Grammar-Based Automated Music Composition in Haskell

Donya Quick
Yale University
donya.quick@yale.edu

Paul Hudak
Yale University
paul.hudak@yale.edu

Abstract

Few algorithms for automated music composition are able to address the combination of harmonic structure, metrical structure, and repetition in a generalized way. Markov chains and neural nets struggle to address repetition of a musical phrase, and generative grammars generally do not handle temporal aspects of music in a way that retains a coherent metrical structure (nor do they handle repetition). To address these limitations, we present a new class of generative grammars called *Probabilistic Temporal Graph Grammars*, or PTGG's, that handle all of these features in music while allowing an elegant and concise implementation in Haskell. Being probabilistic allows one to express desired outcomes in a probabilistic manner; being temporal allows one to express metrical structure; and being a graph grammar allows one to express repetition of phrases through the sharing of nodes in the graph. A key aspect of our approach that enables handling of harmonic and metrical structure in addition to repetition is the use of rules that are parameterized by duration, and thus are actually *functions*. As part of our implementation, we also make use of a music-theoretic concept called *chord spaces*.

Categories and Subject Descriptors H.5.5 [*Information Interfaces And Presentation*]: Information Systems; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages, Algorithms, Design

Keywords Music, Grammar, Algorithmic Composition

1. Introduction

The harmonic analysis of music has long been noted to be analogous to the parsing of natural languages [14]. In the case of a natural language, a sentence, for example, is parsed by starting with the terminal symbols (words), and working backwards to infer their function (noun, verb, etc.), then grammatical phrases (adjective-noun, subject-verb-object, etc.), ending finally with the start symbol representing a sentence. In music, especially in the Schenkerian tradition, a piece of music is parsed by starting with the terminal symbols (notes, rests, and chords), and working backwards to infer local harmonic progressions (say, ii-V-I), song forms (say, AABA), ending finally with a simple I-V-I or even just I (the tonic), serving the function of a start symbol [26, 27].

FARM '13, September 28, 2013, Boston, MA, USA.

ACM 978-1-4503-2386-4/13/09. . . $15.00.
http://dx.doi.org/10.1145/2505341.2505345

Rather than music analysis, however, we are interested in *automated music composition*. One way to approach this is to use grammars *generatively*—that is, to generate sentences from the start symbol. The problem is, with a conventional grammar (such as a context-free grammar, or CFG) the result is usually nonsensical—for example, "The dog wrote the house," or in the case of music, something that just doesn't sound right.

More specifically, conventional grammars intended for automated music composition have the following limitations:

1. They are unable to capture the *sharing* of identical phrases, such as in a song form AABA, where the A sections are intended to be identical (or nearly identical) to one other.
2. They do not take *probabilities* into account. Music analysis has shown that certain productions are more common than others—indeed specific genres of music (say, Bach chorales) have specific distributions of musical characteristics [24].
3. They do not capture *temporal* aspects of music. For example, a production rule stating that a I chord can be replaced with V-I does not capture the fact that the total duration of the two are expected to be the same. Chord symbols in analytical grammars are typically durationless (such as in [23]), despite the importance of rhythm in music [14, 28].

To overcome these problems, we define a new class of generative grammars called *probabilistic temporal graph grammars*, or PTGG's. Technically, this results in a probabilistic context-sensitive grammar with an infinite number of production rules. While such a grammar would be completely intractable if used for parsing, in a generative setting it is highly efficient, and much more expressive.

We use a PTGG to automatically generate the harmonic structure of a composition. However, to go from this harmonic structure to a final composition, we must also choose specific notes to represent the harmonic structure, and special attention must be given to the way these notes form particular "voices" in the harmony (in music theory this task is usually referred to as *voice leading*). To address this task we apply a recent concept from music theory called *chord spaces*, which narrows the search space based on various musical constraints.

In previous work we outlined a grammar similar to PTGG, and also extended the theory of chord spaces and identified some of its inherent computational issues [19, 20]. In this paper we show how PTGG's and chord spaces fit nicely together, and can be elegantly implemented in Haskell [17], with a strong correspondence between mathematical specification and code. We also give several examples demonstrating the success of our approach.

2. Generative System Overview

A summary of our overall framework can be seen in Figure 1. It begins with a PTGG for chord progressions (defined in the Section 3), which is passed to an algorithm for applying the grammar. This

process generates *abstract* musical structure. The chord progressions produced are not tied to any particular style of music.

The next phase of our generative system *interprets* those abstract progressions. We wish to emphasize that the interpretive approach presented in this paper is just one possibility among many. Our approach uses a mathematical construct called *chord spaces* and a constraint satisfaction algorithm to generate music at the level of a MIDI file—roughly the level of representation offered by a paper score. At this stage, our system's output sounds similar to a classical chorale, or a more harmonically diverse equivalent in some cases (we also attempt to produce jazz-like harmonies). However, just because our system produces performable music at this point does not mean that the results are closed to further alteration by either other algorithms or a human. For example, systems such as ours can be employed as an algorithmic component in otherwise human-crafted compositions that could be any number of styles . [1]

Musical features like melody and rhythmic detail are most appropriately handled at the musical interpretation level of this system. However, our implementation does not currently address these features in depth and is primarily concerned with constructing chord progressions. These chord progressions could be further altered to introduce extra melodic or rhythmic features, but the examples contained in this paper have no such additional modification. As a result, all of the chord progressions shown here are homophonic (all voices being rhythmically identical). Post-processing algorithms to address distinctly foreground elements like melody and more complex rhythms are areas of ongoing work.

2.1 Output Format

Our implementation uses the Euterpea library to produce MIDI files as output. Euterpea has its own representation for various musical structures like pitches, notes, and chords. It also supports export of these structures to General MIDI format, which is essentially a collection of note on/off events for each instrument. To produce musical output, the data structures presented here are turned into MIDI via Euterpea's intermediate musical representations. The MIDI data is then easily turned into a visual score using conventional music notation software.

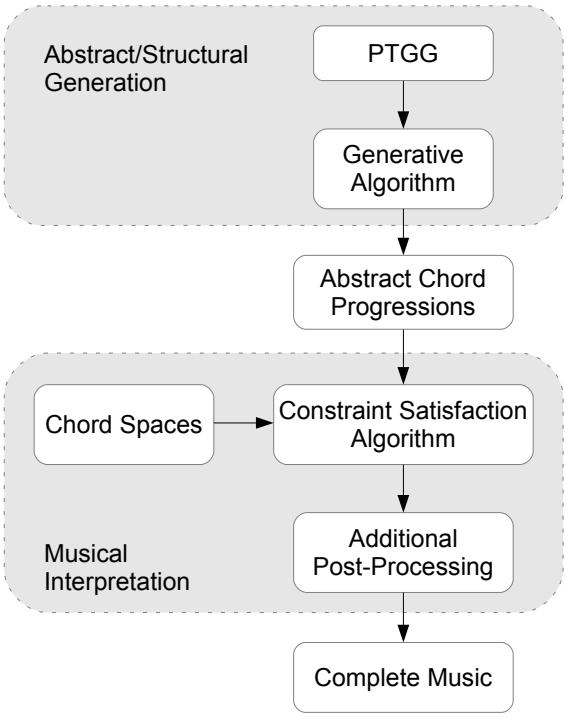

Figure 1. An illustration of the overall structure of our generative system. The first stage of our system creates abstract chord progressions. A generative grammar is used in combination with an algorithm for applying the grammar to produce sequences of Roman numerals. In the second stage of our system, these progressions are fleshed out by using a constraint satisfaction algorithm to traverse chord spaces. The post-processing step in our current system only involves various data type conversions for writing MIDI files, but future systems might include additional post-processing steps for adding melodic and rhythmic development.

3. Grammar Definition

A grammar is a tuple, $G = (N, T, R, S)$ where N is a set of non-terminals, T is a set of terminals, R is a set of rules from $N \to N \cup T$, and $S \in N$ is the start symbol. Terminals are symbols that can only produce themselves, whereas non-terminals have rules that replace them with one or more other symbols.

In [20] we proposed a similar category of grammars to the PTGG defined here. Our PTGG retains the same core concepts:

1. The grammar generates sequences of duration-parameterized abstract chords, written as Roman numerals, and modulation symbols.
2. Chords function as both terminals and non-terminals. Inspired by Schenkerian ideas in music theory, a single, long, abstract chord may be considered representative of a more harmonically diverse elaboration consisting of multiple chords. For example, if a ii-V-I progression may be analyzed as representative of a longer tonic section or I-chord, it is reasonable to allow a long I-chord to produce ii-V-I in a generative setting.
3. Ignoring duration (see below), the grammar is *context free*—the context of a chord does not effect the productions that may be applied to it. However, this does not mean that the *musical interpretation* of the chord is context-free. A Roman numeral appearing in a modulated context implies a different set of pitches than the same Roman numeral in an unmodulated context.
4. Rules are *functions* on the duration of their input symbol. Because durations can be any real number, the set of possible duration-parameterized chords is infinite.

We use the superscript notation c^t to indicate a chord c with duration t. For musical readability, the letters w, h, q, and e are used as shorthands to represent the relative durations of a whole note, half note, quarter note, and eighth note, respectively. Therefore, I^q denotes a I-chord with the duration of a quarter note. Chords can carry any real number as a duration, such as $I^{1.0}$, but those numbers must be assigned a unit of measure (beats, seconds, etc.) to be further musically interpreted.

Chord quality is sometimes captured in Roman numerals based on their case (i being a minor chord, and I being major, for example). However, we do not make this distinction and instead write all chords with upper case Roman numerals to yield the following alphabet:

$$C = \{I,\ II,\ III,\ IV,\ V,\ VI,\ VII\} \quad (1)$$

The simplifying assumption that major and minor modes do not need to be distinguished in the alphabet of Roman numerals was

[1] Our implementation and musical examples of its output are available at `haskell.cs.yale.edu`. Examples provided online include one human-sculpted composition using our system to generate components of a larger electronic work.

made both to allow for a smaller rule set and because it is not clear from existing work how exactly those concepts should be implemented in a generative setting. Sometimes modal disctinctions are ignored in an analytical setting as well. For example, the recently proposed analytical grammar by Martin Rohrmeier in [23] exhibits a small amount of context sensitivity based on mode, but most of the grammar's rules are still context-free and thus make the same simplifying assumption we have made here. Determining how musical contexts such as the current mode should be handled in both the alphabet and construction of rules is an area of ongoing work.

The non-terminals of our grammar are the set of all duration-parameterized chords: $N = \{c^t | c \in C, t \in \mathbb{R}\}$. In keeping with Schenkerian ideas, the start symbol for our grammar is I^t where t is the duration of the entire phrase to be generated.

The chord quality associated with a Roman numeral is determined by the home key and modulation context in which it appears. Modulations can only occur based on diatonic scale degrees. Thus, there are only six possible modulations: one for each scale degree other than the first (which is the current key, or tonic).

$$M = \{M_2, M_3, M_4, M_5, M_6, M_7\} \tag{2}$$

The terminals of our grammar, $T = N \cup M$, include both non-terminals and modulation symbols. Parentheses are used as an additional "meta symbol" for indicating nested structures in generated sequences.

Repetition, or sharing, in our grammar is handled by the use of a let-in syntax to define variables. The notation **let** $x = A$ **in** s means that all instances of x occurring in s should be instantiated with the same value A. The inclusion of these let-in expressions is what creates shared nodes in the graph grammar. Each instance of x in s will point back to the same node (x's definition).

It is important to realize that the let-in notation introduces the concept of variable instances, which is lacking from many generative grammars. For example, **let** $x = A$ **in** xBx is not the same as ABA. This is because in the former, the result of expanding A is shared identically by all instances of x, whereas in the latter each A can be expanded independently.

The set of *sentential forms* K in our grammar is defined recursively as follows:

$$k \in K \ ::= \ c \mid k_1 ... k_n \mid (m\ k_1) \mid \textbf{let}\ x = k_1\ \textbf{in}\ k_2 \mid x \in Var \tag{3}$$

where Var is a set of predefined variable names and $m \in M$ is a modulation.

3.1 Production Rules as Functions

Production rules in our grammar are parameterized by duration, and can thus be thought of as functions. They can be written with concrete durations, such as $I^h \rightarrow V^q\ I^q$ and $I^q \rightarrow V^e\ I^e$. But, in many settings, these are really the "same" rule and can be written as a function of the duration of the input chord: $I^t \rightarrow V^{t/2}\ I^{t/2}$. Duration-parameterized rules allow a finite set of rules to produce an infinite alphabet of duration-parameterized chords.

Not surprisingly, we implement production rules as functions in Haskell [17]. As shown in the following section, treating rules as functions allows the grammar itself to capture many musically relevant behaviors that would otherwise be delegated to an algorithm for applying the grammar. Rules can create repetition as well as exhibit conditional behavior, yielding complex structures with even a very simple generative algorithm. Haskell allows for an elegant implementation of these rules and the generative algorithm.

Finally, a PTGG is a probabilistic grammar, and thus each rule (there may be several rules for each non-terminal) is associated with a probability.

4. Grammar Implementation

We present an implementation of PTGG in Haskell that closely mirrors the presentation above. Simple data types capture the essence of chords, modulations, let-in terms, and sentential forms. As mentioned earlier, functions are used to implement production rules, and are paired with a probability. In addition, we describe a generative algorithm in monadic style that chooses rules based on their associated probabilities.

4.1 Chords, Progressions, and Modulations

Roman numerals represent chords built on scale degrees.

```
data CType = I | II | III | IV | V | VI | VII
  deriving (Eq, Show, Ord, Enum)
```

Key changes, or modulations, in our grammar also take place according to scale degrees. Similarly to the Roman numeral system for labeling chords, we define symbols indicating modulations for the 2^{nd} through 7^{th} scale degrees.

```
data MType = M2 | M3 | M4 | M5 | M6 | M7
  deriving (Eq, Show, Ord, Enum)
```

We now define a data structure to capture the sentential forms of PTGG, called *Term*. This data type has a tree structure to model the nested nature of chord progression features like modulations and repetition. A *Term* can either be a *non-terminal* (*NT*) chord, a *sequence* (*S*) of terms, a term modulated to another key (*Mod*), a let-in expression (*Let*) to capture repetition, or a variable (*Var*) to indicate instances of a particular phrase.

```
data Term =
  NT Chord | S [Term] | Mod MType Term |
  Let Var Term Term | Var Var
type Var = String
```

4.2 Rules

We begin with the following type synonyms for clarity in type signatures.

```
type Prob = Double
type Seed = Int
type Dur = Rational
```

A rule is a function from duration-parameterized chords to a chord progression. Chord progressions are represented as a *Term*. Because more than one rule may exist for a particular Roman numeral, each rule also has a probability associated with it.

```
data Rule = (CType, Prob) :-> RuleFun
type RuleFun = Dur -> Term
```

We also introduce abbreviations for single-chord *Term* values.

```
i, ii, iii, iv, v, vi, vii :: RuleFun
[i, ii, iii, iv, v, vi, vii] =
  map (\c t -> NT (Chord t c)) $ enumFrom I
```

For example, the rule $I^t \rightarrow V^{t/2}\ I^{t/2}$ with probability p would be written:

```
(I, p) :-> \t -> S [v (t / 2), i (t / 2)]
```

The following are some specific rules taken from our implementation that represent the three main forms of our rules. Rules may produce a sequence of chords, a modulated section, or no change (an identity rule).

```
ruleV1 = (V, 0.15) :-> \t -> S [iv (t / 2), v (t / 2)]
ruleV7 = (V, 0.10) :-> (Mod M7 . v)
ruleV9 = (V, 0.10) :-> v
```

Although rules according to Schenkerian theory and the metrical structures in work like Generative Theory of Tonal Music (GTTM) [14] would enforce that the chord durations on the right-hand side sum to 1.0 and follow basic metrical divisions (such as powers of 2), this is not a strict requirement of our grammar. In fact, interesting rhythmic patterns can be created with rules that mix metrical structures and add or subtract duration.

Rules can also create repetition using *Let* expressions. In the rule sets used for our examples, we make use of the following rules:

$$X^t \rightarrow \textbf{let } x = X^{t/2} \textbf{ in } x\ x \tag{4}$$

$$X^t \rightarrow \textbf{let } x = X^{t/4} \textbf{ in } x\ X^{t/2}\ x \tag{5}$$

$$X^t \rightarrow \textbf{let } x = X^{t/4} \textbf{ in } x\ V^{t/2}\ x \tag{6}$$

Because rules are functions, they are more powerful than simply being a table of input and output values. The rules can encapsulate additional aspects of functionality that would otherwise be delegated to the algorithm applying the grammar. Our rules already demonstrate this to some degree by using an infinite alphabet to accommodate durations and by handling repetition within rules. Rules can go further than this though and exhibit *conditional* behavior.

One problematic aspect of the generative process is how to obtain a "nice" distribution of durations that meets musical expectations for some genre. In a chorale, one would expect a lot of quarter notes and perhaps some half and eighth notes, but no notes spanning half the duration of the piece. In jazz, the distribution of durations would be more diverse, but one would still not expect to see very uneven distributions such as a burst of 64^{th} notes followed by a lengthy passage consisting entirely of whole notes.

Even when metrical structure is built into the structure of the rules, stochastic generation can easily create distributions of durations that give no sense of meter and/or absurdly long and short durations. One way to avoid this is to delegate the decision to the algorithm applying the grammar: apply rules left to right whenever possible except for notes that are "too short" for our desired distribution. The distribution of durations is then controlled by other aspects of the grammar and the generative algorithm, such as the probabilities of self-productions (e.g. $I^t \rightarrow I^t$) and the number of generative iterations used.

With a PTGG, there is a more elegant, functional approach to this by encoding the decision making directly into the rules:

```
myRuleFun :: RuleFun
myRuleFun d = if d < durLimit then term1 else term2
```

where $term_1, term_2 :: Term$. This approach allows for a very simple implementation of the grammar's generative algorithm, since the rule set encapsulates all of the complex behavior of the grammar.

4.3 Generating Chord Progressions

Our strategy for applying a PTGG generatively is to begin with a start symbol and choose a rule randomly, but biased by the associated probability. For each successive sentential form, *all* nonterminals are expanded "in parallel."[2]

4.3.1 The *Prog* Monad

Because this strategy is stochastic, randomness must be threaded through the generative process to help with decision making. We achieve this with a simple state monad to thread Haskell's "standard generator" for random numbers. While we could have used Haskell's existing definition for *State*, we opted to define our own monad for added transparency.

[2] This strategy is similar to that used for an L-system or Lindenmayer system [18].

```
newtype Prog a = Prog (StdGen -> (StdGen, a))

instance Monad Prog where
  return a = Prog (\s -> (s, a))
  Prog p0 >>= f1 = Prog $ \s0 ->
    let (s1, a1) = p0 s0
        Prog p1 = f1 a1
    in  p1 s1
```

In addition, we define a single "domain specific" operation to generate a new random number from the hidden standard generator:

```
getRand :: Prog Prob
getRand = Prog (\g ->
  let (r, g') = randomR (0.0, 1.0) g in (g', r))
```

Finally, we define a way to "run" the monad:

```
runP :: Prog a -> StdGen -> a
runP (Prog f) g = snd (f g)
```

4.3.2 Applying Rules

A chord, $X^t \in N$, can be replaced using any rule where X appears on the left-hand side. Since there may be more than one such rule, the *applyRule* function stochastically selects a rule to apply according to the probabilities assigned to the rules. For a rule, $(c,p) :\!\!> rf$, we use the functions *lhs*, *prob*, and *rhs* to gain access to its *CType*, *Prob*, and *RuleFun* respectively.

```
applyRule :: [Rule] -> Chord -> Prog Term
applyRule rules (Chord d c) =
  let rs = filter (\((c', p) :!!> rf) -> c' == c) rules
  in do r <- getRand
        return (choose rs r d)

choose :: [Rule] -> Prob -> RuleFun
choose [] p = error "Nothing to choose from!"
choose (((c, p') :!!> rf) : rs) p =
  if p <= p' || null rs then rf else choose rs (p - p')
```

4.3.3 Parallel Production

The *Prog* monad can be used to write a generative function that runs for some number of iterations, with each iteration making a pass over the entire *Term* supplied as input to that iteration.

In a single iteration of the generative algorithm, a *Term* is *updated* in a depth-first manner to update the leaves (the *NT* values representing chords) from left to right. For *Let* expressions of the form **let** $x = t_1$ **in** t_2, the terms t_1 and t_2 are updated independently, but instances of x are *not* instantiated with their values at this stage. Otherwise, it would be trickier to ensure that all instances of x are generated the same way.

```
update :: [Rule] -> Term -> Prog Term
update rules t = case t of
  NT x      -> applyRule rules x
  S s       -> do ss <- sequence (map (update rules) s)
                  return (S ss)
  Mod m s   -> do s' <- update rules s
                  return (Mod m s')
  Var x     -> return (Var x)
  Let x a t -> do a' <- update rules a
                  t' <- update rules t
                  return (Let x a' t')
```

Finally, we define a function *gen* that iteratively performs the updates:

```
gen :: [Rule] -> Int -> Seed -> Term -> Term
gen rules i s t = runP (iter (update rules) t) (mkStdGen s) !! i
```

The function, *iter*, simply iterates a monadic action infinitely often.

$$
\begin{aligned}
&\mathit{iter} :: \mathit{Monad}\ m \Rightarrow (a \to m\ a) \to a \to m\ [a] \\
&\mathit{iter}\ f\ a = \mathbf{do}\ a' \leftarrow f\ a \\
&\qquad\qquad\quad as \leftarrow \mathit{iter}\ f\ a' \\
&\qquad\qquad\quad \mathit{return}\ (a' : as)
\end{aligned}
$$

Note that Haskell's laziness extends into the monad, and so the infinite list that results from its use is evaluated lazily.

The result of calling *gen* on a *Term* for some number of iterations will be a *Term* that may contain *Let* expressions. Retaining this structure allows us to extract constraints that aid in the musical interpretation of the *Term*.

4.4 Musical Interpretation

A *Term* is a tree data structure with many abstract musical features that must be interpreted in the context in which they appear. Chords must be interpreted within a key, and the key is dependent on the modulation structure of the branch. Variables refer to instances of a specific chord progression, which may have nested Let expressions.

To produce a sequence of chords that can be interpreted musically, the structure of Let statements must be *expanded* by replacing variables with the progressions they represent. This is important because the interpretation of a variable's chords hinges on the context in which the variable appears. Consider the following expression and what happens when variables are instantiated with their values, where the notation $a \Rightarrow b$ means "a evaluates to b."

$$\textbf{let}\ x = I^t\ \textbf{in}\ x\ (M5\ x) \Rightarrow I^t\ (M5\ I^t) \tag{7}$$

In this example, the two instances of x must be interpreted in two different keys in the final progression. If the passage occurs in C-major, then the first x is a C-major chord, but the second is a G-major chord.

When *Let* expressions appear in rules, the variable names in a generated progression are not guaranteed to be unique. In fact, duplicate variable names can be quite common. We use lexical scoping to handle these situations, always taking a variable's nearest (innermost) binding in the *Term* tree as shown below.

$$\textbf{let}\ x = I^t\ \textbf{in}\ x\ (\textbf{let}\ x = V^{t'}\ \textbf{in}\ x\ x)\ x \Rightarrow I^t\ V^{t'}\ V^{t'}\ I^t \tag{8}$$

The *expand* function accomplishes this behavior, replacing instances of variables with their values under lexical scope, by maintaining an environment of variable definitions.

```
expand :: [(Var, Term)] → Term → Term
expand e t = case t of
  Let x a exp → expand ((x, expand e a) : e) exp
  Var x       → maybe (error (x ++ " is undefined")) id $
                  lookup x e
  S s         → S (map (expand e) s)
  Mod m t'    → Mod m (expand e t')
  x           → x
```

These abstract progressions may then be further musically interpreted using *chord spaces* and musical *constraints* as described in the following sections.

5. Chord Spaces

A *chord space* is a way of grouping chords in musically useful ways by using equivalence relations (relations that are reflexive, symmetric, and transitive). The work in [19] and [20] adds additional formalization for four important equivalence relations defined by Tymoczko et al. [29] and Callender et al. [5]: octave equivalence (O), permutation equivalence (P), transposition (T), and cardinality equivalence (C). Chords are O-equivalent if they share the same vectors of pitch classes, P-equivalent chords share the same multisets of pitches, and T-equivalent chords have the same intervallic structure of pitches. These concepts can also be combined. For example, OP-equivalent chords share the same sets of pitch classes, and OPT-equivalent chords have the same intervallic structure of pitch classes—a level of abstraction that captures chord quality. OPC equivalence relates chords that share the same *multisets* of pitch classes, thereby relating chords with different numbers of voices.

Chord spaces can be used to move between different levels of abstraction in music [19, 20]. An *abstract* chord is one lacking information needed for a particular task. A *concrete* chord is one with complete information for the task. The chord spaces we use capture different levels of musical abstraction, with concrete chords being a vector of pitches, sometimes called a voicing.

A chord space is a type of *quotient space*, or the result of applying an equivalence relation to a set. The quotient space formed by applying equivalence relation R to set S is denoted S/R. A chord space is therefore a set of equivalence classes, and we can choose a single *representative point* from each equivalence class to form a *representative subset* of the space. Even if the set of chords in a chord space is infinite, the representative subset of that space may be finite. This is the case for OPC-space, of which we make extensive use in this paper. The representative subset of a chord space can be thought of as a different level of abstraction, where each point represents many other points in the chord space. The smaller the representative subset, the easier it is to traverse. Representative subsets for O-, P-, and OP-space are shown in Figure 3.

The chord spaces used here represent pitches as integers, and a concrete chord is a vector of integers with one element per voice. The notation $\langle x_1, ..., x_n \rangle$ enumerates the elements of a vector.

Our grammar produces abstract progressions consisting of Roman numerals and modulations. We use chord spaces to help simplify the process of mapping *Term* values to performable chords. It becomes a path-finding problem with constraints on the path's shape, as illustrated in Figure 4. For a given *Term*, the following steps convert abstract chords to concrete ones:

1. *Expand* the *Term* to place repeated patterns in their contexts.
2. For each *CType* in a *Term* value, determine its scale degrees and key/mode context from the placement of *Mod* branches in the *Term* value, and then map it to a simple triad (ex: I in C-major would be $\langle 0, 4, 7 \rangle$).
3. Use a chord space and behavioral constraints (such as those from *Let* expressions) to map the chord from step 2 to a more interesting chord. More than one chord space can be used if the decision-making process and constraints can be broken into multiple levels of abstraction.

For example, starting with the phrase a $(Mod\ M5\ I^t)$ interpreted in C-major, the I-chord's scale degrees of 1, 3, and 5 would map to $\langle 7, 11, 14 \rangle$ for a G-major scale starting on pitch 7. Using OPC-space, $\langle 7, 11, 14 \rangle$ could then be turned into a chord of higher-cardinality in a different range, such as $\langle 50, 55, 59, 67 \rangle$ (pitch classes D, G, B, and G respectively). The duration of the chord, t, would remain unchanged.

In our examples, we also make use of another type of chord space that we will refer to as a *jazz space*. The goal of this new chord space is to provide a more diverse interpretation of Roman numerals for 4-voices using seventh chords. While our use of OPC-space takes a traditional interpretation of Roman numerals (a triad with one of the voices being doubled), our "jazzy" interpretation of the chords is based on the mode that would be formed by a scale

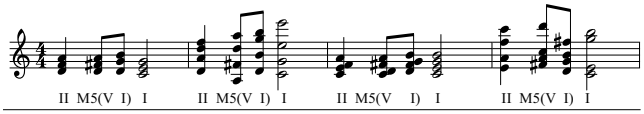

Figure 2. A musical score representation of the example mappings of the progression detailed in section 5.1. Each measure contains a different mapping of the same example. From left to right, they are: block trichords (Equation 10), an OPC-space mapping of those trichords (Equation 11), block jazz chords from our jazz space (Equation 12), and those jazz chords mapped through OPC-space (Equation 13).

starting on the scale degree indicated by the Roman numeral (a *II*-chord in a major key would be the Dorian mode). This allows a single Roman numeral to map to more than one possible set of pitch classes. Once a chord's mode is determined, it can take one of two forms within the mode, covering either the root, third, fifth, and seventh or the second, third, fifth, and seventh. These chords within the jazz space are simply block chords representing collections of pitch classes. To achieve more varied voicings and different ranges, OPC-space must be employed.

5.1 A Small Example

To illustrate the roll of chord spaces in our system's generative process from start to finish, consider the following series of productions.

Progression	**Rules Applied (Left to Right)**
I^w	Start symbol
$II^q\ V^q\ I^h$	$I^t \rightarrow II^{t/4}\ V^{t/4}\ I^{t/2}$
$II^q\ M_5(I^q)\ I^h$	$II^t \rightarrow II^t,\ \ V^t \rightarrow M_5(I^t),\ \ I^t \rightarrow I^t$
$II^q\ M_5(V^e\ I^e)\ I^h$	$II^t \rightarrow II^t,\ \ I^t \rightarrow V^{t/2}\ I^{t/2},\ \ I^t \rightarrow I^t$

The final chord progression, $II^q\ M_5(V^e\ I^e)\ I^h$, contains four different chords, the middle two of which are modulated to the dominant. Suppose this progression were to be interpreted in C-major. The basic pitch classes and corresponding pitch numbers within the range [0,11] would be the following:

$$\langle D,F,A\rangle\ \langle D,F\#,A\rangle\ \langle D,G,B\rangle\ \langle C,E,G\rangle \tag{9}$$

$$\langle 2,5,9\rangle\ \langle 2,6,9\rangle\ \langle 2,7,11\rangle\ \langle 0,4,7\rangle \tag{10}$$

When represented as triads based on their modal context, Roman numerals from our PTGG have a one-to-one mapping to points in representative subsets of both OP- and OPC-space. Once in the form shown above, the chords can be mapped outside of the representative subset to obtain a more interesting progression. Using OP-space, the octaves and order of the voices can be changed, and, with OPC-space, the number of voices can be changed as well. Using OPC-space for four voices, one possible mapping would be:

$$\langle 48,57,65,69\rangle\ \langle 50,57,62,65\rangle\ \langle 50,59,67,71\rangle\ \langle 48,55,64,76\rangle \tag{11}$$

Similarly, one possible mapping through our jazz space can be seen below. Note that the first chord omits its root, while the other three chords include it.

$$\langle 0,4,5,9\rangle\ \langle 0,2,6,9\rangle\ \langle 2,6,7,11\rangle\ \langle 0,4,7,11\rangle \tag{12}$$

Finally, the jazz progression above can be mapped to a less blocky series of chords using OPC-space.

$$\langle 52,57,65,72\rangle\ \langle 54,57,60,74\rangle\ \langle 50,55,59,66\rangle\ \langle 48,52,67,71\rangle \tag{13}$$

A score representation of the following example mappings can be seen in Figure 2. It is important to emphasize that each of these mappings is only one of many possible. Every chord progression has many possible equivalent progressions when mapped through a chord space. Which progression is chosen will depend on what other constraints are applied (selecting for voice-leading smoothness, etc.) and the decision-making process can be stochastic as well. These issues are covered in more detail in later sections.

5.2 Chord Spaces Implementation

Chord spaces are a type of quotient space: a set of elements under an equivalence relation. We implement quotient spaces more generally and then create specific chord spaces using particular representations of chords. For the O, P, T, and C relations, we represent chords as lists of integers, where each integer corresponds to a pitch. Pitches are numbered such that a $(C,0) = 0$, $(C\#,0) = 1$, and so on. For clarity, we will use the following type synonym.

type $AbsChord = [Int]$

An equivalence relation R over set S can be mathematically represented as pairs of elements in S, where $(a\ \in S,\ b \in S) \in R$ implies that a and b are related. However, equivalence relations can also be viewed as a Boolean test: given two elements of S, a and b, return True iff $(a,b) \in R$ [19].

type $EqRel\ a = a \rightarrow a \rightarrow Bool$

Two chords are OPC-equivalent if they have the same sets of pitch classes. A test for OPC-equivalence can, therefore, be implemented as follows.

$opcEq :: EqRel\ AbsChord$
$opcEq\ a\ b = f\ a == f\ b$ **where** $f = nub \circ sort \circ map\ (`mod`12)$

We implement sets as lists. A quotient space, therefore, is implemented as a list of lists. Each sublist represents an equivalence class.

type $QSpace\ a = [[a]]$

Given a set (list) of elements of type a and an equivalence relation represented as a function of type $EqRel\ a$, a quotient space can be formed by the following operator, which mirrors the S/R notation used to denote quotient spaces.

$(//) :: (Eq\ a) \Rightarrow [a] \rightarrow EqRel\ a \rightarrow QSpace\ a$
$[]\ //\ r = []$
$xs\ //\ r =$
  let $sx = [y \mid y \leftarrow xs, r\ y\ (head\ xs)]$
  in $sx : [z \mid z \leftarrow xs, \neg\ (elem\ z\ sx)]\ //\ r$

OPC-space is implemented as a *QSpace AbsChord*. For a collection of chords s (which should consist of all chords that could possibly be needed for a particular application), the corresponding subset of OPC-space would be calculated by $s\ //\ opcEq$.

Functions of type *EqRel* can also be used to find an element's equivalence class given a quotient space.

$eqClass :: (Eq\ a) \Rightarrow QSpace\ a \rightarrow EqRel\ a \rightarrow a \rightarrow EqClass\ a$
$eqClass\ qs\ r\ x =$
  let $ind = findIndex\ (\lambda e \rightarrow r\ x\ (head\ e))\ qs$
  in $maybe\ (error\ (\texttt{"Class not found."}))\ (qs!!)\ ind$

While the type *AbsChord* is suitable for the OPC-equivalence, our jazz space implementation actually requires a slightly different representation. The mode in which a chord appears is very important in jazz, and so chords are represented as a pair of a mode (represented as a scale) and the collection of pitches in the chord (an *AbsChord*). To map abstract chords like Roman numerals through our jazz space, they are first tagged with their corresponding modes before making use of the functions above.

5.3 Novelty

One problem with searching through organized collections of musical features is obtaining a range of reasonably different possible interpretations of a given abstract progression. This is a potential issue for using chord spaces to musically interpret our grammar's abstract progressions. Using depth-first search, adjacent solutions are likely to be very similar or even nearly identical, particularly when the constraints are very relaxed and solutions are abundant. To more effectively explore a range of solutions, random paths would be more likely to capture diversity.

A simple answer to this problem is to randomize the chord space before it is used. This can be done by finding a random permutation on the set of chords used before grouping them according to an equivalence relation. The first solution found by a depth-first search of a chord space whose elements have been randomly permuted will show better diversity relative to the first solution from a differently randomized version of the space than would be the case for two adjacent solutions in the same space. Because the number of chord progressions that can be generated from a chord space are likely to be much larger than the chord space itself (exponentially larger, in fact), randomizing the order of elements in the chord space helps shift some of the burden of "novelty" to a smaller domain that is easier to manipulate.

6. Constraint Satisfaction

Constraints can be added to the path-finding process through chord spaces to obtain solutions with desired characteristics. Voice-leading constraints can be captured as pair-wise constraints between chords [19], and other, more complex structural constraints can be modeled as constraints on entire or partial progressions.

The *Let* expressions in our grammar represent a type of progression-level constraint where abstract chords appearing within certain ranges must be mapped to the same concrete chords. Consider the following progression and its expanded interpretation:

$$\textbf{let } x = (\textbf{let } y = V^{t1}\ I^{t2} \textbf{ in } y\ y) \textbf{ in } x\ IV^{t3}\ I^{t4}\ x \tag{14}$$

$$V^{t1}\ I^{t2}\ V^{t1}\ I^{t2}\ IV^{t3}\ I^{t4}\ V^{t1}\ I^{t2}\ V^{t1}\ I^{t2} \tag{15}$$

The *Let* expressions require that chords at positions 1-2 and 3-4 must be the same subprogressions, and similarly for chords at positions 1-4 and 7-10. Because sub-constraints for chords 1-4 are already specified, there is no need to redundantly assert that chords 7-8 and 9-10 must be the same phrases. The *Let* structure of a generated progression can directly yield these types of constraints by examining the lengths of the variables' values and the positions at which they appear.

For a quotient space q and equivalence relation r, simply calling $head \circ eqClass\ q\ r$ on each chord in a progression will automatically satisfy the constraints of any *Let* expression even when q has had the members of its equivalence classes randomly permuted. This is because each Roman numeral will only be mapped to one value regardless of how it is constrained. However, this *Let*-satisfying progression is unlikely to satisfy any other constraints, and so a more complex traversal of the solution space is required in such a case.

The constraints created by *Let* expressions can be used to aggressively prune the solution space before traversing it. Chord progressions can be viewed as a list of indices into equivalence classes at some level of abstraction, and a naive approach to solving constraints would be to simply perform a depth-first search of all possible progressions sharing the same sequence of equivalence classes.

Consider a progression of length six with the constraints that chords 1-2 must be the same and chords 1-3 and 4-6 must be the same phrases. For simplicity, we will assume there are only two equivalence classes in the chord space with only two elements each: $e_1 = \{a,b\}$ and $e_2 = \{c,d\}$, where a, b, c, and d are chords. If the pattern of equivalence classes is $e_1, e_1, e_2, e_1, e_1, e_2$ (following the constraints), then an incremental search of all progressions sharing the same equivalence classes would look like the following.

Number	Indices	Solution
1	0,0,0,0,0,0	*aacaac*
2	1,0,0,0,0,0	*bacaac*
3	0,1,0,0,0,0	*abcaac*
...	...	
64	1,1,1,1,1,1	*bbdbbd*

Clearly many of these progressions will not even satisfy the *Let* constraints, let alone any extra constraints we may wish to satisfy on top of those. The more constraints exist, the sparser the solutions become in a sea of garbage. On larger problems, traversing the solution space in this way quickly becomes intractable. Fortunately, there is a way to skip the cases that do not satisfy *Let* expressions.

Imagine a search tree consisting *exclusively* of *Let*-satisfying progressions, where all other progressions have been pruned away. In the example above, there are only four such progressions: *aacaac*, *bbcbbc*, *aadaad*, and *bbdbbd*, so there would only be four leafs in such a tree. Doing a depth-first search through this *Let*-satisfying tree as a means to satisfy additional constraints is clearly more efficient, since the only solutions examined are already guaranteed to satisfy the *Let* constraints even if not additional constraints. Applying this strategy to the 6-chord example above, the four *Let*-satisfying solutions would be traversed as follows:

Number	Indices	Solution
1	0,0,0,0,0,0	*aacaac*
2	1,1,0,1,1,0	*bbcbbc*
3	0,0,1,0,0,1	*aadaad*
4	1,1,1,1,1,1	*bbdbbd*

The task of jumping from one *Let*-satisfying progression to another is reducible to the process of incrementing an n-digit number where each digit can have a different number base and some digits' values are tied to others. Indices that are subject to *Let* constraints will move in lockstep, fully avoiding any progressions that do not satisfy the *Let* constraints.

6.1 Constraint Satisfaction Implementation

We denote constraints for *Let* expressions using the type synonym *Constraints* for pairs of indices into a chord progression.

type $Index = Int$
type $Constraints = [[(Index, Index)]]$

Each tuple in the inner lists represents a range of indices inclusive of its endpoints. Each member of the outermost list (elements of type $[(Int, Int)]$) represents ranges of indicies that must be instantiated with the same phrases. Indices start from zero. This structure can be derived directly from an unexpanded *Term* containing *Let* expressions. For example, consider the following (durations are omitted for brevity since they are not relevant):

$$\textbf{let } x = (\textbf{let } y = II \textbf{ in } y\ y\ V\ I) \textbf{ in } x\ x \tag{16}$$

From this we can clearly derive that chords 1 and 2 must be the same and that chords 1-4 and 5-8 must be the same phrases, yielding $[[(0,0),(1,1)],[(0,3),(4,7)]] :: Constraints$. Note that chords 5-6 (indices 4 and 5) will be the same as well if these constraints are satisfied.

Constraint values must be *well-formed* to be used in our algorithms. The value $[...,k_1,...,k_2,...]::Constraints$ is well-formed if there is no index range in k_1 that is further constrained by k_2. In other words, $[[(0,0),(1,1)],[(0,3),(4,7)]]$ is well-formed but

$[[(0,3),(4,7)],[(0,0),(1,1)]]$ is not. Additionally, partially overlapping *Index* pairs are not allowed. $[[(0,3),(4,7)]]$ is well-formed, but $[[(0,3),(2,6)]]$ is not.

To satisfy *Let* constraints, we represent a chord progression as indices into those chords' equivalence classes (indexed from zero) rather than as a list of actual chords (such as the *AbsChord* type described previously). For a progression of length n, there will be n equivalence classes and indices. Two chords having the same equivalence class (such as two C-major triads) do not necessarily need to have the same index into that equivalence class unless constrained by a *Let* expression. As already mentioned, these n indices can be thought of as an n-digit number where each digit has a base determined by the length of its equivalence class. Similarly, depth-first search through the chord space can be viewed as incrementing this list of indices from $0,...,0$ to $l_1-1,...,l_n-1$ where l_i is the length of the i^{th} chord's equivalence class. We treat the leftmost index as least significant.

Indices are referred to as *free indices* if they are not constrained by any indices to their left in the progression. Given the constraints $[[(0,0),(1,1)],[(0,3),(4,7)]]$ for a progression of length 8, the only free indices are 0, 2, and 3. With constraints applied from left to right, indices 1, 4, and 5 will move in lockstep with index 0, and similarly for the phrases defined by indices 6-7 and 2-3.

Finding the next *Let*-satisfying set of indices can be broken down into a three-step process:

1. Derive a list of free indices from the progression's *Let* constraints.
2. Attempt to increment that set of free indices by one, overflowing to the next free index if needed.
3. Apply the constraints to all non-free indices.

We define a function for each of those steps. Given the length of a progression, n, and well-formed $k :: Constraints$, the following function finds indices that can be incremented to traverse the solution space while satisfying *Let* constraints.

```
freeInds :: Int → Constraints → [Int]
freeInds n k =
  let k' = map (map (λ(i,j) → [i..j])) k
      t  = nub $ concat $ concatMap tail k'
  in filter (¬ ∘ (∈ t)) [0..n−1]
```

For $x = freeInds\ n\ k$, indices not in x will be constrained by and move in lockstep with indices within x.

The *incr* function below performs the step of incrementing free indices. Each index is tagged with two values: a Boolean flag indicating whether the index is free and the length of the equivalence class at that index (an *Int*).

```
incr :: [(Bool, Index, Int)] → [Index]
incr [] = error "No more solutions."
incr ((b,i,l):xs) = let is = map (λ(x,y,z) → y) xs in
  if b then if i ⩾ l−1 then 0 : incr xs else i+1 : is
  else i : incr xs
```

The returned value only increments free indices, while other indices remain unchanged. These constrained indices are handled by the *applyCons* function below, which copies the values of free indices over to the other indices that they constrain.

```
applyCons :: [Index] → [(Index, Index)] → [Index]
applyCons inds [] = inds
applyCons inds ((i,j):t) = foldl (f val) inds (map fst t) where
  val         = (take (j−i+1) $ drop i inds)
  f val src i = take i src ++ val ++
                drop (i + length val) src
```

Finally, the *findNext* function makes use of each of three functions above. It takes a set of *Constraints*, the current list of indices, and a list of equivalence class lengths, and returns a new list of *Let*-constraint-satisfying indices.

```
findNext :: Constraints → [Index] → [Int] → [Index]
findNext k is lens =
  let bs = map (∈ freeInds (length is) k) [0..length is − 1]
      xs = zip3 bs is lens
  in foldl applyCons (incr xs) k
```

As already mentioned, the very first progression (indices $[0,0,...,0]$) will always satisfy the *Let* constraints, so subsequent progressions only need to be explored in order to satisfy additional constraints or to obtain more diverse progressions. Any additional constraints are satisfied by recursively calling *findNext* until a solution is found (assuming one exists). The stricter the constraints are, the harder it will be to satisfy them and the longer it will take on average to find a solution.

6.2 Additional Constraints

In previous work [19], we presented a method for generating chord progressions using chord spaces while satisfying the following constraints:

- Avoiding parallel motion, which is when two voices move by the same amount in the same direction.
- Avoiding voice crossing, which is when a higher voice moves below a previously lower voice.
- Regulating voice-leading smoothness by restricting the range of movement in the voices.

This work also presented two approaches for generating constraint-satisfying progressions using chord spaces: a depth-first search that is guaranteed to return a solution if one exists and a greedy approach that will always return a solution but is not guaranteed to satisfy the constraints perfectly. Each general strategy has its merits under different circumstances. The first is ideal when constraints are easily satisfiable and the second can be preferable in a large solution space when constraints are harder to satisfy—particularly if the constraints are "soft" in the sense of many compositional guidelines and do not need to be completely satisfied to obtain a suitable solution.

As we found in [19], constraints like avoiding parallel motion and voice-crossing are actually quite easy to satisfy under musically plausible settings (i.e. that the chord space isn't set up in such a way as to make it deliberately hard or impossible) when they are the only constraints involved. The constraints we used only concerned adjacent pairs of chords and, as a result, were easily satisfied under most circumstances with even the greedy approach. Because solutions were abundant under these pairwise constraints, depth-first search was sufficient to find them in most cases. However, constraints of the form presented by the *Let* expressions here span the entire chord progression. Because of how much the *Let* constraints narrow the number of viable solutions, it suddenly becomes very unlikely to find a solution by simple depth-first search. Although depth-first search would still find a solution if one existed, the runtime will be terrible because of the amount of the solution space that would likely be explored in the process. Similarly, because the *Let* constraints can be hierarchical and span the entire progression, greedy approaches such as the one in [19] are unlikely to find satisfactory solutions.

Our algorithm for generating solutions with chord spaces that satisfy *Let* expressions is useful because it avoids examining anything that wouldn't satisfy at least those constraints. When finding the next solution, it should be possible to also satisfy some addi-

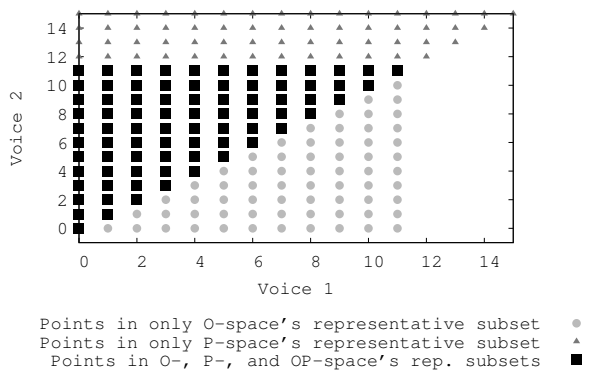

Figure 3. Representative subsets for O-space, P-space, and OP-space for two voices over the range $[0, 14]^2$. When path-finding can be conducted on representative subsets, smaller such sets result in smaller and more tractable solution spaces.

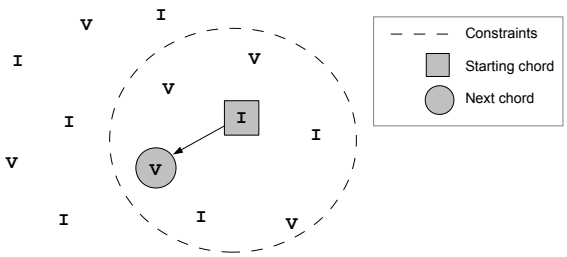

Figure 4. An illustration of the path-finding nature of chord spaces for a *I*-*V* progression. Each Roman numeral can be mapped to many concrete chords, which may literally be thought of as chords floating in space. When we choose a specific *I*-chord, the next transition may be subjected to various voice-leading or other constraints that limit the number of viable choices for the next chord. This defines a region of acceptable solutions for the next chord, which may be chosen stochastically if more than one option exists within that area.

tional pairwise constraints at the same time. This would avoid some problems induced by handling these extra constraints at the progression level. Even some fairly short progressions (such as those of length 12) can result in searching through millions or even billions of unsatisfactory progressions when trying to satisfy additional constraints using our *Let*-based search strategy. The number of solutions searched could be dramatically reduced yet again if some pairwise constraints could be addressed at the same time as the *Let* constraints when incrementing indices into equivalence classes. However, this type of extension to our approach is an area of ongoing work and, therefore, has not yet been implemented.

7. Related Work

Generating harmony is a popular subject in automated composition research. A wide variety of algorithms have been explored, including Markov chain-based approaches [6, 31], neural nets [2, 3, 10], and more specialized systems intended for generating whole compositions [7, 8].

Grammars have been explored both generatively and analytically in music [9, 12, 30]. Studies on brain activity have shown a strong link between language and music in the brain [4], an idea that has become increasingly accepted in music theory through works like GTTM, which presents a grammatical outlook on analyzing music [14] (although it requires additional formalization to be implemented in both analytical and generative settings). Graph grammars, which can account for repetition through the use of shared nodes have been occasionally used in musical settings, such as to aid in composition with audio samples [25] and for representing aspects of musical scores [1].

Martin Rohrmeier introduced a mostly context-free grammar (CFG) for parsing classical Western harmony [23]. The grammar is based on the tonic, dominant, and subdominant chord functions. Terminals are the Roman numerals from I to VII, and the non-terminals are *Piece*, *P* (phrase), *TR* (tonic region), *DR* (dominant region), *SR* (subdominant region), *T* (tonic), *D* (dominant), *S* (subdominant), and four chord function substitutions. However, this grammar has no support for important features like repetition and duration, and so is problematic in a generative setting without additional supervision. The HarmTrace package, written in Haskell, builds on Rohrmeier's grammar to automate harmonic analysis [15]. FHarm, a later system that also uses Haskell, addresses the task of melodic harmonization using HarmTrace to filter out results that best match a particular harmonic model [13]. A fundamental difference between our system and FHarm is that FHarm harmonizes an existing melody, whereas our system performs composition from scratch and does not currently address any melody-specific features.

Meter is clearly an important aspect of music. In work such as GTTM, meter interacts with harmonic aspects of the music through metrical grouping and preference rules [14]. Temperley's work [28] as well as a harmonic analysis algorithm by Raphael and Stoddard [21] also emphasize the role of rhythm and meter in the perception of harmony. However, meter is often treated separately in generative settings, such as in the grammars for jazz riffs presented by Keller and Morrison [12].

Repetition is another feature of music that is often ignored by generative algorithms. Consider a fugue: the subject that opens the piece is expected to appear in modified states later on in the music. If these constraints are ignored, the form of the music is violated. The various musical grammars discussed so far have little to no direct support for this kind of musical feature, and many other algorithms are fundamentally incapable of supporting it as well. Markov chain [11, 31] and most Neural Net-based [2, 10] approaches lack the ability to enforce any sort of pattern repetition over long spans of time without experiencing an explosion in the number of states or nodes.

Our grammar allows easy integration of both metrical features and pattern repetition within the grammar. This allows for the production of complex repeated patterns at multiple levels, even with relatively few rules containing *Let* expresions.

8. Results

We generated examples using one primary rule set and two extra rules sets that offer subtle modifications. Some example results are shown in Figures 5, 6, 7, and 8. The primary rule set was derived from a combination of examples of analyzed classical music in [22] and using the authors' own judgment (such as for creating modulation rules). The rules used are a variation of those presented in [20]. The primary rule set adheres to the Schenkerian notion that the durations of produced symbols should sum to that of the parent symbol (in other words, total duration of the phrase cannot be added to or subtracted from by rules). The modified rule sets deviate from these constraints in some rules and add additional harmonic diversity. All rule sets had a collection of *Let*-based rules that could be either included or excluded from usage to observe the structural impact of rules that create repetition.

We used two chord spaces: OPC-space for voices with the ranges [40,56], [50,62], [55,70], and [60,78] respectively. Some constraints on the music were delegated to the chord spaces, such as forbidding voice-crossing by forcing all chords in the space to have their chords sorted in ascending order and, for some cases, enforcing that the lowest voice doubles either the root or fifth of the chord. Other constraints, such as selecting for smooth voice-leading and satisfying *Let* constraints, could only be applied when traversing the chord space. Our jazz space for producing seventh chords was also used, with results being mapped back into the same OPC-space to achieve more diverse harmony than with the more standard, triad-based interpretation of Roman numerals.

8.1 Structure and Harmony

Although we did not apply any strict metrics to evaluate the performance of our implementation, we were able to make several important qualitative observations. First, the inclusion of rules that create repetition have a profound impact on the quality of the results. Without repetition, even short phrases can have an "unstructured" or "wandering" sound that is atypical of human musicians. The self-similarity added by rules is noticeable both visually in the produced scores and audibly in the structure of the results. The change in quality was most noticeable over short phrases, such as those 4-8 measures long, while long phrases could still begin to sound relatively unstructured as would be the case without repetition. However, the change in observable quality over short progressions suggests that the general strategy for handling repetition may still be a powerful tool for generating longer compositions in the future while retaining a sense of larger structure and coherency in the results, perhaps by using more such rules or by giving repetition rules some degree of conditional behavior.

The harmony of the classically-inspired, strictly Schenkerian version rule set created many classical-sounding phrases. The output shown in Figure 5 and Figure 6 are very consonant with chord transitions that are acceptable for the genre. The voice-leading is much better and more chorale-like in Figure 6 than in Figure 5 due to the use of additional constraints to prevent parallel motion and enforce smoother voice-leadings (no voice moves more than 7 half-steps). In larger examples, some strange transitions occurred. Some of these can be heard in Figure 8, where the selection of chords is more diverse due to the increased number of generative steps allowed by the length. In this example, the transition between the first instance of part A and the beginning of part B is a jarring transition that is not very suitable for the target genre. Similarly, the first measure of part B sounds rather odd with an unexpected major-minor transition in the middle of the measure.

Some of the unusual harmonies exhibited by both Figure 7 and Figure 8 are probably partly due to the lack of chord quality distinction in the alphabet. For example, the scale indices for a *VII* chord in a major scale result in a diminished chord, and similarly for *II* chords in minor scales. As a result, diminished chords sometimes appear in places where human musicians might substitute another type of chord or a modulated chord.

Examples generated using our jazz space had more dissonance and often included rather strange transitions. This was because, in addition to the chord quality problem for the classical-sounding results, the handling of jazz chords purely based on mode was too simplistic to accurately reproduce the sorts of harmonies characteristic of jazz musicians. The harmonies produced in Figure 7 are rather dissonant and leave much room for improvement towards sounding like human-made jazz. Part of this may be due to the root not being present in all chords, but the model is also lacking in other subtleties that are important in jazz. Although Roman numerals may still be a useful level of abstraction, a larger alphabet of chords may be needed before attempting more complex harmonies

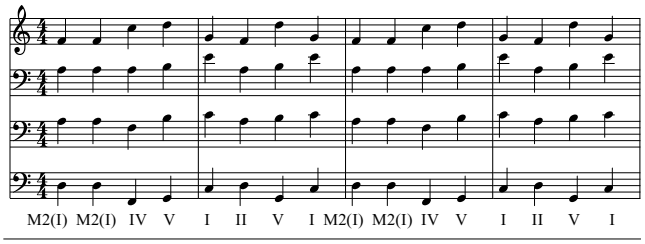

Figure 5. A chord progression generated from the expression: **let** $x = ((\textbf{let}\ x = (M_2\ I^q)\ \textbf{in}\ x\ x)\ IV^q\ V^q\ I^q\ II^q\ V^q\ I^q)\ \textbf{in}\ x\ x$. The chords were interpreted in the key of C-major using OPC-space for voices with the ranges [40,56], [50,62], [55,70], and [60,78] respectively. Note that $M_2(I)$ is the same as an unmodulated *II* chord in this case since the chords did not undergo further generation.

to achieve a more realistic jazz sound. However, although the style was not successfully reproduced, some of the harmonies produced by the model were still rich and diverse with a lot of chromaticism, which is a desirable feature in some styles of modern music.

8.2 Solution Space Traversal

One of the major challenges of working with chord spaces is the sheer size of the solution spaces, even for problems that may seem trivially small to musicians. For four voices, each spanning an octave (12 pitches), there would be 24 ways to assign four pitch classes to pitches in those voices. For a progression of four such chords, there are then $24^4 = 331,776$ solutions. The number of solutions increases exponentially with the length of a chord progression and quickly becomes impossible to enumerate in a reasonable amount of time.

Whether finding a solution in a chord space is tractable is heavily tied to the type of constraints used and whether it is possible to automatically satisfy any of them when choosing a candidate solution. The more relaxed constraints are, the easier it is to find a solution since they are more abundant. As constraints strengthen, the number of solutions becomes sparser—and solutions may not even exist at all.

One way to address the search tractability problem is to utilize different levels of abstraction with multiple chord spaces. By breaking a search problem into multiple steps, committing to partial solutions can dramatically reduce the amount of the final solution space that needs to be examined. Our tests using seventh chords, for example, utilize two separate chord spaces: a space of block seventh chords to choose pitch classes and the larger OPC-space for four-voice chords to choose pitches for each chord. This helps minimize the size of the solution space and allows some problems that would be intractable with OPC-space alone to become both tractable and relatively quick to solve. Our search strategy to automatically satisfy *Let* constraints further reduces the number of progressions that need to be examined. A depth-first search of every chord progression would have been intractable, even for most of the small examples shown in this paper.

9. Future Work

Two obvious extensions to our grammar would be an extended alphabet and an added degree of context-sensitivity in the rules allowing them to generate different productions based on the mode of the current tonic. The combination of these two changes could avoid transitions that were undesirable for the two target genres used here while allowing more expressive possibilities for other modern musical styles. The functional nature of our rules and the monadic style of overall implementation would make these types

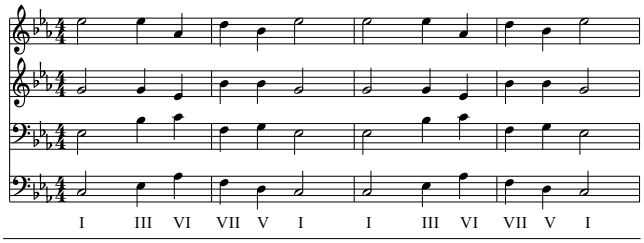

Figure 6. A chord progression generated with both constraints from *Let* expressions and the additional constraints that no voices may exhibit parallel motion (moving by the same amount in the same direction; multiple voices remaining stationary is allowed) and no voice should move more than 7 halfsteps from one chord to another. The progression was generated with the same chord space and as in Figure 5, but in the key of C-minor instead.

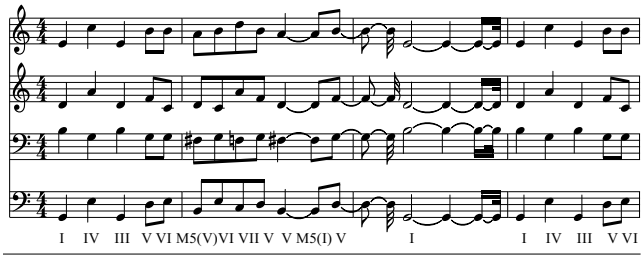

Figure 7. A chord progression generated from a harmonically modified and slightly syncopated rule set, mapping Roman numerals to seventh chords first before mapping them to first a solution in our jazz space and then to another solution in the same chord space used in figure 5 and in the key of C-major.

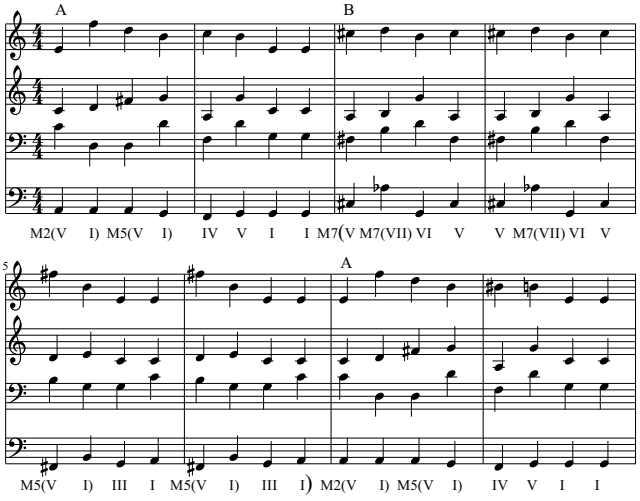

Figure 8. A longer example generated with the same chord spaces and C-major key as in 5 in C-major. It demonstrates an overall ABA format (the start of each section is labeled above the staff) and includes nested modulated sections.

of additions relatively straightforward, since additional information like the current mode could easily be threaded through the generative process much like random numbers and then supplied as an argument to the rules.

The support for repetition could also be expanded. If the alphabet were expanded to include the notion of sections in addition to chords, rules could easily be created to generate a more global composition structure before creating phrases that exhibit self-similarity. This type of addition might correct the less structured sound that larger examples (such as 16+ measures long) produced by our grammar exhibited.

The output from our grammar would benefit from additional post-processing to treat the generated chord progressions as a harmonic backbone to which melodic elaborations can be added. This process might be possible within our existing framework by using context-sensitive rules for rewriting sections. Another type of chord space, Morris's contour spaces [16], may aid in this type of transition.

Additional work is needed in the area of search algorithms relevant for traversing chord spaces. Although the *Let* constraints present one useful tool for "jumping" through a chord space to more relevant candidate solutions, other types of constraints and large chord spaces can still be incredibly time-consuming to traverse. Some musical constraints that can only be evaluated at the progression level in our current implementation may be possible to move into the search algorithm itself as we have done for the *Let* constraints.

Finally, while we have qualitatively assessed the output from our implementation, more formal empirical testing would be required to answer questions such as:

1. How well does the algorithm reproduce a particular style?
2. Does the algorithm's output sound like it was written by a human?
3. How pleasing do average listeners find the algorithm's output?

The current system is perhaps not appropriate for this kind of testing since it does not address many features found in human-made music and would therefore obviously fail to reproduce many styles of music well and would also not sound human-like. However, future extensions to our system that attempt to add additional features like melody and more complex rhythms may be better candidates for empirical testing.

Acknowledgments

This research was supported in part by NSF grant CCF-0811665.

References

[1] S. Baumann. A simplified attributed graph grammar for high-level music recognition. In *International Conference on Document Analysis and Recognition*, pages 1080–1083, 1995.

[2] M. I. Bellgard and C.-P. Tsang. Harmonizing music the Boltzmann way. *Connection Science*, 6(2):281–297, 1994.

[3] M. I. Bellgard and C.-P. Tsang. On the use of an effective Boltzmann machine for musical style recognition and harmonization. In *Proceedings of the International Computer Music Conference*, pages 461–464, 1996.

[4] S. Brown, M. J. Martinez, and L. M. Parsons. Music and language side by side in the brain: a PET study of the generation of melodies and sentences. In *European Journal of Neuroscience*, 2006.

[5] C. Callender, I. Quinn, and D. Tymoczko. Generalized voice-leading spaces. *Science Magazine*, 320(5874):346–348, 2008.

[6] B. J. Clement. Learning harmonic progression using Markov models. In *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1998.

[7] D. Cope. An expert system for computer-assisted composition. *Computer Music Journal*, 11(4):30–46, 1987.

[8] K. Ebcioğlu. An expert system for Schenkerian synthesis of chorales in the style of J.S. Bach. In *Proceedings of the International Computer Music Conference*, 1984.

[9] M. Gogins. Score generation in voice-leading and chord spaces. In *Proceedings of the International Computer Music Conference*, 2006.

[10] D. Hörnel. Chordnet: Learning and producing voice leading with neural networks and dynamic programming. *Journal of New Music Research*, 33(4):387–397, 2004.

[11] D. E. Jean-Francois Paiement and S. Bengio. A probabilistic model for chord progressions. In *Proceedings of the 6th Internationl Conference on Music Information Retrieval*, 2005.

[12] R. M. Keller and D. R. Morrison. A grammatical approach to automatic improvisation. In *Sound and Music Computing Conf.*, 2007.

[13] H. V. Koops, J. P. Magalhaes, and W. B. de Haas. A functional approach to automatic melody harmonisation. In *Proceedings of ACM Workshop on Functional Art, Music, Modeling, and Design (FARM)*. ACM Press DL, September 2013.

[14] F. Lerdahl and R. S. Jackendoff. *A Generative Theory of Tonal Music*. The MIT Press, 1996.

[15] J. P. Magalhaes and W. B. de Haas. Functional modelling of musical harmony: an experience report. In *Proceedings of the 16th ACM SIGPLAN international conference on functional programming*, ICFP '11, pages 156–162. ACM, 2011.

[16] R. D. Morris. *Composition With Pitch-Classes: A Theory of Compositional Design*. Yale University Press, 1987.

[17] S. Peyton Jones. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. URL `www.haskell.org/definition`.

[18] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990.

[19] D. Quick and P. Hudak. Computing with chord spaces. In *Proceedings of the International Computer Music Conference*, September 2012.

[20] D. Quick and P. Hudak. A temporal generative graph grammar for harmonic and metrical structure. In *Proceedings of the International Computer Music Conference*, 2013.

[21] C. Raphael and J. Stoddard. *Computer Music Journal*, 28(3):45–52, 2004.

[22] W. Renwick. *Analyzing Fugue: a Schenkerian Approach*. Pendragon Press, Stuyvesant, NY, 1995.

[23] M. Rohrmeier. Towards a generative syntax of tonal harmony. *Journal of Mathematics and Music*, 5(1):35–53, 2011.

[24] M. Rohrmeier and I. Cross. Statistical properties of tonal harmony in Bach's chorales. In *Int. Conf. on Music Perception and Cognition*, 2010.

[25] G. Roma and P. Herrera. Graph grammar representation for collaborative sample-based music creation. In *5th Audio Mostly Conference*, pages 1–8. ACM, 2010.

[26] H. Schenker. *Harmony*. University of Chicago Press, OCLC 280916, 1954.

[27] S. W. Smoliar. A computer aid for Schenkerian analysis. In *Proc. of the 1979 Annual ACM Conference*, 1979.

[28] D. Temperley. *Music and Probability*. The MIT Press, 2010.

[29] D. Tymoczko. The geometry of musical chords. *Science Magazine*, 313(5783):72–74, 2006.

[30] P. Worth and S. Stepney. Growing music: musical interpretations of l-systems. *Applications on Evolutionary Computing*, 2005.

[31] L. Yi and J. Goldsmith. Automatic generation of four-part harmony. In *UAI Applications Workshop*, 2007.