Abstract

Kulitta: a Framework for Automated Music Composition

Donya Quick

2014

Kulitta is a Haskell-based, modular framework for automated composition and machine learning. A central idea to Kulitta's approach is the notion of abstraction: the idea that something can be described at many different levels of detail. Music has many levels of abstraction, ranging from the sound we hear to a paper score and large-scale structural patterns. Music is also very multidimensional and prone to tractability problems. Kulitta works at many of levels of abstraction in stages as a way to mitigate these inherent complexity problems.

Abstract musical structure is generated by using a new category of grammars called probabilistic temporal graph grammars (PTGGs), which are a type of parameterized, context-free grammar that includes variable instantiation, a feature usually only found in grammars for programming languages. This abstract structure can be turned into full music through the use of constraint satisfaction algorithms and equivalence relations based on music theoretic concepts. An extension to an existing algorithm for learning PCFGs provides a way to learn production probabilities for these grammars using corpora of existing music. Kulitta's modules for these features are able to be combined in different ways to support multiple styles of music.

Kulitta's important contributions include (1) algorithms and a generalized Haskell implementation to support PTGGs, (2) additional formalization of existing musical equivalence relations along with a new equivalence relation for modeling jazz harmony, (3) an empirical evaluation strategy for measuring the performance of automated composition algorithms, and (4) the extension of a machine-learning algorithm for PCFGs to support a

much broader category of grammars (inclusive of PTGGs) via the use of an oracle. Kulitta's musical performance is also promising, demonstrating both stylistic versatility and aesthetically pleasing results.

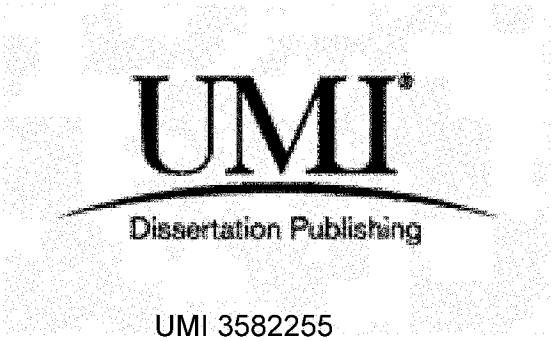
Kulitta: a Framework for Automated Music Composition

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Donya Quick

Dissertation Director: Paul Hudak

December 2014



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Contents

List of Figures

List of Tables

Acknowledgements

I am extremely grateful to my committee members, Paul Hudak, Dana Angluin, Zhong Shao, and Ian Quinn for their help and support over the years. I would also like to thank the Department of Computer Science at Yale University for creating an environment that is welcoming and that fosters interdisciplinary work. Whenever I have been in need of help, there have always been open doors to offer assistance.

I am especially indebted to my advisor, Paul Hudak, who inspired me to start doing research in the field of computer music and who encouraged me to be ambitious with my research goals. He has been the driving force behind the emergence of computer music as a research area in the department, without which the opportunity for my research would not have existed. I am also grateful for the many teaching opportunities he created for me to help further my career.

I would like to thank Dana Angluin and Ian Quinn for their extensive help with the interdisciplinary aspects of my work. Dana guided my explorations into machine learning and has been a source of moral support through much of my time at Yale. Ian has always offered encouragement while keeping my work musically sane.

I would like to thank my husband for his help in constructing the participant study that was used to evaluate Kulitta's performance. Finally, I would like to thank my family for their support during my doctoral years and for lending me their ears on so many occasions during Kulitta's development, which included many dissonant and bizarre musical bumps along the road to producing more refined sounds.

This research was supported in part by the Kempner Graduate Fellowship in the Departmet of Computer Science, the University Fellowship in the Department of Computer Science, NSF Grant CCF-0811665, and NSF Grant SHF-1302327.

Chapter 1

Computer Music as a Field

Computer music is a broad field comprised of many different research areas, and it draws on music theory, mathematics, computer sciennce, and other fields. The styles of music involved are equally diverse, ranging from classical Western music to modern modern Western and also non-Western music. Research topics range from the development of new electronic musical instruments to automation of music analysis and composition. The latter two topics include mathematical modeling of music [11, 52, 80], automated score analysis [43, 74], and construction of artificial intelligence agents to create music [19, 20, 25]. The purpose of this chapter is to provide an overview of some of these research areas and illustrate where Kulitta falls within their scope.

1.1 Composition vs. Performance

Although the definitions of what constitutes a "composition" versus a "performance" of a composition are somewhat blurry in modern music, in general there is a one-to-many relationship: a given composition is likely to have many possible performances, where the composition is an abstract entity that requires additional work or interpretation to be realized as sound.

In traditional Western music, a composition is typically represented as a printed score.

Some musical scores can be very specific, containing detailed information about pitches, timing, and volume. Others are more vague - such as a jazz standard, which only gives limited melodic information and often only abstract information about harmonies, thereby leaving many decisions to the performer. Regardless of the precise level of detail, there is usually room for some amount of further interpretation in concepts. Even a detailed traditional score would allow the performer to interpret features like rubatto (creation of an irregular tempo), the exact volume associated with pianissimo (meaning "very quiet"), and so on. Individual instruments also have additional possibilities for expressive decisions, such as varying timbre (the quality of the sound) or adding vibrato (subtle, rapid pitch fluctuations).

The computer music community often considers composition and performance as two separate tasks, just as a score can be written by one person and performed by another. Algorithms exist for creating novel musical scores [1, 22, 25], and others for performing scores [75, 84]. In fact, even in addition to the one-to-many relationship that exists between a composition and its possible performances, there are good computational reasons for separating composition and performance as independent tasks. Creating a novel, human-like or even just likable musical score is a daunting enough task by itself for a machine without having to worry about additional performance details.

The Kulitta framework addresses composition in the traditional sense: creating scores that require performance. Although Kulitta could easily be used in conjunction with an automated performance algorithm, properties like volume and tempo changes are outside the scope of musical features that Kulitta considers. All of Kulitta's output can, therefore, be easily represented using traditional Western music notation.

1.2 Automated Composition

Automated composition involves generating some amount of a musical score with a computer. Sometimes the term "algorithmic composition" is used interchangeably and also refers to music created at least partially by an algorithm rather than entirely by a human. At its largest possible scope, automated composition would be the creation of a complete, novel score from minimal human input, such as a random number seed. However, many smaller automated composition tasks also exist. For example:

- *Automated harmonization*: given an existing melody and some stylistic constraints, fill in appropriate chords.
- *Automated reharmonization*: given a melody and some harmony, find a slightly different harmony that also sounds good. This a common task done by jazz musicians to add variety to otherwise repeated phrases.
- *Fill-in-the-blank problems*: given a mostly complete piece of music, fill in missing notes while trying to adhere to the same overall style as the rest of the music.
- *Generating variations:* given a melody or short musical phrase, produce a similar but slightly different version of it.

Whether the output from algorithms for these tasks is considered *good* or *human-like* is another matter. Obviously, the larger the scope of the task, the harder it will be for a computer (or even a human for that matter) to consistently produce high-quality results according to some set of standards. However, strict standards do no always exist. Sometimes the "humanity" of the result or exact replication of a style is also irrelevant, and the purpose of the composition is to represent a mathematical model acoustically. For example, fractal-based algorithms have been used to create novel compositions using various music theoretic concepts as a guide [33, 86].

Although many algorithms and implementations in these categories are exclusive to academia, they are not absent from more widely used commercial music composition software. One of the best known examples is Band in a Box, which attempts to solve fill-in-the-blank and automated harmonization problems in different styles [31]. The Fruity Loops Digital Audio Workstation software package also features a "riff generator" to allow users to automatically generate melodies in various styles [58].

The methods discussed so far are all usually handled in offline scenarios: the computer is allowed to work for an arbitrary amount of time before returning a result. Not all styles of music are constructed this way, and some are improvisational - such as jazz. Adding a *real time* component to a musical task such as automated harmonization increases its difficulty, assuming the same level of quality is to be maintained.

A vast array of approaches have been used for tasks in automated composition, including stochastic solvers [19, 20, 87], generative grammars [42], genetic algorithms [1], and more cognitively-inspired models such as neural nets and Boltzmann machines [4, 26, 30, 35]. Each of these approaches has its merits and weaknesses, although there are some common problems relating to the complexity of musical tasks that exist throughout.

1.2.1 Computational Complexity and Music Composition

Consider an 88-key piano. Any skilled pianist in a paritcular style can sit down to such an instrument and play a series of chords that meet that style's constraints. However, consider how a naive computer algorithm might view the piano. There are 88 ways to depress one key, $88 \times 87 = 7,656$ ways to depress two keys, $88 \times 87 \times 86 = 658,416$ ways to depress three keys, and so on. The total number of combinations in which the keys can be depressed (including not depresseing any of them) is the cardinality of the power set of $\{1, ..., 88\}$, or the number of binary numbers representable with 88 bits:

$$2^{88} = 309,485,009,821,345,068,724,781,056 \tag{1.1}$$

Of course, any human musician can tell that this number is actually absurd, since nobody can reasonably play the vast majority of those combinations of pitches. Sill, even given a classifier to determine which sets of pitches are reasonable and which are not, such a naive algorithm would still need to compute each one in order to determine if it is viable.

These types of exponential patterns are everywhere in music. The problem above is a vertical one in a musical score: choosing what to write on the staff at a particular beat or point in time. However, the same problem also exists horizontally when considering changes in those pitches over time. If there are n possible chords to pick from for each of m melody notes, then there are n^m possible chord progressions to explore. Even when n can be whittled down to a reasonably small number of candidates, creating a progression of those chords under styistic constraints can still become intractable. Given these problems, efficient representations for musical structures and methods for minimizing unnecessary computation are incredibly important in automated composition algorithms. Kulitta employs an important principal in order to tackle these sorts of problems in a tractable way: musical abstraction. This helps to break daunting tasks with large solution spaces into smaller problems, allowing a solution to emerge in progressively finer levels of detail, much like a sculpture being chiseled out of stone.

1.2.2 Assessing Compositional Quality

The subjects of composition and performance are often conflated when people listen to music and make a judgement about its quality. If someone hears a piece of music and says it is "bad," is that because he/she didn't like the score, the way the performers interpreted it, or some combination of the two? Even if there exists some performance of a composition that would be deemed "good," it is still very easy to make that same composition sound bad to many people: simply have it performed by an orchestra of out-of-sync, novice theremin players [1]. This makes assessment of a *score* rather tricky, since we don't hear the score—we

1. The theremin is a notoriously difficult-to-play electronic instrument where the performer's hands control pitch and volume via their proximity to metal rods. Even minor unsteadiness of the hand and small shifts in

hear its performance.

Automated composition is also a strangely volatile subject, particularly amongst musicians. As the author has directly experienced, it is quite common for musicians to actually be offended—sometimes dramatically so—by the existence of automated composition research, while others embrace it readily. Similar phenomena are described by David Cope [21]. In contrast, research on natural language processing that attempts to let machines communicate with us using grammatically correct sentences does not appear to elicit such a sharply divided and emotional reaction. Voice-communication with machines and machines that talk to us are increasingly prevalent and accepted features in modern society, and yet a machine that essentially sings is controversial. The strong attitudes that exist about automated composition research add further difficulty to objectively assessing the performance of algorithms that produce music.

Currently, there is no standard set of metrics or methods by which to assess the performance of an automated composition system. Also, what constitutes "good" music varies across the human population. Many aspects of "goodness" are also style-specific. For some styles of music, such as chorales in the style of J.S. Bach, various music theoretic analyses can be used to determine the acceptability of a composition for its style. However, for other styles, particularly new ones, there are fewer or no such formal approaches beyond simply observing how other people respond to the music. Additionally, people without musical training would also be unable to analyze a score visually in the way that a music theorist could analyze a Bach chorale, therefore requiring a performance of that score for any sort of assessment—bringing the problem of composition quality versus performance quality into the mix. Chapter 9 addresses these issues in more detail and presents one possible way of assessing an automated composition system's performance empirically using human subjects testing.

---

the performer's posture while playing a note can have noticable impact on the generated pitch.

1.2.3 Systems for Automated Composition

Two notable automated composition systems exist with goals similar to Kulitta's: a chorale-harmonization system created by Kemal Ebcioglu and David Cope's learning-based Experiments in Musical Intelligence. These two systems are both capable of producing complete compositions of high compositional quality by music theoretic standards.

Kemal Ebcioglu created a system for harmonizing chorales in the style of J.S. Bach [25]. The system uses a domain-specific programming language called Backtracking Specification Language (BSL) and attempts to harmonize a melody by operating on the solution from many different musical representations, or "viewpoints." Some viewpoints include the harmonic backbone of the chorale as a series of chords, the melodic detail of the chorale, and the Schenkerian analysis [2] of the chorale. Constraints at each of these levels must be satisfied in order to find a suitable solution, with backtracking being a fundamental part of the overall generate-and-test search process. The system is capable of producing harmonizations on par with those produced by skilled human composers.

David Cope's Experiments in Musical Intelligence (EMI) is another system capable of generating chorales in the style of J.S. Bach [19, 20, 22], although with a significantly different overall approach. EMI is a machine-learning based system for automated composition that attempts to emulate styles by analyzing a corpus of music. EMI's general strategy for style emulation is to attempt to do mostly what has already appeared in the training data—but to reject solutions that are too similar to the training data. Existing patterns are recombined at various levels to produce a new, but not too new, result. In this way, by generating primarily features that have already been observed, many of the otherwise tricky aspects of style emulation are avoided. Rather than backtracking, if EMI does not find a solution, it starts over from the beginning using a slightly different set of generative parameters. EMI is also generalizable to other types of data, such as spoken language.

2. Schenkerian analysis is a method of analyzing a score to derive its abstract harmonic structure.

Automated composition systems suffer from a tradeoff between novelty or scope and quality. Systems that produce very novel or "creative" results often produce a lot of garbage, while those that consistently produce high-quality results tend to produce many things that sound the same. Ebcioglu's system sacrifices novelty for high-quality output. Cope's system also makes this same sacrifice, although perhaps to a lesser extent. The advantages of Cope's approach in EMI is that it is able to very convincingly reproduce a given style when the corpus is large, as is the case for Bach chorales. Because it closely emulates its input data, it also will retain fairly high quality. However, novelty will suffer as the training corpus shrinks.

1.3 Computer Music's Interdisciplinary Nature

Research in computer music is highly interdisciplinary, drawing from areas like artificial intelligence and machine learning, linguistics, and psychology. A few field intersections relevant to Kulitta are highlighted here.

1.3.1 Music, Artificial Intelligence, and Machine Learning

Algorithms for atuomated composition can also often be viewed as artificial intelligence agents. While the term "artificial intelligence" (AI) more commonly conjures up images of interactive game opponents such as Deep Blue [12] or IBM's Watson [27], music composition has many sub-tasks that share features in common with more classical AI problems, namely constraint-satisfaction over large domains and emulation of human behaviors or decision-making.

A machine learning algorithm is one that attempts to derive a concept from a collection of data. The concept may or may not have generative usage. For example, an algorithm for classifying music by genre may need to learn what properities each genre has from training examples, but does not necessarily need to be able to generate new compositions in those

styles. However, some systems can do both tasks [5].

Many, although not all, artificial intelligence algorithms also include forms of machine learning. While it is possible to build artificial intelligence agents for simple situations without a learning component, such as a board game opponent that bases its decisions purely on traversal of a pre-defined tree of possibilities, learning is appealing in more complex scenarios. Adding a learning component to an AI algorithm allows it to tailor its behavior to a specific situation more succintly than trying to account for each situation by hand. Learning is commonly employed in musical algorithms when attempting to emulate styles or create compositions that sound humanly plausible.

One of the most commonly applied learning algorithms in computer music is the Markov chain [41, 13, 87]. There are two reasons for this: the algorithm is simple, and music is sequential in nature, lending itself to modeling a score as a series of state transitions [2]. Figure 1.1 shows a simple example of this type of representation. It is relatively straightforward to take a corpus of music and derive some sort of Markov model from it. Unfortunately, when used generatively, such models tend to result in random-sounding or distinctly non-human-sounding music. These problems are further described in Chapter 4.

1.3.2 Natural Language and Music

Evidence from recent studies suggests that spoken language and music are related in the brain [7]. In fact, the structure of music may be best described by grammars, just as is the case for spoken language, and there has been substantial work on this idea in music theory [48, 67, 85]. Grammars are, therefore, an appealing category of mathematical models to explore for the purpose of both analyzing and generating music. However, exactly which category of grammars would be best for describing music is very much an open question.

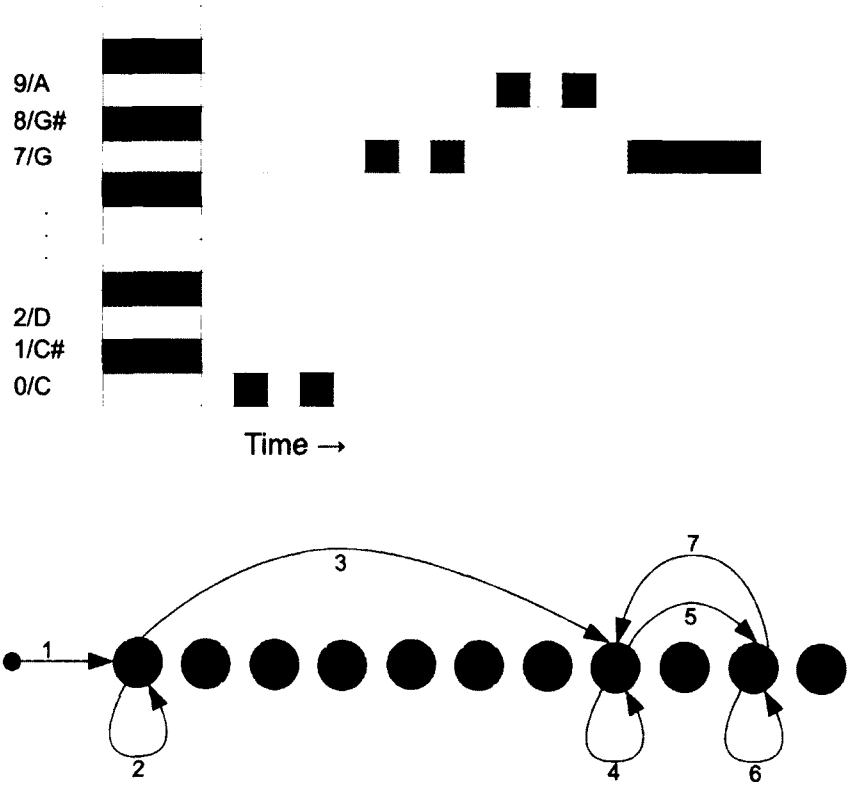


Figure 1.1: Music is commonly represented using state spaces [2]. The illustrations above show the opening refrain of "Twinkle, Twinkle Little Star" represented as a piano roll and as a finite state machine over musical states. The top representation is a graph of depressed keys on a piano over time (a gray box is a key depressed for some amount of time) and the bottom representation shows the path for the same melody through musical states, where each represents a key on a piano.

1.3.3 Programming Languages

A number of domain-specific languages exist for both representing music as well as composing music [36, 38, 44, 53, 60]. A fundamental problem in any musically-oriented computer program is how to actually represent various musical concepts, both mathematically and inside a computer. What is the appropriate way to represent a pitch? Should it be an integer and discrete like the keys on a piano, or a continuous value like the range possible on a violin? Should a chord (a collection of simultaneous pitches held for some time) be a set, multiset, or vector? What about several notes played in sequence or in parallel, or more abstract structures like the notion of a developmental "part A" and its variations? Can certain musical structures be polymorphic for better reusability? These are questions that enter the realm of programming languages. Kulitta's methods of representing musical features, which are described in Chapters 3, 4, and 5, even include some programming-language-specific features, such as variable instantiation to indicate repeted phrases.

Euterpea is a library for music representation and manipulation in Haskell [36]. The Kulitta framework is implemented in Haskell and uses the Euterpea library for some of its levels of musical representation. However, Kulitta also contains its own embedded category of grammars for representing harmonic and metrical structure, called Probabilistic Temporal Graph Grammars (PTGGs). A PTGG can contain statements representing variable instantiation, similar to the let-in constructs found in programming languages. Sentences written using this category of grammar must then be interpreted to create music, much as a program must be executed to know its result. PTGGs are described in chapter 4.

Chapter 2

An Overview of Kulitta

Kulitta is a modular framework for automated composition in a variety of styles. The name,"Kulitta," comes from a musician in Hittite mythology [81]. A central idea to Kulitta's approach is the notion of abstraction: the idea that something can be described at many different levels of detail. Music has many levels of abstraction, ranging from the sound we hear to a paper score and large-scale structural patterns. Music is also very multidimensional and prone to tractability problems. Kulitta uses this principal of abstraction to mitigate these computational problems and flesh out a composition in stages. Kulitta is also able to learn some musical features from a corpus of analyzed music.

2.1 Introduction

A summary of Kulitta's overall structure can be seen in Figure 2.1. There are three general components to the system: a learning step, a structural generation step, and a musical interpretation step. Structural generation begins with a musical grammar for abstract chord progressions called a *probabilistic temporal graph grammar* (PTGG). Production probabilities for aspects of this grammar can either be defined by hand (no learning step) or inferred from existing musical phrases using machine learning techniques. This PTGG and its associated production probabilities are passed to a generative algorithm. This process generates

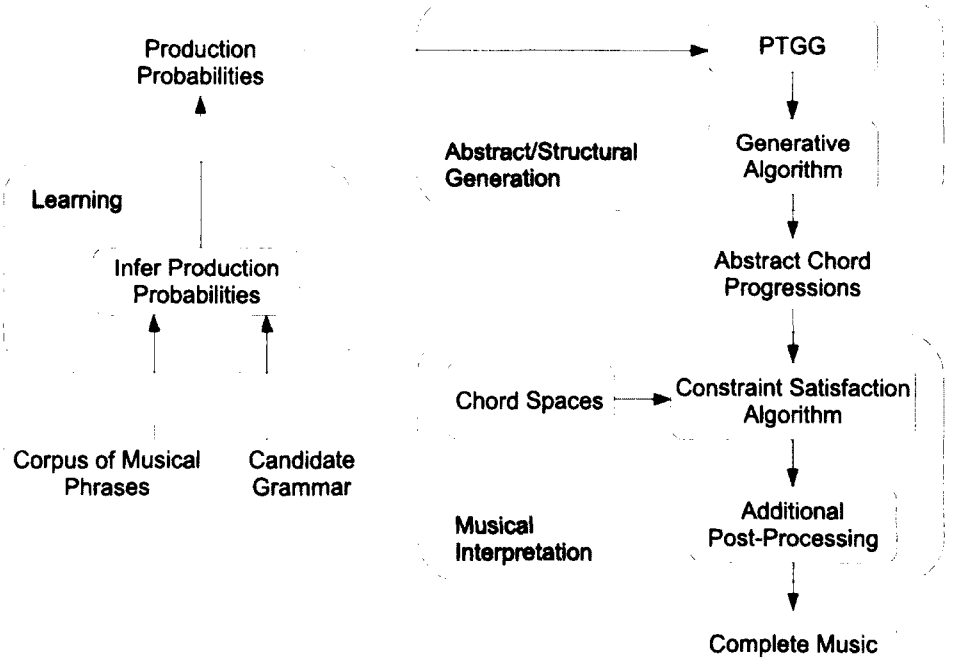


Figure 2.1: An illustration of the overall structure of Kulitta. The first stage of our system creates abstract chord progressions. A generative grammar called a *probabilistic temporal graph grammar* (PTGG) is used in combination with an algorithm for applying the grammar to produce abstract chord progressions. Production probabilities for aspects of this grammar can be inferred from examples of existing musical phrases. 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.

abstract musical structure. At this stage, the chord progressions produced are not tied to any particular style of music.

The next phase of our generative system *interprets* those abstract progressions. As part of the musical interpretation process, Kulitta uses a mathematical construct called *chord spaces* to turn an abstract chord progression into one that could be represented as a score. At this stage, the chord progressions will be homophonic (all voices being rhythmically identical). However, generation does not need to stop there. Various style-specific melodic and rhythmic elements can still be added. Two main styles are currently supported by Kulitta: classical chorales and simple jazz.

2.2 Musical Abstraction

Kulitta revolves around the principal of abstraction: the notion that a musical passage can be represented at different levels of detail and that two distinct musical passages may differ in the details while being the same in more fundamental ways. Kulitta's notion of "abstraction" is very similar to the definition of the term used in programming languages. The more abstract something this, the more information must be filled in before that thing can be used. An abstract function is a type signature lacking a function body: we know something about the function's interface, but we don't know exactly how it will behave, and many different implementations are possible. Similarly, a series of chord symbols from a jazz standard contain information about musical flavor, but we don't know exactly what interpretation a performer will take—and there are many such interpretations.

Music contains many levels of abstraction. Although one rarely thinks of it while listening to a piece of music, ideas like *melody* and *harmony* are abstract concepts, as are specific patterns within those broader features. Musical scores are abstract representations as well, with one score having multiple possible performance interpretations. This section addresses several areas of music that have multiple possible levels of abstraction.

2.2.1 Pitches

A musical *pitch* is a sound that has a particular *fundamental freqeuncy*, which is the lowest frequency in a series of harmonics. Some instruments produce many harmonics, which is part of what causes the *timbre* of one instrument to be different from another. Two pitches on instruments sound "the same" when the fundamental frequency is the same, even if the series of additional harmonics produced is different (creating a different timbre or texture).

In the modern Western tuning system, pitches are represented as tuples of a *pitch class* (*C*, *C#*, *D*, etc.) and *octave* (an integer). Pitches on a musical score can typically be represented using integers, where each number corresponds to a key on an infinitely long piano keyboard. There are also different ways of mapping numbers to piano keys, depending on where "octave 0" is placed. Euterpea uses the convention that $(C,0)$ is 0, $(C\#,0)$ is 1, $(C,5)$ is 60 (middle C on a piano), and so on for all pitch classes and octaves. Negative octaves produce negative pitch numbers, such as $(B,-1) = -1$. Enharmonically equivalent pitches [1] are mapped to the same integer. In other tuning systems, fractional pitch numbers may be allowed. For example, a pitch of 60.5 would be partway between $(C,5)$ and $(C\#,5)$. However, Kulitta does not support microtones, so pitches and pitch numbers will be treated as integers from this point on.

Given this numbering system for pitch classes, the relationship between a pitch number, p, where $(C,0) = 0$, and its fundamental frequency, f (in Hz), is calculated by Equation 2.1. Note that the offset of 69 added in Equation 2.1 is specific to Euterpea's placement of octave 0; other pitch numbering systems require adding a different offsets.

$$p = 69 + \lceil 12 \times log_2(f/440Hz) \rceil \tag{2.1}$$

Pitch classes are essentially abstract pitches. While one can play a (C,4) concretely on

1. Pitches can be written in more than one way on a musical score. Pitches are enharmonically equivalent when they indicate the same key on a piano. For example, within the same octave, E# and F would be enharmonically equivalent.

an instrument, there is no such option for just "C" by itself - one needs more information to be concrete, such as the octave in which the pitch class is to be played. Pitch classes can be indexed using $[0,11]$, where $C = 0$, $C\# = 1$, and so on up to $B = 11$. For a pitch class, pc, and octave, o, the pitch number, p, is calculated by the formula in Equation 2.2.

$$p = 12 \times o + pc \tag{2.2}$$

2.2.2 Chords

The term "chord" has ambiguous musical meaning. The term can be used to refer to a specific collection of simultaneously-sounding notes on a musical score. In this case, the concept of a chord involves both duration and the notion of voices within the chord. Such a chord may be best mathematically represented as a vector of pitches and a duration if the start and end times are uniform.

Chords are also notated more abstractly in music using Roman numerals, where one numeral represents many possible score-level interpretations. In this setting, a chord is often durationless and carries only some information about the pitch content of the music. In the key of C, a Roman numeral I would indicate a C-major chord, perhaps with the pitch classes C, E, and G. However, it tells the reader nothing about the octaves associated with these pitch classes or even the number of voices involved.

Even more abstract is the idea of "chord quality." Chord quality refers to concepts like "major chord" and "minor chord." A chord's quality gives some information about the intervallic structure of its pitch classes. The term "major chord" usually implies the structure of a major triad: picking scale indices 1, 3, and 5 from a major scale (indexed from 1). Again, such a chord is durationless and the pitch classes can occur in any octave.

2.2.3 Chord Progressions

A *chord progression* is a sequence of chords in time. The chords may be concrete, with specific pitches and durations, or abstract, lacking information about duration and/or specific pitches. Progressions can be described in even more abstract terms. For example, a *cadence* is a chord progression that ends a musical phrase. There are several types of cadences, each described using abstract chords—usually Roman numerals—to indicate harmonic structure. Two examples are authentic cadences (V-I) and plagal cadences (IV-I).

Jazz often employs *chord substitutions*, the idea that one chord may be substituted for another in a progression. Chord substitutions add variety to repeated phrases without breaking harmonic continuity. A progression that is described using possible chord substitutions exists at a higher level of abstraction than one described only in terms of a single string of Roman numerals.

2.2.4 Melodies

Melodies are sequential patterns of notes, although the distinction between which patterns are considered tuneful or melodic and which are not is poorly defined. A number of different approaches have been proposed for melodic analysis and modeling [18, 88]. In Schenkerian theory, melodies contain a mixture of harmonic tones and other notes, many (but not necessarily all) of which are analyzed away to determine the structure of the music [73, 74]. This suggests that there are also multiple levels of abstraction present within a single melody.

Melodies can also be thought of as belonging to categories. In classical Western music, a *theme and variations* is a peice that consists of an opening melodic motif that is repeated with small alterations throughout the music. These variations of the original melody all sound similar, and, in an algorithmic composition setting, one might view many of them as equally reasonable candidates when trying to create a new melody from scratch while adhering to various other musical constraints, such as the underlying harmonic structure of

the melody.

2.2.5 Developmental Structure

Repetition of patterns and variations on a repeating pattern are fundamental to musical structure. Repetition and variation create a heirarchical structure in long sections of music, and an absence of this structure is likely to result in complaints of the music sounding "directionless" or "wandering."

Patterns of repetition in music are often described using strings of letters. For example, *ABA* form would imply that there is an *A* section and a *B* section, and that both instances of *A* are the same — or at least sufficiently similar as to be recognizable as instances of the same musical idea. Sometimes a "prime" notation is used to indicate slight variation. The pattern $AA'BA$ would indicate that the first two instances of *A* are similar, but the one denoted A' is slightly different in some way. Exactly what constitutes a variation versus a completely new section in a mathematically formal way is an open question.

2.3 Mathematical Models

Kulitta models music using two primary mathematical models: equivalence relations and grammars. Grammars are used to generate abstract structure in the music and equivalence relations are used to move between levels of abstraction.

2.3.1 Equivalence Relations and Chord Spaces

How should musical abstraction be mathematically represented? For a number of the abstract musical features discussed above, one approach is to use *equivalence relations* to partition a set of concrete examples into categories representing the desired level of abstraction.

Relations are mathematically represented as sets of pairs. For some relation R, $(a,b) \in R$ means that a is related to b. An equivalence relation is a relation that is reflexive, symmetric, and transitive. These properties are defined below, where unidirectional and bidirectional arrows represent implication and bi-implication respectively.

- Reflexivity: $(a,a) \in R$.
- Symmetry[2]: $(a,b) \in R \leftrightarrow (b,a) \in R$.
- Transitivity: $(a,b) \in R \wedge (b,c) \in R \rightarrow (a,c) \in R$.

Kulitta uses equivalence relations to move between different levels of abstraction in music, such as to move from Roman numerals to vectors of pitches. Kulitta's implementation supports equivalence relations in a generalized way, making the system more modular and more easily extensible to include additional equivalence relations for new musical features.

The musical equivalence relations used in Kulitta are also called *chord spaces*. Some chord spaces are derived directly from music theory. We make use of both the classical chord spaces presented by Tymoczko et al. [80] and Callender et al. [11] as well as proposing a new space to capture elements of jazz harmony. These are further described in chapter 3.

2.3.2 Musical Grammars

Grammars have been explored both generatively and analytically in music [33, 42, 86]. Studies on brain activity have shown a strong link between language and music in the brain [7], an idea that has become increasingly accepted in music theory through works like GTTM, which presents a grammatical outlook on analyzing music [48] (although it requires additional formalization to be implemented in both analytical and generative settings).

2. The property of symmetry in relations is sometimes referred to as symmetricity.

Kulitta uses a category of musical grammars called *Probabilistic Temporal Graph Grammars* (PTGGs). These grammars incorporate both traditional features like those from probabilistic context free grammars (PCFGs) as well as features more common in programming languages, such as "let" expressions to allow variable creation and instantiation. The latter are used to support higher-level musical structures such as *ABA* form where each *A* must be identical, as well as to capture the more subtle $AA'BA$, where each *A* is expected to to be identical but A' is expected to be slightly different. PTGGs are described in chapter 4.

2.3.3 Machine Learning

Although the generative part of Kulitta can be run using hand-built grammars and other musical models, these models can also be *learned* from a data. Kulitta's support for learning makes it more adaptable to handling different styles of music than it would be if these models had to be hand-built each time. Given a corpus of music, Kulitta is able to infer certain properties that can then be emulated in the generative steps. Kulitta's learning process is described in chapter 7.

2.4 Implementation

Kulitta is implemented in the Haskell programming language. Many of the system's features lend themselves to a functional approach, leading to an elegant Haskell implementation[3]. Kulitta also attempts to avoid being tied to a particular musical style by using strategies that are general and highly modular. Haskell's type system lends itself to this, allowing functions to be defined in the most abstract way possible through the use of type variables. Kulitta's modularity also allows for different models to be combined in multiple ways, creating a diverse range of results.

3. Kulitta's complete source code, MIDI files of the examples in subsequent chapters, and recordings of additional compositions created by Kulitta are online at `http://www.donyaquick.com`.

Kulitta's 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 Kulitta's output data structures 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. Examples shown here were produced using MuseScore [6], an open source music notation system.

Chapter 3

Musical Equivalence Relations

Kulitta uses a construct called a *chord space* to capture different levels of musical abstraction. This allows musical problems to be solved iteratively with smaller, more easily searchable solution spaces at each step [62]. Chord spaces are formed using *equivalence relations*. This chapter presents a general implementation of equivalence relations in Haskell that supports many different chord spaces. The following notations and definitions are used throughout the chapter:

- Function composition: $(f_2 \cdot f_1)x = f_2(f_1(x))$.
- Function equality: $f_1 = f_2$. This means that f_1 and f_2 will have the same input/output mapping even if their definitions and/or complexities are different.
- Vectors: $\vec{x} = \langle x_1, ..., x_n \rangle$.
- Vectors created from a constant: $k^n = \langle k, ..., k \rangle$.
- Addition of two vectors: $\vec{x} + \vec{y} = \langle x_1 + y_1, ..., x_n + y_n \rangle$.
- Adding a constant to a vector: $\vec{x} + k = \langle x_1 + k, ..., x_n + k \rangle$.

3.1 Equivalence Relations

A relation is mathematically represented as a set of pairs. For some relation $R \subseteq A \times B$, the notation $(a,b) \in R$ means that $a \in A$ is related to $b \in B$. An *equivalence relation*, $R \subseteq S \times S$, is reflexive, symmetric, and transitive. These three properties are formalized below, where unidirectional and bidirectional arrows indicate logical implication and bi-implication respectively.

- Reflexivity: $\forall a \in S,\ (a,a) \in R$.
- Symmetry: $\forall a,b \in S,\ (a,b) \in R \longleftrightarrow (b,a) \in R$.
- Transitivity: $\forall a,b \in S,\ (a,b) \in R \wedge (b,c) \in R \longrightarrow (a,c) \in R$.

Relations can also be thought of as digraphs, where a directed edge exists from a to b if and only if $(a,b) \in R$. Because of symmetry, equivalence relations are often represented as undirected graph, where reflexivity is assumed and where an edge connecting a and b implies the existence of both $(a,b) \in R$ and $(b,a) \in R$. An undirected graph of an equivalence relation will be a collection of cliques, where each clique represents an *equivalence class*. The equivalence class of an element is the clique to which it belongs. Given a relation, R, and element, a, this is formalized as:

$$eqClass(a,R) = \{b \mid (a,b) \in R\} \tag{3.1}$$

The notation $a \sim_R b$ means that a is related to b under equivalence relation R, or that a and b are R-equivalent. This means that $(a,b) \in R$. If R is an equivalence relation, then it will also be the case that $b \sim_R a$, such that the notation is symmetric.

Composition of functions is defined as $(g \cdot f)\ x = g(f(x))$, and composition of two relations follows a similar convention.

$$R_2 \cdot R_1 = \{(a,c) | (a,b) \in R_1,\ (b,c) \in R_2\} \tag{3.2}$$

However, composing two equivalence relations does not necessarily produce a new equivalence relation. Two equivalence relations, R_1 and R_2, can be combined to make a new equivalence relation using the *join* operation, $R_1 \vee R_2$ [51]. We will use the notation R^+ to denote the *transitive closure* of relation R, which involves adding pairs (or edges in the digraph) possible until R is transitive.

$$R^+ = R \cup (R \cdot R) \cup (R \cdot R \cdot R)... \tag{3.3}$$

$$R_1 \vee R_2 = (R_1 \cdot R_2 \cup R_2 \cdot R_1)^+ \tag{3.4}$$

The join operation is commutative, such that $R_1 \vee R_2 = R_2 \vee R_1$. For simplicity, we will abbreviate $R_1 \vee R_2$ as simply $R_1 R_2$. As will be shown later with some musical equivalence relations, although combining equivalence relations is simple in concept, it is not always straightforward in practice to preserve properties like transitivity when combining two or more equivalence relations.

3.1.1 Quotient Spaces

A quotient space is the result of applying an equivalence relation to a set, thereby forming a partition of the set's elements or "gluing" related elements together to form a set of sets. For a set S and relation R, the quotient space formed by applying R to S is denoted S/R and sometimes referred to as R-space.

For example, consider the equivalence relation formed by the integers modulo 2:

$$a \sim_{mod\,2} b \longleftrightarrow a \bmod 2 = b \bmod 2, \;\; a, b \in \mathbb{Z} \tag{3.5}$$

The quotient space formed by $\mathbb{Z}/mod\ 2$ partitions the integers into even and odd equivalence classes, which can be represented by the points 0 and 1 respectively. All even numbers are "glued" to 0, and all odd numbers are "glued" to 1. This particular quotient

space is usually denoted $\mathbb{Z}_2$. Quotient spaces formed by taking the integers modulo other values are similarly denoted $\mathbb{Z}_x$ for some $x \in Z$. [1]

3.1.2 Groups

A group is a pair consisting of a set, S, and an operator, $*$, with the following properties[24]:

- Closure: $\forall a, b \in S, a * b \in S$
- Associativity: $\forall a, b, c \in S, a * (b * c) = (a * b) * c$
- Identity element: $\exists e \in S | \forall a \in S, a * e = e * a = a$
- Inverse element: $\forall a \in S, \exists a^{-1} \in S | a * a^{-1} = a^{-1} * a = e$

Abelian groups are also commutative: $\forall a, b \in S, a * b = b * a$.

The *symmetric group* of order n is the set of all permutations of n elements. It is denoted S_n. The symmetric group is a collection of permutations on a list of length n, and there are $n!$ such permutations. $S_n = \{\sigma_1, ..., \sigma_{n!}\}$ is a group with function composition as the operator [24].

As will be shown later in this chapter, several operators that define equivalence classes on chords also form groups where the elements are functions, much like is the case for the symmetric group.

3.1.3 Normalizations

An equivalence class is a set of elements that are all related to one another, forming a clique when represented as a graph. Points a and b are related under R if $(a, b) \in R$. However, if R is large (possibly infinite), simply searching for $(a, b) \in R$ can be problematic as a means to determine whether $a \sim_R b$ holds. This process of checking whether $a \sim_R b$ holds, or

1. The integers modulo n is also sometimes denoted $\mathbf{Z}/n\mathbf{Z}$ [24]

whether $(a,b) \in R$, is called testing for equivalence under R or testing for class membership (since $a \sim_R b$ implies that a and b belong to the same equivalence class).

Normalizations are one way to address this problem: for a set S, relation R, and quotient space S/R, rather than enumerate the entire equivalence class of an element $s \in S$ when determining class membership of a new element, one can instead compute a *representative point* of that equivalence class. The set of all representative points is referred to as the *representative subset* of S/R, denoted by S_R. If a function $f : S \to S_R$ has the property that every point in S is R-equivalent to exactly one point in S_R, it is called a *normalization*. More formally:

Definition 1. *$S_R \subseteq S$ is a representative subset for S/R iff $\forall x \in S$, there is exactly one $y \in S_R$ such that $x \sim_R y$.*

Definition 2. *f is a normalization for the quotient space S/R whenever $\forall x,y \in S$, $f(x) = f(y) \longleftrightarrow x \sim_R y \wedge f(x) \sim_R x$*

Theorem 1. *A function $f : S \to S' \subseteq S$ is a normalization for some equivalence relation, R, if $\forall y \in S', f(y) = y$.*

Proof. Since f is a function, it will map every element of S to exactly one element in $S' \subseteq S$, forming a partition of S. If we group elements using the criteria that $a \sim b$ iff $f(a) = f(b)$, then the f can be used to partition S into a set of cliques or equivalence relations. Therefore, f is a normalization for some equivalence relation. □

Corollary 1. *R is an equivalence relation and $f : S \to S' \subset S$ is a normalization for R when $a \sim_R b \longleftrightarrow f(a) = f(b)$.*

Equivalence relations can have more than one normalization, and different normalizations may be needed under different circumstances. Normalizations can also sometimes be composed to produce new normalizations. The conditions under which this can happen are described below.

Definition 3. *Let f_1 and f_2 be normalizations for equivalence relations R_1 and R_2 respectively on set S and $f_3 = f_2 \cdot f_1$ with range S_3. The function f_3 is a normalization for $R_3 = R_1 \vee R_2$ iff $\nexists x, y \in S_3, x \sim_{R_3} y$.*

The concept of a *fundamental domain* of a quotient space is similar to the definition we have presented for representative subsets, and fundamental domains exist for a number of musical equivalence relations [11, 80]. However, although the exact definition of a fundamental domain can be slightly different from one source to another, the fundamental domain of a quotient space usually preserves some aspects of the quotient space's geometry. These sorts of additional constraints are not required to have a representative subset, although every fundamental domain will also be a representative subset.

3.1.4 Path-Finding with Equivalence Relations

Just as one element is equivalent to many others under an equivalence relation, a sequence of many elements can be related to many other such sequences. The sequence of elements can be viewed as a path through equivalence classes.

For example, the first step of traditional harmonic analysis would be the process of turning collections of pitches, or chords, into a series of Roman numeral labels, where each Roman numeral represents a particular equivalence class of chords in the context of a key and mode. The same set of Roman numeral labels can correspond to many unique compositions.

An important feature of this approach of path-finding through equivalence relations is that it can dramatically reduce the size of the solution spaces explored for a particular problem. Consider an infinite set of elements S and an equivalence relation, R that produces quotient space S/R with a *finite* number of equivalence classes. The integers modulo n, $\mathbb{Z}_n$ are an example of this sort of relationship. For example, the representative subset of $\mathbb{Z}_{12}$ is finite, with exactly 12 members (the numbers 0 through 11), while $\mathbb{Z}$ is infinite. It can be more efficient to partially solve a problem by first traversing a representative subset of a

quotient space rather than diving into the set of elements directly.

3.1.5 Musical Spaces

A *chord space* is a way to organize chords in musically meaningful ways. They provide convenient, intermediate levels of organization between various abstract and concrete chords. Mathematically, a chord space is a type of quotient space formed by applying an equivalence relation to a set of chords. One such chord space groups chords based on pitch class content, providing a useful level of abstraction for voice-leading assignment, but there are also many other possible chord spaces that relate chords in different ways.

One way to construct musically-meaningful equivalence relations is to exploit existing concepts in music theory, such as the ideas of pitch class and transposition. Tymoczko and Callender et al. introduce several such relations on chords, each based on some concept in music theory [11, 80]. Other musical quotient spaces are also possible. There is no reason that the concept must be constrained to grouping individual chords. It would also be possible to have a *progression space* by grouping chord progressions or even a *melody space* by grouping melodies. Regardless of the musical concept used, the same mathematical principles of quotient spaces apply. Algorithms designed to operate on quotient spaces generally will also support any such musical space. Here we consider two broad categories of spaces, the OPTIC spaces [11, 80] and contour spaces [56], along with a new category inspired by jazz music theory called *mode space*.

3.2 Equivalence Relations in Haskell

Given a quotient space, S/R, there are two types of questions that will commonly be asked in the Kulitta framework when working with musical equivalence relations:

1. For some $x, y \in S$, is $x \sim_R y$?

2. For some $x \in S$, what is x's R-equivalence class, $eqClass(x, R)$?

We implement equivalence relations by creating a function to answer the first question.

```
type EqRel a = a → a → Bool
```

This is easy to do for equivalence relations where normalizations exist.

```
type Norm a = a → a

normToEqRel :: (Eq a) ⇒ Norm a → EqRel a

normToEqRel f x y = f x == f y
```

We implement sets as lists. A quotient space is then a list of lists, or $[[a]]$. The "slash" operator in the notation S/R and equivalence class lookup can be defined as follows.

```
(//) :: (Eq a) ⇒ [a] → EqRel a → QSpace a

[] // r = []

s // r =
  let e = [y | y ← s, r y (head s)]
  in e : [z | z ← s, ¬ (elem z e)] // r

eqClass :: (Eq a, Show a) ⇒ QSpace a → EqRel a → a → EqClass a

eqClass qs r x =
  let ind = findIndex (λe → r x (head e)) qs
  in maybe (error ("No class for " ++ show x)) (qs!!) ind
```

3.3 The OPTIC Relations

Callender et al. introduce five equivalence relations on chords [11]. Chords in these relations are represented as vectors of pitch numbers. The relations, therefore, partition $\mathbb{Z}^n$ (the set of all integer vectors of length n). Vectors are written as $\vec{x}$ or as $\langle x_1, ..., x_n \rangle$ to show the elements individually. The notation 1^n refers to a vector of lenth n whose elements are all 1, and the notation $\mathbb{Z}^n$ refers to the set of all integer vectors of length n.

- *Octave equivalence*, O. Chords belong to the same equivalence class if they have the same vectors of pitch classes: $\vec{v} \sim_O \vec{v} + 12\vec{i}$, $\vec{i} \in \mathbb{Z}^n$ [11]. For example, $\langle 0,4,7 \rangle$ and

$\langle 12,4,7\rangle$ are O-equivalent; they are both C-major triads where the voices have the pitch classes C, E, and G respectively. [2]

- *Permutation equivalence*, P. Chords with the same multisets of pitches belong to the same equivalence class under this relation. P can be defined using the symmetric group of order n, S_n (the set of all permutation functions for n elements): $\vec{v} \sim_P \sigma(\vec{v})$, $\sigma \in S_n$ [11]. For example, $\langle 0,4,7\rangle$ and $\langle 4,0,7\rangle$ are P-equivalent.

- *Transposition equivalence*, T. Chords with the same intervallic content belong to the same equivalence class. For example, $\langle 0,4,7\rangle$ and $\langle 1,5,8\rangle$ are T-equivalent. The relation was originally defined as $\vec{v} \sim_T \vec{v} + c1^n$, $c \in \mathbb{R}$ for continuous, microtonal systems [11]. For Kulitta, however, it is further constrained by requiring $c \in \mathbb{Z}$ to model discrete tonal systems, such as those relevant to a piano.

- *Inversion equivalence*, *I*. Chords are related to their negations, which are a reflection around the origin. For example $\langle 0,4,7\rangle \sim_I \langle 0,-4,-7\rangle$.

- *Cardinality equivalence*, C. Chords with duplicate neighboring voices are related to each other. For example $\langle 0,4,7\rangle$ is related to $\langle 0,0,4,7\rangle$ but not to $\langle 0,4,7,0\rangle$.

The reflexive, symmetric, and transitive properties are easy to prove for O, P, and T. However, I and C are problematic since their definitions do not account for all three properties. The definition of I-equivalence is not reflexive, although this is an easy modification to make to the definition. Cardinality equivalence is somewhat more complicated, and, as shown later in this chapter, is more easily dealt with by defining a normalization for the equivalence relation.

The OPT relations can be combined to make new relations by using the join operation: $R_1 \vee R_2$, written as $R_1 R_2$ for simplicity. For example:

2. Note that Octave equivalence is essentially $\mathbf{Z}_{12}$, the integers modulo 12.

- Octave and Transposition equivalence, OT. $\vec{v} \sim_{OT} \vec{v} + 12\vec{i} + c1^n$, $\vec{i} \in \mathbb{Z}^n$, $c \in \mathbb{Z}$. Chords in the same equivalence class have the same intervallic structure when represented as vectors of pitch classes. For example, $\langle 0,4,7 \rangle \sim_{OT} \langle 13,5,8 \rangle$.

- Octave and Permutation equivalence, OP. $\vec{v} \sim_{OPT} \sigma(\vec{v} + 12\vec{i})$, $\vec{i} \in \mathbb{Z}^n$, $\sigma \in \mathbb{S}_n$. Chords in the same equivalence class have the same multisets of pitch classes. For $n = 3$ voices, OP-space contains an equivalence class for all C-major triads, another for all C-minor triads, and so on.

- Permutation and Transposition equivalence, *PT*. $\vec{v} \sim_{PT} \sigma(\vec{v} + c1^n), \sigma \in \mathbb{S}_n, c \in \mathbb{Z}$ (or $c \in \mathbb{R}$ for microtonal systems). Chords in the same equivalence class share the same intervallic structure of their multisets of pitches. For example: $\langle 0,4,7 \rangle \sim_{PT} \langle 5,1,8 \rangle$.

- Octave, Permutation, and Transposition equivalence, OPT. $\vec{v} \sim_{OP} \sigma(\vec{v} + 12\vec{i} + c1^n)$, $\vec{i} \in \mathbb{Z}^n$, $\sigma \in \mathbb{S}_n, c \in \mathbb{Z}$. Chords in the same equivalence class have the same intervallic structure of their multisets of pitch classes, capturing the notion of chord quality. For example, $\langle 0,4,7 \rangle \sim_{OPT} \langle 0,3,8 \rangle$, where $\langle 0,4,7 \rangle$ is a C-major triad and $\langle 0,3,8 \rangle$ is an A-flat-major triad. This can be seen as follows: $\langle 0,4,7 \rangle \sim_O \langle 12,4,7 \rangle \sim_T \langle 8,0,3 \rangle \sim_P \langle 0,3,8 \rangle$

Proofs of these definitions are in Appendix A. Chord spaces involving cardinality equivalence are more easily formalized using their normalizations. Two such examples are PC-equivalence (permutation and cardinality) and OPC-equivalence (octave, permutation, and cardinality). PC-equivalent chords share the same sets of pitches, and OPC-equivalent chords share the same sets of pitch classes. Definitions for these are covered later in the chapter.

3.3.1 Applications of OPTIC

A sequence of representative points from a chord space represents a sequence of equivalence classes. Such a path also represents many possible other paths through non-representative

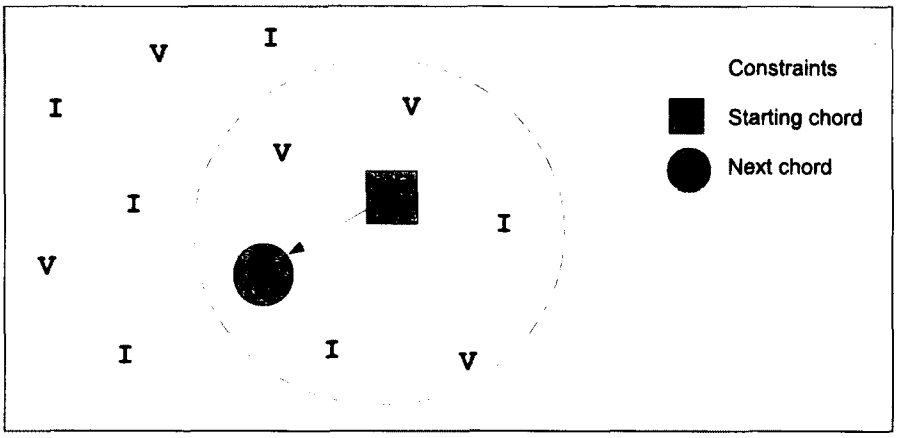


Figure 3.1: An illustration of the path-finding nature of chord spaces for a *I-V* progression. Each Roman numeral can be mapped to many concrete chords, which may literally be thought of as chords floating in space. When we choose a specific *I*-chord, the next transition may be subjected to various 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.

points in those equivalence classes. Given a chord space that has the same level of abstraction as an abstract progression (such as one written as Roman numerals), the task of turning that abstract progression into a concrete progression becomes a path-finding problem.

3.3.2 Normalizations for OPTIC

In order to use the OPTIC relations, there must be ways to test whether two chords are equivalent under each individual relation and combination of relations. For the individual relations and for many combinations of relations, the normalizations can be used for this task. Normalizations for O, P, T, I, are as follows, where *sort* is a function that sorts a vector's elements in ascending, lexicographic order. Proofs of the property in Definition 2 for these normalizations are somewhat trivial, following directly from the simple arithmetic and sorting operations involved. Proofs for these normalizations can be found in Appendix A.

$$normO(\langle x_1,...,x_n\rangle) = \langle x_1 \bmod 12,...,x_n \bmod 12\rangle \tag{3.6}$$

$$normP(\vec{x}) = sort(\vec{x}) \tag{3.7}$$

$$normT(\langle x_1,...,x_n\rangle) = \langle x_1 - x_1,...,x_n - x_1\rangle \tag{3.8}$$

$$normI(\langle x_1,...,x_n\rangle) = \text{if } x_i < 0 \text{ then } \langle -x_1,...,-x_n\rangle \text{ else } \langle x_1,...,x_n\rangle,$$
$$\text{where } x_i \text{ is the first non-zero element of the vector.} \tag{3.9}$$

Although one normalization is shown for T-equivalence above, it serves as a good example of an equivalence relation for which more than one obvious normalization exist. In *normT* above, the first element of a vector, x_1 is subtracted from the entire vector. However, as shown below, any x_i can be used.

Theorem 2. *Let $F = \{f_1,...,f_n\}$ be the set of functions $f_i = (\langle x_1,...,x_n\rangle) = \langle x_1 - x_i,...,x_n - x_i\rangle$. An algorithm $A : \mathbb{Z}^n \to \mathbb{Z}^n$ is a normalization for $\mathbb{Z}^n/T$ if, for all $\vec{x} \in \mathbb{Z}^n$, it applies the same f_i to all members of $\vec{x}$'s equivalence class, $E(\vec{x}, \mathbb{Z}^n/T)$.*

Proof. Recall that $\vec{x} \sim_t \vec{y} \longleftrightarrow \exists c \in \mathbb{Z}, \vec{x} = \vec{y} + c1^n$. Two chords are T-equivalent if they have the same intervallic structure. Adding a constant to a chord produces a T-equivalent chord, so $\forall f \in F$, $\vec{x} \sim_T f(\vec{x})$. Now we must show that $\forall f \in F, l\vec{x} \sim_T \vec{y} \longleftrightarrow normT(\vec{x}) = normT(\vec{y})$.

Let $\vec{x} = \langle x_1,...,x_n\rangle, \vec{y} = \langle y_1,...,y_n\rangle$.

Let $\vec{x'} = \langle x_1 - x_i,...,x_n - x_i\rangle, \vec{y'} = \langle y_1 - y_i,...,y_n - y_i\rangle$ for some $i \in \mathbb{Z}$.

If $\vec{x}$ and $\vec{y}$ have the same intervallic structure (the definition of T-equivalence), then $\vec{x'}$ and $\vec{y'}$ will be equal and the i^{th} element of both $\vec{x'}$ and $\vec{y'}$ will be 0. We therefore have that $c = x_i - y_i$ and $\vec{x} \sim_T \vec{x'} = \vec{y'} \sim_T \vec{y}$. If $\vec{x}$ and $\vec{y}$ are *not* T-equivalent, then the intervallic structures are different and $\vec{x'} \neq \vec{y'}$. Therefore, $A(\vec{x}) = A(\vec{y}) \longleftrightarrow \vec{x} \sim_T \vec{y}$. □

Corollary 2. *The following function:*

$$normT(\langle x_1,...,x_n\rangle) = \langle x_1 - x_1,...,x_n - x_1\rangle \tag{3.10}$$

which subtracts the first element of a vector from all other elements, is a normalization for $\mathbb{Z}^n/T$.

The original definition for C-equivalence technically only relates elements that differ by one set of duplications and is neither symmetric nor transitive. Symmetry is easily assumed, but transitivity is more problematic. Consider the following:

$\langle x,y,z\rangle \sim_C \langle x,y,y,z\rangle \sim_C \langle x,y,y,z,z\rangle$.

To retain transitivity, it must be the case that $\langle x,y,z\rangle \sim_C \langle x,y,y,z,z\rangle$. Cardinality equivalence's definition, therefore, needs to be extended to include any number of sequential duplications in any voice (including zero duplications to ensure reflexivity) such that two chords are C-equivalent if they share the same vectors of pitches where adjacent duplicates are eliminated. Given this definition, it is easiest to formalize C-equivalence by creating its normalization. A normalization for C-equivalence is most succinctly defined recursively, using the list notation from Haskell to represent vectors, where a vector of length n, $\langle x_0,...,x_{n-1}\rangle$ can be written as $[x_0,...,x_{n-1}]$ or $x_0 : ... : x_{n-1} : []$. The code for *normC* below presents this normalization for C-equivalence using Haskell.

$normC :: [Int] \to [Int]$

$normC\ [x0 : x_1 : t] = \textbf{if}\ x0 == x_1\ \textbf{then}\ normC\ (x0 : t)\ \textbf{else}\ (x0 : normC\ (x_1 : t))$

$normC\ x = x$

We then have C-equivalence defined as follows:

$$\vec{x} \sim_C \vec{y} \longleftrightarrow normC(\vec{x}) = normC(\vec{y}) \tag{3.11}$$

Combining Normalizations

Some of the normalizations for O, P, T, and C can be combined to create new normalizations for compound equivalence relations. Proofs for these normalizations can be found in Appendix A.

$$normOP = normP \cdot normO \tag{3.12}$$

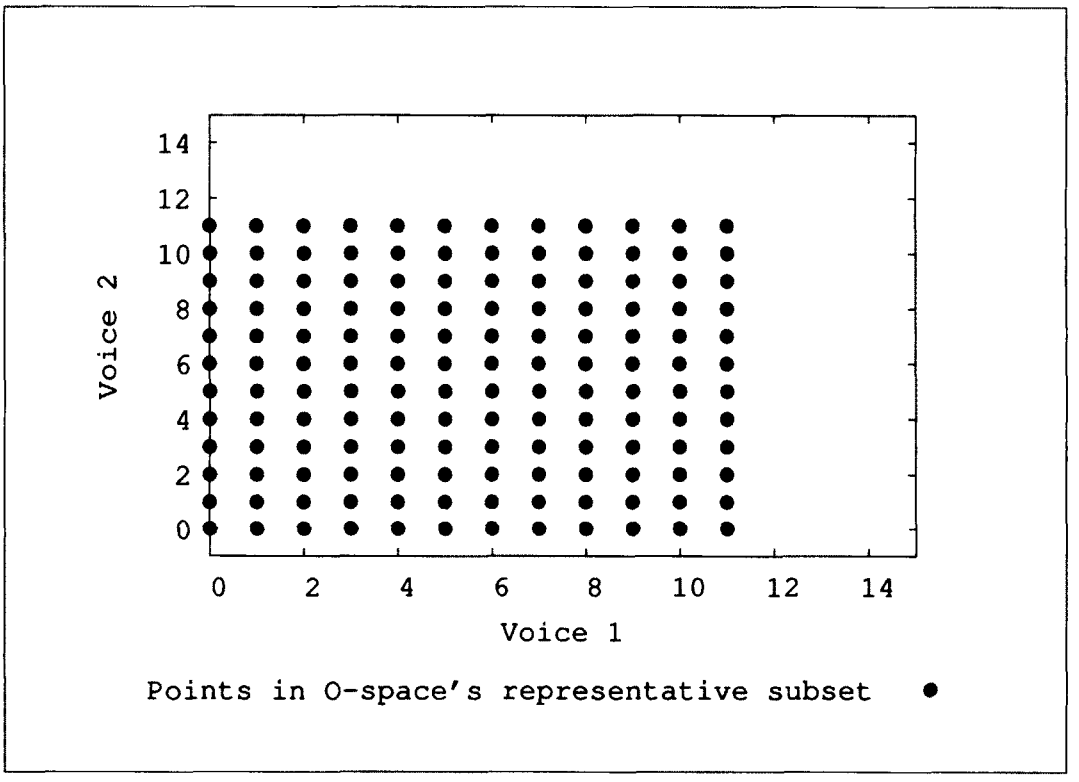


Figure 3.2: The representative subset of O-space for two voices as defined by *normO*.

$$normOT = normO \cdot normT \tag{3.13}$$

$$normPC = normC \cdot normP \tag{3.14}$$

$$normPT = normT \cdot normP \tag{3.15}$$

$$normOPC = normC \cdot normP \cdot normO \tag{3.16}$$

Finding a normalization for an equivalence relation is not always the simplest way to check for equivalence class membership. An example of this is OPT-equivalence, for which a normalization is somewhat more complicated than checking class membership. While representative subsets of OPT-space can be defined [3], it is not easy to normalize chords into this subset of $\mathbb{Z}^n/OPT$. The reason for this is illustrated by the points $\langle 0,2,7 \rangle$

3. For example, Tymoczko et al. define a fundamental domain for OPT-space for three voices [80]. As noted previously in this chapter, fundamental domains are also representative subsets.

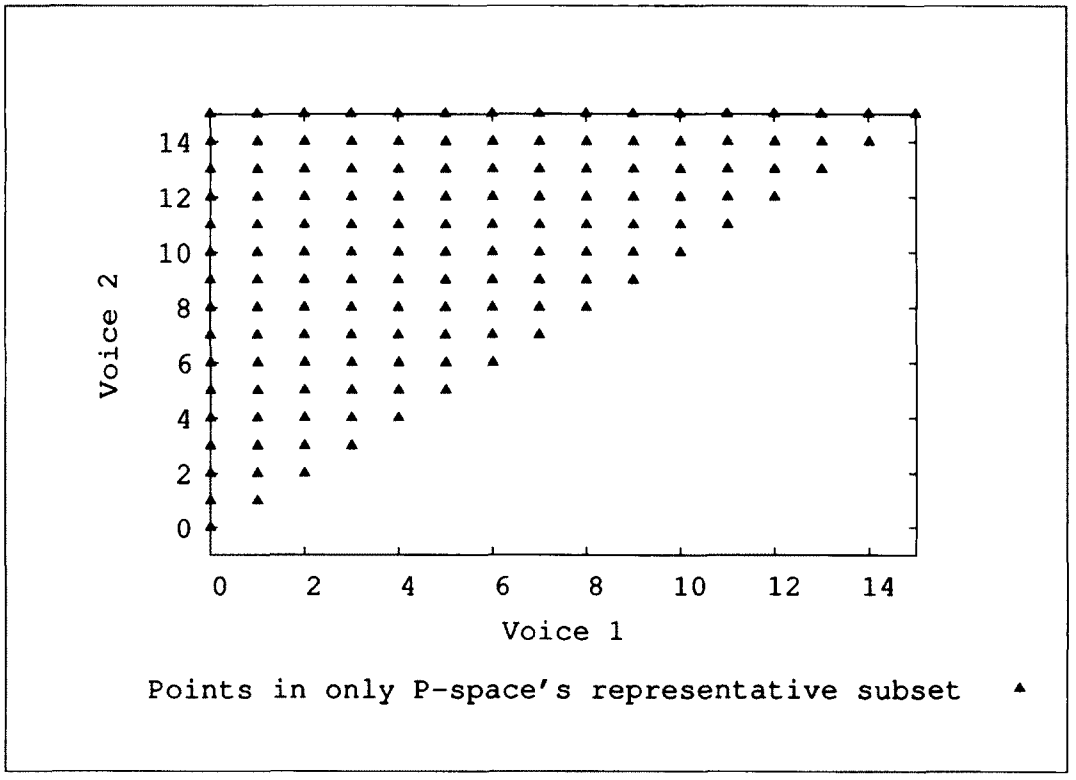


Figure 3.3: The representative subset of P-space for two voices as defined by *normP*.

and $\langle 0,5,7\rangle$, which are related by:

$$\langle 0,5,7\rangle \sim_O \langle 12,5,7\rangle \sim_P \langle 5,7,12\rangle \sim_T \langle 0,2,7\rangle$$

The point $\langle 0,5,7\rangle$ should, therefore, be normalized to $\langle 0,2,7\rangle$ under the conventions of our representative subset. However, we cannot use any of the normalizations discussed so far to accomplish this. $\langle 0,5,7\rangle$ will be mapped to itself with *normP*, *normO*, and *normT*. The same thing happens with $\langle 0,2,7\rangle$ as well. Therefore, we have two choices: create one or more new normalizations, or use another algorithm to test whether two chords are OPT-equivalent.

Testing for OPT-Equivalence

Because the O, P, and T normalizations cannot be combined to create a new normalization for OPT-equivalence, testing equivalence under OPT requires either a different algo-

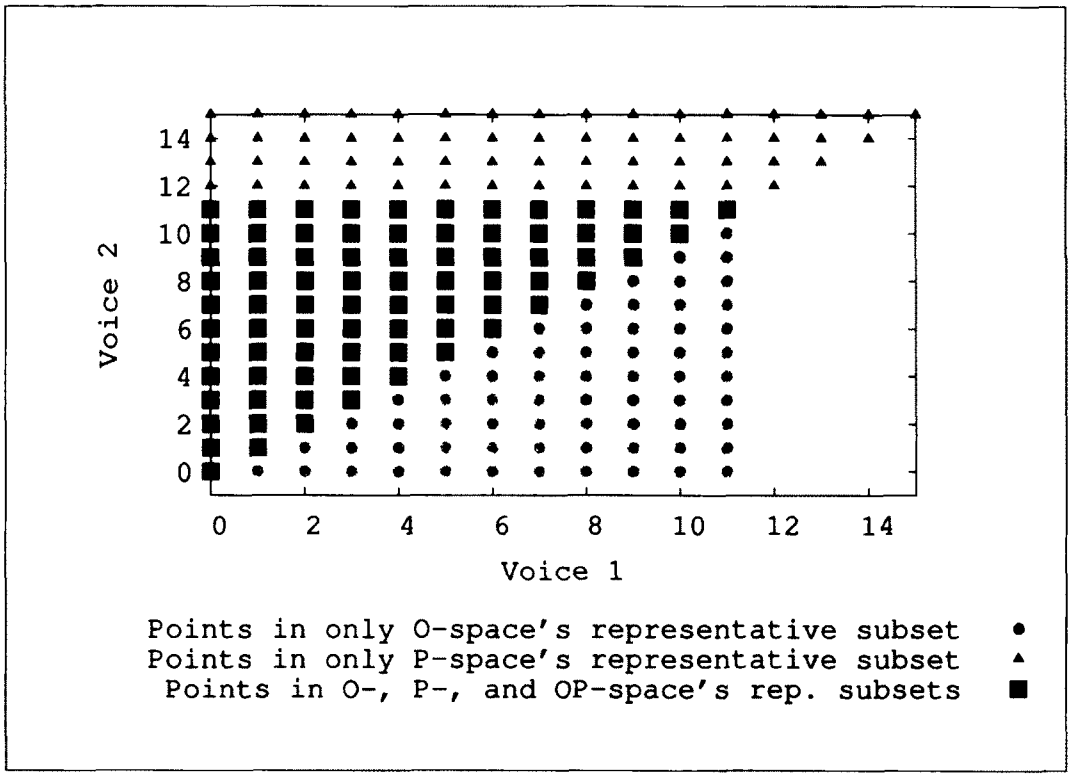


Figure 3.4: The relationship between the representative subsets of O-space, P-space, and OP-space as defined by *normO*, *normP*, and *normOP*.

rithm or a new normalization not based on composing existing normalizations. For OPT-equivalence, algorithm 1 is a function that, although it makes use of the O, P, and T normalizations, does not define a normalization for the OPT relation itself. This algorithm returns true if and only if two chords are OPT-equivalent. It makes use of the vector concatenation operator, $+\!\!+$, defined in Equation 3.17

$$\langle x_1,...,x_n\rangle +\!\!+ \langle y_1,...,y_m\rangle = \langle x_1,...,x_n,y_1,...,y_m\rangle \tag{3.17}$$

Algorithm 1. *Let $\vec{x}$ and $\vec{y}$ be two vectors of length n.*

$optEq(\vec{x},\vec{y}) =$

Let $\vec{x'} = normT(normOP(\vec{x}))$, $\vec{y'} = normT(normOP(\vec{y}))$.

Let $S_{\vec{y}} = \{normPT(\vec{y'} + 12\vec{i}) \mid \vec{i} = 1^m +\!\!+ 0^{n-m},\ 0 \leqslant m < n\}$.

If $\vec{x'} \in S_{\vec{y}}$ *then return true, otherwise return false.*

Theorem 3. *Algorithm 1, optEq, correctly tests for OPT-equivalence.*

Proof. We also have that $\vec{x'}$ and $\vec{y'}$ are sorted vectors in $[0,11]^n$ whose first element is zero. We have that $\vec{x'}$ and $\vec{y'}$ must be OPT-equivalent to $\vec{x}$ and $\vec{y}$ respectively by transitivity. Similarly, we know that $\forall \vec{a} \in S_{\vec{y}}$, $\vec{a} \sim_{OPT} \vec{y}$. Finally, observe that $S_{\vec{y}}$ is the set of all T-normalized *rotations* of $\vec{y'}$ within the range $[0,12]^n$.

We must now show that $\vec{x'} \in S_{\vec{y}} \longleftrightarrow \vec{x'} \sim_{OPT} \vec{y'}$. Chords that meet $\vec{x'}$'s structural constraints will be referred to as "useful" chords. We will make use the following equation, where $min(\vec{v})$ returns a vector's smallest element and and $max(\vec{v})$ returns its largest:

$span(\vec{v}) = max(\vec{v}) - min(\vec{v})$.

The definition of $S_{\vec{y}}$ contains all OPT-equivalent chords to $\vec{y}$ that have a span of < 12, are sorted, and whose first elements are zero. We show the correctness of $S_{\vec{y}}$'s definition in four steps.

1. At least one field of each $\vec{i}$ used to create $S_{\vec{y}}$ must be zero. Otherwise, because of the *normPT* operation, redundant chords will be created: $normPT(\vec{x}) = normPT(\vec{x}+k)$.

2. Octave shifts where $\vec{i}$ contains at least one field that is 0 and at least one field that is > 1 will produce chords with too large a span to be useful.

   Case 1:

   - Let $\vec{v} = \langle ..., a, ..., b, ... \rangle$ where $0 \leqslant a \leqslant b \leqslant 11$.
   - Let $\vec{i} \in [0,2]^n$ be a vector of octave shifts such that
     $normPT(\vec{v} + 12\vec{i}) = \langle a, ..., b+24 \rangle$
     where a is the smallest field non-shifted field and b is the largest field shifted by 2 octaves.
   - Case $a = b$: $span(\langle a, ..., b+24 \rangle) = 24$ (too big)
   - Case $a < b$: $span(\langle a, ..., b+24 \rangle) > 24$ (too big)

- $\vec{v} = \langle ..., a, ..., b, ... \rangle$ $\leqslant a \leqslant b \leqslant$
- $\vec{i} \in [\ ,\ ]^n$

$normPT(\vec{v} + \vec{i}) = \langle b, ..., a + \rangle$

b a

- $a = b$ $span(\langle b, ..., a + \rangle) =$
- $a < b$ $span(\langle b, ..., a + \rangle) =$ $-k \in [\ ,\ ] \geqslant$

$>$

- $\vec{v} = \langle ..., a, ..., b, ... \rangle$ $\leqslant a \leqslant b \leqslant$
- $\vec{i} \in [\ ,\ ]^n$

$normPT(\vec{v} + \vec{i}) = \langle a, ..., b + \rangle$

a b

$\vec{v}$

- $a = b$ $span(\langle a, ..., b + \rangle) =$
- $a < b$ $span(\langle a, ..., b + \rangle) \geqslant$

$\vec{v} = \langle ..., a, b, ... \rangle$ $\leqslant a \leqslant b \leqslant$

$\vec{i}$

$normPT(\vec{v} + \vec{i}) = \langle b, ..., a + \rangle$

b a

Case $a = b$: $span(\langle b, ..., a+12 \rangle) = 12$ (too big)

Case $a < b$: $span(\langle b, ..., a+12 \rangle) < 12$ (useful)

Because of these properties, we know that $S_{\vec{y}}$ contains all possible useful chords—the only chords that might be equal to $\vec{x'}$. Therefore, *optEq* correctly tests for OPT-equivalence. □

A normalization for OPT-equivalence would not be much different from this algorithm. Since any two OPT-equivalent chords will, in fact, generate the same sets of chords for $S_{\vec{y}}$, we can simply take the lexicographically smallest element of the set.

Algorithm 2. $normOPT(\vec{x}) =$

Let $\vec{x'} = normOP(\vec{x})$

Let $S_{\vec{x}} = \{normPT(\vec{x} + 12\vec{i}) \mid \vec{i} = 1^m +\!\!+\ 0^{n-m},\ 0 \leqslant m < n\}$.

Return minimum$(S_{\vec{x}})$*, where minimum returns the lexicographically smallest memeber of the set.*

For example, since $\langle 0,2,7 \rangle$ is lexicographically smaller than $\langle 0,5,7 \rangle$, $\langle 0,5,7 \rangle$ will be normalized to $\langle 0,2,7 \rangle$ and $\langle 0,2,7 \rangle$ will be normalized to itself.

Theorem 4. *normOPT is a normalization for OPT-equivalence.*

Proof. The correctness of *normOPT* follows directly from the correctness of *optEq*. $S_{x'}$ will contain all OPT-equivalent chords falling within $[0,11]^n$ that are sorted in ascending order and whose first element is zero. This set will be the same for all members of an OPT-equivalence class of chords. □

Using these methods of testing for OPT-equivalence, a test and normalization can be defined for OPTC-equivalence. This relates chords whose sets of pitch classes are OPT-equivalent. The *normOPC* operation can be used to reduce a vector to its set of pitches, which can then be compared using *optEq* to determine OPTC-equivalence or further normalized by *normOPT* to achieve a normalization for OPTC.

$$optcEq(\vec{x}, \vec{y}) = optEq(normOPC(\vec{x}), normOPC(\vec{y})) \tag{3.18}$$

$$normOPTC = normOPT \cdot normOPC \tag{3.19}$$

It is important to note that *normOPTC* cannot be defined using *normPC* instead of *normOPC*. The reason for this is that *normPC* will only remove duplicate *pitches*, while *normOPTC* clearly needs duplicate *pitch classes* to be removed as well. The corresponding proof for *normOPTC*'s normalization properties can be found in Appendix A.

3.3.3 Groups

The O, P, T, and I relations can also be represented as parameterized functions, where equivalent chords can be produced from an input chord. These functions will be written with the Haskell currying notation, where $f(x,y)$ is written $f\ x\ y$ and $(f\ x)$ is a function that takes one additional argument.

$$o\ \vec{i}\ \vec{x} = \vec{x} + 12\vec{i},\quad \vec{i} \in \mathbb{Z}^n \tag{3.20}$$

$$p\ \sigma\ \vec{x} = \sigma(\vec{x}),\quad \sigma \in S^n \tag{3.21}$$

$$t\ k\ \vec{x} = \vec{x} + k\vec{1}^n,\quad k \in \mathbb{Z} \tag{3.22}$$

$$i\ k\ \vec{x} = k\vec{x},\quad k \in \{1, -1\} \tag{3.23}$$

Each of the functions above has the form $f_R\ p\ \vec{x} = y$ for some parameter $p \in P_R$ for relation R. The equivalence relations could then be described as follows:

$$\vec{x} \sim_R \vec{y} \longleftrightarrow \exists p \in P_R \mid f_R(\vec{x}, p) = \vec{y} \tag{3.24}$$

For a relation R that can be defined using operation f_R and parameters P_r, the set of all

functions that can be applied to any chord $\vec{x}$ is $F_R = \{f_R\ p \mid p \in P_r\}$. For the parameterizations above, o, p, t, and i form groups.

$$G_O = (\{o\ \vec{i} \mid \vec{i} \in \mathbb{Z}^n\}, \cdot) \tag{3.25}$$

$$G_P = (\{p\ \sigma \mid \sigma \in S^n\}, \cdot) \tag{3.26}$$

$$G_T = (\{t\ k \mid k \in \mathbb{Z}\}, \cdot) \tag{3.27}$$

$$G_I = (\{i\ k \mid k \in \{1, -1\}\}, \cdot) \tag{3.28}$$

G_P is a group because S_n is a group, and the two are synonyms for the same group, just with slightly different notation. Proofs of the group properties of G_O, G_T, and G_I follow from properties of addition and multiplication and can be found in the Appendix A. G_O, G_T, and G_I are also Abelian, since the order of composition for the functions does not matter. S_n, and, therefore, G_P are not Abelian.

3.3.4 OPTIC in Haskell

Since Kulitta operates on chords in $\mathbb{Z}^n$, we define chords as vectors or lists of integers. Euterpea contains the type *AbsPitch* as a type synonym for *Int*. We extend this to represent chords similarly.

type *AbsChord* = [*AbsPitch*]

Many of the various combinations of OPTIC operations are individually most easily implemented in Haskell using the normalizations described previously.

normO, *normT*, *normP*, *normOP*, *normPT*, *normOPC* :: *Norm AbsChord*

normO = *map* (‘*mod*‘12)

normT x = *map* (*subtract* \$ *head x*) *x*

normP = *sort*

normOP = *sort* ∘ *normO*

$normOT = normO \circ normT$

$normOPC = nub \circ normOP$

These are then easily turned into equivalence relations of type *EqRel* using *normToEqRel*.

$oEq, pEq, tEq, opEq, opcEq :: EqRel\ AbsChord$

$[oEq, pEq, tEq, opEq, otEq, opcEq] = map\ normToEqRel$

$[normO, normT, normP, normOP, normOT, normOPC]$

Group operators can also be defined for the O, P, T, and I. Each operator in Haskell mirrors its mathematical definition. Vectors are represented as lists. The octave operator, *o*, takes a list of octave shifts and a chord. The *zipWith* operator combines the two vectors.

$o :: [Int] \rightarrow AbsChord \rightarrow AbsChord$

$o\ is\ xs = zipWith\ (\lambda i\ x \rightarrow x + 12 * i)\ is\ xs$

The permutation operator, *p*, takes a permutation, *s* (for "sigma"), as its first argument, which is represented as a list of indices into a list. The *s* argument must, therefore, be the same length as *xs* and be a permutation of $[0..length\ xs - 1]$. For example, $p\ [3,1,2]\ [0,4,7]$ evaluates to $[7,0,4]$.

$p :: [Int] \rightarrow AbsChord \rightarrow AbsChord$

$p\ s\ xs = map\ (xs!!)\ s$

The transposition operator, *t*, simply adds a constant to a vector, and the inversion operator, *i*, takes a Boolean value that determines whether the chord is multiplied by 1 (left unchanged) or by -1.

$t :: Int \rightarrow AbsChord \rightarrow AbsChord$

$t\ c\ xs = map\ (+c)\ xs$

$i :: Bool \rightarrow AbsChord \rightarrow AbsChord$

$i\ neg\ xs = \textbf{if}\ neg\ \textbf{then}\ map\ (*(-1)))\ \textbf{else}\ xs$

As already discussed, OPT is problematic and is more easily defined using a different algorithm that makes use the group operator for octave equivalence, *o*.

```
optEq :: EqRel AbsChord
optEq x y =
  let n = length y
      (x', y') = (normT $ normOP x, normT $ normOP y)
      is = map (λk → take k (repeat 1) ++ take (n − k) (repeat 0)) [0..n]
      s = map (normT ∘ normP) $ map (λi → o i y') is
  in or (map (== x') s)
```

In the definition above, *is* is the set of all octave shifts that result in rotations of the vector, and s is $S_{\vec{y}}$ from Algorithm 1. From this algorithm, as described previously, OPTC-equivalence can be tested by first normalizing into OPC space and then testing for OPT-equivalence.

```
optcEq :: EqRel AbsChord
optcEq a b = optEq (normOPC a) (normOPC b)
```

3.4 Contour Equivalence

Contour equivalence is a concept introduced by Morris [56]. Contours exist over a sequence of pitches. These pitch sequences are most intuitively thought of as pitches in a melody, but they can actually be any musical feature that would be represented as a vector of pitches, such as a chord. A pitch vector's contour is a ranking of its elements from smallest to largest. This is defined by the following algorithm, where *sort* is a function that sorts a vector's elements in ascending order (e.g. $sort(\langle 3,1,2\rangle) = \langle 1,2,3\rangle$).

Algorithm 3. $rank(\vec{x}) =$

Let $\vec{x'} = normPC(\vec{x})$

Replace each field of $\vec{x}$ *with its index in* $\vec{x'}$

The Haskell definition is very similar to the algorithm above, where fields in vectors are indexed from zero. After finding $\vec{x'}$, the *ranks* value is computed as a list of tuples, which

serves as a lookup table for each pitch's rank.

$$
\begin{aligned}
&\mathit{rank} :: [\mathit{AbsPitch}] \to [\mathit{Int}] \\
&\mathit{rank}\ xs = \\
&\quad \textbf{let}\ x' = \mathit{normPC}\ xs \\
&\qquad \mathit{ranks} = \mathit{zip}\ x'\ [0\,..\,\mathit{length}\ x' - 1] \\
&\quad \textbf{in}\ \mathit{map}\ (\lambda x \to \mathit{fromJust}\ \$\ \mathit{lookup}\ x\ \mathit{ranks})\ xs
\end{aligned}
$$

Contour equivalence can be defined as an equivalence relation, *Con*.

$$\vec{x} \sim_{Con} \vec{y} \longleftrightarrow \mathit{rank}(\vec{x}) = \mathit{rank}(\vec{y}) \tag{3.29}$$

For example, $\mathit{rank}(\langle 5,7,5,10\rangle) = \langle 0,1,0,2\rangle$, and $\mathit{rank}(\langle 3,10,3,12\rangle) = \langle 0,1,0,2\rangle$, so $\langle 5,7,5,10\rangle \sim_{Con} \langle 3,10,3,12\rangle$. A *Con*-equivalence class consists of pitch vectors that all have the same relative ranking of elements, or the same general type of shape. The function *rank* both defines the equivalence relation and is a normalization for it. Reflexivity, symmetry, and transitivity follow from the definition of *Con* using a normalization.

The concept of "melodic contour" in a less mathematically strict sense has been used as a form of musical abstraction in automated composition tasks [41]. Although Kulitta currently does not make direct use of contour equivalence for generating melodies, it would be easily usable within the existing framework and is an appealing avenue of future work.

3.5 Modal Equivalence

The harmony of a lot of classical Western music is centered around primarily two modes: major and natural minor, with the intervallic structures $\langle 2,2,1,2,2,2,1\rangle$ and $\langle 2,1,2,2,1,2,2\rangle$ respectively. The minor scale is actually a rotation of the major scale's intervallic structure. There are seven such rotations, each yielding a different mode as shown in Table 3.1.

A mode can be represented using several levels of abstraction: as a collection of intervals, as a collection of pitch classes, or as a collection of pitches. In keeping with the OPTIC

Rotation	Name	Intervallic structure	Scale rooted at 0
0	Ionian (Major)	$\langle 2,2,1,2,2,2,1\rangle$	$\langle 0,2,4,5,7,9,11\rangle$
1	Dorian	$\langle 2,1,2,2,2,1,2\rangle$	$\langle 0,2,3,5,7,9,10\rangle$
2	Phrygian	$\langle 1,2,2,2,1,2,2\rangle$	$\langle 0,1,3,5,7,8,10\rangle$
3	Lydian	$\langle 2,2,2,1,2,2,1\rangle$	$\langle 0,2,4,6,7,9,11\rangle$
4	Mixolydian	$\langle 2,2,1,2,2,1,2\rangle$	$\langle 0,2,4,5,7,9,10\rangle$
5	Aeolian (Minor)	$\langle 2,1,2,2,1,2,2\rangle$	$\langle 0,2,3,5,7,8,10\rangle$
6	Locrian	$\langle 1,2,2,1,2,2,2\rangle$	$\langle 0,1,3,5,6,8,10\rangle$

Table 3.1: The intervallic structure of all seven modes based on the major scale and an example scale rooted at 0 (pitch class C) for each.

way of handling chords, modes can be thought of as a 7-voice chord where each voice is a unique pitch class. We use this representation to define two new concepts: *modally related* chords and *modal equivalence*.

We define the set of all modes as chords to be transpositions of members of the rightmost column from Table 3.1:

$$\mathbb{M} = \{\vec{m} = t\ k\ \vec{m'} \mid k \in [0,11],\ \vec{m'} \in \{\langle 0,2,4,5,7,9,11\rangle, ..., \langle 0,1,3,5,6,8,10\rangle\}\} \quad (3.30)$$

We will refer to a chord as being a *member* of a mode if its pitch classes are a subset of those allowed in the mode. Consider the power set operation, normally written as $\mathbb{P}(S)$ for set S: $\mathbb{P}(S) = \{S' \subseteq S\}$. This operation can also be defined over a set represented as a vector (i.e. the elements are sorted and no duplicates exist).

$$\mathbb{P}(\langle x_1,x_2,...,x_n\rangle) = \{\langle x_1\rangle,\ ...,\ \langle x_n\rangle,\ \langle x_1,x_2\rangle,\ ...,\ \langle x_1,x_n\rangle, ..., \langle x_1,x_2,...,x_n\rangle\} \quad (3.31)$$

For example:

$$\mathbb{P}(\langle 1,2,3\rangle) = \{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle, \langle 1,2,3\rangle\} \quad (3.32)$$

For a chord $\vec{x} \in \mathbb{Z}^n$ and a mode $\vec{m} \in \mathbb{M}$, a chord belongs to a mode if its pitch classes belong to the mode. The normalization for OPC-equivalence reduces a chord to its sorted

set of pitch classes, creating the right level of abstraction for this test.

$$member(\vec{x},\vec{m}) \longleftrightarrow normOPC(\vec{x}) \in \mathbb{P}(\vec{m}) \tag{3.33}$$

Two chords are then *modally related* if their pitch classes are subsets of the same mode.

$$modallyRelated(\vec{x},\vec{y}) \longleftrightarrow \exists \vec{m} \in \mathbf{M},\ member(\vec{x},\vec{m}) \wedge member(\vec{y},\vec{m}) \tag{3.34}$$

The predicate *modallyRelated* defines a relation, but it is not an equivalence relation due to the fact that some chords have ambiguous modal membership. The two-note chord, $\langle 0,7 \rangle$, is one such ambiguous chord, being a member of all modes except Locrian. Therefore, we have $modallyRelated(\langle 0,7 \rangle, \langle 0,4,7 \rangle)$ and $modallyRelated(\langle 0,7 \rangle, \langle 0,3,7 \rangle)$, but $\langle 0,4,7 \rangle$ and $\langle 0,3,7 \rangle$ are not modally related since there are no modes in $\mathbf{M}$ that contain $\{0,3,4,7\}$.

The ambiguity issue already discussed means that vectors of pitches are not specific enough to create an equivalence relation grouping chords in a way that allows them to be explored by mode. One way to do this is to "tag" the chords with additional information, namely the modes to which they belong, since we need $\langle 0,7 \rangle$ in one mode to be differentiated from $\langle 0,7 \rangle$ in another mode.

$$S_{M0} = \{(\vec{m} \in \mathbf{M},\ \vec{x} \in \mathbb{Z}^n) \mid member(\vec{x},\vec{m})\} \tag{3.35}$$

This space is infinite because of $\vec{x} \in \mathbb{Z}^n$. However, the subset of this space where $\vec{x}$ is a member of OPC's representative subset is far more manageable.

$$S_M = \{(\vec{m} \in \mathbf{M},\ normOPC(\vec{x} \in \mathbb{Z}^n)) \mid member(\vec{x},\vec{m})\} \tag{3.36}$$

This can be redefined using the definition of $\mathbb{P}$ over vectors. If $member(\vec{x},\vec{m})$, then $\vec{x} \in \mathbb{P}(\vec{m})$.

$$S_M = \{(\vec{m} \in \mathbf{M},\ \vec{x} \in \mathbb{P}(\vec{m})\} \tag{3.37}$$

This set of chords is finite: $S_M = 10,752$. If grouped by the mode member of the tuple, $\vec{m}$, there are 84 equivalence classes (one per each mode rooted at a particular pitch class), each containing 128 chords. Because chords are represented as tuples with a mode as context, modal equivalence is trivial to define over S_M.

$$(\vec{m}_1, \vec{x}_1) \sim_M (\vec{m}_2, \vec{x}_2) \longleftrightarrow \vec{m}_1 = \vec{m}_2 \tag{3.38}$$

We refer to this new quotient space, S_M/M, as *mode space*. Mode space is easily enumerable and can also be generated more efficiently than the other quotient spaces discussed so far by utilizing its relationship to the power set operation. As shown in chapter 6, mode space represents an appealing level of abstraction for Jazz, bridging the gap between representative subsets of Roman-numeral-level OP-space and the more complex set of chords present in Jazz, as seen in Table 3.2.

	Major tonic		Minor tonic	
Roman numeral	Triad	Mode	Triad	Mode
I	Major	Major	Minor	Minor
II	Minor	Dorian	Diminished	Locrian
III	Minor	Phrygian	Major	Major
IV	Major	Lydian	Minor	Dorian
V	Major	Mixolydian	Minor	Phrygian
VI	Minor	Minor	Major	Lydian
VII	Diminished	Locrian	Major	Mixolydian

Table 3.2: Modal interpretation of Roman numerals.

3.6 Musical Equivalence Relations in Kulitta

Kulitta uses musical equivalence relations, or chord spaces, to transition between different levels of musical abstraction: a path through an abstract space is converted to a path in a more concrete space. However, this only solves part of the compositional problem, and does not address how to create the starting, abstract path or how to satisfy other musical constraints while transforming that path into a more concrete one. Kulitta uses *musical grammars* to create the abstract path, and then uses *constraint satisfaction* algorithms during path finding through chord spaces. These topics are covered in Chapters 4 and 5 respectively. Chapter 6 shows an integrated view of how chord spaces, musical grammars, and constraint-satisfaction interact to create complete pieces of music.

Chapter 4

A Grammar for Harmonic and Metrical Structure

The harmonic analysis of music has long been noted to be analogous to the parsing of natural languages. In the Schenkerian tradition, harmonic structure in music is viewed hierarchically, yielding essentially a parse tree of harmonic sections. Recent work has shown that music and spoken language involve the same parts of the brain [7], and work such as Generative Theory of Tonal Music [48] presents a grammatical outlook on many aspects of musical structure.

In natural language, a sentence would be parsed by starting with the terminal symbols (words), working backwards to infer their function (noun, verb, etc.). These symbols would then be grouped into grammatical phrases (adjective-noun, subject-verb-object, etc.), forming a hierarchical structure that ends with the start symbol representing a sentence. In music, especially in the Schenkerian tradition, a piece of music would be parsed by starting with the terminal symbols (notes, rests, and chords), and working backwards to infer local harmonic progressions (such as ii-V-I), song forms (such as AABA[1]), creating a similar,

1. Large-scale patterns of repetition in music are typically denoted using capital letters. ABA form would indicate a 3-section piece with identical (or sufficiently identical) first and last sections. Similarly, AABA indicates a 4-section piece where the first, second, and fourth sections are the same.

hierarchical structure that ends with a simple I-V-I or even just I (the tonic), serving the function of a start symbol [73, 74].

In the context of Kulitta, however, we are primarily interested in automated music composition rather than analysis. One way to approach this is to use grammars generatively—that is, to generate sentences from the start symbol. Unfortunately, with many conventional grammars (such as context-free grammars, or CFGs) 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 [68].

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 durations of those chords. Chord symbols in analytical grammars are typically duration-less (such as those in Martin Rohrmeier's grammar for harmony [67]), despite the importance of rhythm in music [48, 77, 78]. When chords are durationless, the Schenkerian idea that the V-I would occupy the same duration as their parent I-chord is impossible to capture.

To overcome these problems, we define a new class of generative grammars called *probabilistic temporal graph grammars* [63], or PTGGs [2]. These grammars operate on

2. PTGGs are based on a similar category of grammars called Temporal Generative Graph Grammars

duration-parameterized chords and the rules are functions of those parameters. In a generative setting, this added complexity over a traditional CFG is highly efficient and much more expressive.

4.1 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 [16, 87], neural nets [4, 5, 26, 29, 30, 35], and more specialized systems intended for generating whole compositions [19, 25].

Grammars are an appealing representation for music because of their ability to capture long-spanning structural constraints such as those found in the harmonic structure of music—for example, starting and ending in the same key. Other popular representations used in algorithmic composition algorithms have problems capturing more than short-term structure in music. Markov Chains and Neural nets are two commonly used approaches that suffer from this problem.

A Markov Chain of order n represents a finite state machine where each state captures n steps of production "history." Each state has a collection of transition probabilities to other states. Markov chain-based approaches are commonly used both for small-scale algorithmic composition tasks and for tasks where partial musical information is already given, such as melodic harmonization [70, 87]. However, Markov chains are doomed to perform poorly at more complex tasks where larger scale musical structure must be generated, since they can only "keep track of" as many productions as their order, n, allows, resulting in state explosion when trying to capture constraints over longer generated sections. Although approaches such as variable-length Markov chains [9, 70] can help to mitigate the state explosion for some tasks, they do not eliminate the problem. For even a

defined by Quick and Hudak[64].

variable-length Markov chain to capture a constraint that spans from the first symbol to the last symbol, the order, n, would have to be the length of the generated section—a clearly unreasonable approach.

Neural nets have been used for problems in automated harmonization [26, 29, 30, 35, 57]. A related type of network, called a Boltzmann machine, has been applied to a wider variety of musical tasks: classification of existing music, "fill in the blank" problems (like automated harmonization), and free composition [4, 5]. Boltzmann machines are particularly appealing for their versatility in this regard, since the same model can be reused for each task by simply "clamping" (holding constant) different nodes in the net. However, Boltzmann machines as well as other neural net systems still suffer from complexity problems when dealing with music, since output nodes must be tied to pitches or pitch classes. Representing a decision about a single note choice out of n possibilities requires n output nodes. For m independent choices with n possibilities each, most representations require n^m output nodes. This quickly become problematic for representing complex structures.

4.1.1 Macro Grammars

Macro grammars [28] are a category of context-free grammars that allow both standard productions, such as $A \rightarrow a$, as well as productions that are functions $F(x) \rightarrow w$, where x is an argument or variable and w is an expression that uses x. These function-based productions can capture features that would otherwise require a context-sensitive grammar. For example, a macro grammar can be used to generate strings of the form $a^n b^n c^n$ (some number of as followed by the exact same number of bs and cs):

$S \rightarrow F(a,b,c)$

$F(x,y,z) \rightarrow (xa,yb,zc)$

$F(x,y,z) \rightarrow xyz$

A more typical CFG would have no way to capture the constraint that there must be the same number of each of the three characters in the string, being able to capture $a^n b^n$, but

not $a^n b^n c^n$. PTGGs are similar to macro grammars in that they are context-free grammars that use functions to capture certain features (including repetition) that would otherwise require a context-sensitive grammar.

4.1.2 Musical Grammars

Grammars have been explored both generatively [33, 42, 54, 86] and analytically [48, 67, 85] in music. Studies on brain activity have shown a strong link between language and music in the brain [7], an idea that has become increasingly accepted in music theory through works like GTTM, which presents a grammatical outlook on analyzing music [48] (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 [69] and for representing aspects of musical scores [3].

Martin Rohrmeier introduced a mostly context-free grammar (CFG) for parsing classical Western harmony [67]. The grammar is based on the tonic, dominant, and subdominant chord functions. Terminals are the Roman numerals from I to VII, and the nonterminals 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 [50]. 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 [45]. A fundamental difference between our system and FHarm is that FHarm harmonizes an existing melody, whereas Kulitta can compose from scratch without existing musical input from the user.

The recently proposed analytical grammar by Martin Rohrmeier exhibits a small amount

of context sensitivity based on mode. For example, tonic chords, denoted as T, are given different, modally-determined productions [67].

$T \rightarrow$ I

$T \rightarrow TP$

$T \rightarrow TCP$

$TP \rightarrow$ VI when in major

$TP \rightarrow$ III when in minor

$TCP \rightarrow$ III when in major

$TCP \rightarrow$ VI when in minor

However, consider a PCFG formed from the grammar above. If the production probabilities for TP and TCP are the same, this collection of rules is really equivalent to a reduced set of completely context-free rules:

$T \rightarrow$ I

$T \rightarrow$ VI

$T \rightarrow$ III

The TP and TCP nonterminals would allow the production probabilities to differ based on mode, but determining exactly how they differ is an open problem best addressed in a machine learning context. Determining how musical contexts such as the current mode should be handled in both the alphabet and construction of rules is the subject of later chapters, where production probabilities are derived from musical corpora.

Meter is another clearly important aspect of music. In work such as GTTM, meter interacts with harmonic aspects of the music through metrical grouping and preference rules [48]. Temperley's work [76, 77, 78] as well as a harmonic analysis algorithm by Raphael and Stoddard [66] 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 [42].

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 or no direct support for this kind of musical feature, and many other algorithms are fundamentally incapable of supporting it as well. Markov chain and most Neural Net-based 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* expressions.

4.2 Generating Music with a PTGG

A graphical representation of the generative portion of Kulitta discussed in this chapter can be seen in Figure 4.1. It begins with a PTGG for chord progressions (defined in the Section 4.3), 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 Kulitta's generation *interprets* those abstract progressions. We wish to emphasize that there are many possible algorithms and mathematical models to use at this stage, since it determines many of the stylistic elements of the music. Kulitta uses a mathematical construct called *chord spaces* and style-specific embellishment algorithms to generate music at the level of a MIDI file—roughly the level of representation offered by a paper score. Musical interpretation is discussed in more detail in Chapters 5 and 6.

Additionally, just because Kulitta can produce performable music does not mean that

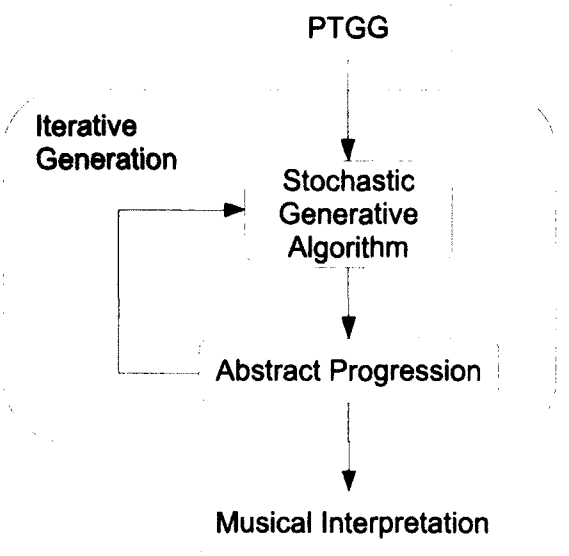


Figure 4.1: An illustration of the generative process for a probabilistic temporal graph grammar (PTGG). A PTGG is used in combination with an iterative algorithm for applying the grammar to produce sequences of abstract progressions consisting of Roman numerals, modulations, and *Let* expressions to capture repetition. After generation, *Let* expressions must be interpreted by instantiating variables.

the results are closed to further alteration by either other algorithms or a human. For example, Kulitta's support for generating abstract structure with musical grammars could be employed as an algorithmic component in otherwise human-crafted compositions that could be in any number of styles.

4.3 Grammar Definition

A grammar is a tuple, $G = (N, T, R, S)$ where N is a set of nonterminals, T is a set of terminals, R is a set of rules from $N \rightarrow (N \cup T)+$, and $S \in N$ is the start symbol. Terminals are symbols that cannot be replaced (or, alternatively, can only produce themselves), whereas nonterminals have rules that replace them with one or more other symbols. Rules have the form $A \rightarrow v$ where v is a sequence of one or more terminals and nonterminals.

A PTGG has several core concepts that distinguish it from more standard CFGs:

1. The grammar generates sequences of duration-parameterized abstract chords, written

as Roman numerals, and modulation symbols.

2. Chords function as both terminals and nonterminals. 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 affect 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 can be infinite even with a finite set of rules.

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 by using both uppercase and lowercase Roman numerals. When this distinction is made, i would indicate a minor chord and I a major chord. However, this distinction is not made within Kulitta. Therefore, all chords are

written with upper case Roman numerals to yield the following alphabet:

$$C = \{I, II, III, IV, V, VI, VII\} \tag{4.1}$$

The simplifying assumption that major and minor modes do not need to be distinguished in the alphabet of Roman numerals was made both to allow for a smaller rule set and because it is not clear from existing work how best to capture those concepts in a generative setting. Sometimes modal distinctions are ignored in an analytical setting as well [67].

The nonterminals of a PTGG are the set of all duration-parameterized chords:

$$N = \{c^t | c \in C, t \in \mathbb{R}\} \tag{4.2}$$

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).

$$M = \{M_2, M_3, M_4, M_5, M_6, M_7\} \tag{4.3}$$

The terminals of our grammar include both nonterminals and modulation symbols. Parentheses are used as an additional "meta symbol" for indicating nested structures in generated sequences.

$$T = N \cup M \tag{4.4}$$

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, the expression **let** $x = A$ **in** xBx, where A and B are nonterminals, is not the same as the expression 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:

$$k \in K \ ::= \ c \mid k_1 ... k_n \mid (m\ k_1) \mid \textbf{let}\ x = k_1\ \textbf{in}\ k_2 \mid x \in Var \tag{4.5}$$

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

4.3.1 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.

We implement production rules as functions in Haskell [59]. 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 nonterminal) is associated with a probability and the probabilities for a particular left-hand side (a single nonterminal) must sum to 1.

4.4 Haskell Implementation

This section presents an implementation of PTGG in Haskell that closely mirrors the mathematical presentation above. Simple data types capture the essence of chords, modulations, let-in expressions, 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.

4.4.1 Chords, Progressions, and Modulations

Roman numerals represent chords built on scale degrees, of which there are seven.

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. The first scale degree is the root, and there is no need to indicate staying within the current key.

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 *nonterminal* (*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*

4.4.2 Rules

We begin with the following type synonyms for clarity in type signatures for probabilities (*Prob*), random number seeds (*Seed*), and duration (*Dur*).

type *Prob* = *Double*

type *Seed* = *Int*

type *Dur* = *Rational*

Rules are a functions from duration-parameterized chords to chord progressions. 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. To capture this, we define a constructor that takes a lefthand-side tuple of a *CType* and production probability, and pairs it with a *RuleFun* (a function from duration to chord progressions).

data *Rule* = (*CType*, *Prob*) :-> *RuleFun*

type *RuleFun* = *Dur* → *Term*

We also introduce abbreviations for single-chord *Term* values to allow chord progressions to be written more concisely.

i, *ii*, *iii*, *iv*, *v*, *vi*, *vii* :: *RuleFun*

[*i*, *ii*, *iii*, *iv*, *v*, *vi*, *vii*] = *map* (λ*c t* → *NT* (*Chord t c*)) $ *enumFrom I*

Note that the usage of lower-case numerals is required to define these abbreviations as functions in Haskell, but the quality of a chord indicated by the above functions is still determined by the modal context in which it appears.

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

(*I*,*p*) :-> λ*t* → *S* [*v* (*t* / 2), *i* (*t* / 2)]

Table 4.1 shows a complete PTGG. The following are some specific rules taken from our implementation of that table 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) :-> λ*t* → *S* [*iv* (*t* / 2), *v* (*t* / 2)]

ruleV9 = (*V*, 0.10) :-> *v*

$$ruleV10 = (V, 0.10) :-> (Mod\ M5 \circ i)$$

Rules according to Schenkerian theory and the metrical structures in work like Generative Theory of Tonal Music (GTTM) [48] 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. However, 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, although they may yield little or no sense of meter. Therefore, we do not explore these types of grammars, but simply note that they are legal as PTGGs.

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

$$X^t \rightarrow \textbf{let}\ x = X^{t/2}\ \textbf{in}\ x\,x \tag{4.6}$$

$$X^t \rightarrow \textbf{let}\ x = X^{t/4}\ \textbf{in}\ x\,X^{t/2}\,x \tag{4.7}$$

$$X^t \rightarrow \textbf{let}\ x = X^{t/4}\ \textbf{in}\ x\,V^{t/2}\,x \tag{4.8}$$

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. The rules shown so far already demonstrate this to some degree by using an infinite alphabet to accommodate durations and by handling repetition within rules. Rules can also exhibit *conditional* behavior.

One problematic aspect of the generative process that can be solved by adding conditional behavior to rules 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 an elegant, functional approach to this by encoding the decision making directly into the rules:

$$myRuleFun :: RuleFun$$

$$myRuleFun\ d = \textbf{if}\ d < durLimit\ \textbf{then}\ term_1\ \textbf{else}\ term_2$$

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.

4.4.3 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."[3]

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

3. This strategy is similar to that used for an L-system or Lindenmayer system [61].

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.

```
newtype Prog a = Prog (StdGen → (StdGen, a))

instance Monad Prog where

  return a = Prog (λs → (s, a))

  Prog p0 >>= f1 = Prog $ λs0 →

    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 (λg →

  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)
```

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 (λ((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')
```

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 alter 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 by iterating a monadic action infinitely often.

```
gen :: [Rule] → Int → Seed → Term → Term
gen rules i s t = runP (iter (update rules) t) (mkStdGen s) !! i
iter :: Monad m ⇒ (a → m a) → a → m [a]
iter f a = do a' ← f a
              as ← iter f a'
              return (a' : as)
```

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*.

Num.	Probability	Rule
1	0.20	$I^t \rightarrow II^{t/4}\ V^{t/4}\ I^{t/2}$
2	0.20	$I^t \rightarrow I^{t/4}\ IV^{t/4}\ V^{t/4}\ I^{t/4}$
3	0.20	$I^t \rightarrow V^{t/2}\ I^{t/2}$
4	0.20	$I^t \rightarrow I^{t/4}\ II^{t/4}\ V^{t/4}\ I^{t/4}$
5	0.20	$I^t \rightarrow I^t$
6	0.80	$II^t \rightarrow II^t$
7	0.20	$II^t \rightarrow M_2(V^{t/2}\ I^{t/2})$
8	0.70	$III^t \rightarrow III^t$
9	0.30	$III^t \rightarrow M_3(I^t)$
10	0.80	$IV^t \rightarrow IV^t$
11	0.20	$IV^t \rightarrow M_4(I^{t/4}\ V^{t/4}\ I^{t/2})$
12	0.15	$V^t \rightarrow IV^{t/2}\ V^{t/2}$
13	0.10	$V^t \rightarrow III^{t/2}\ VI^{t/2}$
14	0.10	$V^t \rightarrow I^{t/4}\ III^{t/4}\ VI^{t/4}\ I^{t/4}$
15	0.10	$V^t \rightarrow V^{t/4}\ VI^{t/4}\ VII^{t/4}\ V^{t/4}$
16	0.10	$V^t \rightarrow V^{t/2}\ VI^{t/2}$
17	0.10	$V^t \rightarrow III^t$
18	0.10	$V^t \rightarrow V^{t/2}\ V^{t/2}$
19	0.05	$V^t \rightarrow VII^{t/2}\ V^{t/2}$
20	0.10	$V^t \rightarrow V^t$
21	0.10	$V^t \rightarrow M_5(I^t)$
22	0.70	$VI^t \rightarrow VI^t$
23	0.30	$VI^t \rightarrow M_6(I^t)$
24	0.40	$VII^t \rightarrow VII^t$
25	0.50	$VII^t \rightarrow I^{t/2}\ III^{t/2}$
26	0.10	$VII^t \rightarrow M_7(I^t)$

Table 4.1: Production rules of a sample PTGG.

Figure 4.2 shows an example of the steps of this algorithm and the resulting parse tree. Because of the presence of identity rules, which can be applied many times, parse trees for generated progressions can often be constructed with fewer rule applications than actually occurred.

4.4.4 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

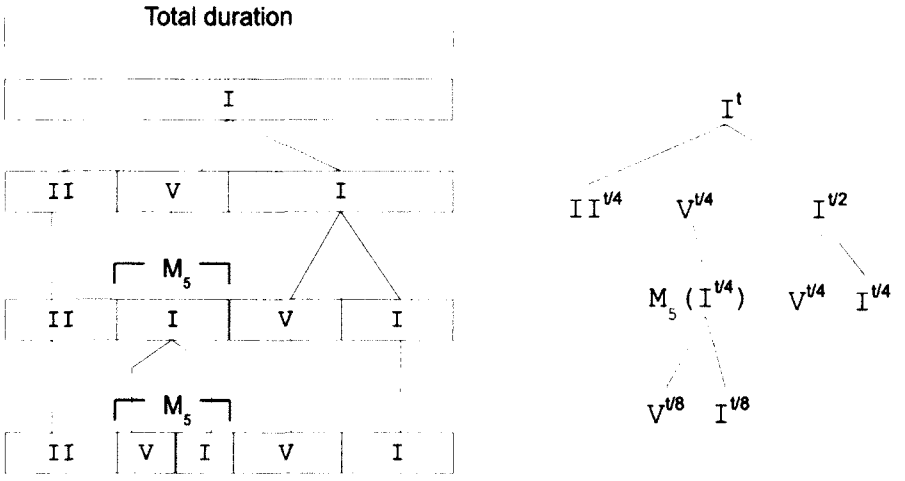


Figure 4.2: Two parse tree representations of the same progression, created by applying the rules 1, 3, and 21 from Table 4.1, along with identity rules 5, 6, and 12. The left representation more closely mirrors the iterative generation algorithm, where each row of chords represents an iteration.

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."

$$\textbf{let } x = I^t \textbf{ in } x\ (M5\ x) \Rightarrow I^t (M5\ I^t) \tag{4.9}$$

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.

$$\textbf{let } x = I^t \textbf{ in } x \ (\textbf{let } x = V^{t'} \textbf{ in } x\ x)\ x \Rightarrow I^t\ V^{t'}\ V^{t'} I^t \tag{4.10}$$

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 chapters. Figure 4.3 shows a small example of this process.

4.5 Modal Context-Sensitivity

The PTGGs discussed so far are context-free for everything except duration of the symbols. However, although the harmonies produced by context-free PTGGs are interesting, they also demonstrate the need for considering mode when applying rules. Here we only consider two modes, major and minor, although the extension to a larger number of modes is also possible within the same framework. There are two possible ways to address the issue of modal context-sensitivity:

1. Increase the alphabet size and allow for major/minor chords. The simplest approach would be to double the alphabet and create modulation rules such as $VI_{major} \rightarrow M_6(I_{minor})$ to indicate that a VI-chord in a major key (which is a minor triad) would need to be replaced by a minor modulated section.

2. Add mode-handling to the monad and allow rules to use conditional logic on the mode.

The monadic implementation lends itself to easy introduction of certain contextual features. The current mode is a type of information that is easily handled in the same way as threading randomness through the computation. This allows for a smaller rule set, since rules like $I^t \rightarrow I^t$ and $V^t \rightarrow M_5(I^t)$ for which mode does not matter do not need to be duplicated for major and minor modes. Instead, only those rules that are prone to making undesirable harmonies in one mode or another need to be modified. This type of implementation does not preclude the level of detail that would be possible in a non-redundant rule set for an alphabet of major and minor Roman numerals, but it allows for simplifications when the sets of rules relevant to each mode and their associated probabilities demonstrate overlap.

It is important to note that modal context-sensitivity implemented at the monadic level is not the same as creating a traditional context-sensitive grammar, where presence of symbols elsewhere in the sequence can influence the selection of rules. All of the rules still have the context-free form of $A \rightarrow XY$ and traditionally context-sensitive rules of the form $AB \rightarrow XYB$ are still illegal. Rather, where we would have two rules with the same left-hand side, $A \rightarrow XY$ and $A \rightarrow X'Y'$, where XY is appropriate in major and $X'Y'$ is appropriate in minor, the mode-handling logic is once again encapsulated in the *rules* (in the same way as handling a minimum duration) rather than being delegated to the applying algorithm.

Table 4.2 shows an example of a modally context-sensitive PTGG. It also includes additional conditional behavior to aid in duration distribution. This rule set contains 26 rules using a monad-level handling of mode.

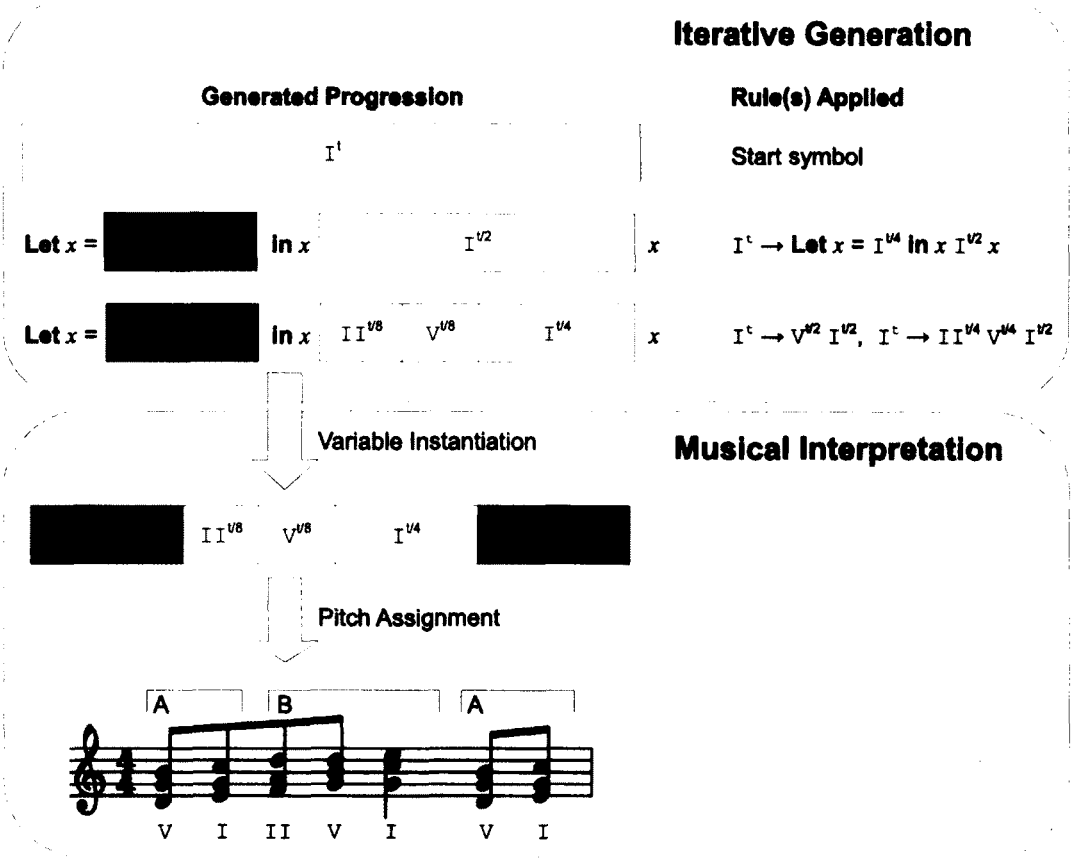


Figure 4.3: Example of the generative process and musical interpretation for *Let* expressions. The pitch assignment step shows only one of many possible outcomes, with the important feature being that chosen pitches adhere to the overall ABA pattern defined by the *Let* expression. Handling of *Let* expressions at the pitch assignment step is discussed in Chapter 5.

The changes to the implementation needed to support modal context-sensitivity are relatively small. Details of the modified implementation are shown in Appendix B.

4.6 Other Alphabets

So far, the chords used in PTGGs have been limited to Roman numerals, yielding a seven chord alphabet. This is, in fact, a very limited musical alphabet, and other, more diverse labeling systems are commonly used. Fortunately, the same framework developed in the previous sections is still applicable to other alphabets of chords, and it can be generalized without requiring a redefinition of *CType* or any invasive changes into the algorithms. Rather, a chord simply can be redefined in a polymorphic way: as "something" with a duration.

```
data Chord a = Chord Dur a
  deriving (Eq, Show)
```

This type change then propagates to to *Term*'s definition.

```
data Term a = NT (Chord a) | S [Term a] | Mod MType (Term a) |
  Let String (Term a) (Term a) | Var String
  deriving (Eq, Show)
```

Changes elsewhere are trivial to accommodate this difference, merely substituting *Term a* for the instances of *Term* in type signatures. The PTGGs shown in Tables 4.1 and 4.2 would produce progressions of type *Term CType*.

This more general definition of the *Chord* type is useful for constructing PTGGs over different harmonic systems, and allows the easy extension of the code to new data types representing different alphabets of chords. In chapter 8, PTGGs are constructed from a modified version of Rohrmeier's grammar for harmony, which introduces new chordal concepts in addition to Roman numerals. For example, in this grammar, a "tonic chord," denoted as T, is a separate and more general entity than a I-chord. The *Chord a* type allows this type

of change easily within the framework. This extension would also allow the use of rules with more harmonic granularity, such as those found in Terry Winograd's work [85], which specify the intervallic structure or inversion of the chord: $I \rightarrow I_3^5$, $I \rightarrow I_4^6$, and so on. More details on this more generalized version of the implementation are contained in Appendix B.

4.7 Other Possible Extensions

Because a PTGG generates abstract structure for music that must be further interpreted, it would be easy to augment the alphabet and allowable sentential forms to handle more musical concepts. One common musical feature is the notion of a variation. This is often denoted using a "prime" notation. For example, AA' would indicate a section A followed by a very similar but not identical version of A. One might notate this with a PTGG as:

$$\textbf{let } x = A \textbf{ in } (x \ \textbf{variation}(x)) \tag{4.11}$$

where **variation** above denotes an operation that must be performed at the musical interpretation level, similarly to let-expressions. However, these sorts of additional features are not currently implemented. Exact repetition in the form of a let-expression is easily implemented in Kulitta because the musical meaning is unambiguous: each instance of a variable must be exactly the same. On the other hand, the idea of musical variation is far less well defined. We usually know a variation of a theme or section when we hear it, but those observations lie in an area between exact repetition and total distinction that is still fuzzy and poorly defined. A formal definition of what constitutes a variation would be needed in order to proceed from abstract to concrete in the generative process.

Num.	Probability	Rule
1	0.187	$I^t \rightarrow$ **if** *major* **then** $II^{t/4}\ V^{t/4}\ I^{t/2}$ **else** $IV^{t/4}\ V^{t/4}\ I^{t/2}$
2	0.187	$I^t \rightarrow I^{t/4}\ IV^{t/4}\ V^{t/4}\ I^{t/4}$
3	0.187	$I^t \rightarrow V^{t/2}\ I^{t/2}$
4	0.187	$I^t \rightarrow$ **if** *major* **then** $I^{t/4}\ II^{t/4}\ V^{t/4}\ I^{t/4}$ **else** $I^{t/4}\ IV^{t/4}\ V^{t/4}\ I^{t/4}$
5	0.252	$I^t \rightarrow$ **if** $t \leqslant h$ **then** I^t **else** $I^{t/2}\ I^{t/2}$
6	0.400	$II^t \rightarrow$ **if** *major* **then** II^t **else** IV^t
7	0.400	$II^t \rightarrow$ **if** *major* **then** (**if** $t > q$ **then** II^t **else** $M_2(I^t)$) **else** $M_2(I^t)$
8	0.200	$II^t \rightarrow$ **if** *major* **then** $VI^{t/2}\ II^{t/2}$ **else** $VI^{t/2}\ IV^{t/2}$)
9	0.900	$III^t \rightarrow III^t$
10	0.100	$III^t \rightarrow M_3(I^t)$
11	0.900	$IV^t \rightarrow IV^t$
12	0.100	$IV^t \rightarrow M_4(I^t)$
13	0.100	$V^t \rightarrow V^t$
14	0.150	$V^t \rightarrow IV^{t/2}\ V^{t/2}$
15	0.100	$V^t \rightarrow III^{t/2}\ VI^{t/2}$
16	0.100	$V^t \rightarrow I^{t/4}\ III^{t/4}\ VI^{t/4}\ V^{t/4}$
17	0.100	$V^t \rightarrow V^{t/4}\ VI^{t/4}\ VII^{t/4}\ V^{t/4}$
18	0.100	$V^t \rightarrow V^{t/2}\ VI^{t/2}$
19	0.100	$V^t \rightarrow III^t$
20	0.050	$V^t \rightarrow V^{t/2}\ V^{t/2}$
21	0.100	$V^t \rightarrow VII^{t/2}\ V^{t/2}$
22	0.100	$V^t \rightarrow M_5(I^t)$
23	0.700	$VI^t \rightarrow VI^t$
24	0.300	$VI^t \rightarrow M_6(I^t)$
25	0.500	$VII^t \rightarrow$ **if** $t > q$ **then** VII^t **else**$M_7(I^t)$
26	0.500	$VII^t \rightarrow I^{t/2}\ III^{t/2}$

Table 4.2: Example of a modally context-sensitive PTGG with conditional behavior on both mode and duration within rules. Note that implementing this rule set without modal context-sensitivity would require an alphabet of 14 Roman numerals (I_{major}, I_{minor}, etc.) and would therefore require twice as many individual rules, many of which would be redundant. The letters h and q denote a durations of a half note and quarter note respectively.

Chapter 5

Constraint Satisfaction

A musical grammar provides a backbone or abstract path that can be transformed into more concrete music using chord spaces. However, there are multiple ways that this can be done, and many of them are likely to sound bad or violate various music theoretic principles. If we wish to generate music in a particular style, there are likely to be various rules or constraints that define that style beyond what can be captured in a chord space, such as specific ways that voices must behave when transitioning between chords. These sorts of musical behaviors can be addressed with the use of a *constraint satisfaction* algorithm when traversing a chord space.

Artificial intelligence systems often involve constraint-satisfaction algorithms[46]. When operating in a discrete environment, a brute-force approach to constraint-satisfaction is depth-first search of all possible outcomes. This approach is also guaranteed to succeed if the constraints are satisfiable. However, many solution spaces are too large for this approach to be realistically usable, requiring the uses of stochastic alternatives instead.

5.1 Musical Constraints

Music theory describes many rules that separate good music from bad, and one style from another. Many of these rules can also be viewed as constraints that the composer must

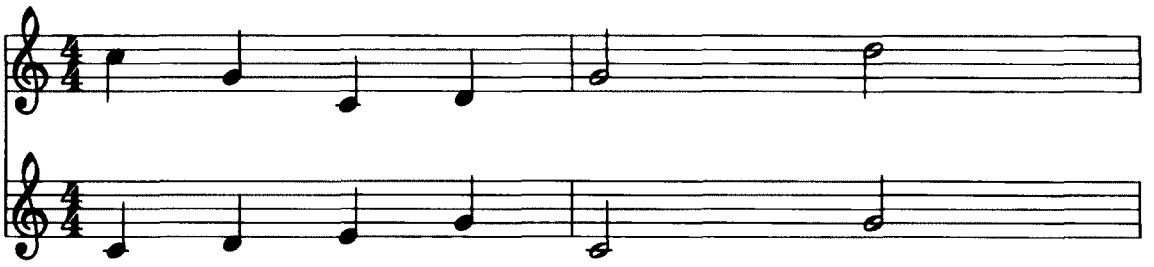


Figure 5.1: An example of undesirable voice-leading behavior. The voices cross twice between beats 2-3 and 4-5, and they exhibit parallel motion (discussed in Section 5.3) in the second measure.

satisfy in order to write good music. One of the difficulties in capturing these rules or constraints in a formal setting is that not all of them are strict or "hard" constraints. Quite often the constraints are softer, and must only be met most of the time in order to produce a satisfactory result.

The term "voice" refers to a musical line that contains only one note at a time. For many instruments, one voice corresponds to one staff on the score[1]. In a chorale, voices are not allowed to cross—in other words, the n^{th} highest voice must play the n^{th} highest note. Figure 5.1 shows an example of voices crossing. This is a hard constraint in the sense that the only acceptable solutions are those completely satisfying the constraint. Similarly, when writing a part for violin, the range of the instrument must be considered and is again a hard constraint. Exact repetition of a phrase would also be a hard constraint. However, other common transformations, such as the formation of a counter subject in a fugue by transposition of the subject, are not so strict, and sometimes small deviations can even be more pleasing than strict adherence to the rule. Because soft constraints are inherently more difficult to formalize, the constraints presented here are hard constraints: a musical feature either does or does not meet the constraints in a Boolean sense. All constraints are presented alongside their Haskell implementations.

Kulitta uses chord spaces as a fundamental part of the music generation process to move

1. Instruments capable of playing many notes at once are an exception to this. For example, score for a piano will often have more than one voice represented on the same staff.

from abstract progressions to concrete music. Some musical constraints are easily captured within the chord space itself, while others are better handled during the path-finding process to traverse the chord space. Kulitta includes three path-finding constraint satisfaction algorithms. Two of these algorithms will only return solutions that satisfy 100% of the imposed constraints. The third is a greedy algorithm that seeks to mostly satisfy the constraints but might break them sometimes—a property that "softens" otherwise hard constraints. All three approaches have merit in different settings.

5.1.1 Predicates

All of the constraints discussed in this chapter are formalized as hard constraints. Most of these constraints can be formalized as a Boolean test, or *predicate*, that returns *True* when a musical value satisfies the constraint(s) and *False* otherwise.

type $Predicate\ a = a \rightarrow Bool$

Predicates can exist over many different types of values. Using the *AbsChord* type defined in Chapter 3, a *Predicate AbsChord* would be a predicate over single chords, a $Predicate\ (AbsChord, AbsChord)$ would be over pairs of chords, and a $Predicate\ [AbsChord]$ would be over chord progressions. Kulitta makes use of several musical constraints of these types.

5.2 Single Chord Constraints

Kulitta uses a number of single-chord constraints to make style-appropriate choices for pitch assignment in chords. The following constraints can all be handled at the single chord level:

- *Voice order.* A simple way to ensure that voices do not cross is to enforce that pitches in all chords are sorted in ascending order.

- *Voice spacing*, or intervallic structure of the chord. Spacing of voices is important; if the voices are too tightly packed, they may difficult to distinguish (for example, tightly packed clusters of low pitches on a piano are often described as sounding "muddy"). Voices can also be spaced too far apart.

- *Doubling*. When a pitch class is played by two voices in different octaves, it is said to be doubled. Some styles have rules for when different pitch classes can be doubled. Sometimes it is appropriate to double the root or fifth of a chord, but not other voices such as the third or seventh.

- *Voice ranges*. Each voice can only utilize a finite range of pitches. This means that the total set of possible chords for a given piece of music is also finite when discrete pitches are used, even if the set is prohibitively large or intractable to explore.

Single-chord constraints are most effectively employed in the construction of the quotient space itself, since single-chord predicates can be viewed as acting on the set of chords before any equivalence relations are applied. Pruning the set of available chords before applying an equivalence relation requires less computation than would be required to first form the quotient space and then filter out chords during path-finding.

We can represent single-chord constraints with predicates on single chords, or type *Predicate AbsChord*. Note that this is a type synonym for $AbsChord \rightarrow Bool$. Expressing the voice ordering constraint above with this type is simple.

$sorted :: Predicate\ AbsChord$

$sorted\ x = x == sort\ x$

One way to constrain voice spacing is to set lower and upper bounds on how far apart the various voices can be. For example, it may be appropriate to allow the two lowest voices more freedom of movement than the two highest voices. This can be done by creating a predicate with an extra argument tailored to a particular style.

$spaced :: [(Int, Int)] \rightarrow Predicate\ AbsChord$

$spaced\ lims\ x = and\ \$$

$zipWith\ (\lambda(l,u)\ diff \rightarrow l \leqslant diff \wedge diff \leqslant u)\ lims\ \$\ zipWith\ subtract\ x\ (tail\ x)$

Another example of a spacing-related predicate would be the notion of what chords can be played with a single hand on a piano when the pianist only has a reach of an octave.

$pianoChord :: Predicate\ AbsChord$

$pianoChord\ x = length\ x \lambda leq\ 5 \wedge maximum\ x - minimum\ x \leqslant 12$

The OPTIC equivalence relations discussed in chapter 3 are a useful tool for defining voice doubling behavior. For example, suppose we are only interested in triads expressed with four voices, which is a common scenario for chorales. We will allow major and minor triads where the root and fifth are doubled, and diminished triads where the root is doubled.

$triads :: [AbsChord]$

$triads = [[0,0,4,7],[0,4,7,7],[0,0,3,7],[0,3,7,7],[0,0,3,6]]$

$doubled :: Predicate\ AbsChord$

$doubled\ x = elem\ (normOP\ x)\ triads$

Voice ranges are best handled in a slightly different way for reasons of efficiency. The functions above are designed for use with a function like *filter* to prune away unwanted chords from a set or list. Chords that are pruned are really wasted computation, so it is best to avoid computing them in the first place when possible. Voice range constraints can be satisfied during generation, such that only satisfactory chords are ever computed. The function, *makeRange*, below incorporates the single-chord constraint of voice ranges to generate a finite set of chords for use in a chord space. The function takes a list of lower and upper bounds for each voice and returns all possible chords within that range.

$makeRange :: [(AbsPitch, AbsPitch)] \rightarrow [AbsChord]$

$makeRange = foldr\ (\lambda(l,u)\ xs \rightarrow [(a:b)\ |\ a \leftarrow [l..u], b \leftarrow xs])\ [[]]$

This yields chords satisfying the voice ranges, but we may wish to apply further predicates. Although some computation will be wasted on discarded chords, it is still better to

waste the computation at the single chord level rather than expend it later when, for example, forming or traversing a quotient space. Consider the following two methods of forming a quotient space using *makeRange* and some additional constraints, f.

$$filter\ f\ (makeRange\ ranges)\ //\ r$$

$$map\ (filter\ f)\ (makeRange\ ranges\ //\ r)$$

The first version will be more efficient than the second, since the second performs comparisons against potentially undesirable chords when forming the quotient space. That extra computation is avoided in the first version.

As a more extensive example, all of the above can be combined to create the OP-space for standard Soprano, Alto, Tenor, and Bass ranges where the voices do not cross, voices are separated by no more than an octave, and voice doubling is controlled.

$$satbOP :: QSpace\ AbsChord$$

$$satbOP = filter\ f\ (makeRange\ ranges)\ //\ opEq\ \textbf{where}$$

$$f\ x = and\ \$\ map\ (\$x)\ [sorted, spaced\ limits, doubled\ triads]$$

$$limits = repeat\ (3,12)$$

$$ranges = [(40,60),(47,67),(52,76),(60,81)]$$

This forms a quotient space with 959 chords grouped into 60 equivalence classes, one for each chord quality in each key, and unwanted chords are pruned away before forming the quotient space. This space handles a quite a few musical constraints, but it cannot capture other constraints of voice-leading behavior, such as avoiding parallel motion. To do that, constraints over pairs of chords must be considered, which involves the use of path finding algorithms.

5.3 Pairwise Constraints

Kulitta makes use of two constraints that are most easily defined over pairs of chords:

- Avoiding parallel motion. Voices move in parallel when they go up or down by the

same number of pitches. Figure 5.1 shows an example of this.

- Voice-leading smoothness. In some styles, smooth voice-leading is desirable and means that the voices should move very little from one chord to the next. In other styles, leaping behavior in one or more voices may be more desirable.

Parallel motion can be easily detected for two chords, $\vec{x}$ and $\vec{y}$, with the same number of voices by checking whether $\vec{x} - \vec{y}$ contains any duplicate values.

```
notParallel :: Predicate (AbsChord, AbsChord)
notParallel (x, y) =
  let diff = zipWith subtract x y
  in nub diff == diff
```

Voice-leading smoothness can be detected similarly. For generality, we will use a sub-predicate for single voices to establish what desirable movement is.

```
voiceLeading :: [Predicate (AbsPitch, AbsPitch)] →
  Predicate (AbsChord, AbsChord)
voiceLeading preds (x, y) = and $ zipWith ($) preds $ zip x y
```

An instance of this constraint would be restricting voices to moving no more than 7 halfsteps between chords.

```
vl7 :: Predicate (AbsChord, AbsChord)
vl7 = voiceLeading (repeat f) where f (a, b) = abs (a - b) ⩽ 7
```

Finally, while voice-crossing is most efficiently implemented by only using chords whos voices are sorted in ascending order, it is possible to define a more general test for voice crossing over pairs of chords. If there exists at least one permutation that sorts the pitches in two chords, then the voices do not cross from one chord to the other.

```
noCrossing :: Predicate (AbsChord,AbsChord)
noCrossing (c1,c2) =
  let sn = permutations [0..length c1 - 1]
      ps1 = filter (λs → p s c1 == sort c1) sn
      ps2 = filter (λs → p s c2 == sort c2) sn
  in ¬$null [p | p ← ps1,elem p ps2]
```

Alternatively, when the chords are known to be sets of pitches (no two voices having duplicate pitches), contour equivalence can also be used.

```
noCrossing2 :: Predicate (AbsChord,AbsChord)
noCrossing2 (x,y) = rank x == rank y
```

Both approaches remove the need to have the voices sorted in ascending order to check for voice-crossing. However, it is less efficient than the single-chord approach discussed previously. For the complexity reasons discussed in the next section, it is always preferable to satisfy constraints within the chord space when it is possible to do so.

5.3.1 Depth-First Search

Finding a chord progression that meets certain constraints is analogous to the satisfiability problem in computer science, which is NP-complete for arbitrary formulas on Boolean variables [32]. Because of this, there is a tractability issue involved in finding candidate solutions that satisfy one or more potentially arbitrary predicates. If there are k possible choices for each of m chords in a progression, there are k^m total possibilities. For a given quotient space S/R and predicate H we clearly need a more efficient method for finding solutions than generating all R-equivalent solutions, storing them, and then looking for cases satisfying H. An algorithm generating H-acceptable solutions must perform more aggressive pruning of the solution space.

As already mentioned, the most efficient way to prune the solution space is to prune the set of available chords. However, there are many constraints that simply can't be captured

at that level. Fortunately, other predicates can also help prune the solution space. The algorithm below uses pairwise predicates during a depth-first search through the solution space. While this general strategy does not change the complexity class of the problem, it avoids computing and storing unnecessary progressions.

Algorithm 4. $pairProg(R, S, h_{pair}, [\vec{x}_1, ..., \vec{x}_m]) =$

1. *If* $m = 1$, *return* $E(\vec{x}_1, S/R) = \{\vec{z} \mid \vec{z} \in S \wedge \vec{z} \sim_R \vec{x}\}$, *otherwise continue.*
2. *Let* $Y = pairProg(R, S, h_{pair}, [\vec{x}_2, ..., \vec{x}_m])$
3. *Return* $\{[\vec{y}_1, \vec{y}_2, ..., \vec{y}_m] \mid \vec{y}_1 \in E(\vec{x}_1, S/R),\ [\vec{y}_2, ..., \vec{y}_m] \in Y,\ h_{pair}(\vec{y}_1, \vec{y}_2)\}$.

This is leads to a concise Haskell definition:

```
pairProg :: (Eq a, Show a) ⇒ QSpace a → EqRel a → Predicate (a,a) →
            [a] → [[a]]
pairProg qs r h [] = [[]]
pairProg qs r h (x:xs) =
  let newSolns = [(y:ys) | y ← eqClass qs r x, ys ← pairProg qs r h xs,
                           h (y, head ys)]
  in if ¬ $ null newSolns then newSolns else error "No solutions!"
```

Even when solutions are filtered using predicates, the work involved in traversing the entire set of solutions to locate desirable ones and even the number of desirable solutions can be intractable in situations involving many chords, many voices, and/or large ranges for the voices. Finding the very first solution using *pairProg* is often easy with the types of predicates discussed so far (assuming the chord spaces used are musically reasonable), but what if we want a deeper solution, or a solution chosen uniformly at random?

5.3.2 Stochastic Search

As mentioned in the beginning of the chapter, rules in music are not always strict, and so it may be sufficient to find a solution that mostly satisfies a set of predicates, even if some parts of the solution violate the predicates. Algorithm 5, *greedyChord*, illustrates the process of picking the next chord, and Algorithm 6, *greedyProg*, describes the process of generating the entire progression. While *greedyProg* is not guaranteed to find a solution satisfying a progression predicate, it will attempt to satisfy a pairwise predicate when choosing each chord. *greedyProg*'s success rate in satisfying the constraints at each step will be proportional to the abundance of constraint-satisfying solutions in the solution space.

Algorithm 5. *Let $\vec{x}_i$ be the chord for which we wish to find a new R-equivalent member of $S \subset \mathbb{Z}^n$, $\vec{y}_i$. Let $\vec{y}_{i-1}$ be the previously chosen chord, choose(S') be a function to stochastically select an element from a set, and $f(\vec{x}_i, \vec{y}_{i-1}, E(\vec{x}_i, S/R))$ be a fall-back method for choosing $\vec{y}_i$.*

$greedyChord(\vec{x}_i, \vec{y}_{i-1}, S, R, H_{pair}, f) =$

Let $S_H = \{\vec{y} \in E(\vec{x}_i, S/R) \mid H_{pair}(\vec{y}_{i-1}, \vec{y})\}$

If $S_H = \emptyset$ *then return* $\vec{y}_i = f(\vec{x}_i, \vec{y}_{i-1}, E(\vec{x}_i, S/R))$. *Otherwise, return* $\vec{y}_i = choose(S_H)$.

Algorithm 6. $greedyProg([\vec{x}_1, ..., \vec{x}_m], S, R, H_{pair}, f) = [\vec{y}_1, ..., \vec{y}_m]$, *where*

$$\vec{y}_i = \begin{cases} choose(E(\vec{x}_1, S/R)) & for\ i = 1 \\ greedyChord(\vec{x}_i, \vec{y}_{i-1}, S, R, H_{pair}, f) & otherwise \end{cases}$$

The main advantage to this approach is that the solution space is maximally pruned at each step. This allows the algorithm to operate on inputs that would cause tractability problems for *pairProg*. The downside is that *greedyProg* is not guaranteed to find predicate-

satisfying solutions. It is possible to find a partial solution with no subsequent choices that satisfy the supplied predicate or search strategy. In such a situation, there are three options: (1) fail and return an error message, (2) backtrack to try to find a better solution (analogous to lazy evaluation of *pairProg*), or (3) try another predicate. For our implementation, we chose option 3: a fall-back method for choosing a next chord is therefore required if we wish to ensure that *greedyProg* produces a solution. In practice, it may be sufficient to have a result that mostly satisfies a predicate even if some chord transitions do not. Since this greedy approach must make progress at each step, it presents a more tractable option for larger-scale composition problems.

Using a fixed input progression, a predicate p_1 that forbids voice-crossing, parallel motion, and leaps greater than 7 halfsteps, the *greedyProg* algorithm was tested to estimate its success rate [62]. This input progression used, shown in Figure 5.2, demonstrates some of the tractability issues associated with the general problem of rewriting chord progressions. There are 11 chords in the input progression, and, within the range $[36,57]^3$, there are 24 *OP*-equivalent ways to choose each of the first, second, and third chords and 48 ways to choose each of the remaining eight chords. This gives a total of $24^3 \times 48^8$ possible *OP*-equivalent solutions. Of those solutions, we used *pairProg* to determine that 901728 of them satisfied the constraints within the range $[36,57]^3$.

greedyProg was run for 10000 trials under these conditions to find *OP*-equivalent solutions to the input progression in Figure 5.2 within the range $[36,57]^3$. We used a predicate, p_2, that restricts voice-movement to 7 halfsteps as a fall-back method for *greedyProg*. Under these conditions, 8165 of *greedyProg*'s returned progressions satisfied the target predicate. 7406 of those p_1-satisfying progressions were unique solutions. Of the 1835 returned progressions that violated p_1, an average of only 1.1 chord transitions per progression violated p_1, with the highest number of p_1-violating transitions being 2. This indicates that *greedyProg* performed relatively well in this experiment. However, the algorithm's performance will be directly affected by the range of the voices and specific

predicate used. Decreasing the range of each voice, for example, would increase the odds that *greedyProg* would become stuck and rely on its fall-back function.

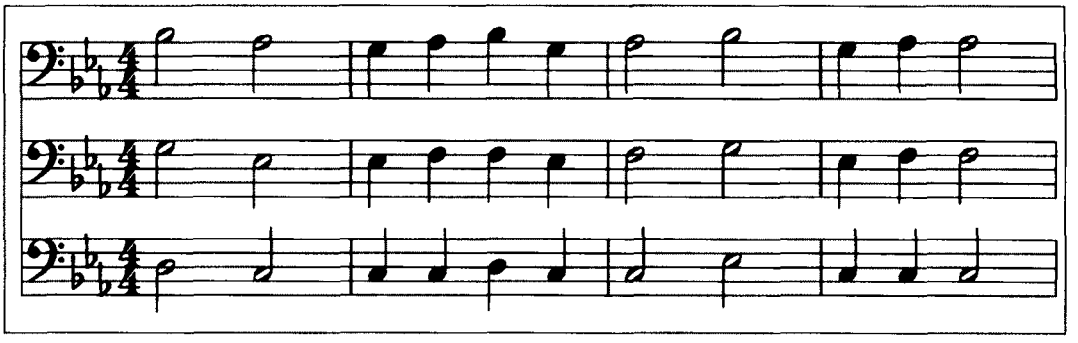


Figure 5.2: A simple chord progression for three voices used for testing the performance of the *greedyProg* algorithm.

5.3.3 Delegation of Equivalence Class Lookup

One notable limitation of both *pairProg* and *greedyProg* is the direct interaction with the quotient space. An alternative approach is to delegate the lookup of equivalence classes to some other algorithm. This useful in situations where single chord constraints exist that are position-specific. For example, perhaps we would like the bass to only play the root of a chord on the very last chord in a passage. This approach is formalized in Algorithms 7, 8, and 9.

Algorithm 7. $pairProg'(h_{pair}, [E_1, ..., Em]) =$

1. *If* $m = 1$, *return* E_1, *otherwise continue.*
2. *Let* $Y = pairProg'(h_{pair}, [e_2, ..., e_m])$
3. *Return* $\{[\vec{y}_1, \vec{y}_2, ..., \vec{y}_m] \mid \vec{y}_1 \in E_1, [\vec{y}_2, ..., \vec{y}_m] \in Y, h_{pair}(\vec{y}_1, \vec{y}_2)\}$.

Algorithm 8. *Let E_i be an equivalence class of chords, $\vec{y}_{i-1}$ be the previously chosen chord in the sequence, choose(S) be a function to stochastically select an element from a set, and $f(E_i, \vec{y}_{i-1})$ be a fall-back method for choosing $\vec{y}_i$ if no solutions exist.*

$$greedyChord'(E_i, \vec{y}_{i-1}, H_{pair}, f) =$$

Let $S_H = \{\vec{y} \in E_i \mid H_{pair}(\vec{y}_{i-1}, \vec{y})\}$

If $S_H = \emptyset$ *then return* $\vec{y}_i = f(E_i, \vec{y}_{i-1})$. *Otherwise, return* $\vec{y}_i = choose(S_H)$.

Algorithm 9. $greedyProg'(E_1, \ldots, E_m, H_{pair}, f) = [\vec{y}_1, \ldots, \vec{y}_m]$, *where*

$$\vec{y}_i = \begin{cases} choose(E) & for\ i = 1 \\ greedyChord'(E, \vec{y}_{i-1}, H_{pair}, f) & otherwise \end{cases}$$

5.4 Repetition

The **let** expressions in PTGGs 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:

$$\textbf{let}\ x = (\textbf{let}\ y = \mathrm{V}^{t1}\ \mathrm{I}^{t2}\ \textbf{in}\ y\ y)\ \textbf{in}\ x\ \mathrm{IV}^{t3}\ \mathrm{I}^{t4}\ x \tag{5.1}$$

$$\mathrm{V}^{t1}\ \mathrm{I}^{t2}\ \mathrm{V}^{t1}\ \mathrm{I}^{t2}\ \mathrm{IV}^{t3}\ \mathrm{I}^{t4}\ \mathrm{V}^{t1}\ \mathrm{I}^{t2}\ \mathrm{V}^{t1}\ \mathrm{I}^{t2} \tag{5.2}$$

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 now, we will only consider *exact* repetition of a phrase—that is to say that both phrases appear in the same key and should, therefore, be assigned identical patterns of pitches. This is the only type of repetition possible in the example rules sets shown in Tables 4.1 and 4.2. However, repetition is not always this simple in music, and some more complicated types of repetition are actually allowed in the sentential forms for PTGGs. For example:

$$\textbf{let } x = I^t \textbf{ in } x\ (M_5\ x)\ x \Rightarrow I^t\ (M_5\ I^t)\ I^t \tag{5.3}$$

It is not entirely clear what the "correct" interpretation of this should be beyond the level of Roman numerals. It is common to have repetition with transposition: playing the same overall pattern, but offset up or down by some number of pitches from the original. Under this model, if x is interpreted as $\langle 0,4,7\rangle$ (C-E-G), then $(M_5\ x)$ should be T-equivalent to $\langle 0,4,7\rangle$ to retain the same intervallic structure and O-equivalent to $< 7,11,2 >$ (G-B-D) to have the correct pitch classes. However, finding a satisfying pitch assignment is more complicated because of the modulation: a pitch assignment for the un-modulated x may satisfy the outer chords but not be transposable due to the ranges of the voices. An extreme example would be a collection of voices with only a single-octave range from 0 to 11. Under these conditions, it is actually *impossible* to simply slide the pitches for x in Equation 5.3 up or down enough to produce the same intervallic structure in the modulated key. Because of these issues and the resulting ambiguity of how to handle them, our **let**-satisfying algorithm is only equipped to handle un-modulated, exact repetition of phrases.

For a quotient space q and equivalence relation r, simply calling $head \circ eqRel\ q\ r$ 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 the following progression, which is six chords long.

$$\textbf{let } y = (\textbf{let } x = A \textbf{ in } xxB) \textbf{ in } yy \tag{5.4}$$

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: $e_1 = \{a,b\}$ for A's equivalence class and $e_2 = \{c,d\}$ for B's equivalence class, where a, b, c, and d are chords. The pattern of equivalence classes for the progression above would be e_1, e_1, e_2, e_1, e_1, e_2, and 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 leaves 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.

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 indices

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):

$$\textbf{let } x = (\textbf{let } y = \text{II } \textbf{in } y\ y\ \text{V I}) \textbf{ in } x\ x \tag{5.5}$$

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 $[[(0,0),(1,1)],[(0,3),(4,7)]] :: \textit{Constraints}$. 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. $[k_1,...,k_n]$::*Constraints* is well-formed if it has no overlapping index lists (although nested ranges are permitted) and there is no index range in k_i that is further constrained by k_j for $j > i$. 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. When there are no overlapping index ranges, well-formedness can be ensured by calling $\textit{sort} \circ (\textit{map sort})$ on the *Constraint*.

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 (or cardinality) 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 $k :: \mathit{Constraints}$, the following function finds indices that can be incremented to traverse the solution space while satisfying **let** constraints.

$$
\begin{aligned}
&\mathit{freeInds} :: \mathit{Int} \to \mathit{Constraints} \to [\mathit{Index}] \\
&\mathit{freeInds}\ n\ k = \\
&\quad \textbf{let}\ k' = \mathit{map}\ (\mathit{map}\ (\lambda(i,j) \to [i..j]))\ k \\
&\qquad h\ = \mathit{nub}\ \$\ \mathit{concatMap}\ \mathit{head}\ k' \\
&\qquad t\ = \mathit{nub}\ \$\ \mathit{concat}\ \$\ \mathit{concatMap}\ \mathit{tail}\ k' \\
&\qquad t'\ = [\mathit{maximum}\ (\mathit{concat}\ \$\ \mathit{concat}\ k')..n-1] \\
&\quad \textbf{in}\ \mathit{filter}\ (\neg \circ (\in t))\ h +\!\!+ t'
\end{aligned}
$$

For $x = \mathit{freeInds}\ n\ k$, 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*).

$$incr :: [(Bool, Index, Int)] \to [Index]$$

$$incr\ [\,] = error\ \texttt{"No more solutions."}$$

$$incr\ ((b,i,l) : xs) = \textbf{let}\ is = map\ (\lambda(x,y,z) \to y)\ xs\ \textbf{in}$$

$$\quad \textbf{if}\ b\ \textbf{then if}\ i \geqslant l-1\ \textbf{then}\ 0 : incr\ xs\ \textbf{else}\ i+1 : is$$

$$\quad \textbf{else}\ i : incr\ xs$$

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.

$$applyCons :: [Index] \to [(Index, Index)] \to [Index]$$

$$applyCons\ inds\ [\,] = inds$$

$$applyCons\ inds\ ((i,j) : t) = foldl\ (f\ val)\ inds\ (map\ fst\ t)\ \textbf{where}$$

$$\quad val \quad = (take\ (j-i+1)\ \$\ drop\ i\ inds)$$

$$\quad f\ val\ src\ i = take\ i\ src +\!\!+ val +\!\!+$$

$$\qquad drop\ (i + length\ val)\ src$$

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.

$$findNext :: Constraints \to [Index] \to [Int] \to [Index]$$

$$findNext\ k\ is\ lens =$$

$$\quad \textbf{let}\ bs = map\ (\in freeInds\ n\ k)\ [0\,..\,length\ is - 1]$$

$$\qquad xs = zip3\ bs\ (is)\ (lens)$$

$$\quad \textbf{in}\ foldl\ applyCons\ (incr\ xs)\ k$$

As already mentioned, the very first progression (indices $[0,0,\ldots,0]$) 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).

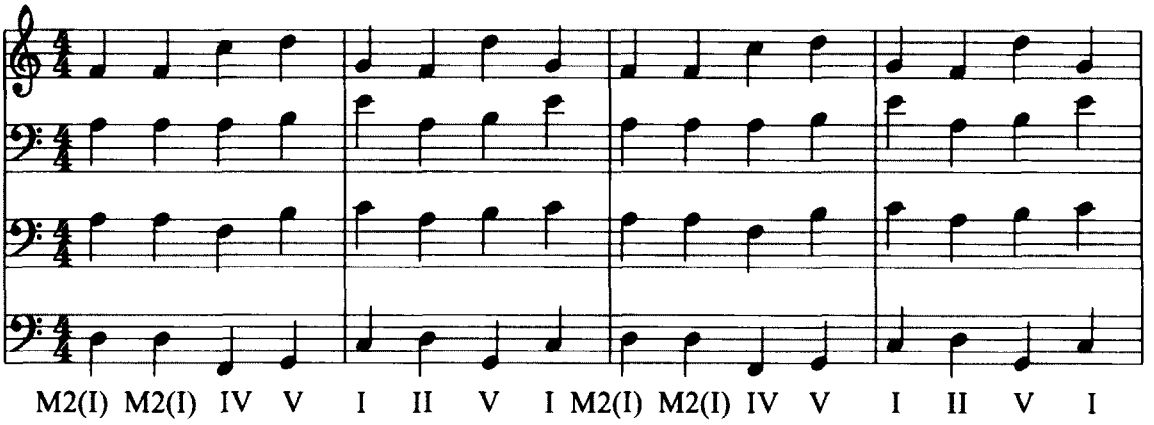


Figure 5.3: A chord progression generated from the expression: **let** $x = ($(**let** $x = (M_2\ I^q)$ **in** $x\ x)\ IV^q\ V^q\ I^q\ II^q\ V^q\ I^q)$ **in** $x\ x$. The chords were interpreted in the key of C-major using OPC-space for voices with the ranges [40,56], [50,62], [55,70], and [60,78] respectively. Note that $M_2(I)$ is the same as an unmodulated *II* chord in this case since the chords did not undergo further generation.

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.

5.4.1 Greedy Algorithm for *Let* Expressions

The *greedyProg* algorithm can be easily modified to handle *Let* expressions. Using a similar approach to that of the previous section, chords can be greedily assigned to free indices and then copied over to constrained indices. This approach will satisfy *Let* expressions exactly but is likely to violate any other pairwise or larger-spanning constraints at junctions between variables in *Let* expressions.

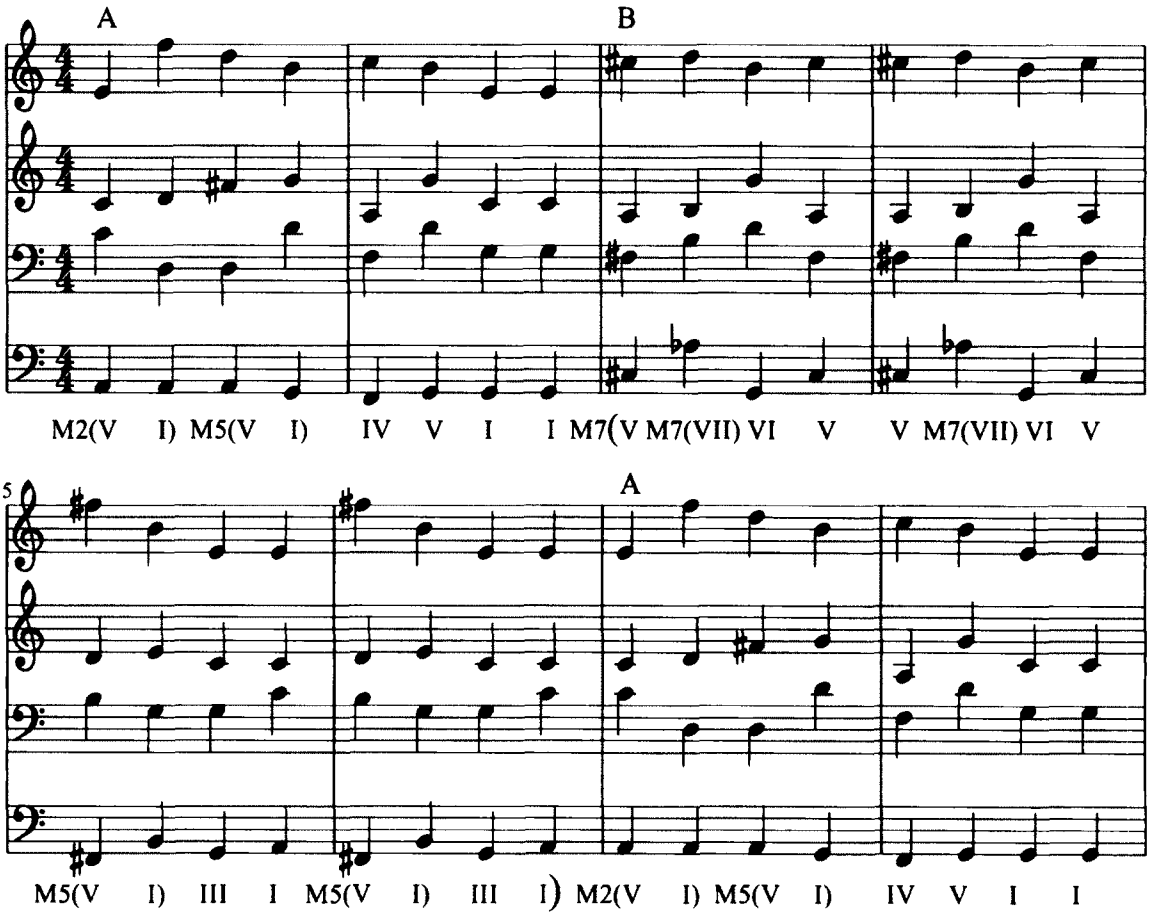


Figure 5.4: A longer example generated with the same chord spaces and C-major key as in 5.3 in C-major. It demonstrates an overall ABA format (the start of each section is labeled above the staff) and includes nested modulated sections.

```
greedyLet :: (Eq a, Show a) ⇒ Predicate (a,a) → Fallback a → Constraints →
                [EqClass a] → StdGen → [a]
greedyLet p f k es g =
  let n = length es
      cs = greedyProg' p f g es
      consPat = foldl applyCons [0..n-1] (sort k)
  in map (cs!!) consPat
greedyLetT :: QSpace AbsChord → EqRel AbsChord →
                Predicate (AbsChord,AbsChord) → Fallback AbsChord →
                Constraints → [TChord] → StdGen → [TChord]
greedyLetT q r p f k cs g =
  let justCs = map tnP cs
      es = map (eqClass q r) justCs
      justCs' = greedyLet p f k es g
  in zipWith newP cs justCs'
```

5.5 The Problem of Novelty

One problem with searching through organized collections of musical features is obtaining a range of reasonably different possible interpretations while adhering to various rules. 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. Although this isn't necessarily a problem from a very literal constraint-satisfaction standpoint, it poses a problem for human evaluation of the range of results.

When evaluating the performance of a system, it is useful to hear more than one result from a set of starting conditions. In Kulitta's case, if for some reason we don't like the first solution or simply want to hear what else the system can do under the same conditions, it

is rather unsatisfying to hear one progression after another that only differ by a few notes. When solutions are abundant, finding significantly different solutions can actually become more difficult. For easily satisfied constraints such as *vl7*, the first several solutions will likely only differ by one chord. What we really want to hear to evaluate the system's performance are solutions that are mostly different rather than nearly identical.

Both *pairProg* (pruned depth-first search) and *findNext* (depth-first search for **let** constraints) are potentially impacted by this diversity problem. To more effectively explore a range of solutions, random paths would be more likely to capture diversity. The *greedyProg* algorithm avoids the diversity problem by being stochastic. When solutions are sparse, *greedyProg* may also end up bending the rules more than might be desired in order to produce a solution.

One possible answer to the problem of creating novelty for *pairProg* and *findNext* is to randomize the chord space before it is searched. This can be implemented by finding a random permutation of the set of chords used in the chord space and then grouping the chords according to an equivalence relation. For *findNext*, one could even go a step further and randomize the individual equivalence classes supplied to the algorithm. This would prevent an unconstrained *I* chord from always being the same as another *I* chord constrained by a **let** expression.

The first solution found by a depth-first search of a chord space whose elements have been randomized in different ways 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 is 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.

Chapter 6

Generating Music

Probabilistic Temporal Graph Grammars can be used with Chord Spaces to bridge the gap between abstract progressions and performable musical scores. Kulitta uses the following generative workflow. First, Kulitta stochastically generates an abstract chord progression with a PTGG. Each chord in this progression is then mapped to a point in the representative subset for a chord space. Next, Kulitta uses a path-finding algorithm to map this representative subset path to a more diverse path through the larger chord space. This may be done more than once in succession with different chord spaces to alter the style while moving closer towards concrete, performable chords. Finally, Kulitta adds more complex melodic and rhythmic elements to the chord progression, yielding a complete musical score. A visual representation of this workflow can be seen in Figure 6.1.

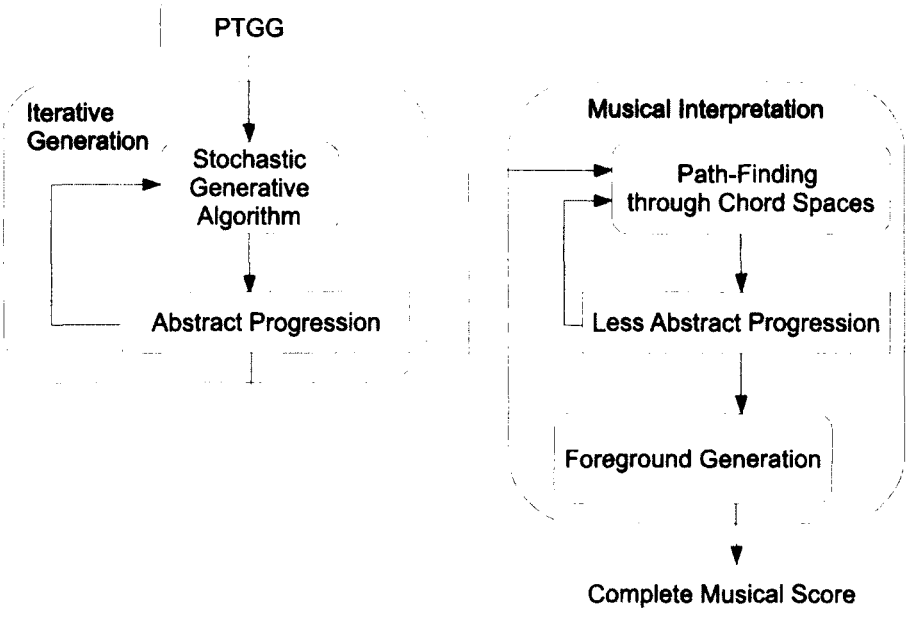


Figure 6.1: Generative workflow in Kulitta: a PTGG is used to generate an abstract progression, which is then iteratively turned into a concrete progression and finally processed by a style-specific foreground algorithm to yield a complete musical score.

6.1 A Simple Example

Step	Progression	Rules Applied (Left to Right)
1	I^w	Start symbol
2	$II^q\ V^q\ I^h$	$I^t \to II^{t/4}\ V^{t/4}\ I^{t/2}$
3	$II^q\ M_5(I^q)\ I^h$	$II^t \to II^t,\ V^t \to M_5(I^t),\ I^t \to I^t$
4	$II^q\ M_5(V^e\ I^e)\ I^h$	$II^t \to II^t,\ I^t \to V^{t/2}\ I^{t/2},\ I^t \to I^t$

Table 6.1: Generating a short chord progression with a PTGG where w, h, q, and e indicate the durations of a whole note, half note, quarter note, and eighth note respectively.

To illustrate the role of chord spaces in our system's generative process from start to finish, consider the series of productions in Table 6.1. 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:

$$\langle D,F,A\rangle\ \langle D,F\#,A\rangle\ \langle D,G,B\rangle\ \langle C,E,G\rangle \tag{6.1}$$

$$\langle 2,5,9\rangle\ \langle 2,6,9\rangle\ \langle 2,7,11\rangle\ \langle 0,4,7\rangle \tag{6.2}$$

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:

$$\langle 50,57,62,65\rangle\ \langle 45,54,62,69\rangle\ \langle 50,59,67,71\rangle\ \langle 48,55,64,76\rangle \tag{6.3}$$

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

$$\langle 0,4,5,9\rangle\ \langle 0,2,6,9\rangle\ \langle 2,6,7,11\rangle\ \langle 0,4,7,11\rangle \tag{6.4}$$

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

$$\langle 52,57,65,72\rangle\ \langle 54,57,60,74\rangle\ \langle 50,55,59,66\rangle\ \langle 48,52,67,71\rangle \tag{6.5}$$

A score representation of the mappings above can be seen in Figure 6.2. 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.

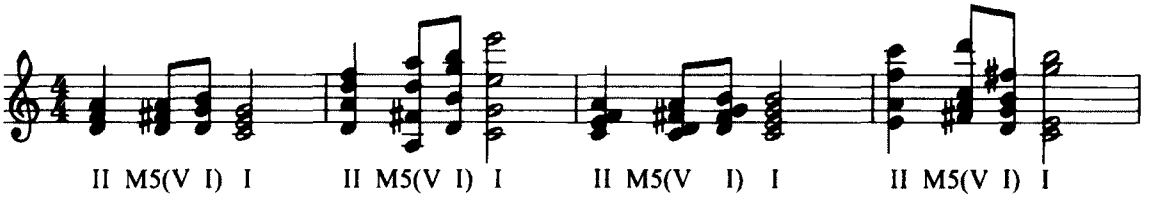


Figure 6.2: A musical score representation of the example mappings of the progression detailed in section 6.1. Each measure contains a different mapping of the same example. From left to right, they are: block trichords (Equation 6.2 transposed up by several octaves), an OPC-space mapping of those trichords (Equation 6.3), block jazz chords from mode space (Equation 6.4), and those jazz chords mapped through OPC-space (Equation 6.5).

6.2 Generating Complete Music

The examples shown so far have been chord progressions. Performed as is, they will sound stiff and simplistic. One of the things that accounts for this is the lack of *nonchordal* tones in the score. A *chordal* tone is one that is a member of the triad representing the Roman numeral. For I in C-major, chordal tones would have the pitch classes C, E, and G, and other pitch classes like F would be nonchordal. Adding nonchordal tones has the potential to both add richness and melodic patterns, but it must be done carefully, for it is very easy to also create a bizarre-sounding mess out of an otherwise pleasing chord progression by using nonchordal tones in the wrong way.

In Schenkerian analysis, music is analyzed at three different levels of abstraction: the *background*, *middleground*, and *foreground*. The background is the most abstract, typically reduced to a $V-I$ or $I-V-I$ pattern [73, 74], and the foreground is the musical score. To move from the concrete score to more abstract, underlying features, *foreground elements* are stripped away from the music. These include many of the "faster" notes and ornamentations in classical music, and they are often non-chordal tones—meaning that they are not part of the triads representing the harmony of the piece. The middleground includes representations with intermediate levels of musical detail between the background and foreground.

Using PTGGs and chord spaces to generate chord progressions creates a musical mid-

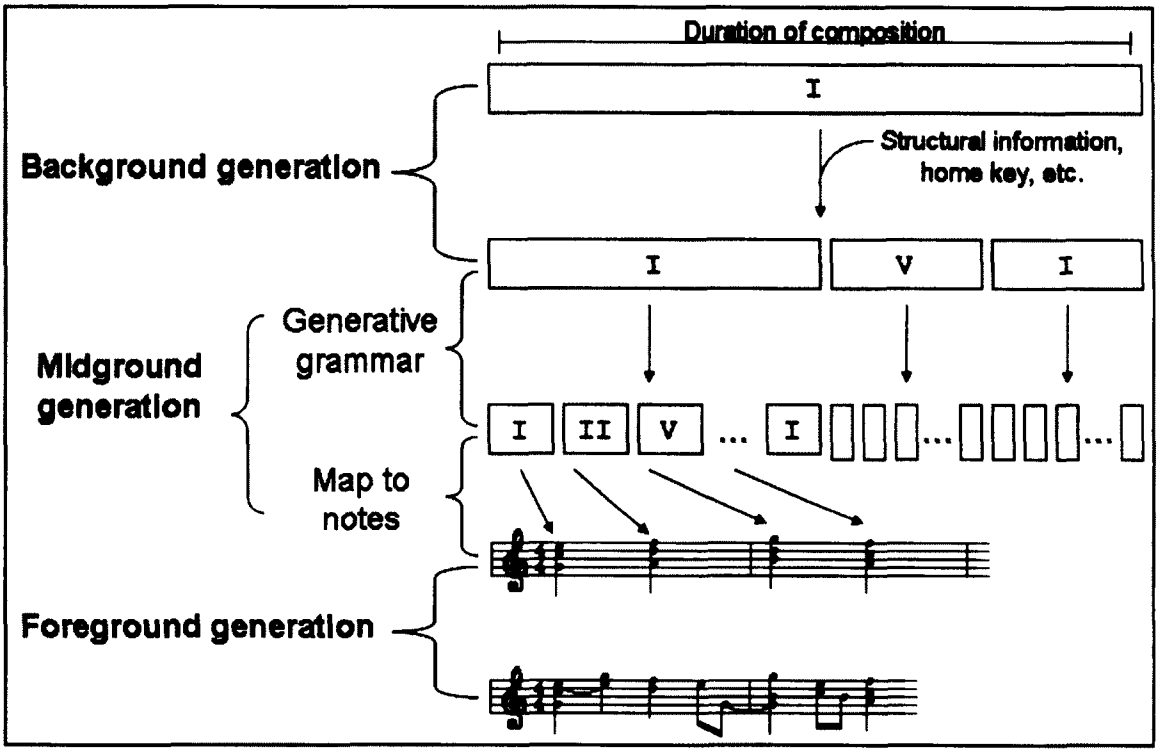


Figure 6.3: Graphical representation of Kulitta's generative process showing the background, middleground, and foreground levels of abstraction.

dleground as shown in Figure 6.3. How to make the transition from middleground to foreground depends on the style of music, since the definitions of desirable rhythmic and melodic features can differ greatly from one genre to another. Here we present methods for creating foregrounds in two styles: chorales and jazz.

6.2.1 Classical Foreground

A PTGG generates structure at the middleground level. While the first few rule applications may result in a structure that still more closely resembles a background in Schenkerian analysis, by the time a 4-measure phrase has been expanded to include mostly quarter notes, it will contain middleground structure: the harmonic backbone onto which foreground structure can be added. In chorales in the style of J.S. Bach, the harmonic backbone is quite prominent and foreground features are relatively simple: notes added between chords, slight rhythmic offset of chordal tones, etc.

For two chordal notes in sequence with pitches i and j, there are several types of new, foreground notes that can be added between them. Examples include:

- Passing tones: pitches between i and j, creating a line in one direction (either all ascending or all descending).
- Neighboring tones: nearby pitches that are higher than the largest of i or j or lower than the smallest of i or j. Neighboring tones are generally close to i or j, such as one or two halfsteps away.
- Anticipations: another instance of pitch j placed between the two chordal notes.
- Suspensions: pitch i is carried over into some of j's "temporal space" (or even all of it such that j never appears). If i and j are the same, then the notes are merged into one that spans the duration of both.

These types of foreground notes generally appear on pitches that are within the current key's scale. In the key of C-major, (F,5) would be an acceptable passing tone between (E,5) and (G,5), but (F#,5) would usually not, since F# is not a member of the C-major scale.

Kulitta's classical foreground algorithm introduces some of these features with four operations over adjacent pairs of chordal tones. The definitions are close to but do not exactly match those given above.

1. $passing(l,t_1,t_2)$ = stochastically add note that is temporally between t_1 and t_2, has a pitch between $min(t_1,t_2)$ and $max(t_1,t_2)$, within the current scale, and no more than l halfsteps away from either t_1 or t_2.
2. $neighboring(l,t_1,t_2)$ = stochastically add a note that is temporally between t_1 and t_2, outside the interval formed by $[min(t_1,t_2), max(t_1,t_2)]$, within the current scale, and no more than l halfsteps away from either t_1 or t_2.
3. $anticipation(t_1,t_2)$ = insert a note at t_2's pitch between the onset of t_1 and t_2.

4. *repetition*(t_1, t_2) = repeat t_1's pitch once before t_2.

5. *nothing*(t_1, t_2) = do nothing to the pair of notes.

These operations are applied on pairs of chordal notes from left to right over each voice independently. For situations where two adjacent notes are from chords in different keys, acceptable scale tones for the *passing* and *neighboring* operations are taken from the intersection of the two sets of pitch classes. If no suitable pitches exist, then the original two notes are left unchanged and *nothing* is the default operation.

When adding new notes between two others, one of two possible rhythmic modifications takes place. If notes t_1 and t_2 have durations d_1 and d_2 respectively, the possibilities for inserting a new note, t_3 with duration d_3 are as follows:

1. Set $d_3 = d_1/2$ and set t_1's new duration to be $d_1/2$.

2. If $d_1 \geqslant q$, then set $d_3 = e$ and set t_1's new duration to be $d_1 - e$, otherwise resort to the previous option.

The Haskell implementation of this approach for generating chorale foregrounds can be found in Appendix B. An example of the results of this algorithm can be seen in Figures 6.4 and 6.5. Figure 6.4 presents a homophonic (all voices moving at the same time) chord progression, and Figure 6.5 shows this progression with foreground elements added by the methods described so far.

This approach to generating a classical foreground does not take **let** constraints into consideration. To achieve exact repetition for a pattern like **let** $x = A$ **in** $x\ x$, phrase A would have to be interpreted through chord spaces and have a foreground added before expanding the **let** expression. However, allowing different foreground interpretation of each instance of a variable is not entirely a bad thing, and actually has an interesting side effect: the creation of variations. Given a homophonic progression created by interpreting **let** $x = A$ **in** $x\ x$ through some chord space, the foreground may differ between instances of

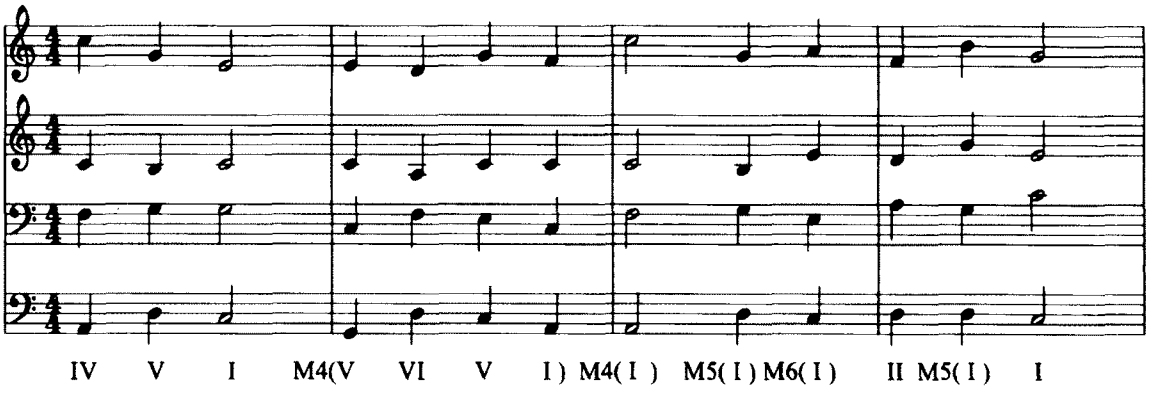


Figure 6.4: An example of a phrase generated by Kulitta without a foreground. It consists of just chordal tones and is homophonic (all voices moving at the same time).

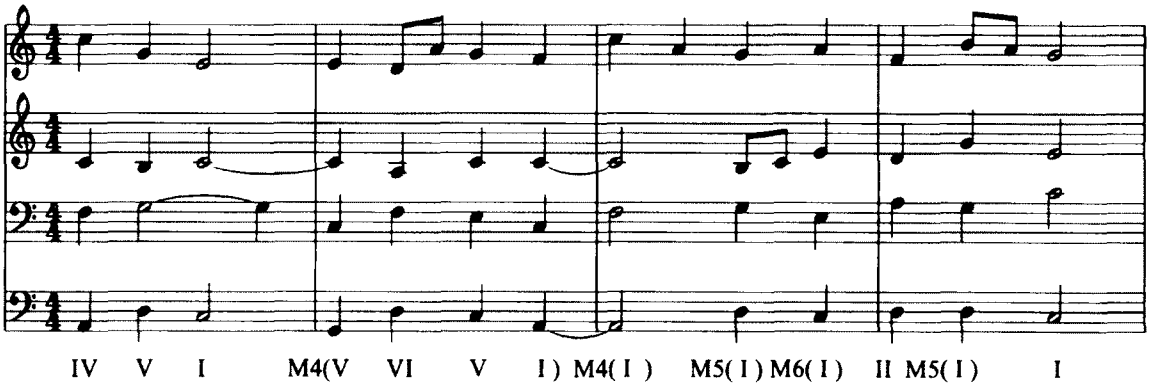


Figure 6.5: The phrase from Figure 6.4 with foreground elements added stochastically.

x. Because the harmonic backbone will be the same, the phrases will sound very similar, as though the second part of the phrase is a variation of the first. This is illustrated in Figures 6.6 and 6.7.

6.2.2 Jazz Foregrounds

Jazz is fundamentally different from the chorale-like style discussed in the previous section in that its harmonies are based around seventh chords, which are triads containing the root, third, fifth, seventh, and sometimes various additions and substitutions such as the second (sometimes referred to as the ninth) [49]. Jazz, therefore, requires working with at least 4 voices that can all be different pitch classes.

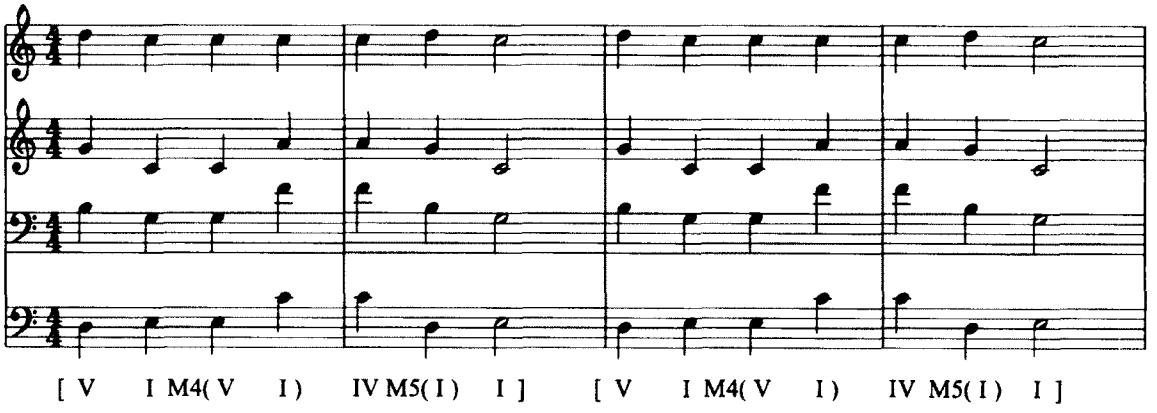


Figure 6.6: An example of a phrase generated by Kulitta without a foreground. It consists of just chordal tones and is homophonic (all voices moving at the same time). Bracketed sections indicate repeats formed from a *Let* expression.

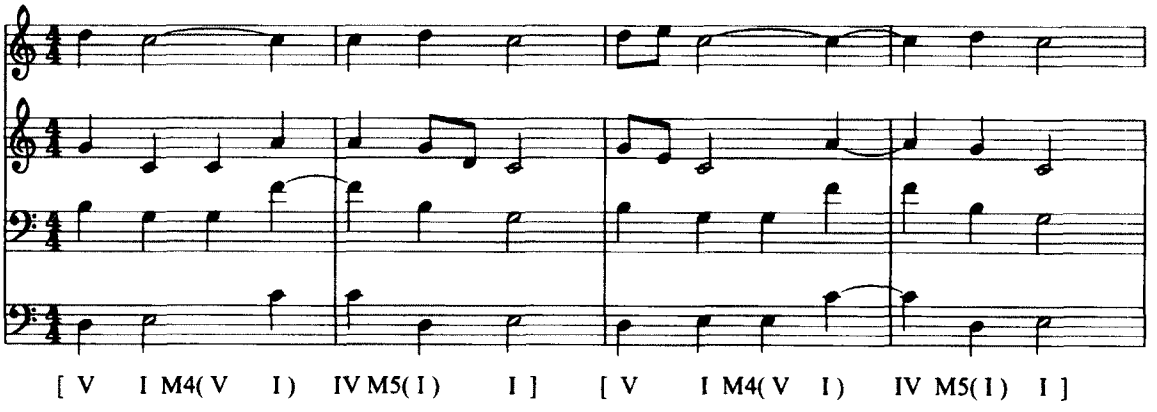


Figure 6.7: The phrase from Figure 6.4 with foreground elements added stochastically. Bracketed sections indicate repeats formed from a *Let* expression. Note that the foreground was applied without knowledge of the *Let* expression and therefore creates a variation rather than an exact repeat.

One of the more interesting aspects of Kulitta's modular design is that a PTGG suitable for generating chorale-like music with a classical foreground algorithm can also create very jazzy harmonies. In fact, the same middlegound can produce *both* a chorale and jazz harmonies, as seen in Figure 6.2.

Jazz foregrounds are somewhat more difficult to create than those for chorales, not just because of the extra voices they require compared to Kulitta's chorale foreground algorithm, but also because the style is broader and less formalized than is the case for chorales. Jazz compositions are also usually specified in less detail than their classical cousins, leaving a lot of decisions up to the performer. The algorithms demonstrated here are very simple and close to the rudimentary level of fake book notation than the intricate level of a live jazz performance, and there are currently no features addressed that a beginner-level performer couldn't manage—in other words, no elaborate solos or clever chord substitutions. However, the results are still stylistically distinct from the results of the classical foreground algorithm. Kulitta has two methods for generating jazzy music: a simple algorithm that produces mainly jazz chords and an algorithm for a simple bossa nova interpretation of a progression. Examples of output from these algorithms can be seen in Figures 6.10, 6.11, and 6.12.

Simple Jazz Foreground

Kulitta's first approach to jazz is essentially just a mapping of the chords from a progression of Roman numerals to seventh chords (chords containing scale degree 7) instead of triads. This simple jazz approach utilizes a five-voice version of mode space for what are referred to as chord *templates*. These are collections of scale indices that can be used to represent a chord in a reasonable way. One example would be to use the root, third, fifth, and seventh of the mode. However, there are many other ways of playing a chord, and some do not include the root. Another way to play the chord if the root is elsewhere would be to use the second, third, fifth, and seventh. Kulitta will map Roman numerals to one of these two

configurations. The bass doubles the root when it is present in the other four voices or adds the root when it is not present.

The bass is required to play the root in this algorithm for an important reason: if this voice is left out and only the 4-voice chords are used, the results often sound unusually dissonant. This is likely because of chords that drop the root, which may end up sounding like they are in a different key if heard in the wrong context particularly when more than one such chord occurs in sequence. Therefore, while it is locally acceptable for a jazz chord to drop certain chordal tones, such as the root, other parts must clearly provide some sort of broader harmonic context to avoid misinterpretation of the chord.

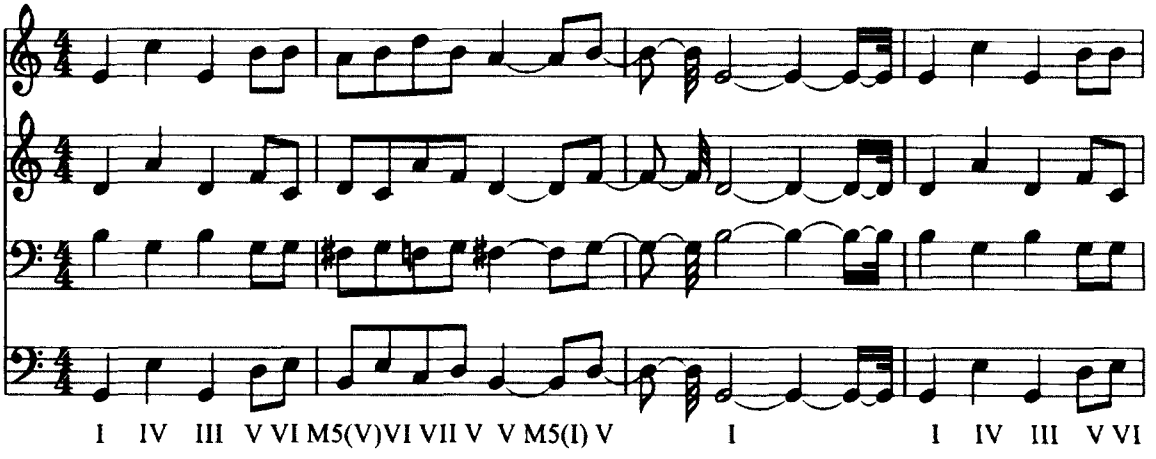


Figure 6.8: An example of a 4-voice, "jazzy" phrase. However, this phrase sounds strange because of the lack of a root in some of the chords.

Bossa Nova Foreground

Kulitta also uses a slightly more complex jazz foreground algorithm that yields a bossa nova-like interpretation of a chord progression. This approach, shown graphically in Figure 6.9, uses seven voices, which would be computationally tricky if forced into the same chord space. To cut down on the combinatoric explosion that would result from such an approach, three different chord spaces are used in *parallel*, one for each "band member" in the music: bass, harmony, and lead. This means that constraint-satisfaction can only

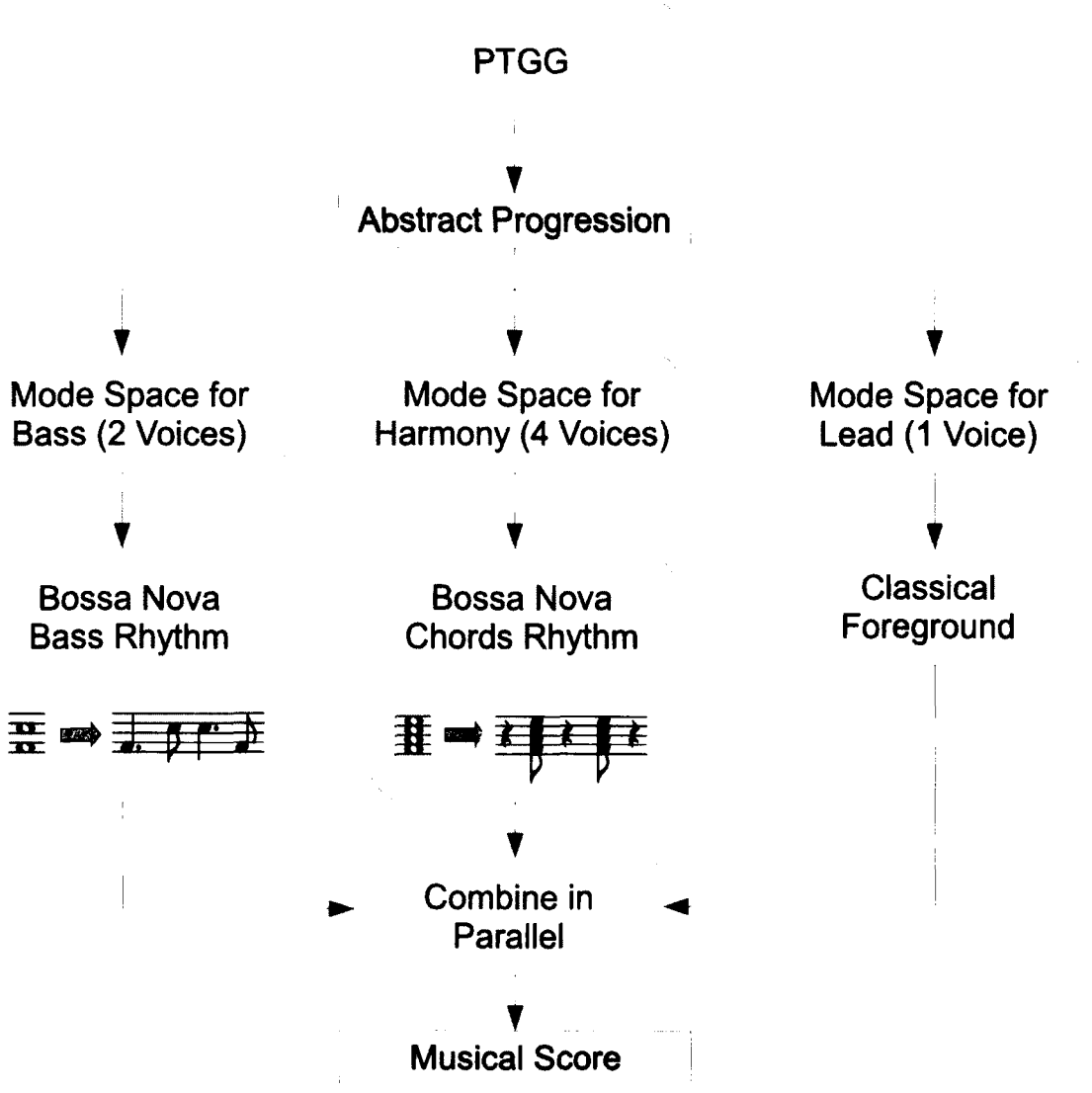


Figure 6.9: Graphical representation of Kulitta's bossa nova foreground algorithm.

take place *within* one of these musical roles, but that is not unreasonable for this genre of music. Jazz is highly improvisational in nature, and musicians will very often have to make decisions about their own part that only rely on abstract information about what the other performers will do rather than exact knowledge of the pitches. This algorithm is shown graphically in Figure 6.9. The forking and merging of the generative process is similar to the interactive ensemble models described by Hudak and Berger [37], although Kulitta's bossa nova algorithm is simpler in that it does not involve any further interaction between musical roles after the fork occurs.

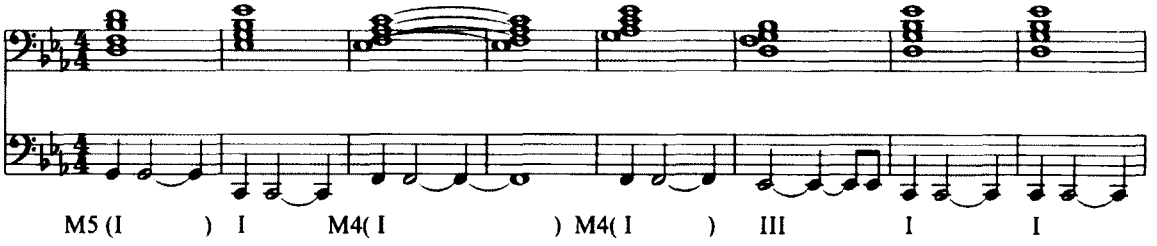


Figure 6.10: An example of a phrase in C-minor with a simple jazz foreground.

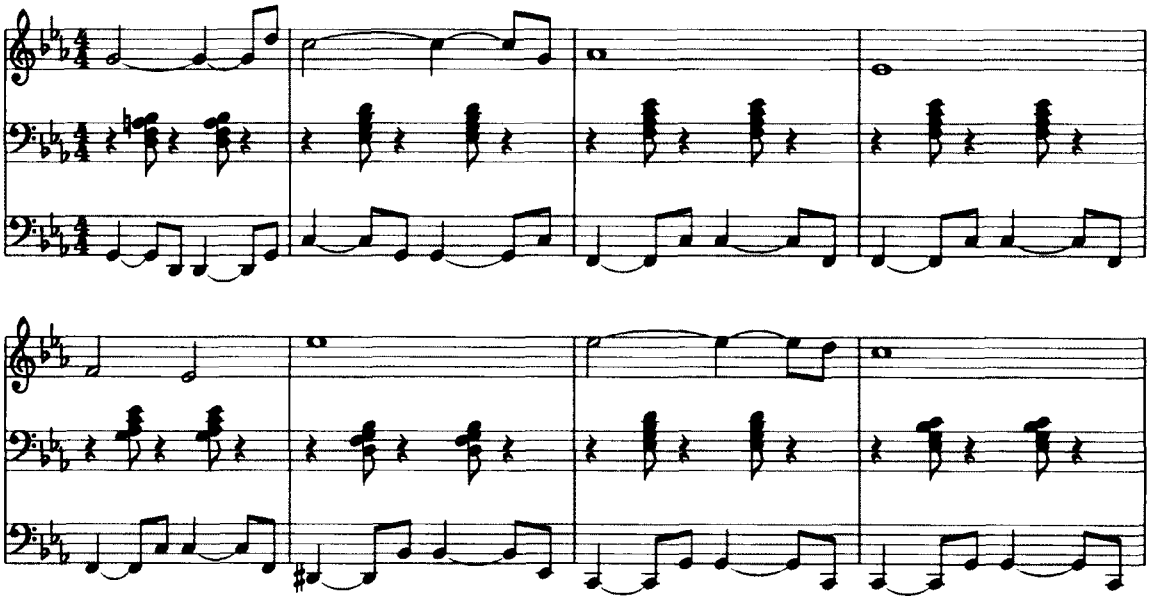


Figure 6.11: An example of a phrase in C-minor with a bossa nova foreground. It uses the same Roman-numeral-level, abstract progression as in Figure 6.10.

Obviously the results from the simple and bossa nova algorithms are still very much at the level of a beginner in jazz. To handle a greater breadth of jazz styles with more complicated features, it would be preferable to learn the behaviors of each instrument or performer from examples of human performances rather than to build unique algorithms for each by hand. This remains as an area of possible future work for Kulitta.

6.2.3 Other Styles

Kulitta is also capable of generating more modern-sounding harmonies through the use of other chord spaces as shown in Figure 6.14, and by mixing chord spaces and foreground

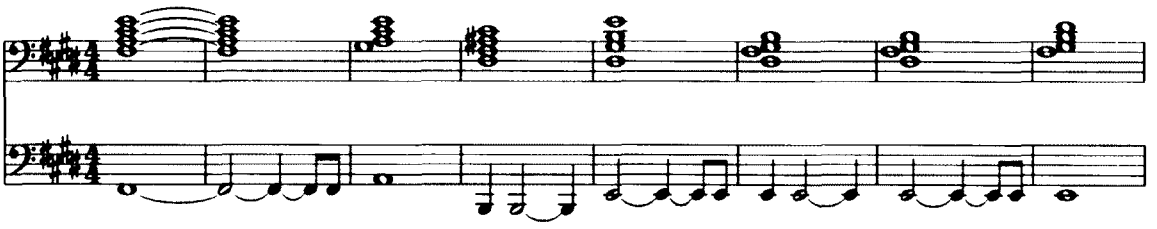


Figure 6.12: An example of a phrase in E-major with a simple jazz foreground.

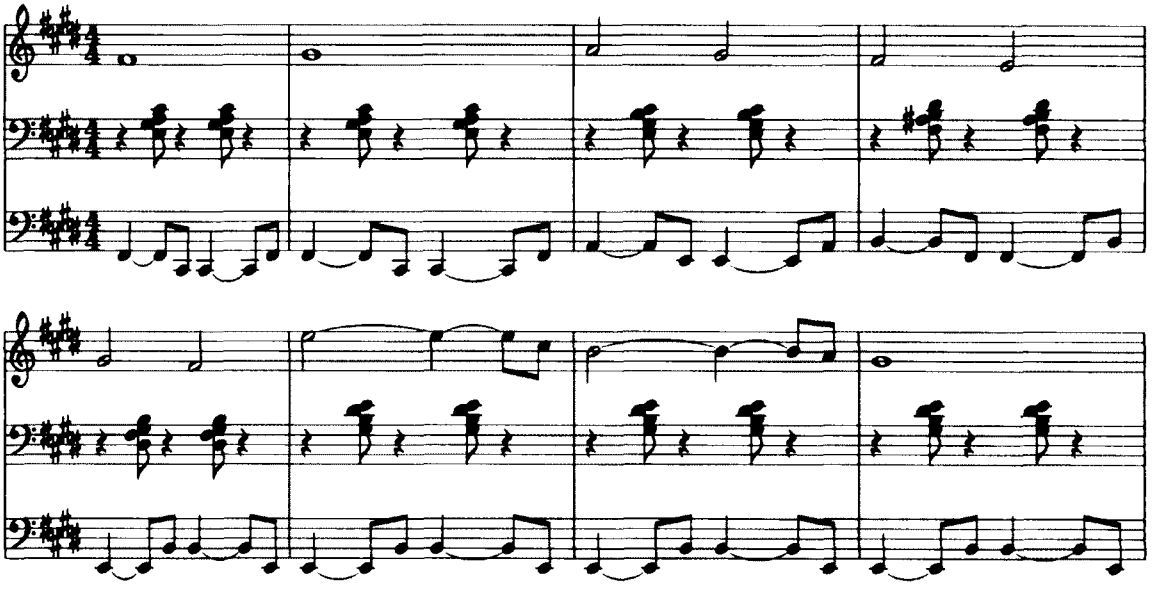


Figure 6.13: An example of a phrase in E-major with a bossa nova foreground. It uses the same Roman-numeral-level, abstract progression as in Figure 6.12.

algorithms from different styles as shown in Figure 6.15. Also, although capable of generating complete musical scores, Kulitta does not have to be the sole author of its (or her) output. In fact, there is a wide range of possible further human interpretation for Kulitta's output at many levels. Abstract progressions generated by a human can be fed into Kulitta's constraint-satisfaction and foreground algorithms, or progressions made by a PTGG can provide the harmonic backbone or inspiration for further details added by a human composer. There are even real-time, interactive possibilities for Kulitta, such as using an an online constraint-satisfaction algorithm in conjunction with chord spaces to create music in response to user input. Figure 6.16 shows an example of how Kulitta's generative modules can be used in this way. This type of system could be used to create an algorithmic back-

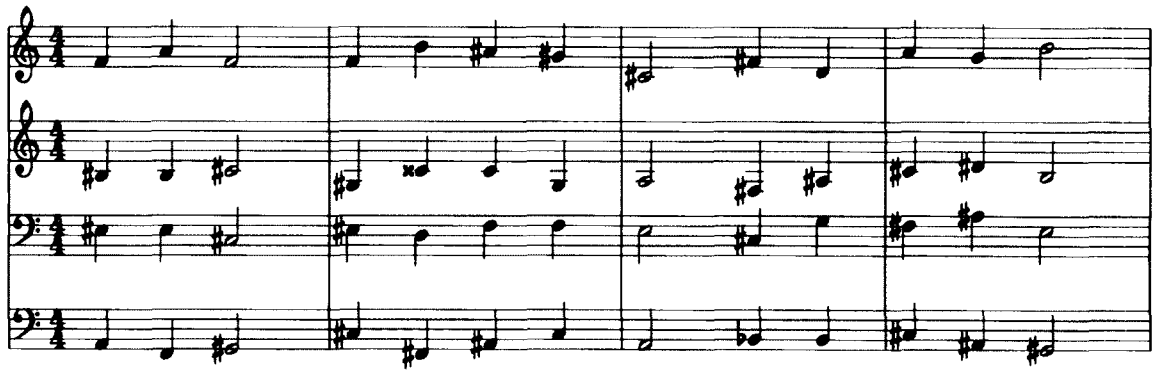


Figure 6.14: An example of a phrase (without a foreground) that is OPTC-equivalent to the one shown in Figure 6.4. The use of OPTC-equivalence causes a much more diverse set of harmonic transitions that are uncharacteristic of classical chorales, but more frequent in modern music.

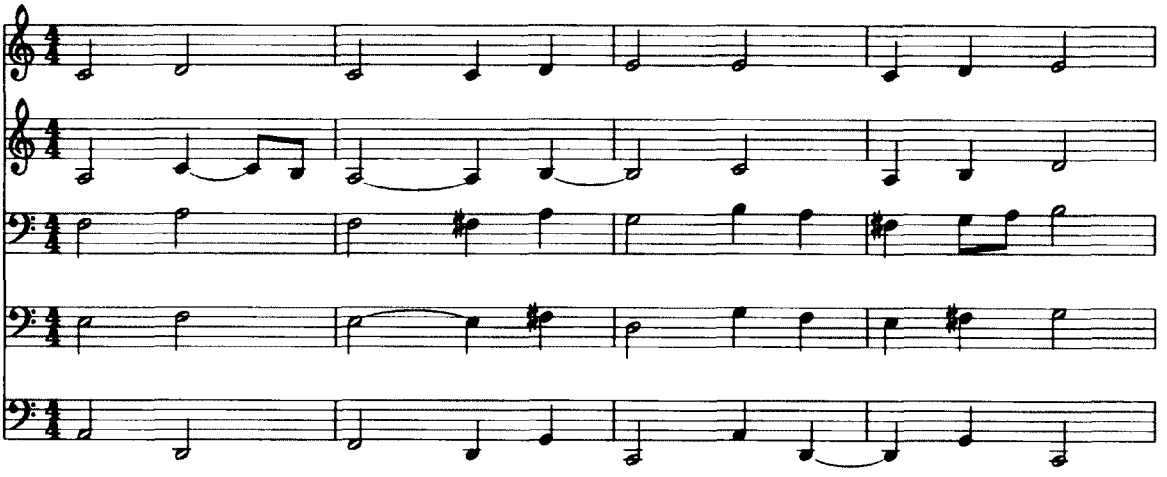


Figure 6.15: A five-voice "jazz chorale" generated by mixing jazz chord spaces with a classical foreground algorithm.

end to a more traditional step sequencer or, if extended to feature learning and/or prediction algorithms in the statistical analysis step, it would be possible to "jam" interactively with a human performer.

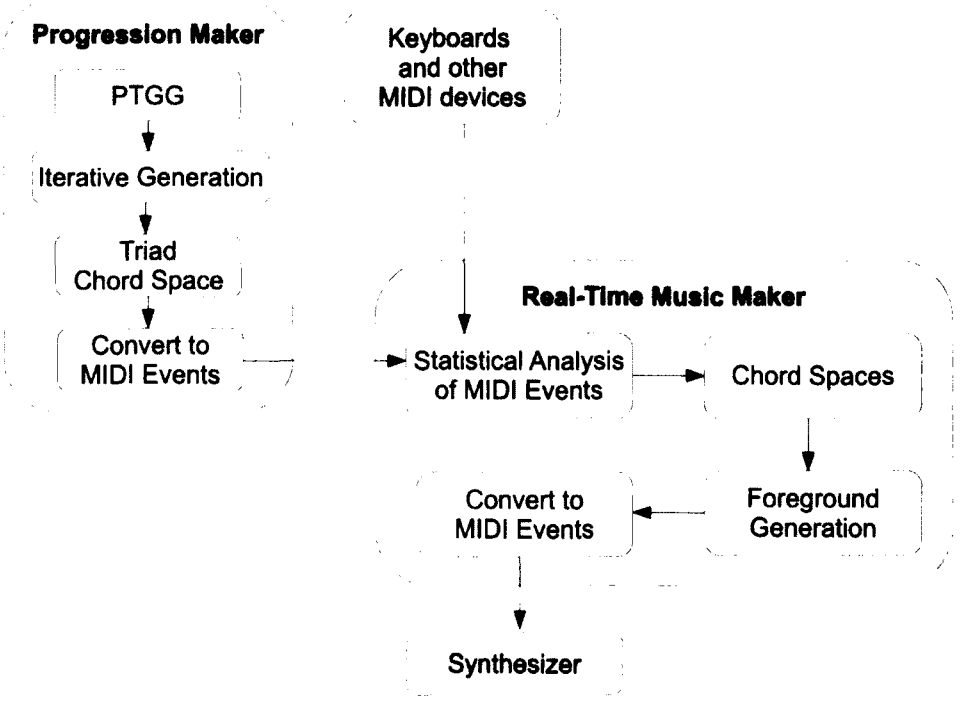


Figure 6.16: An example of Kulitta's modules used in a real-time setting with two different types of input: MIDI events from a "progression maker" program that converts chords to a timed stream of MIDI events and arbitrary MIDI events from a human user or other source. The "real-time music maker" takes streams of MIDI events and analyzes them to determine properties like the current key. This information can then be used to generate new music that changes based on the input stream of MIDI events.

Chapter 7

Learning Musical Structure

One of Kulitta's goals is to not just produce music from hand-constructed grammars, but to also learn features of these grammars from collections of existing compositions. Kulitta currently supports two methods of learning musical features:

1. Learning production probabilities for a musical PCFG, which is then converted to a PTGG for use with Kulitta's generative algorithms.

2. Learning production probabilities for a PTGG directly.

Kulitta uses an extended version of the inside-outside algorithm to learn musical grammars. This chapter describes various modifications to that algorithm, first for learning a musical PCFG and later for learning a PTGG. Chapter 8 shows use of these algorithms with corpora of data to learn production probabilities and then generate music.

7.1 Related Work

Machine learning-based approaches to automated composition are appealing since they can yield more diverse results with less human effort. David Cope's EMI [19, 20] is one such system, and learning algorithms such as Markov decision processes, neural nets, and Boltzmann machines have also been used to generate various musical features [4, 5, 35, 87].

Music has many features in common with spoken language and is now often analyzed using methods inspired by linguistics [48]. Just as spoken language has the notion of parts of speech (noun, verb, etc.), music has abstract labels that are applied to various features (I-chord, passing tone, etc.). Machine learning techniques have been applied to tasks such as part-of-speech tagging [34] as well as harmonic and rhythmic anaysis [13, 66].

Kulitta uses musical grammars, PTGGs, as one step in its generative process. Learning grammars and grammatical features are common subjects of computational linguistics research and learning algorithms exist for various categories of grammars. CFGs are popular subjects for their simplicity, and PCFGs in Chomsky Normal Form are possible to learn in $O(n^3)$ time with the *inside-outside algorithm* [17, 47]. Learning algorithms for some categories of context-sensitive grammars have also been proposed [14, 15].

7.2 The Inside-Outside Algorithm

The inside-outside algorithm is an approach for learning production probabilities for a PCFG analagously to how the forward-backward algorithm learns state transition probabilities for HMMs [47]. Given a PCFG and a data set presumed to be generated by that PCFG, the algorithm's goal is to find production probabilities for that PCFG that maximize the probability of the data set. Instead of computing probabilities over a linear sequence of symbols, the inside-outside algorithm computes probabilities over parse trees. The *inside* probability of a particular node in the parse tree is the probability of generating the subtree rooted at that node, and the outside probability of a node is the probability of the rest of the parse tree minus that node's subtree. Inside and outside probabilities are respectively analagous to the forward and backward probabilities of a HMM.

The rules of a PCFG must be supplied up-front to the inside-outside algorithm along with initial estimates for the production probabilities. The algorithm then iteratively re-estimates the production probabilities for each rule. Rules are expected to be in *Chomsky*

normal form for a PCFG. This form allows the two types of rules shown below[1], where capital letters indicate nonterminals and lowercase letters represent terminals.

$$A \rightarrow BC \tag{7.1a}$$

$$A \rightarrow x \tag{7.1b}$$

Before production probabilities can be reestimated, strings in the training data must first be parsed to determine which rules might have been applied. Parsing must also be done in a way that accounts for ambiguity, since some strings may have more than one possible parse tree. The CYK algorithm can be used for this.

7.2.1 CYK Parsing

John Cocke, Daniel Younger, and Tadao Kasami described a parsing algorithm that is now called the CYK algorithm, or sometimes CKY algorithm [55]. It approaches the parsing task by filling in a table rather than building a parse tree directly. The resulting table can be used to determine whether a given string is accepted by a language, but finding a specific parse requires some extra work.

A CYK parse table shows which nonterminals can produce which substrings of the input sequence. For a string of length n, rows are numbered from 0 to n and columns from 1 to n. Row 0 contains the string itself. A symbol at row r and column c must be able to produce symbols c through $c + r - 1$ via some series of rule applications. Consequently, strings accepted by a given language will yield the start symbol in the topmost cell, which accounts for the entire string. Tables are constructed from the bottom up. For example, given the rules $S \rightarrow AA$, $A \rightarrow AA$ and $A \rightarrow a$, the CYK table for *aaa* would be:

1. Sometimes a third type of rule, $S \rightarrow \varepsilon$, is also included to allow the start symbol to produce the empty string. However, such a rule is not useful in the context of this chapter and therefore is not considered.

	1	2	3
3	S,A		
2	S,A	S,A	
1	A	A	A
0	a	a	a

Individual cells will be referred to using the notation (r,c) for row r and column c. The start symbol appears in $(3,1)$, thereby indicating that at least one parse exists. However, the table does not indicate any particular parse. The parse for *aaa* above is ambiguous, since it could either be grouped as $a(aa)$ or $(aa)a$. This is captured in the table, since $(3,1)$ can produce $a(aa)$ by generating the A symbols in cells $(1,1)$ and $(2,2)$ or it can produce $(aa)a$ by generating the A symbols in cells $(2,1)$ and $(1,3)$.

All possible parses are represented in the table as well as extraneous symbols that can produce portions of the string but are not involved in any full parse of the string. The A appearing at (3,1) is an example of this, as are the start symbols, S, appearing below row 3.

7.2.2 Learning Production Probabilities

The symbol ψ is used to denote the probability mass function over rules. The inside probability of a nonterminal A spanning terminals i through j is denoted $\alpha(A,i,j)$. For a sequence of symbols, $x_1,...,x_n$, forward probabilities are computed by:

$$\alpha(A,i,i) = \psi(A \rightarrow x_i) \tag{7.2}$$

For $i < j$:

$$\alpha(A,i,j) = \sum_{k=i,\ A \rightarrow BC \in R}^{k=j-1} [\psi(A \rightarrow BC) \times \alpha(B,i,k) \times \alpha(C,k+1,j)] \tag{7.3}$$

Given the start symbol for the grammar, S, the probability of the entire sequence of length n can be found by: $\alpha(S,1,n)$.

The outside probability of a nonterminal, A, spanning symbols i through j, is denoted $\beta(A,i,j)$. It accounts for all portions of the tree not addressed by $\alpha(A,i,j)$. In other words, $\beta(A,i,j)$ is the probability of generating the sequence: $x_1,\ldots,x_{i-1},A,x_{j+1},..,.x_n$.

$$\beta(S,1,n) = 1 \tag{7.4}$$

For $A \neq S$, $i \leqslant j$:

$$\beta(A,i,j) = \sum_{k=j+1,\ B\to AC\in R}^{k=n} [\psi(B \to AC) \times \alpha(C,k,n) \times \beta(B,i,k)] + \sum_{k=1,\ B\to AC\in R}^{k=i-1} [\psi(B \to CA) \times \alpha(C,k,i-1) \times \beta(B,k,j)] \tag{7.5}$$

The α and β values are combined to calculate the probability of a rule appearing at a particular point in a string's parse tree. This quantity is called μ.

$$\mu(A,i) = \alpha(A,i,i) \times \beta(A,i,i) \tag{7.6a}$$

$$\mu(A \to BC,i,k,j) = \psi(A \to BC) \times \alpha(B,i,k) \times \alpha(C,k+1,j) \times \beta(A,i,j) \tag{7.6b}$$

For a given rule, this value is then summed over all instances of a rule and normalized to calculate the new production probability for that rule.

$$count(A \to x) = \sum_i \mu(A \to x,\ i) \tag{7.7a}$$

$$count(A \to BC) = \sum_{i,k,j} \mu(A \to BC,\ i,\ k,\ j) \tag{7.7b}$$

Let $count^s$ denote the counting equation over a particular string in the data set (all equations to this point have been for single strings). Re-estimation of the production probability for a rule $A \rightarrow v$ with data set $S = \{s_1, ..., s_m\}$ is defined by:

$$\psi'(A \rightarrow v) = \frac{\sum_{s \in S} count^s(A \rightarrow v)}{\sum_{A \rightarrow v' \in R}(\sum_{s \in S} count^s(A \rightarrow v'))} \tag{7.8}$$

This recalculation of production probabilities can be done iteratively until the values converge to within some threshold.

The algorithm's representation of symbols over spans mirrors that of the CYK algorithm. For example, the value $\alpha(A, i, j)$ will be nonzero if A appears in column i of row $j - i$ in the parse table, and zero if A is not present in that cell. Simialrly, $\beta(A, i, j)$ will only be nonzero if A is part of a parse tree. The CYK parse table can be used directly to locate (A, i, j) tuples over which calculations should be done.

7.3 Learning a Musical PCFG

A slightly modified version of the inside-outside algorithm can be used to learn production probabilities for Martin Rohrmeier's CFG for harmony. However, Rohrmeier's grammar is not in Chomsky normal form. Rather than modify the grammar to place it in Chomsky normal form, we can also modify the inside-outside algorithm to handle new types of rules. Rules such as $TR \rightarrow T$ and $I \rightarrow I\ IV\ I$ in Rohrmeier's grammar require that the algorithm handle rules of the following forms.

$$A \rightarrow BCD \tag{7.9a}$$

$$A \rightarrow B \tag{7.9b}$$

The *rank* of a rule is the number of symbols that appear on the righthand side. Rohrmeier's

grammar uses rules of rank 1, 2, and 3. Rank 3 and rank 1 for rules of the form $A \rightarrow B$ are relatively straightforward additions to the equations, although rules of rank 3 increases the worst-case complexity for the algorithm. The definition for the $i \neq j$ case of the inside probability formula becomes:

$$\begin{aligned}\alpha(A,i,j) = & \sum_{k,l,\ A\rightarrow BCD\in R}^{i\leqslant k<l<j} [\psi(A\rightarrow BCD)\times\alpha(B,i,k)\times\alpha(C,k+1,l)\times\alpha(D,l+1,j)] \\ & + \sum_{k,\ A\rightarrow BC\in R}^{i\leqslant k<j} [\psi(A\rightarrow BC)\times\alpha(B,i,k)\times\alpha(C,k+1,j)] \\ & + \sum_{A\rightarrow B\in R} [\psi(A\rightarrow B)\times\alpha(B,i,j)]\end{aligned} \tag{7.10}$$

The changes to β are also fairly straightforward, although rather verbose. To simplify the definition, we will denote the previous definition of β in Equation 7.5 for rules of rank 2 as β_2 and the new definition for rules of rank 3 as β_3.

$$\beta_3(A,i,j) = \beta_2(A,i,j)$$
$$+ \sum_{k,l,\ B\to ACD\in R}^{j<k<l\leqslant n} [\psi(B\to ACD)\times\alpha(C,j+1,k)\times\alpha(D,k+1,l)\times\beta(B,i,l)] +$$
$$+ \sum_{k,l,\ B\to CAD\in R}^{k<i\leqslant j<l} [\psi(B\to CAD)\times\alpha(C,k,i-1)\times\alpha(D,j+1,l)\times\beta(B,k,l)] +$$
$$+ \sum_{k,l,\ B\to CDA\in R}^{k<l<i} [\psi(B\to CDA)\times\alpha(C,k,l-1)\times\alpha(D,l,i-1)\times\beta(B,k,j)] +$$
$$+ \sum_{B\to\in R} [\psi(B\to A)\times\beta(B,i,j)] \tag{7.11}$$

A final modification must take place in the formula for μ and *count* as well to handle rules of rank 3, although the definitions of μ and *count* for rules of rank 2 remain the same.

$$\mu(A\to B,i,j) = \beta_3(A,i,j)\times\psi(A\to B)\times\alpha(B,i,j) \tag{7.12}$$

$$\mu(A\to BCD,i,k,l,j) = \beta_3(A,i,j)\times\psi(A\to BCD)\times$$
$$\alpha(B,i,k)\times\alpha(C,k+1,l)\times\alpha(D,l+1,j) \tag{7.13}$$

$$count(A\to B) = \sum_{i,j}\mu(A\to B,\ i,\ j) \tag{7.14}$$

$$count(A\to BCD) = \sum_{i,k,l,j}\mu(A\to BCD,\ i,\ k,\ l,\ j) \tag{7.15}$$

7.4 Learning a PTGG

A PTGG is a parameterized grammar with an infinite alphabet and rules that are functions. All of these features are problematic for the inside-outside algorithm even with the extensions discussed so far. Three key features of the grammar must be addressed:

1. PTGGs make no distinction between terminals and non-terminals.

2. Symbols in PTGGs must carry extra information (the parameters indicating duration) and the parameter list is potentially infinite. However, a CYK-style parse table of a given progression will still be finite.

3. Rules are *functions* that are later instantiated with concrete values. Productions such as $I^w \rightarrow V^h \ I^h$ and $I^h \rightarrow V^q \ I^q$ must be recognized as *instances* of the same rule, $I^t \rightarrow V^{t/2} \ I^{t/2}$.

In a more general sense, what we need is a way to learn a grammar where the full extent of the alphabet is unknown and where rules have the form of $X \rightarrow f(X)$. Here we show an oracle-based approach to the inside-outside algorithm that enables learning grammars of this form when $f(X)$ has certain properties.

7.4.1 An Oracle Approach to the Inside-Outside Algorithm

Suppose we don't know exactly what the rules for a grammar look like except for the assumption that they are context-free (in the sense that only one symbol can appear on the left-hand side of a rule). The rules could even exhibit conditional behavior based on symbols' parameters, as is possible in a PTGG. In fact, the details of the rule set are unnecessary for the learning production probabilities as long as the learning algorithm has access to the following things:

1. An identifier for each abstract rule. This can simply be a number (i.e. "rule 1," "rule 2," and so on).

2. The production probability for each abstract rule (or the initial estimates of those probabilities).

3. A partition of the abstract rules' identifiers into groups that share the same left-hand side.

4. An oracle that takes a sequence of symbols and returns all *rule instances* that could have directly produced it along with their associated identifiers. The distinction between a rule and its instance is described in Section 7.4.3.

Note that for a typical PCFG, there is no function/instance separation in the rules, so an oracle would return the rule itself. However, for some other grammar like a PTGG, the oracle would only return a rule instance like $I^w \rightarrow V^h \ I^h$, while the exact function that created it, $I^t \rightarrow V^{t/2} \ I^{t/2}$, would remain unknown to the learning algorithm. Importantly, the learning algorithm has neither an enumeration of the alphabet (which is potentially infinite for a PTGG) nor an enumeration of all possible rule instances. The algorithm only needs information about the symbols and rule instances that can be used to accomplish a CYK-style parse of the training data. Given the four pieces of information described above, production probabilities can be re-estimated by:

1. Building a CKY-style parse table of rule instances for each string in the training data. Storing the full rule instances and their associated identifiers avoids any subsequent queries to the oracle once the parse table is complete.

2. Traversing the parse table to compute α and β values as described in Section 7.4.5.

3. Summing counts for rule instances by their rule identifiers when re-estimating production probabilities.

Step 1 above, building a CYK-style parse table, must address the lack of terminal/nonterminal distinction and the rule function/instance distinction for PTGGs. These cause some small,

but cascading changes through the probability calculations. These issues are described in more detail in the following sections.

7.4.2 Removing the Terminal/Nonterminal Distinction

Mathematically, the inside-outside algorithm does not actually enforce any important distinction between terminals and nonterminals in the traditional sense. Terminals are simply symbols that exist in row 0 and provide a stopping point for the recursive α calculations, much like the start symbol serves as a stopping point for β.

Removing the terminal/nonterminal distinction in a grammar implies that any point during generation of a sequence of symbols is a valid stopping point. In many of the grammars used for spoken language, this would be absurd, since a non-terminal like *noun phrase*, often abbreviated as *NP*, is not a spoken entity and must be further instantiated to something more meaningful. A string such as "This is a brown horse" is accepted by the English language, but "This is *NP*" is not—the instance of *NP* must be further expanded. However, this is not the case for all grammars.

A lack of terminal/non-terminal distinction is an important property of L-Systems, a category of grammars commonly used for modeling fractals, where infinite self-similarity must be accounted for but only finite sequences can realistically be calculated. Consider the following L-System with A as its start symbol:

$$\begin{aligned} A &\rightarrow AB \\ B &\rightarrow A \end{aligned} \tag{7.16}$$

This grammar, defined by Prusinkiewicz and Lyndenmaier [61], consists entirely of non-terminals and produces strings such as *ABAAB*. All strings produced by the grammar can be further expanded. How much they are expanded depends on the generative algorithm used.

In a grammar without terminals, rules of the form $A \rightarrow x$ are mathematically no different from rules of the form $A \rightarrow B$ where $A \neq B$ (situations involving "identity rules" or "self productions" of the form $A \rightarrow A$ will be addressed later). Rohrmeier's grammar has many of these, such as $TR \rightarrow T$ and $DR \rightarrow D$. Allowing for these rules, the equation for inside probabilities for single symbols in the sequence $X_1, ..., X_n$ becomes:

$$\alpha(A,i,i) = \begin{cases} 1.0 & \text{if } A = X_i \\ \sum_{A \rightarrow B \in R} [\psi(A \rightarrow B) \times \alpha(B,i,i)] & \text{otherwise} \end{cases} \tag{7.17}$$

The definition for $\alpha(A,i,j)$ where $i \neq j$ remains the same as in 7.19. True terminals will cause no problems; the only difference is that they are now candidates to be supplied to the α calculations.

7.4.3 Rule Functions and Rule Instances

One of the trickiest aspects of PTGGs from a learning standpoint is the distinction that exists between rules, which are functions, and their instances, which are applications of those functions to specific values. For clarity, we will refer to PTGG rules as *rule functions* when refering to the function itself, such as $I^t \rightarrow V^{t/2}\ I^{t/2}$ where t is a variable. We will refer to the applications of those rules to specific values as *rule instances*. For the rule function $I^t \rightarrow V^{t/2}\ I^{t/2}$, the productions $I^w \rightarrow V^h\ I^h$ and $I^h \rightarrow V^q\ I^q$ are just two of many possible *instances* of the function.

For a traditional PCFG, the probability mass function, ψ, can simply perform a table lookup, pattern maching against either the lefthand side or righthandside (or both) for calculating α and β values respectively. For a PTGG, the process is more complicated. In training data, chords will have concrete durations, such as h (half note) rather than a variable, meaning that parsing must take place with concrete values as well. The parse tree must, therefore, be constructed using rule instances rather than rule functions.

The concept of rule functions and instances of those functions is broader than simply those in PTGGs. In fact, any grammar with rules of the form $X \rightarrow f(X)$ where X is a single symbol and $f^{-1}(X)$ is computable are covered by this paradigm, including the more standard category of PCFGs. The rules in PCFGs can be thought of as functions where each has only one instance.

7.4.4 Parsing with Rule Instances

With a PCFG, rule instances are no different from the actual rules in the grammar. However, with a PTGG, one rule can have many instances. How can we efficiently find these instances to recursively compute α and β values without knowing about the rules themselves? The answer lies in a simple change to the parse tree representation.

Consider the progression $II^q\ V^q\ I^q\ I^q$ produced by the rules $I^t \rightarrow V^{t/2}\ I^{t/2}$, $I^t \rightarrow I^{t/2}\ I^{t/2}$ and $V^t \rightarrow II^{t/2}\ V^{t/2}$. A CYK-style parse tree would look like the following.

4	I^w			
3				
2	V^h	I^h	I^h	
1	II^q	V^q	I^q	I^q
	1	2	3	4

Notice that the removal of the terminal/nonterminal distinction now means that there is no need for a 0-row. This representation can be used to derive the portion of the alphabet relevant to the string and the spans over which each symbol should have α and β computed. We can derive the following combinations for which α and β would need to be computed:

i	j	Symbols
1	2	V^h
2	3	I^h
3	4	I^h
1	4	I^w

For all other symbols and spans, α and β would be zero. However, we also need to be able to determine which rules produced each cell in order to recursively compute α and β probability values. Unfortunately, the standard CYK representation does not tell us which rules were applied at each point, since, under the oracle model, we don't know anything about the abstract rules in the grammar and can only ask about rule instances. Rather than querying the oracle again to re-derive which rules were applied to which cells, a better solution is to simply label the cells with rule instances during the parsing process. The table now becomes:

4	(0) $I^w \rightarrow V^h\ I^h$			
3				
2	(2) $V^h \rightarrow II^q\ V^q$	(0) $I^h \rightarrow V^q\ I^q$	(1) $I^h \rightarrow I^q\ I^q$	
1	II^q	V^q	I^q	I^q
	1	2	3	4

This operation is delegated to an oracle and must be defined for a given category of grammars. For PCFGs, the operation is just a matter of checking symbol membership in the strings appearing on the right-hand side of rules. For PTGGs, the operation is a little more complicated and involves the notion of *valid* and *invalid* rule instances. A valid rule instance for a sequence of symbols is one where the duration parameters associated with each symbol have ratios that can account for portions of the sequence. For example, with a start symbol of I^4 and rules that only divide the temporal parameters of symbols by powers of two, $I^4 \rightarrow V^2\ I^2$ would be a valid instance of $I^t \rightarrow V^{t/2}\ I^{t/2}$, but $I^1 \rightarrow V^{1/3}\ I^{2/3}$ would be invalid. Algorithm 10 shows the process for backtracking through rule instances of a PTGG that would be carried out by the oracle.

Algorithm 10. *Given a validity function for a PTGG, valid(r), that returns true for valid rule instances:*

backtrack$_{PTGG}(x^p)$ =

1. *Let $Y = y^t \rightarrow x_1^{f_1(t)} \ldots x_n^{f_n(t)} \mid \exists i,\ x = x_i$ be the collection of rule functions where x appears on the right-hand side at position i.*
2. *$Y' = \emptyset$ be a set of rule instances.*
3. *For each rule function, $y^t \rightarrow x_1^{f_1(t)} \ldots x_n^{f_n(t)} \in Y$ where $x = x_i$:*
   (a) *Let $p' = f_i^{-1}(p)$ be the unique parent parameter that could have produced p.*
   (b) *If valid$(y^{p'} \rightarrow x_1^{f_1(p')} \ldots x_n^{f_n(p')})$, then add $y^{p'} \rightarrow x_1^{f_1(p')} \ldots x_n^{f_n(p')}$ to Y'.*
4. *Return Y'*

7.4.5 Modifications to the Inside-Outside algorithm

The changes required to support the oracle model and CYK parse table described in the previous sections cause subtle changes throughout the inside-outside algorithm's equations. Changes must be made where the equations for α and β make reference to the rule set, R, since the rule functions are inaccessible and only rule instances can be accessed through an oracle. However, these changes are fairly small, substituting some reference to the rule set's oracle for the rule set itself. For an Oracle, O, we will use the notation $P(O,i,j)$ to refer to the cell in the Oracle-made parse table that can produce symbols from i through j. This cell will contain a list of rule instances. Using this notation, α would be redefined as follows:

$$\alpha(A,i,i) = \begin{cases} 1.0 & \text{if } A = X_i \\ \sum_{A \rightarrow B \in P(O,i,i)} [\psi(A \rightarrow B) \times \alpha(B,i,i)] & \text{otherwise} \end{cases} \tag{7.18}$$

$$\alpha(A,i,j) = \sum_{k,l,\ A\to BCD\in P(O,i,j)}^{i\leqslant k<l<j} [\psi(A\to BCD)\times\alpha(B,i,k)\times\alpha(C,k+1,l)\times\alpha(D,l+1,j)]$$
$$+ \sum_{k,\ A\to BC\in P(O,i,j)}^{i\leqslant k<j} [\psi(A\to BC)\times\alpha(B,i,k)\times\alpha(C,k+1,j)] \quad (7.19)$$

Equations for β and μ would be modified similarly, replacing instances of R with $P(O,i,j)$.

7.4.6 Identity Rules

So far, productions of the form $A \to B$ have been allowed, but with the requirement that $A \neq B$. Allowing for rules of the form $A \to A$ is somewhat problematic for various statistical reasons. Consider the following, terminal-less PCFGs:

Grammar 1	Rule	Probability
	$A \to A\,A$	0.4
	$A \to A$	0.2
	$A \to B$	0.4

Grammar 2	Rule	Probability
	$A \to A\,A$	0.5
	$A \to B$	0.5

Both of these can generate strings defined by the regular expression $[A|B]^*$, which is all strings consisting of As and Bs. However, for a fixed number of generative steps, the probability of generating the string AB is lower for the first grammar than for the second. For Grammar 1, the probability of generating AB is at most $0.4 \times 0.4 = 0.16$ if $A \to A$ is never used, and this value would decrease for each additional instance of $A \to A$, which can be applied infinitely many times to yield as many distinct parse trees. For Grammar 2, the probability of generating AB is exactly $0.5 \times 0.5 = 0.25$.

From this, it is clear that the two distributions above can exhibit different behavior even

under the same applying algorithm. For the same number of total productions, grammar 1 is likely to produce a shorter string than grammar 2. However, what if the generative algorithm and number of productions are unknown? Distinguishing between the two distributions becomes tricky, since each string producible by grammar 1 can be produced by grammar 2 with fewer steps.

To fully accommodate the features of PCFGs, we extend the inside-outside algorithm once more with a *heuristic* for counting occurrences of identity rules. It must be emphasized that this heuristic will not suit all generative algorithms, since knowing where identity rules are likely to be applied requires knowing how the generative algorithm works. Unlike other rules, identity rules cannot simply be inserted into the parse tree wherever possible, since they can be applied infinitely many times in sequence while achieving the same overall result. Identity rules also cannot be ignored; some probability mass should be assigned to them if they are present in the grammar.

As a middle ground, we count one identity rule per parse tree in the μ calculation. In Equations 7.12-7.15, the requirement that $A \neq B$ is simply removed. However, importantly, we do *not* count identity rules in the α and β calculations. Doing so would risk placing disproportionate weight on identity rules when re-estimating production probabilities for a PCFG. By factoring identity rules into the μ calculation as shown above but nowhere else, it avoids assigning truly excessive weight to identity rules. Unfortunately though, this does not guarantee a conservative guess for the probability of identity rules, and the estimated probability can still end up being disproportionately high. Consider the following grammar.

Probability	Rule
0.3	$A \rightarrow AA$
0.3	$A \rightarrow A$
0.4	$A \rightarrow B$
0.5	$B \rightarrow B$
0.5	$B \rightarrow C$

It takes a minimum of two productions to generate a string containing at least one C. If

a data set is generated with very few iterations of an L-System-like algorithm, such as only three iterations, attempting to learn the probability for the $B \rightarrow B$ using the heuristic described above will fail by assigning a disproportionately large probability to the rule. This is because the restricted number of productions makes it unlikely that C will be encountered relative to B, and each instance of B will constitute one count for applying $B \rightarrow B$. This is a difficult problem to avoid with "deep" grammars that require many productions to reach certain symbols. Symbols closer to the top of the symbol hierarchy are more likely to have accurate probabilities learned for their identity rules.

7.4.7 Computational Complexity

For a PCFG, the complexity of this oracle-based version is actually unchanged, since oracle would have a worst case runtime of $O(l)$, where l is the number of rules in the grammar, and will perform the same search as would have to be done in an oracle-less model anyway. Therefore, for PCFGs, the complexity is still $O(n^3)$ for grammars with rules of rank 2 and $O(n^4)$ for rules of rank 3, where n is the length of of the string to be parsed. Since the oracle model requires that $f^{-1}(X)$ be computable for rules of the form $X \rightarrow f(X)$ but does not bound the complexity of that computation, it is possible that the oracle's complexity could overtake that of the CKY-parsing for other categories of grammars.

The validity function for a PTGG requires checking whether the durations in a rule instance are possible to produce by a given grammar. One way to do this is to simply enumerate the number of possible durations and test them. This set of values will be controlled by the format of the rules, the overall duration of the progression, d_{total}, and the smallest duration present in the progression, d_{min}. For example, for rules that only divide duration evenly, there will be $log_2(d_{total}/d_{min}) + 1$ possible durations. This value will grow much more slowly than the length of the input sequences, which is bounded by (d_{total}/d_{min}). Therefore, the complexity for parsing will still be dominated by $O(n^3)$ from the CYK algorithm, where n is the number of symbols in the input sequence.

7.5 Learning Additional Grammatical Features

In addition to learning production probabilities, it would also be useful to learn the collection of rules or even a collection of relevant non-terminals for forming rules. Lari and Young show that the inside-outside algorithm can be used in conjunction with HMMs to perform precisely this task for PCFGs [47]. However, the grammars learned are linear, much like a HMM that must emit a symbol before moving on. This category of learnable grammars does not allow for any symmetry or more complex branching in the parse tree. This kind of structure seems a poor fit for music, where two phrases of equal length would normally be viewed as branches of a fairly balanced tree.

Lari and Young proposes a method of using the inside-outside algorithm with an initially too-large collection of rules and pruning it down to a smaller collection of rules [47]. Although this could be applied to music, without a way to learn the non-terminals it doesn't serve much purpose. Nonterminals like T for "tonic" do not make sense when separated from their musical meanings to form a set of more general rules. For example, starting with the collection of rules $\{x \rightarrow yz \mid x,y,z \in \{T,S,D\}\}$ is fundamentally problematic, since the grammar is simply too ambiguous to allow sufficient pruning.

PTGGs take the hypothesis that observable harmonic transitions at the level of chord progressions are representative of large-scale patterns as well. If this hypothesis is true, then it may be possible to detect short phrases that would be likely candidates for right-hand-sides of rules and to then derive the left-hand side through music theoretic principles. The task of identifying right-hand-sides of the rules is somewhat analogous to the task of automatic text segmentation in computational linguistics. However, this step in the learning process has not yet been added to Kulitta and is a subject of ongoing work.

Chapter 8

Putting It All Together

One of Kulitta's goals is to not just generate music, but to "learn by listening"— to be able to derive desirable musical features and patterns from a corpus of data while still being able to produce original pieces of music. Production probabilities for PCFGs and PTGGs are one such musical feature that can be learned from a corpus of data as shown in Chapter 7 using modifications to the inside-outside algorithm. Two different generative experiments are described in this section: training Kulitta on a corpus of Bach chorales and a modified version of Rohrmeier's grammar for harmony and training on a synthetic data set generated by a hand-built PTGG. Each set of learned production probabilities was used to generate multiple novel phrases of music.

8.1 Training on Bach Chorales

Bach chorales offer a source of abundant and stylistically consistent musical examples that are also relatively easy to analyze to the level of Roman numerals. Because of this, it is a good data source for testing the performance of grammars like Rohrmeier's. Here we present the results of such an experiment. A corpus of Bach chorales was analyzed to the Roman numeral level, and short phrases were extracted as training data. A modified version of Rohrmeier's grammar for harmony [67] was used as the candidate grammar to

parse the data and learn production probabilities. Finally, Kulitta's generative framework (described in Chapter 6) was used to create short, chorale-styled phrases according to the learned production probabilities.

8.1.1 Data Set

A collection of Bach chorales analyzed by Christopher W. White [83] were taken as the starting data. The chorales in this data set had already been analyzed to determine the key of each chord. Further processing was then done to assign a Roman numeral to each chord, which was relatively straightforward, since most of the chords were simple triads (e.g. $\langle 0,4,7 \rangle$ in C-major is a I-chord). Phrases were taken as 4-measure sections if all within the same key, and same-key segments for 4-measure sections that changed key. These phrases were then divided by mode, creating major and minor training data sets containing 2,650 and 1,446 phrases respectively.

8.1.2 A Modification of Rohrmeier's PCFG for Harmony

Rohrmeier's grammar for harmony is a CFG suitable for learning with the inside-outside algorithm as described in the previous chapter. A reduced version of it that lacked modulations was used to parse the Bach corpus. However, not all of the data was parsable with this reduced grammar. Some of this may have been due to noise in the Roman numeral assignment, but other instances were likely due to limitations of Rohrmeier's grammar. To parse a larger subset of the data while minimizing redundancy in the rules, a modified version of Rohrmeier's grammar was used as the candidate grammar for learning production probabilities. This grammar is shown in Table 8.1. Rules 13, 14, 19, 20, and 21 were suggested by Ian Quinn as mechanisms to increase the number of parsable phrases in the data set.

Important changes to the grammar include the re-purposing of the P nonterminal to mean "plagal cadence" rather than "phrase." The "phrase" level in Rohrmeier's grammar added redundancy and was largely irrelevant to parsing such small sections of music. Sim-

1	TR	→	T
2	TR	→	TR TR
3	TR	→	DR T
4	TR	→	TR DR
5	DR	→	DR DR
6	DR	→	D
7	DR	→	SR D
8	SR	→	S
9	SR	→	SR SR
10	T	→	VI
11	T	→	III
12	T	→	I
13	T	→	I II VI
14	T	→	T P
15	D	→	VII
16	D	→	V
17	S	→	IV
18	S	→	II
19	S	→	IV III IV
20	P	→	IV I
21	P	→	IV P

Table 8.1: A modified version of Rohrmeier's grammar for harmony, where TR is the start symbol.

ilarly, "piece" as a nonterminal was not needed either. Instead, *TR*, or "tonic region" was used as the start symbol. Repetitions at the Roman numeral level were also removed to avoid overly ambiguous parses. The grammar from Table 8.1 was able to parse a total of 1,335 phrases out of the 2,650 present in the major data set. The minor data set was not tested with this grammar.

8.1.3 Method

Using the grammar in Table 8.1, a total of 1,335 phrases in major keys were parsable from the Bach corpus. The extended version of the inside-outside algorithm described in Chapter 7 was run on samples of 200 of these phrases. Samples were taken uniformly at random using a pseudorandom number generator and a fixed seed to ensure reproducibility of the samples. A total of five random samples were taken (seeds 0 through 4). The inside-outside

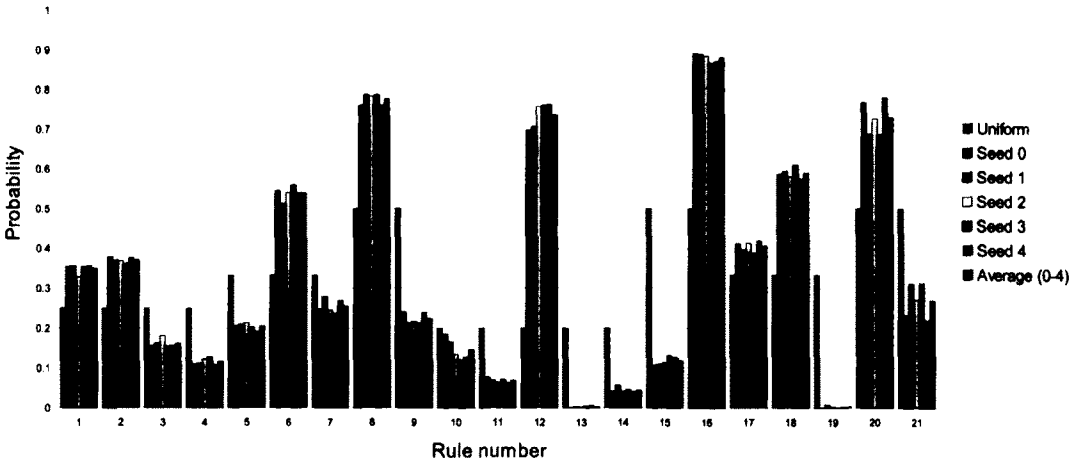


Figure 8.1: Production probabilities for a PCFG based on Rohrmeier's grammar using a corpus of major phrases from Bach chorales. A list of numbered rules can be found in Table 8.1. The graph shows the results of five different runs, each with a different seed for randomly selecting 200 data from the corpus. All runs were given uniform initial probabilities and ran until the change in distributions between iterations fell below 1% of the total probability mass. The "average" data series represents the average of all five runs.

algorithm was run until the change in probability mass fell below a specified threshold. This threshold was set to be 1% of the total probability mass, or 0.07 (since there were 7 nonterminals and the probabilities for each nonterminal must sum to 1.0).

8.1.4 Results

The results of 5 runs of the inside-outside algorithm on different samples from the data set are shown in Figure 8.1. All runs of the algorithm began with a uniform probability mass distribution and converged within only a few iterations. As can be seen from Figure 8.1, the results are all fairly similar regardless of the particular sample of data points used, indicating that the data is consistent within itself. The average of these probabilities was used to qualitatively assess the learned production probabilities using Kulitta's generative algorithms.

8.1.5 From PCFG to PTGG

Kulitta's generative algorithms are designed to take PTGGs rather than more typical PCFGs. A PCFG for harmony can be converted into a PTGG, but there is no clear best way to do this. PCFGs like Rohrmeier's and the version in Table 8.1 are unable to handle any temporal information and there is a terminal/nonterminal distinction, which must be removed in some way to produce a PTGG. Results from three different approaches to this conversion are presented here.

- **Approach 1**: assign all symbols a constant duration of a quarter note, q. For example: $TR^t \rightarrow TR^q\ TR^q$. After some number of generative iterations, nonterminals may still exist in the string. These are forced to terminals using the following convention.

  TR, T, TP, TCP, P $\Rightarrow$ I

  DR, D, DP $\Rightarrow$ V

  SR, S, SP $\Rightarrow$ IV

- **Approach 2**: divide durations according to the following patterns for different right-hand sides from 1 to 3 symbols long (rules of rank 1 to 3):

  $A^t \rightarrow B^t$

  $A^t \rightarrow B^{t/2}\ C^{t/2}$

  $A^t \rightarrow B^{t/4}\ C^{t/4}\ D^{t/2}$

  As with approach 1, nonterminals are not guaranteed to be converted to terminals after a fixed number of iterations, so approach 1's mapping is taken to force any remaining nonterminals to terminals.

- **Approach 3**: Convert all non-terminals in the rule set to Roman numerals according to the same mapping used by the previous approaches. This results in the grammar shown in Table 8.2. Generation then proceeds as in approach 2, but with no need to coerce leftover nonterminals after generation has finished. This actually results in a more general grammar than the original PCFG. This approach was tested to see if certain temporal problems observed with the other two approaches would be resolved

1	I^t	$\rightarrow$	I^t
2	I^t	$\rightarrow$	$I^{t/2}\, I^{t/2}$
3	I^t	$\rightarrow$	$V^{t/2}\, I^{t/2}$
4	I^t	$\rightarrow$	$I^{t/2}\, V^{t/2}$
5	V^t	$\rightarrow$	$V^{t/2}\, V^{t/2}$
6	V^t	$\rightarrow$	V^t
7	V^t	$\rightarrow$	$IV^{t/2}\, V^{t/2}$
8	IV^t	$\rightarrow$	$IV^{t/2}$
9	IV^t	$\rightarrow$	$IV^{t/2}\, IV^{t/2}$
10	I^t	$\rightarrow$	VI^t
11	I^t	$\rightarrow$	III^t
12	I^t	$\rightarrow$	I^t
13	I^t	$\rightarrow$	$I^{t/4}\, II^{t/4}\, VI^{t/2}$
14	I^t	$\rightarrow$	$I^{t/2}\, I^{t/2}$
15	V^t	$\rightarrow$	VII^t
16	V^t	$\rightarrow$	V^t
17	IV^t	$\rightarrow$	IV^t
18	IV^t	$\rightarrow$	II^t
19	IV^t	$\rightarrow$	$IV^{t/4}\, III^{t/4}\, IV^{t/2}$
20	I^t	$\rightarrow$	$IV^{t/2}\, I^{t/2}$
21	I^t	$\rightarrow$	$IV^{t/2}\, I^{t/2}$

Table 8.2: A PTGG constructed from the PCFG in Table 8.1 using approach 3 for PCFG to PTGG conversion.

by a more general grammar.

Approaches 2 and 3 mirror the patterns of duration division used in the PTGGs shown in Chapter 4, while approach 1 makes no attempt to preserve meter. Phrases produced with approach 1 may, therefore, have a strange number of chords and end in the middle of a measure and merely substitutes a duration for the sake of fitting into Kulitta's generative framework. Approach 3 is overall most similar to the PTGGs in Chapter 4 since the terminal/nonterminal distinction is removed by allowing all symbols to be generative.

These three approaches for converting PCFGs into PTGGs were tested using the average of the learned production probabilities from the Bach chorale data set and the modified version of Rohrmeier's grammar. To evaluate the performance of the two methods, each were tested with stochastic generation using the classical approach from Chapter 6 on 20 different random number seeds. The lengths of these progressions are summarized in Table

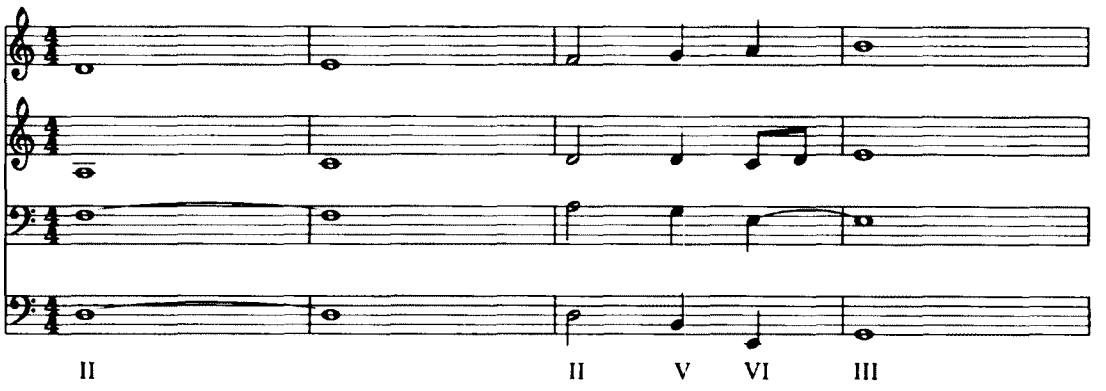


Figure 8.2: The phrase from seed 0, approach 3 from Table 8.3. Note that no I-chords were produced. The distribution of durations is also skewed due to the appearance of II-chords (which cannot create any other chords) early in the generative process.

8.3. Sample phrases from this set are shown in Figures 8.2-8.5.

Approach 1 suffers from a lack of temporal consideration, being likely to stop in the middle of a measure in a way that is quite uncharacteristic of the training data (Bach chorales). Both approaches 1 and 2 also suffer from a chord distribution problem—they are prone to becoming "stuck" on very few chords, often only a single tonic chord. Similarly, they are prone to not generating terminals.

Approaches 2 and 3 have the temporal benefits of the PTGGs in Chapter 4, although both allow strange chord transitions that are not representative of the input data. Approach 3, for which the PTGG was actually more general than the learned PCFG), demonstrated temporal behavior more like the PCFGs from Chapter 4, but it also had the most unusual-sounding transitions. Still, the results from this learning-based approach seemed on par with that of the hand-build grammars in Chapter 4—allowing for the fact that no modulations are present in the learned version. The phrases are also quite prone to ending on *V* rather than *I*, which does occur in the training data, but would present a problem if used to generate a complete piece of music where ending on *I* is required. Approach 3 also had more consistent quality of results than approaches 1 and 2. This was largely because the temporal distribution of chords and also the number of chords was more consistent between examples, lacking the big swings in density present in the the other two approaches.

Seed	Approach 1	Approach 2	Approach 3	Key
0	2	2	5	C Major
1	5	5	6	F Major
2	1	1	6	B-flat Major
3	1	1	6	B-flat Major
4	3	3	6	E-flat Major
5	1	1	9	A-flat Major
6	1	1	6	G Major
7	6	5	10	D-flat Major
8	2	2	6	F-sharp Major
9	27	8	1	B Major
10	28	9	5	B-flat Major
11	12	6	6	E Major
12	2	2	5	A Major
13	2	2	2	D Major
14	4	4	6	D-flat Major
15	1	1	6	G Major
16	1	1	6	C Major
17	3	3	13	F Major
18	1	1	9	E Major
19	1	1	10	B-flat Major

Table 8.3: Progression lengths resulting from using PCFG to PTGG conversion approaches 1, 2, and 3 on 20 random number seeds. Approaches 2 and 3 were run using a minimum duration for chords of a quarter note and a phrase length of 4 measures. All progressions were generated using 8 iterations of the generative algorithm. Progressions with only one chord are the result of $TR \rightarrow T$ occurring first. Examples of the musical phrases from seeds 0 and 7 can be seen in Figures 8.2 through 8.5.

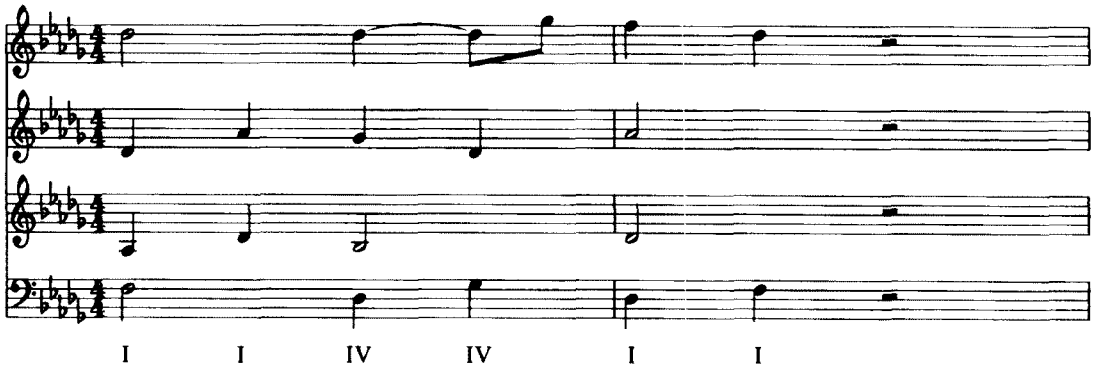


Figure 8.3: The phrase from seed 7, approach 1 from Table 8.3. Note that the phrase ends in the middle of a measure.

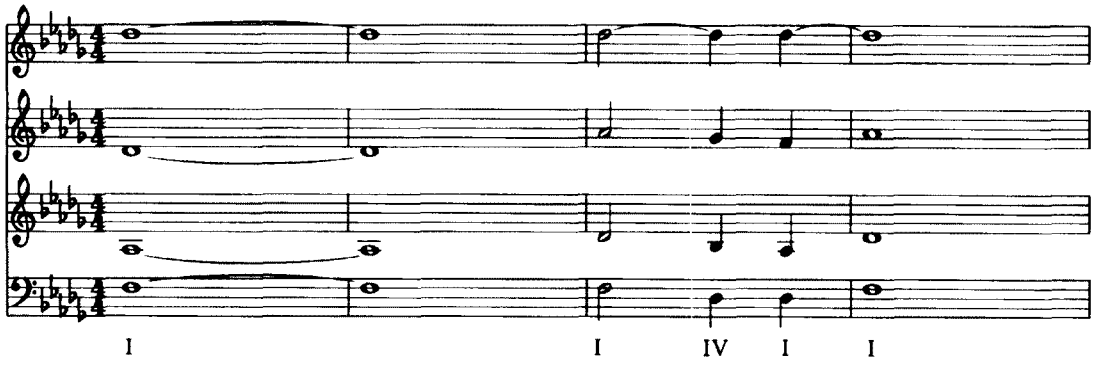


Figure 8.4: The phrase from seed 7, approach 2 from Table 8.3. Although it has I-chords, unlike the phrase in Figure 8.2, it still has a relatively poor distribution of chord durations.

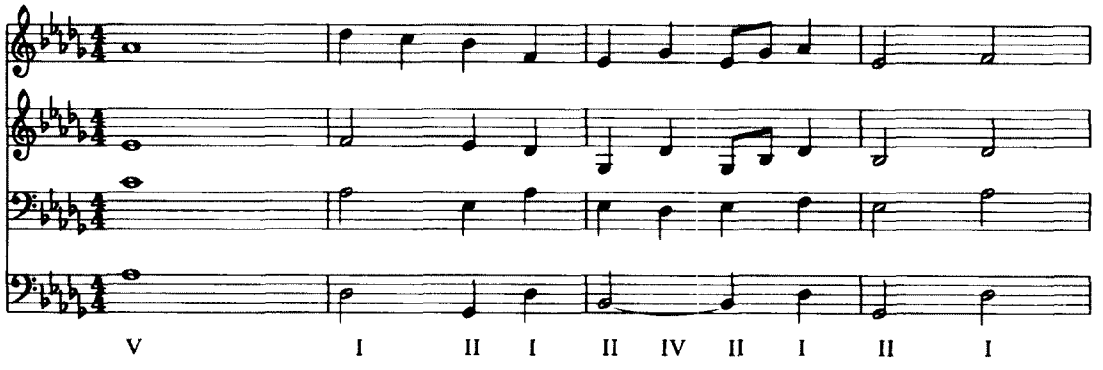


Figure 8.5: The phrase from seed 7, approach 3 from Table 8.3. This phrase has a somewhat more reasonable distribution of durations and chords than in Figures 8.2-8.4.

1	T	$\rightarrow$	T
2	T	$\rightarrow$	TT
3	T	$\rightarrow$	DT
4	T	$\rightarrow$	TD
5	D	$\rightarrow$	D
6	D	$\rightarrow$	DD
7	D	$\rightarrow$	SD
8	S	$\rightarrow$	S
9	S	$\rightarrow$	SS

Table 8.4: A further simplification of the grammar in Table 8.1.

8.1.6 Another Approach

Many of the problems in the results seen from training on the grammar in Table 8.1 stem from the possibility that generation can become "stuck" on certain chords. For example, $TR \rightarrow T$ occurring first will always result in a single-chord progression, no matter how many generative iterations are used. Although this is less of a problem with approach 3, many short progressions still occur due to productions such as $I \rightarrow III$ occurring either first or in a fairly early iteration. Additionally, when progressions are longer with approach 3, they become fairly bizarre and unrepresentative of the input data.

To get a better distribution of chords within a progression, the same data set was tested using a yet further simplification: only working with the three symbol alphabet of T, S, and D. Rules for producing series of these can be derived from Table 8.1 and are shown in Table 8.4. This grammar parsed 1,095 of the 2,650 phrases in the major data set and 582 of the 1,446 phrases in the minor data set. From this selection of each data set, samples of 200 were used with a convergence threshold of 0.03 (1% of the total probability mass). The learned probabilities over five runs for the major data set is shown in Figure 8.6. Averages of five runs over the major and minor data set are shown in Figure 8.7.

Roman numeral	I	II	III	IV	V	VI	VII
Occurrences in Bach corpus	12758	4443	1451	3689	8449	2061	1209
$T/S/D$ category	T	S	T	S	D	T	D

Table 8.5: Roman numeral frequencies in the Bach corpus.

The grammar in Table 8.4 was converted to a PTGG with the following method of assigning durations based on the rank of the rule:

$A^t \rightarrow B^t$

$A^t \rightarrow B^{t/2}\, C^{t/2}$.

This resulted in sequences of T, S, and D, which require further conversion to be used in Kulitta's generative framework. Two methods of mapping from $T/S/D$ to Roman numerals were tried:

1. **One-to-one mapping**: $T = I$, $S = IV$, and $D = V$. With this approach, the other four Roman numerals (II, III, VI, and VII) are never used.

2. **Stochastic, one-to-many mapping**: $T = \{I, III, VI\}$, $S = \{II, IV\}$, and $D = \{V, VII\}$ based on the statistical prevalence of each type of chord in the corpus (shown in Table 8.5). This was implemented by using *greedyProg* on a chord space populated with Roman numerals in the proportions shown in Table 8.5 and grouped by $T/S/D$ category.

These mappings were then run through the same chord spaces as used in previous examples in this chapter. Examples of the results can be seen in Figures 8.8-8.11.

Overall, this method performed notably better for creating a PTGG and generating musical phrases. Results from the simple, one-to-one mapping were generally consonant and had a reasonable temporal distribution of chords. However, the harmony was also very restricted. The stochastic one-to-many mapping performed similarly due to the fact that the most likely chords in each category are those from the one-to-one mapping. Harmony was slightly more diverse, but strange chord transitions were also periodically introduced, prob-

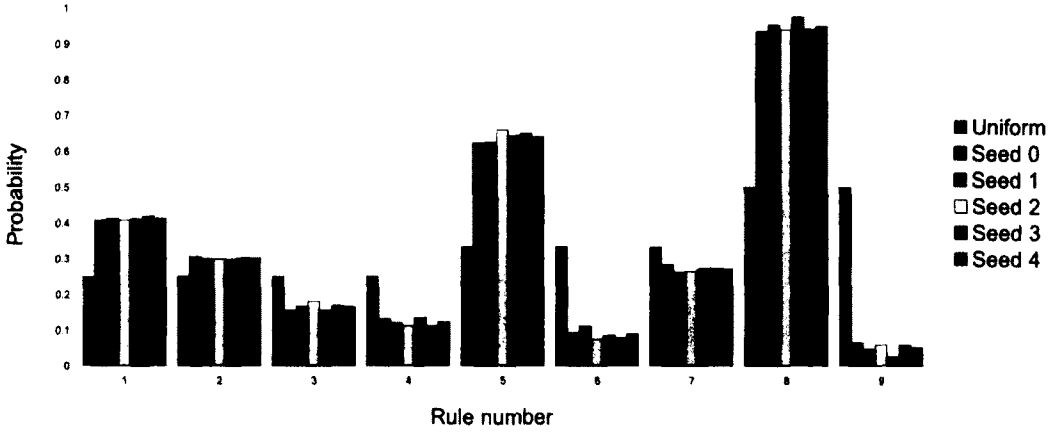


Figure 8.6: Production probabilities for the PCFG in Table 8.4 after training on major Bach chorales. The graph shows the results of five different runs, each with a different seed for randomly selecting 200 data from the corpus. All runs were given uniform initial probabilities and ran until the change in distributions between iterations fell below 1% of the total probability mass. The "average" data series represents the average of all five runs.

ably due to a combination of noise in the data and the lack of broader context-sensitivity in the one-to-many mapping step.

For the generative purposes described in Chapter 9, the process described so far on major chorales was repeated on the minor subset of the Bach corpus. There were significantly fewer minor examples, and only 582 were possible to parse. A comparison of results is shown in Figure 8.7. In general, the learned production probabilities were very similar to the major distribution, with one main exception: the probabilities for $S \rightarrow S$ and $S \rightarrow SS$.

8.2 Training on Synthetic Data

As shown in Chapter 7, the inside-outside algorithm can be extended with an oracle that allows for rule functions to be learned. However, it is not obvious how to construct a PTGG that will parse the Bach chorale corpus, so training from a real data set is problematic even though the inside-outside algorithm can handle the rule formats for PTGGs. Still, parsing on artificially generated data sets is possible and also serves as a proof of concept for the

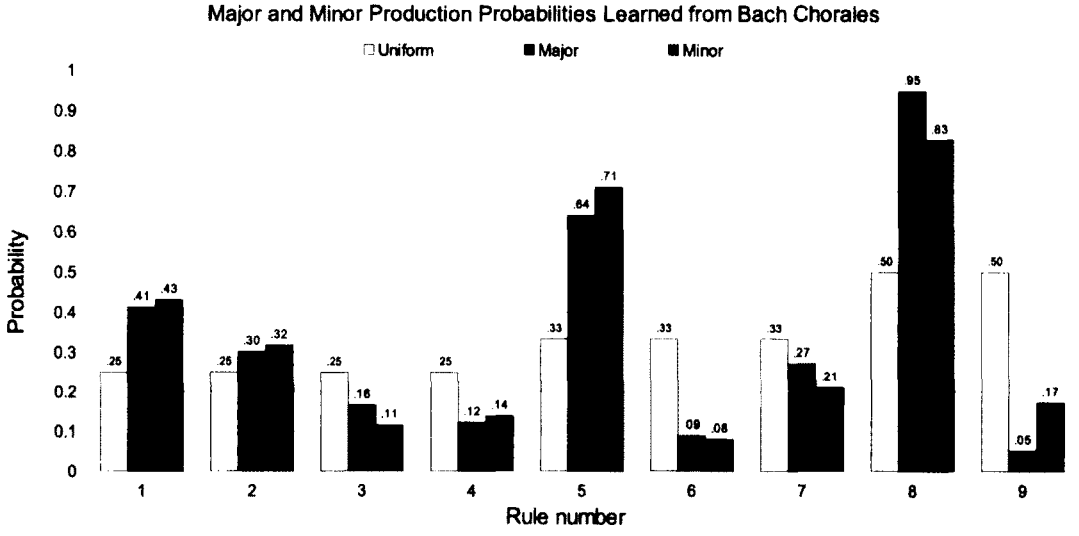


Figure 8.7: Average production probabilities for the PCFG in Table 8.4 from major and minor Bach chorales. The graph shows the average of five different runs for each mode, each with a different seed for randomly selecting 200 data from the corpus. All runs were given uniform initial probabilities and ran until the change in distributions between iterations fell below 1% of the total probability mass. The "average" data series represents the average of all five runs.

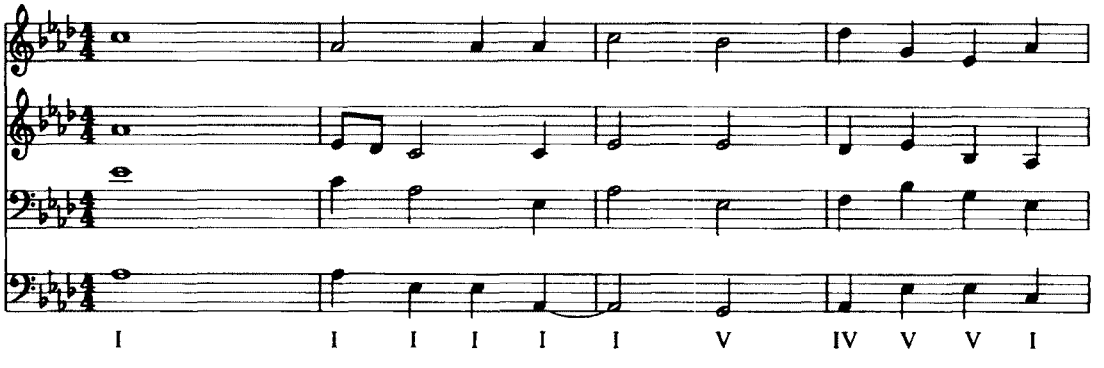


Figure 8.8: A progression generated by training the grammar in Table 8.4 on Bach chorales and using the simple, one-to-one mapping for Roman numerals.

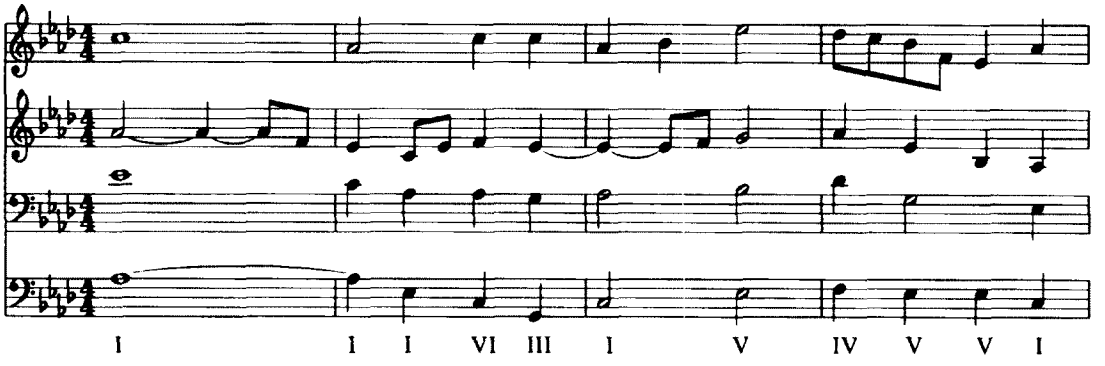


Figure 8.9: A progression generated from the same abstract progression at the $T/S/D$ level as Figure 8.8, but using the stochastic, one-to-many mapping for Roman numerals.

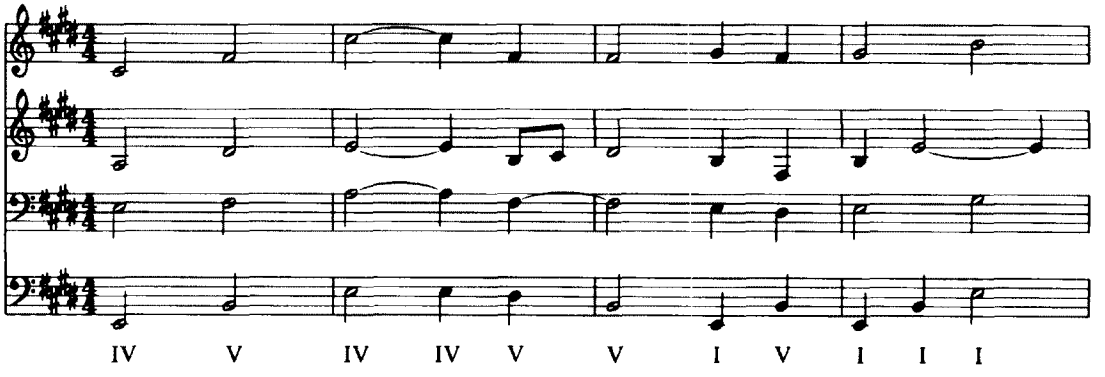


Figure 8.10: A progression generated by training the grammar in Table 8.4 on Bach chorales and using the simple, one-to-one mapping for Roman numerals.

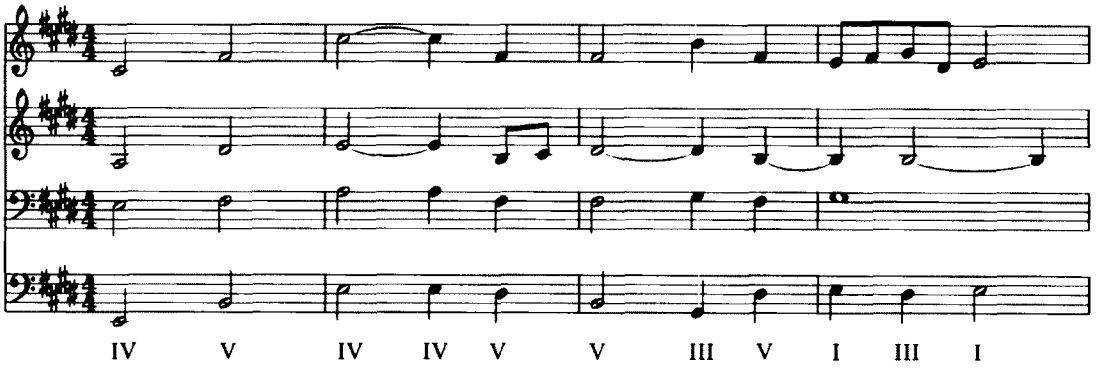


Figure 8.11: A progression generated from the same abstract progression at the $T/S/D$ level as Figure 8.10, but using the stochastic, one-to-many mapping for Roman numerals.

Num.	Probability	Rule
1	0.7	$I^t \rightarrow V^{t/2}\ I^{t/2}$
2	0.3	$I^t \rightarrow IV^{t/4}\ V^{t/4}\ I^{t/2}$
3	0.7	$III^t \rightarrow III^{t/2}\ VI^{t/2}$
4	0.3	$III^t \rightarrow I^{t/2}\ III^{t/2}$
5	0.6	$IV^t \rightarrow IV^{t/4}\ III^{t/4}\ IV^{t/2}$
6	0.4	$IV^t \rightarrow V^{t/2}\ IV^{t/2}$
7	0.2	$V^t \rightarrow IV^{t/2}\ V^{t/2}$
8	0.3	$V^t \rightarrow II^{t/2}\ V^{t/2}$
9	0.4	$V^t \rightarrow III^t$
10	0.1	$V^t \rightarrow VII^t$

Table 8.6: A small PTGG.

feasibility of learning PTGGs from corpera.

In the experiment presented here, a synthetic data set was generated using the rules shown in Table 8.6. A total of 1000 phrases was created using the PTGG rules and probabilities shown in Table 8.6 and the *gen* function from Chapter 5. Phrases were created using 4 generative iterations, a starting duration of $4w$ (a whole note in 4/4), and a minimum duration of a quarter note q.

Kulitta's learning algorithm was run using samples of 100 phrases taken from the total set of 1000. Learning was considered complete when the change in probability mass from one iteration to another fell below 1% of the total (0.04, since the rules in Table 8.6 have a total mass of 4). Phrases were selected uniformly at random using a random number seed. Learning was repeated with five different samples from random number seeds 0 through 4. The results of this training are graphed in Figure 8.12. Finally, the average of the learned probabilities over all five runs was used to generate new musical phrases. Because the grammar in Table 8.6 will create new chords at every generative step, only three generative iterations were needed to fill 4 measures with a reasonable distribution of notes (compared to 8 iterations needed for the grammar in Tables 8.2 and 8.4, which have many productions of rank 1).

8.2.1 Results

The learned production probabilities are shown in Figure 8.12. All runs of the learning algorithm converged to the same result rapidly. This is like due to the small rule set and the fact that the candidate grammar was, in fact, the grammar used to produce the data set. The learned probabilities differ from the probabilities shown in 8.6, but are nevertheless close for most of the rules. Some deviation is likely not just because of having a finite sample of data but also because the generative algorithm is not guaranteed to produce data with the exact ratios of rule applications as dictated by the production probabilities (due to psuedo-randomness and the fact that the data produced is finite). One significant deviation occurred: rules 5 and 6 have learned probabilities almost the reverse of those used to generate the data set. This is likely to be another artifact of the minimum duration threshold in the generative algorithm, since rule 5 (which produces 3 symbols) would be less likely to be usable further down in generation than rule 6 (which produces only 2 symbols).

Musically, the results from this test are more dissonant than those from the Bach corpus with PCFG to PTGG conversion. This will be partly because the PTGG used for this experiment was designed mainly to test the learning algorithm rather than to model a particular type of music well. Cearly it would be preferable to perform learning on a corpus of real music rather than synthetic data in order to obtain results that match a particular style.

8.3 Conclusion

The results from the experiments in this chapter are promising, since they indicate that results that sound similar to those from the PTGGs in Chapter 4 can be obtained by learning production probabilities from a corpus of real, human-made music. The synthetic data experiment also serves as a proof of concept that PTGGs are feasible to learn when given suitable data.

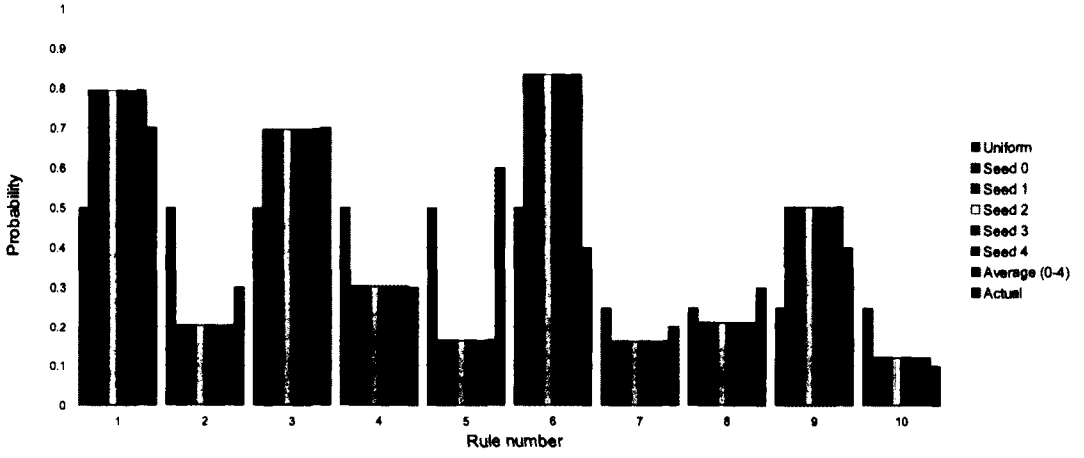


Figure 8.12: Production probabilities for the PTGG shown in 8.6 using a corpus of major phrases generated from the same grammar. The graph shows the results of five different runs, each with a different seed for randomly selecting 100 data from the corpus. All runs were given uniform initial probabilities and ran until the change in distributions between iterations fell below 1% of the total probability mass. The "average" data series represents the average of all five runs and the "actual" data series shows the values used to generate the data set (from Table 8.6).

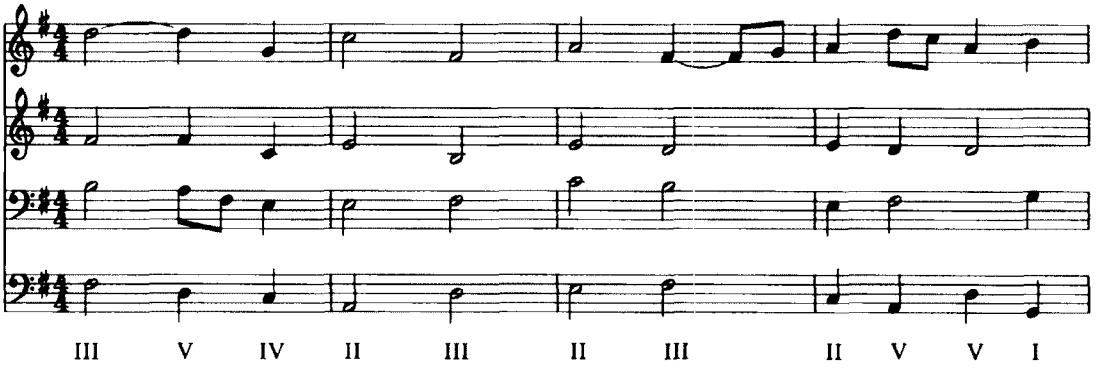


Figure 8.13: Phrase generated using the production probabilities in Figure 8.12.

The various phrases generated after training show a clear need for a temporal element in the grammar and also a need for ways to control the number of produced chords—or, rather, a way to avoid being immediately funneled into a single-chord series of productions. PTGGs like those in Chapter 4 do this well, but parsing real music with them is tricky. Small amounts of noise in the analysis could easily render a phrase unparsable by a PTGG if the durations of chords were affected, whereas a more standard, non-temporal PCFG is able to handle these types of deviations with the inclusion of a few extra rules. Currently, this actually makes PCFG to PTGG conversion a somewhat more robust approach when trying to learn features from human-made music. However, learning a PTGG directly could be equally robust if error tolerance was built into the parsing process.

Chapter 9

Empirical Assessment

There are currently no standard metrics or experimental procedures for assessing the performance of automated composition algorithms, and there are many problems associated with creation of such tests. First, what does "quality" mean in the context of an automated composition algorithm? There are many possible interpretations of quality, such as how well a work adheres to music theoretic rules, whether it sounds convincingly like it was produced by a particular composer, whether it sounds human, or whether people simply "like" it.

If we are interested in a measure like "humanness," who is to be the judge? For example, if the goal is to determine how well an algorithm can sound convincingly like J.S. Bach (without duplicating existing work), then the question must be asked: *to whom* is it convincing? The algorithm's creator, an expert, an arbitrary other person, or some combination of all of them? There is a tendency in the field of computer music to assume that a panel of music experts should always be used to determine whether an algorithm has met its compositional goals. However, there are some serious problems associated with having an algorithm be judged by such a panel.

Suppose we wish to evaluate whether an algorithm writes music that is in the style of J.S. Bach, but without duplicating existing work by the composer. The odds are stacked

against the algorithm when viewed by a panel of experts, simply because the Bach's corpus is both finite and well-known. Even if the algorithm produced something that Bach might have written, the mere fact that it is novel risks carrying a negative bias with an expert in Bach's existing music. In order to gain an unbiased result, novel chorales by Bach himself would be required for anonymous comparison against those from the algorithm.

Expert analysis in these sorts of cases is clearly problematic. However, it is also not the only means of evaluation, particularly when the questions of interest are less composer-specific. If there is such a quality as how "human" a work sounds, it is not unreasonable to assume that non-experts should be able to pick up on it intuitively. This quality may even be a factor in the likability of a piece of music, or even inseparable from it. Both may be rooted to some degree in abstract structural elements, or the anticipation of what is to come next when listening to a piece of music [39].

Expert analysis can take place visually by reading a score, which seems appealing for testing algorithms that generate musical scores without their corresponding performances. Because of the expert-related problems in comparison against famous works, however, novel features may be unfairly punished for their novelty. On the other hand, a non-expert would be unable to analyze a printed score, and requiring the non-expert to listen to a performance of the work risks conflating compositional features with performance features.

9.1 Experiment Overview

We present an experimental format that attempts to mitigate the problems discussed so far and show results from a study using this approach. Using a structure similar to a Turing test [79], participants were asked to rate their confidence or belief that a musical phrase was written by either a human or a machine. To avoid the issue of bias against certain styles, participants were first shown examples of phrases created by both humans and computers in two contrasting styles, atonal and classical chorales. Phrases used as stimuli during

the subsequent experiment came from three categories: (1) Kulitta, (2) a random-walk algorithm that will be referred to as Random, and (3) chorales written by J.S. Bach. These three phrase categories are referred to as "composers" for simplicity, despite the fact that two are algorithms.

The two computer composers, Kulitta and Random, utilized the same chord spaces and similar foreground algorithms. Phrases from all categories were rendered to audio on computer to ensure uniform performance features. Phrases used in the experiment were all four measures long, contained four voices, and were recorded at 120bpm with virtual instruments for oboe, clarinet, English horn, and bassoon assigned to soprano, alto, tenor, and bass respectively. A small amount of reverb effect was added to make the recordings sound more natural, since the instruments could sound rather harsh when dry. All notes within each phrase assigned a volume of 127 (the maximum possible value in MIDI). This resulted in "flat" performances with no dynamics or tempo variation. Because these very mechanical performances could have impacted participants' ability to judge the underlying scores, the presence of human-made phrases in the stimuli was important to serve as a sanity check. This also yielded a baseline measure of how confident the participants were that real humans were actually human under the particular performance conditions.

Two experimental conditions were used, and each only differed in the phrases taken from Kulitta. In the first condition, phrases were generated using hand-built PTGGs. The second condition featured phrases generated with models learned from Bach chorales as described in Chapter 7. The Bach and random phrases remained the same. Each participant was only allowed to take a single survey corresponding to one of these experimental conditions.

Results were analyzed within each experimental condition to derive raw score distributions for Bach, Kulitta, and the random-walk algorithm. Although the raw scores exhibited a bimodal distribution, mean scores by participant for each composer (Bach, Kulitta, and Random) exhibited a more typical normal distribution. T-Tests on these distributions

shows that all three were distinct (using $p < 0.01$) in both experimental conditions, although Kulitta's mean was closer to Bach's than it was to that of the random-walk algorithm.

T-Tests comparing distributions for the same composer between experimental conditions showed that Bach's distributions were not distinct ($p > 0.05$). Therefore, Bach was rated consistently across both experimental conditions. Kulitta's distributions between conditions were subtly different but still very similar, indicating that the learning approach performed roughly the same as the hand-built grammars. However, a T-Test of the two Random distributions fell under $p < 0.01$, which indicates distinct distributions. Since Kulitta's phrases were the only ones to change between experimental conditions, this difference suggests that perception of Kulitta's phrases may have impacted perception of the Random phrases.

9.1.1 Likert Scale

A Likert scale is a measurement strategy commonly employed in psychology experiments for measuring degree of agreement with statements. They are often used in situations where a binary classification such as "yes/no" or "agree/disagree" is considered to limited, such as when answers like "uncertain," "no preference," or "slightly agree" would be useful in addition to more extreme alternatives.

Participants in this experiment were asked to classify musical examples by using a Likert scale measuring confidence that an example was produced by either a human or a computer. The ratings used the 7-point scale shown in Figure 9.3. This type of scale was used because there is a potentially meaningful difference between someone being totally confident in a classification and merely leaning towards it but being forced to choose due to the limited options. If participants really are unsure of their choice, that information can be useful.

It was suspected at first that Kulitta might fail a more standard Turing test, and the binary classification used in a Turing test would remove some of the detail that could be

observed from a Likert scale. One of the reasons it was assumed that Kulitta might not fare well in a more standard Turing test was the fact that the human examples used were not just human-made, but also the product of an *expert* human, J.S. Bach. In a classical Turing test, participants would be asked to observe phrases from unknown sources (humans or computers) and classify them decisively as either human or computer.

A Turing test to compare Bach and Kulitta is, in some ways, like a Turing test for comparing a primitive/early chess algorithm with known shortcomings against a Grandmaster. Obviously the bar is set quite high for the machine in such circumstances, since the machine's performance will inevitably be compared to that of the expert in successive trials. Therefore, if the goal is to determine whether the machine performs like an arbitrary human, the test is really rather unfair. Nevertheless, even if Kulitta performed miserably when compared to human composers, the Likert scale would still provide more information on the algorithm's performance than would be the case for a binary classification method.

To illustrate the information captured by a Likert scale that is lost with binary classification, consider two hypothetical algorithms, A and B. Suppose that both would both would fail a Turing test by being classified as a machine 100% of the time. That does not mean that the two algorithms performed equally during the test. In fact, the two algorithms could still show differences when using the Likert scale used in this experiment. If algorithm A scored closer to the middle of the scale (but still on the computer side) while algorithm B was classified firmly as a computer, that difference is meaningful and would indicate that algorithm A was closer to exhibiting human-like behavior than algorithm B. Using binary classification, there is no room to allow participants to express doubt or uncertainty in their answers, and so information on how swayed a participant is in one direction or the other is lost.

9.2 Musical Phrases

A total of 40 phrases, all lasting about 10 seconds, were recorded for this experiment. 10 phrases were taken from Bach chorales, 10 from the random walk algorithm, and 20 from Kulitta. Kulitta's phrases occupied the majority of the examples because Kulitta has a fairly diverse range of behavior. Therefore, it was important to try to capture a representative sample while keeping the total number of trials reasonable, since long experiments can cause participants to become frustrated, bored, or otherwise fatigued.

9.2.1 Phrases from Kulitta

Two versions of Kulitta were tested: one using hand-built grammars and one trained on Bach chorales. Both versions used the same OPC-space for four voices (using the ranges shown in Table 9.1) and the simple foreground algorithm described in Chapter 6 to add melodic elements in a classical, chorale-like style.

In both cases, Kulitta used a minimum duration of a quarter note to generate abstract structure for 4-measure long phrases with a 4/4 meter. Additionally, the starting structure of the phrase was also varied uniformly at random to be one of the following, where w indicates a whole note (the duration of one measure in 4/4):

- I^{4w}
- $I^{2w}\ I^{2w}$
- **Let** $x = I^{2w}$ **in** $x\,x$

The bass was forced to double either the root or the fifth for major and minor chords and the root for diminished chords. This was accomplished by using a single-chord predicate that checked for OPC-equivalence to $\langle x,x,x+m,k+7\rangle$, $\langle x,x+m,x+7,x+7\rangle$, or $\langle x,x,x+3,x+6\rangle$ for $x \in [0,11]$ and $m \in \{3,4\}$. OP-space mapping was done using *greedyProg'* from Chapter 5 and a filter over equivalence classes that selected for the bass doubling the

Voice	Pitch number range	Pitch range in Euterpea
Soprano	[60, 81]	(C,5) to (A,6)
Alto	[52, 76]	(E,4) to (E,6)
Tenor	[47, 67]	(B,3) to (G,5)
Bass	[40, 60]	(E,3) to (C,5)

Table 9.1: Voice ranges used for Kulitta's phrases. "Middle C" or $(C,5)$ is pitch 60.

root only with 80% probability. Foregrounds were added to the phrases using the classical foreground approach described in Chapter 6. Phrases were generated in both major and minor. In each condition (hand-built vs. trained), 10 major and 10 minor examples were taken. Of each modal group, 8 progressions ending on I and 2 progressions ending on V were selected at random.

Phrases from Hand-Built Grammars

Two hand-built rule sets were used: the rule set shown in Table 4.2 and an additional rule set modified to end on V instead of I. This additional V-ending rule set was added because some of the phrases taken from Bach chorales ended on V. For each mode and grammar, 20 phrases were generated, each with a different random seed. A home key was chosen uniformly at random for each abstract phrase. From the total of 80 phrases generated, a smaller sample was selected for the experiment. This smaller sample consisted of 16 phrases ending on I, half of them major and the other half minor, and 4 phrases ending on V, again half major and half minor. Within each ending/mode category, phrases were randomly chosen out of the 20 that were generated. Details on the distribution of starting structures can be found in Table9.2.

Phrases from Trained Model

Kulitta was trained on Bach chorales using the approach described in Chapter 7 that used a 3-letter alphabet: T (tonic), D (dominant), and S (subdominant). Alphabets were expanded to Roman numerals using the one-to-many approach described in Chapter 7 to obtain a

Mode	Ending Type	Starting Structure	Hand-Built Phrases	Trained Phrases
Major	I	I^{4w}	1	2
Major	I	$I^{2w}\ I^{2w}$	6	5
Major	I	**Let** $x = I^{2w}$ **in** $x\ x$	1	1
Major	V	$I^{2w}\ I^{2w}$	2	2
Minor	I	I^{4w}	1	2
Minor	I	$I^{2w}\ I^{2w}$	3	2
Minor	I	**Let** $x = I^{2w}$ **in** $x\ x$	4	4
Minor	V	I^{4w}	1	0
Minor	V	$I^{2w}\ I^{2w}$	1	1
Minor	V	**Let** $x = I^{2w}$ **in** $x\ x$	0	1

Table 9.2: Distribution of starting structures in Kulittas phrases for emperical evaluation.

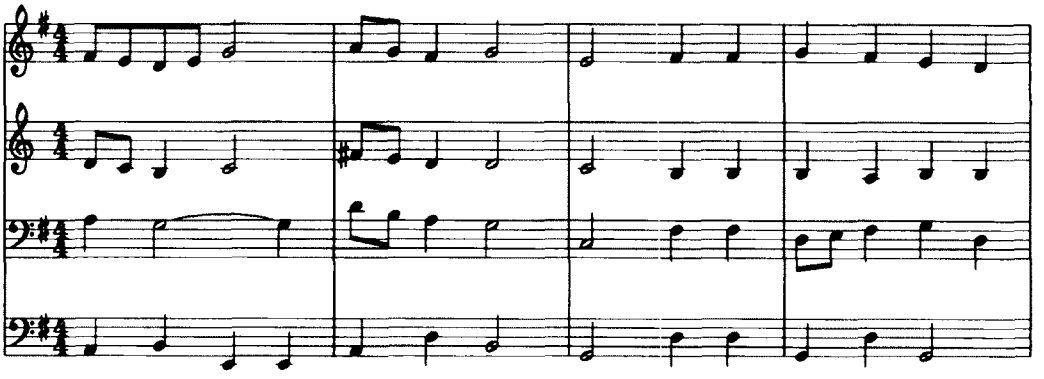


Figure 9.1: An example of one of the phrases used in the experiment that was generated by Kulitta using a hand-built grammar. It is in G-major and uses the grammar from Table 4.2, ending on I.

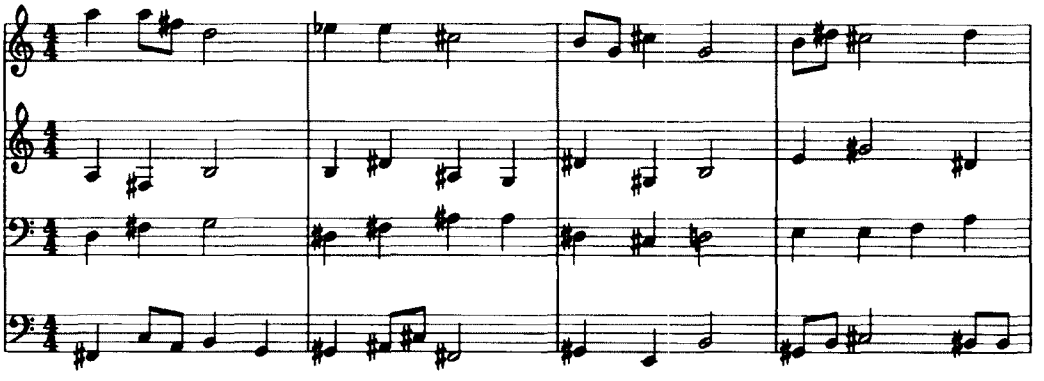


Figure 9.2: An example of one of the phrases generated by the random walk through chord spaces.

distribution of Roman numerals similar to that of the training data. Once mapped to an expanded set of Roman numerals, the phrases were interpreted in the same way as those from the hand-built grammars, using the same chord spaces and foreground algorithm.

9.2.2 Randomly Produced Phrases

The Random phrases were intended to provide another computer-generated point of comparison, but using a simpler approach than Kulitta. Instead of using a grammar as the source of abstract harmonic structure, the algorithm performed a stochastic walk through the chorale chord space described in section 9.2.1 using the greedy algorithm described in 5. This resulted in very dissonant phrases, since they were created with no notion of key. Foreground elements were added stochastically using an approach similar to the one described in Chapter 6 for classical foregrounds, but without the scale-related constraints on choosing non-chord tones as foreground elements.

The main difference between Random and Kulitta is in the abstract musical structure: Kulitta's phrases have a hierarchical structure, while Random's did not. The Random phrases were texturally similar to Kulitta's phrases due to the similar foreground algorithm. However, they differed sharply by the lack of a clear tonal center and therefore more diverse transitions between chords. An example of a randomly-produced phrase can be seen in Figure 9.2.

9.2.3 Phrases from Bach Chorales

A set of 10, 4-measure-long phrases were taken from some Bach chorales and are listed in table 9.3. Most of the phrases ended on I, but some did not. Chorales and individual phrases were selected by the author with a few selection criteria: chorales needed to be in 4/4, have at least one 4-measure phrase, and relatively simple foregrounds (i.e. no trills) that were texturally/rhythmically similar to the Kulitta and Random phrases.

Phrase	BWV Number
1	12
2	18
3	39
4	67
5	293
6	293
7	334
8	64
9	101
10	101

Table 9.3: List of Bach phrases showing source chorale numbers.

9.3 Experimental Procedure

The experiment was run online and implemented using a combination of Javascript, PHP, and HTML5 to control transitions between stimuli, randomize the order of the audio examples, deliver audio in an automated way, and record participants' responses. The experiment's interface was a slideshow-like format where participants were moved from one screen to another either automatically or when clicking a button.

The experiment was run using Amazon Mechanical Turk[10, 40, 72] (MTurk) as a source of participants. Data from a total of 237 participants was obtained on MTurk, which 121 in the first category (Kulitta with hand-built PTGGs) and 116 in the second (Kulitta trained on Bach chorales). Participants in this study had various levels of education and musical training, were taken from the United States only, and included non-composers as well as some composers.

After viewing the consent form, participants first entered their MTurk ID and then pressed a button to continue. They were then given another button to press to play a sound, specifically a chord produced by the virtual woodwind instruments at MIDI volumes of 127 each. They were asked to adjust the volume of their audio system using this chord as a reference. Participants had to press the button to hear a sound at least once before continuing, but had the option to press the button as many times as needed for volume adjustment.

Style	Human Phrase Source	Computer Phrase Source
Atonal	"Six Little Piano Pieces: Rasch, aber Leicht" by Arnold Schoenberg	An L-System similar to those in Euterpea
Chorale	A chorale by J.S. Bach (BWV NUMBER)	A chorale by David Cope's artificial intelligence algorithm, Emmy [23]

Table 9.4: Labeled examples presented to participants at the beginning of the experiment.

After the volume setup was complete, participants were taken to a set of labeled examples that were not part of the experimental trials.

Labeled Examples

After the volume adjustment step, participants were given four example phrases to listen to in two different styles: modern/atonal and chorales in the style of J.S. Bach. The purpose of these examples was to encourage listeners to think about the diversity of results they might encounter in the experimental trials and to be aware that more than two distinct sources of music could be involved. The examples were labeled truthfully as being written by either a computer (algorithm) or a human, but only to that level of detail—information on authors and composition titles were not included. All examples were approximately the same length (about 10sec) and rendered to audio using a virtual piano instrument. The virtual woodwinds were not used because not all of the examples contained exactly four voices. The examples are described in Table 9.4. Once all four examples had been played at least once, participants were allowed to continue to the series of 40 experimental trials.

Experimental Trials

Following the training examples, participants were instructed that they would be given 40 phrases to listen to, would only be able to listen to each phrase once, and would rate their confidence that each was written by either a human or a computer. Participants were asked to rate examples quickly after hearing them while the examples were still fresh in their

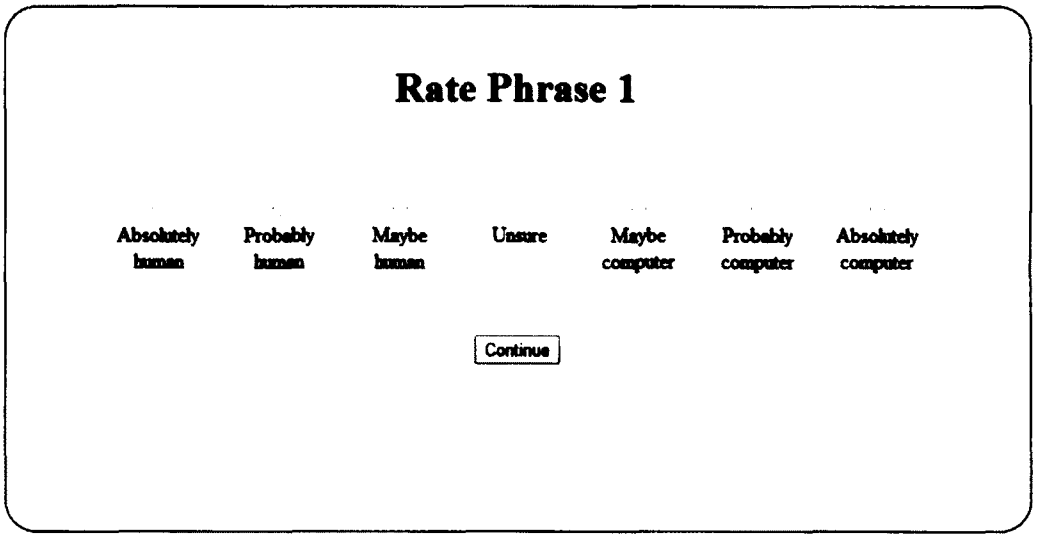


Figure 9.3: An example screen shot from the experiment. Having just heard a phrase (in this case the first phrase of 40) participants were asked to rate it using a 7-point Likert scale.

minds. Phrases were played automatically; once playback had finished, participants were asked to rate the example. Participants had to select a rating in order to continue to the next example by pressing a button. The rating could be changed as many times as desired before continuing. The scale presented is shown in Figure 9.3.

For each phrase in the experiment, the following items were recorded and written to a file:

- Loading time of the page playing the phrase
- Trial number
- Audio file played
- Rating given by the participant

Ratings were recorded using a scale of 0-6 corresponding to the labels shown in Figure 9.3, where 0 was "absolutely human" and 6 was "absolutely computer." The loading time of the first page of the survey was also recorded to be able to compute the total time spent on each of the 40 phrases. After listening to and rating each of the 40 phrases, participants were asked to complete a short survey on demographics and musical background.

Participant Demographics		
Value	**Condition 1**	**Condition 2**
Minimum Age	19	20
Maximum Age	67	74
Average Age	35	35
Male	65	67
Female	48	49
Other Gender	1	0
Undefined Gender	7	0

Table 9.5: Basic demographics for participants by experimental condition. The "undefined" cases in condition 1 were due to a browser compatibility-related data collection problem on one of the pages of demographics questions (all other data for the experimental trials was still recorded for these participants)

Demographics and Musical Experience

Standard demographics were collected for participants along with information on their musical training. Demographics information was collected on age, gender, ethnicity, nationality, first language, and highest level of education. Musical background information collected included whether the participants played an instrument, how many years of musical training they had, and whether they had taken a music theory course.

9.4 Results

A total of 237 people participated in the experiment on MTurk. 121 participants took the experiment under condition 1, where Kulitta's phrases were generated from hand-built grammars. The remaining 116 participants were in condition 2, where Kulitta used a grammar with production probabilities derived from Bach chorales. Some basic demographics are summarized in Table 9.5.

Participants took an average of 18.2 seconds per trial. Participants in condition 1 (121 participants) took an average of 18.4 seconds and those in condition 2 (116 participants) averaged 18.0 seconds. In addition to time spent making a decision, this number includes the 10 seconds during which the trial's phrase would be played along with any additional

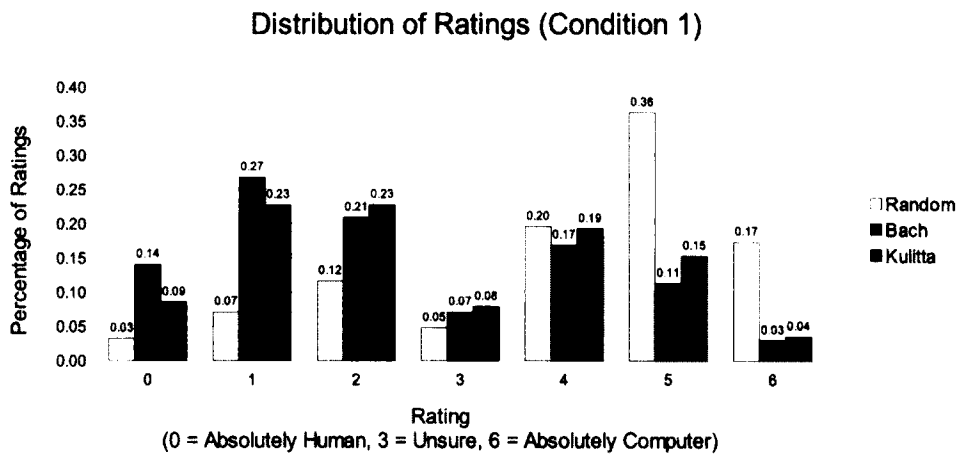


Figure 9.4: Distribution of raw scores for condition 1 of the participant study (hand-built grammar).

latency factors, such as a slow connection causing delays when loading audio files. Subtracting the playback time, this means that participants took an average of 8.2 and 8.4 seconds to respond to each trial, which is relatively fast.

A summary of raw scores as a histograms is shown in Figures 9.4 and 9.5. Participants' average scores for each composer are shown as a histogram in Figures 9.6 and 9.7. Both representations show that there are differences in the scores for each musical source, but the patterns exhibited by the Bach and Kulitta scores are more similar than either is to the random walk.

Table 9.6 shows p-values from performing a paired, two-tailed Student T-Test on the averages of participants' scores for all three categories. All comparisons are statistically significant ($p < 0.01$), indicating that the three categories of music yielded distinct distributions of scores.

As expected, the Bach phrases' scores averaged closest to 0 ("absolutely human") and Random's scores averaged closest to 6 ("absolutely computer") with Kulitta's results falling in the middle. Overall average scores for each composer are shown in Table 9.7. Notably, Kulitta's scores fall much closer to Bach's than to the random walk's. Music training

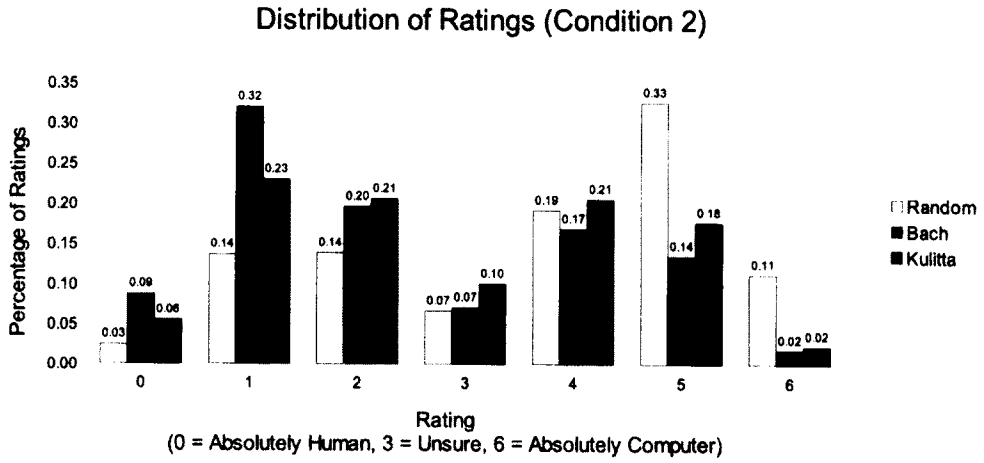


Figure 9.5: Distribution of raw scores for condition 2 of the participant study (trained on Bach chorales).

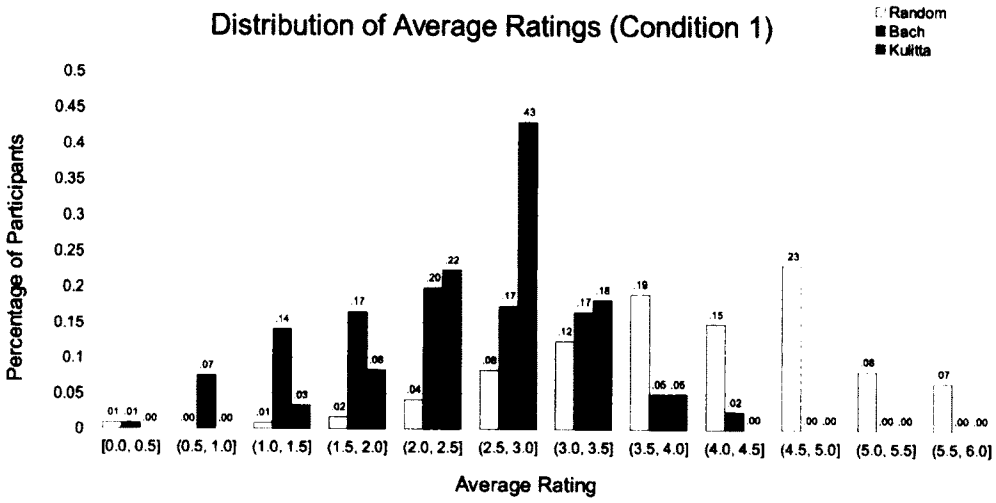


Figure 9.6: Distribution of average scores for condition 1 of the participant study (hand-built grammar).

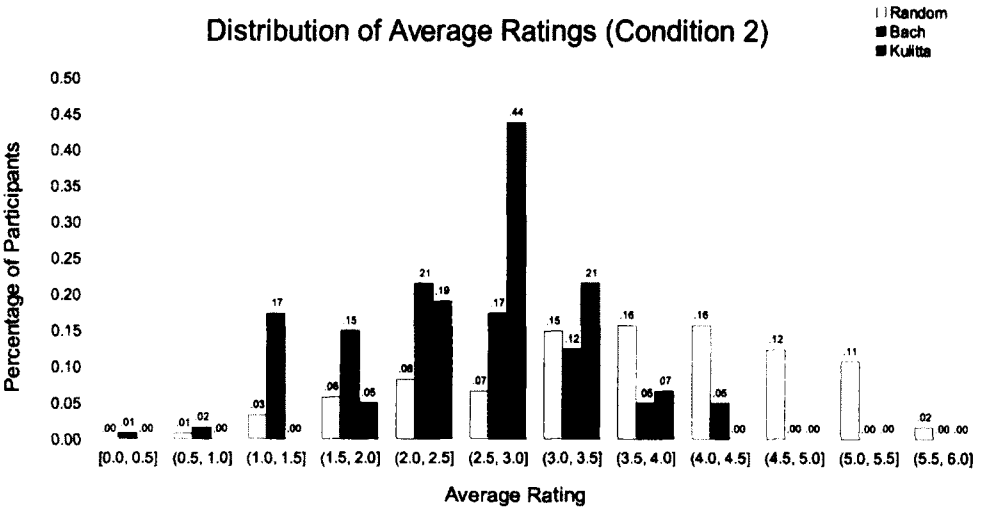


Figure 9.7: Distribution of average scores for condition 2 of the participant study (trained on Bach chorales).

T-Test Comparisons			
Condition	Random/Bach	Bach/Kulitta	Random/Kulitta
1 (Hand-Built)	4.83×10^{-20}	1.06×10^{-5}	1.14×10^{-22}
2 (Trained)	1.07×10^{-11}	4.70×10^{-7}	7.24×10^{-11}

Table 9.6: P-values from paired, two-tailed T-Tests to compare average scores of each composer. All fall well below the typical threshold of $p < 0.01$, indicating that the distributions are different.

and music theory training showed no correlation with the average scores given to each composer ($R^2 < 0.1$ for all comparisons).

T-Test comparisons of each composer to itself across the two conditions are shown in Table 9.8. As seen in the table, Bach was scored consistently (a large p value), but the Random distribution changed ($p < 0.01$). It may be that the differences in Kulitta's

Average Scores			
Condition	Random	Kulitta	Bach
1 (Hand-Built)	4.09	2.67	2.32
2 (Trained)	3.70	2.80	2.39

Table 9.7: Average scores for each composer across all participants. Kulitta averaged both on the human side of the scale (< 3.0) and closer to Bach than to Random. Kulitta and Bach differed by 0.35 and 0.41 in conditions 1 and 2 respectively. Kulitta and Random differed by 1.42 and 0.90, and Random and Bach differed by 1.77 and 1.31.

Within-Composer T-Tests	
Bach	0.506
Kulitta	0.0413
Random	0.00748

Table 9.8: T-Test comparison of composers across experimental conditions. Large p values are not considered significant. The results indicate that Bach was scored consistently, while there is a clear difference ($p < 0.01$) for the Random distributions between the two experimental conditions. Kulitta's distributions may or may not have been different.

behavior from one condition to the other (hand-built vs. trained on Bach chorales) affected participants judgments of the Random cases. In the second condition (trained on Bach chorales), Kulitta exhibited some strange chord transitions that, to the author, did sound somewhat more similar to the Random phrases than was the case in the first condition. In other words, when the version of Kulitta trained on Bach chorales made a "mistake," it created chord transitions that were perhaps more similar to the Random phrases than when non-learning version of Kulitta made a mistake. Whether Kulitta's score distributions differed between the two experimental conditions is somewhat unclear. While the p-value of 0.0413 is within the standard of $p < 0.05$ used in many psychology studies, it is very close to the threshold and not nearly as strong an effect as exhibited in Table 9.6 ($p < 0.01$ in all cases).

9.4.1 Discussion

Kulitta's phrases performed surprisingly well in this experiment. The fact that Kulitta averaged on the human side of the scale suggests that the system may have passed a more standard Turing test with a binary answering scheme rather than a Likert scale. However, the Likert scale provided important information about Kulitta's performance that would have been lost with a binary system. Kulitta's placement relative to the Bach and Random scores shows that, although Kulitta was clearly more similar to Bach, there is plenty of room for improvement.

Some interesting observations can be also be made about the other two composer's

phrases from this experiment. Although lowest-scored on average (most human), Bach's phrases did not score consistently on the human side of the scale and some phrases scored consistently higher (more computer-like) than others. It may be that, as suspected before the experiment was run, that the "flat" or un-expressive performance of the phrases led to participants doubting that some Bach phrases were written by a human. Alternatively, it may be that a lack of larger context for the phrases makes them more difficult to interpret. These hypotheses could be tested using experimental designs similar to this one.

The random phrases had the highest scores on average (most computer-like), which was expected given the lack of structural consideration when generating the phrases. However, the phrases did have a similar texture to the Bach and Kulitta phrases as well as brief passages that were more tonal-sounding purely due to chance. As a result, it is not surprising that there was a range of scores for the random walk phrases that sometimes wandered onto the human side of the scale.

An unexpected outcome of the study was the bimodal nature of the Likert scale scores, even though the distribution of averages was still normal. Although avoiding the extreme ends of a Likert scale is a well-known phenomenon, avoidance of the middle suggests that the scale may have actually functioned as two separate Likert scales joined at the middle. Further testing would be needed on the exact wording and organization of the rating system to determine whether this was the case.

Chapter 10

Conclusion

Kulitta is a modular framework for automated composition that has demonstrated potential with two distinctly different styles of music. Kulitta breaks down the compositional process into a series of generative steps using musical grammars, chord spaces, and constraint-satisfaction algorithms before adding style-specific foreground elements—melodic and rhythmic features as they would appear on a musical score. Kulitta also features a learning module that allows production probabilities for a musical grammar to be learned from a corpus.

10.0.2 PTGGs and Chord Spaces

PTGGs are a new category of musical grammars that use an alphabet of chords parameterized by duration, resulting in an alphabet that is technically infinite. This parameterization allows a PTGG to capture both abstract harmonic structure as well as metrical constraints. Rules are functions of duration that can exhibit conditional behavior as well as modal context sensitivity. PTGGs also have a feature that is normally associated with programming languages: **let-in** expressions that capture the notion of repetition through variable declaration and instantiation. This helps to capture an aspect of self-similarity in music that is missing from many other proposed musical grammars. While the features of PTGGs in-

crease parsing difficulty, parsing data with them and learning production probabilities for their rules (which are really functions) is still feasible.

With a very simple generative algorithm similar to what would be used for an L-System, PTGGs are able to generate abstract harmonic structure in music that is suitable for a variety of styles of music. The conditional behavior of the rules both helps to reduce redundancy in the rule set as well as ensuring a reasonable distribution of durations. When combined with chord spaces and constraint-satisfaction algorithms, PTGGs can be used to create a complete musical score.

Chord spaces are a way of grouping chords (collections of pitches) in musically meaningful ways. Mathematically, they are the result of applying an equivalence relation to a set of chords to form a quotient space (a "gluing together" of related points). Many chord spaces can be defined according to music theoretic concepts and geometric transformations. Kulitta makes use of several different chord spaces to convert abstract progressions from PTGGs into concrete chord progression for both Western classical music and jazz. This transformation from abstract to concrete is a path-finding problem that requires a constraint-satisfaction algorithm to capture desirable musical behaviors.

10.0.3 Constraint Satisfaction

Constraint-satisfaction algorithms are needed to turn abstract progressions from PTGGs into concrete chords while enforcing the presence or avoidance of various concrete musical features. For a classical chorale, voice-leading should be relatively smooth and both parallel motion and voice-crossing should be strictly avoided. Some of these constraints can be captured by filtering the quotient space before it is traversed, but others such as those concerning pairs of chords (as is the case for avoiding parallel-motion and large leaps) require satisfying constraints during path-finding.

Kulitta's constraint-satisfaction techniques are rather basic: depth-first search; a single-pass, greedy search; and a search for satisfying **let** expressions. These algorithms perform

well on fairly simple musical constraints, such as voice-leading constraints characteristic of chorales. However, the harder musical constraints become to satisfy, the more these algorithms become intractable, in the case of the depth-first and let-satisfying searches, or unlikely to find a good solution in the case of the greedy algorithm.

The main benefit of Kulitta's greedy search is that it runs quickly, fast enough that it is possible to use in an interactive setting. It will also always produce a result, even if that result violates some constraints. On the other hand, the other two search strategies, which are fundamentally depth-first, are unsuitable for any sort of application where a solution must be found rapidly. They might get lucky with a particular chord space and find a solution quickly, or they might end up searching through billions of solutions when a perfect solution may not even exist.

The time vs. quality trade-off of producing a quick but imperfect result or performing a slow search for a perfect solution are far from unique to music. It is a problem that plagues any area of computer science with a large search space. Many decision-making problems in AI suffer the same trade-off, and the problem is amplified in a real-time setting.

10.0.4 Music Generation

Kulitta is able to generate chorale-like, classical music in a style resembling J.S. Bach by using PTGGs derived from existing analyses of Bach's work, chord spaces, and a foreground algorithm to add basic melodic elements such as passing and neighboring tones. The system's modularity allows for the style of music to easily be changed by utilizing different chord spaces and a different foreground algorithm. Jazzy harmonies can be produced by either reinterpreting classical progressions or mapping Roman numerals through a mode space with jazz chord templates and then through OPC-space for some number of voices.

One of Kulitta's notable features is the ability to blend styles through the use of different chord spaces and foreground algorithms. Kulitta's modularity allows for multiple

chord spaces to be combined in different ways and for foreground algorithms to be used interchangeably. For example, a "jazz chorale" can be created simply by passing through an additional chord space, mode space, between the classical chord space and classical foreground steps in generation.

The generative algorithms used by Kulitta suffer from one major limitation: they must always move forward and produce a result. In some ways, this makes a qualitative comparison against human-made music exceptionally difficult for Kulitta, since humans are generally not held to the same, linear type of workflow. Consider writing a lengthy passage of text with a pen and paper in one pass. Mistakes in the result will obviously be more likely under those constraints than if the same task was performed with a pencil, eraser, and proof-reader. In general, outside the special settings such as academic exams, humans working at the level of a paper score—or a passage of text—are free to abandon partially-complete work when it becomes problematic. Human composers can also rework ideas multiple times in the process of trying to create a desirable result, just as a writer will iteratively edit wording and punctuation. Kulitta does not currently have this luxury, and the compositional process for every given random seed must be carried through to completion in a single pass. It is therefore not at all surprising that there is a fairly wide range of quality in the output.

10.0.5 Learning

Kulitta uses an extended version of the inside-outside algorithm to learn musical PCFGs as well as PTGGs. The inside-outside algorithm must be modified to handle rules of higher rank (the number of symbols on the right-hand side of rules) and to handle the distinction between rule functions and rule instances that exist for PTGGs. The latter is done by the use of an oracle that allows a CYK-style parse table to be derived for a particular category of grammars.

Learning for musical PCFGs was tested by using a corpus of major phrases from Bach

chorales using modified and simplified versions of Rohrmeier's grammar for harmony [67]. Production probabilities for this grammar were iteratively re-estimated until the change in probability mass fell below a 1% of the total. The average of five runs of the learning algorithm over different subsets of the Bach corpus was used to generate new phrases of music. To do this, the PCFGs were converted to PTGGs by three different means. Although phrases generated by two of those approaches suffered problems related to the PCFG's lack of support for metrical structure, the third produced phrases that were qualitatively very similar to those produced by the hand-built grammars described in Chapter 4.

As described in Chapter 7, PTGGs have difficulty parsing real musical data. This is due to the strict nature of the temporal divisions of PTGGs like those in Chapter 4. Musicians are rarely to strict with their temporal decisions, so training data would either need to be coerced into a suitably strict form or the temporal constraints of the PTGG would need to be softened during the parsing process. In the absence of such heuristics, it was not possible to test a PTGG with the Bach corpus. Instead, the performance of the learning algorithm was evaluated using a synthetically generated data set, supplying the same PTGG to the learning algorithm that was used to generate the data set but with uniform initial probabilities instead of the actual probabilities used to generate the data set. In general, the actual probabilities were recovered successfully, with exceptions likely being due to deviations introduced by limitations on the generative algorithm.

10.0.6 Empirical Assessment

A participant study was designed and conducted on Amazon's Mechanical Turk to evaluate Kulitta's performance compared to J.S. Bach and a random walk algorithm. Participants were asked to listen to a random permutation of 40 phrases (10 from Bach, 10 from the random-walk, and 20 from Kulitta). After hearing a phrase, participants were asked to rate their confidence or belief that the phrase was written by a human or computer using a 7-point scale from "absolutely human" to "absolutely computer."

The results of the study showed that Kulitta performed quite well, averaging on the human side of the scale used, although still close to the middle of the scale. Kulitta scored more similarly to Bach than to the random walk algorithm. So far, only Kulitta's chorale phrases have been evaluated using this empirical approach. Whole pieces as well as other styles, such as jazz or perhaps more complex styles of classical music, should eventually be evaluated in a similar manner.

10.1 Future Work

Figure 10.1 shows an example of possible improvements to Kulitta's workflow. Both the generative and learning aspects of Kulitta have many avenues of possible future work: handling more musical features in PTGGs, improved constraint-satisfaction algorithms, and learning new musical features.

10.1.1 PTGGs and Constraint Satisfaction

The grammars used in Kulitta, PTGGs, have the potential to be expanded to include representations of more musical features. Currently only repetition is captured, but other concepts such as variations and perhaps other musical transformations such as retrograde and inversion could also be added. Context-sensitivity may also be useful in avoiding undesirable harmonic transitions.

Kulitta's constraint satisfaction algorithms are currently somewhat basic, with the greedy approach, *greedyProg*, being the only stochastic search. In artificial intelligence, stochastic search algorithms are often employed as a means to traverse large solutions spaces that would otherwise be very difficult to traverse in a depth-first manner. Tactics like simulated annealing (such as the approach used in Boltzmann machines [4, 5]) and Monte Carlo search [8] may yield better performance over large chord spaces than Kulitta's current search methods.

It would also be useful to allow some sort of bi-directional workflow during generation. Given an analysis algorithm to detect problematic musical features, the overall quality of Kulitta's results could obviously be raised if random number seeds that produced undesirable results could be identified "thrown out" at the first sign of a problem rather than pressing ahead to try to turn them into concrete music. Similarly, introduction of a backtracking approach to rework ideas that only have minor problems would allow for improved performance.

10.1.2 Learning New Musical Features

Kulitta's learning component has the potential to be greatly expanded. It would be ideal to learn more musical features than just production probabilities for an existing grammar. The most obvious extension would be to learn the grammar itself, deriving a candidate collection of rules from a corpus for which production probabilities can be learned using the same data.

Additional chord spaces would also be useful to learn from data sets rather than being defined by hand. For example, Bach sometimes used altered versions of triads that, if analyzed to the correct Roman numeral, are not currently possible for Kulita to create due to the particular chord spaces used during generation[1]. Approaches similar to that of Quinn and Mavromatis to learn harmonic function [65] or White for chordal alphabet reduction [82] could be used to accomplish this. Chord substitution in jazz is another example of a feature that may be possible to learn from a data set and would likely be best modeled in a way similar to mode space, by tagging chords with additional contextual information. It may also be useful to learn weights or probability distributions over chord spaces to favor some chords over others during generation.

The ability to learn foreground behavior would also be a beneficial extension to Kulitta,

1. Diminished chords, II in a minor key and VII in a major key, often sound inappropriately used in Kulitta's progression. This is probably due to the chord spaces not containing more style-appropriate versions of those chords.

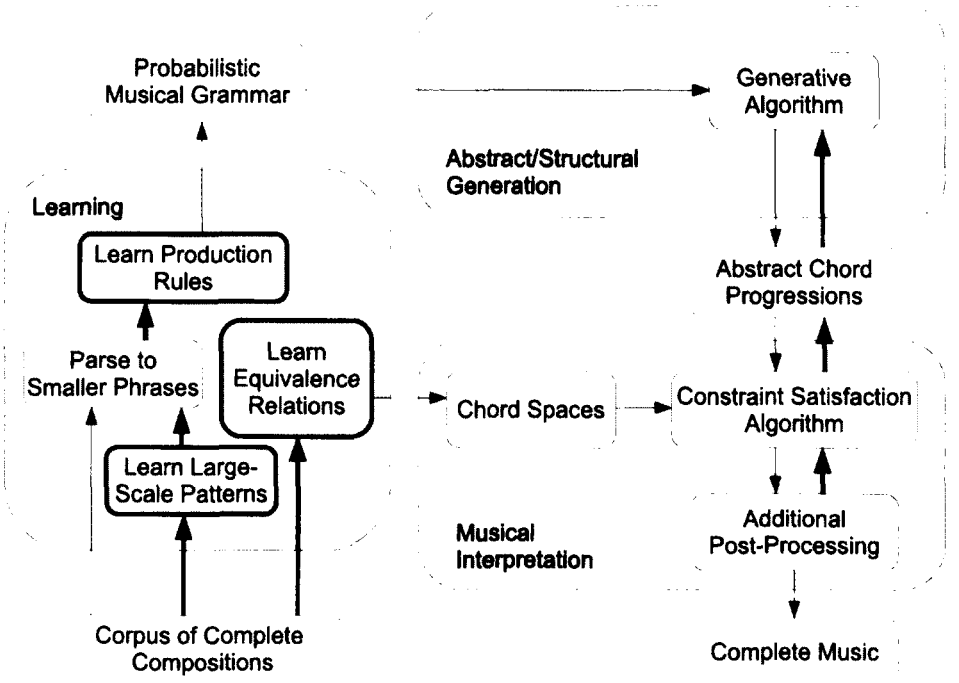


Figure 10.1: An example of a possible extension to Kulitta, featuring a more extensive learning component and backtracking or bidirectional workflow between the generative steps. Bold lines indicate new features that are not currently included in Kulitta.

since defining individual styles by hand is both time-consuming and difficult to do in a generalized way. This would require new data sets and new analysis algorithms to effectively handle different types of music.

Finally, patterns of repetition and variation are an important part of musical styles. There are key differences at this level between, for example, a fugue and a rock song. In addition to learning style-specific behavior at the foreground level to produce style-appropriate phrases, it would be useful to identify larger-scale developmental patterns that can be used as starting patterns for generation with a PTGG or similar type of grammar.

Because of music's similarity to spoken language, other learning approaches used in computational linguistics may be relevant to musical data. Some of the additional musical features mentioned so far may be best learned using algorithms intended for text processing. For example, phrase detection is a shared problem between the two types of data.

10.2 Concluding Remarks

Kulitta serves as a useful hypothesis testing mechanism for modeling musical constraints and examining candidate grammars for harmonic structure, since Kulitta can act in both an analytical and generative capacity. Kulitta also serves as a way to generate new and interesting music based on a user-supplied description of the constraints and style. That syle may be something well-known, or something new—even something totally unexpected. Used as a coding framework, Kulitta provides an interesting and unique way to compose novel works by writing a high-level or abstract description of the structure, which Kulitta then instantiates. Overall, Kulitta is a promising automated composition system whose strengths are "her" modularity and adaptability.

Appendix A

OPTIC Proofs

This appendix contains supporting proofs for the OPTIC normalizations, tests for equivalence, and group properties. Recall the following notations used in Chapter 3:

- Function composition: $(f_2 \cdot f_1)x = f_2(f_1(x))$.
- Function equality: $f_1 = f_2$. This means that f_1 and f_2 will have the same input/output mapping even if their definitions and/or complexities are different.
- Vectors: $\vec{x} = \langle x_1, ..., x_n \rangle$.
- Vectors created from a constant: $k^n = \langle k, ..., k \rangle$.
- Addition of two vectors: $\vec{x} + \vec{y} = \langle x_1 + y_1, ..., x_n + y_n \rangle$.
- Adding a constant to a vector: $\vec{x} + k = \langle x_1 + k, ..., x_n + k \rangle$.
- Vector concatenation: $\langle x_1, ..., x_n \rangle +\!\!+ \langle y_1, ..., y_m \rangle = \langle x_1, ..., x_n, y_1, ..., y_m \rangle$

Additionally, the notation $\exists!$ is used to denote unique existence.

A.1 OPTIC Normalizations

Recall the properties in Definition 2 from Chapter 3. For a function, $f : S \to S' \subseteq S$, to be a normalization for an equivalence relation, R, two properties must hold:

1. $\forall x \in S,\ x \sim_R f(x)$

2. $\forall x, y \in S, x \sim_R y \longleftrightarrow f(x) = f(y)$

The normalizations for O-, P-, OP-, OT-, PT-, PC-, and OPC- are simple and follow from basic properties of the symmetric group (permutations) and simple arithmetic on vectors. This section gives proofs that the two properties above exist for each normalization discussed in Chapter 3 for which a proof was not already given in the chapter.

Theorem 5. *$normO\langle x_1, ..., x_n\rangle = \langle x_1 \ mod\ 12, ..., x_n \ mod\ 12\rangle$ is a normalization for O-equivalence.*

Proof. Recall that octave equivalence is defined as follows.

$\vec{x} \sim_O \vec{x} + 12\vec{i},\ \vec{i} \in \mathbb{Z}^n$

This is related to the division algorithm, which is defined by Kenneth H. Rosen [71] as follows:

> Let a be an integer and d a positive integer. Then there are unique integers q and r with $0 \leqslant r < d$, such that $a = dq + r$.

In this algorithm, $r = a \ mod\ d$ is the remainder. The *normO* operation is performing this operation field-wise on a vector, where $d = 12$:

$\vec{a} = 12\vec{q} + \vec{r}$

or:

$\vec{r} = \vec{a} - 12\vec{q}$

Since $\vec{r} = normO(\vec{a})$ is equivalent to an octave shift operation, we know that the first normalization property holds: $\vec{x} \sim_O normO\vec{x}$. The division algorithm also means that any $\vec{y} \in \mathbb{Z}^n$ can be defined as:

$\vec{y} = \vec{x} + 12\vec{i}$, $\vec{x} \in [0,11]^n$, $\vec{i} \in \mathbb{Z}^n$

We can now use this to rewrite the definition of octave equivalence. First, we will start by relating $\vec{x} \in [0,11]^n$ to two arbitrary other vectors.

$\vec{x} + 12\vec{i} \sim_O \vec{x} \sim_O \vec{x} + 12\vec{j},\ \vec{x} \in [0,11]^n$

Now, because equivalence relations must be transitive, we can drop the $\vec{x}$ in the middle.

$\vec{x} + 12\vec{i} \sim_O \vec{x} + 12\vec{j},\ \vec{x} \in [0,11]^n$

To show that octave equivalent vectors normalize to the same value, we can show that for a particular $\vec{x} = \langle x_1, ..., x_n \rangle$ and an arbitrary $\vec{i} = \langle i_1, ..., i_n \rangle$, $normO(\vec{x} + 12\vec{i}) = \vec{x}$.

$normO(\langle 12i_1 + x_1, ..., 12i_n + x_n \rangle)$

$= \langle (12i_1 + x_1)\ mod\ 12, ..., (12i_n + x_n)\ mod\ 12 \rangle$ Definition of *normO*.

$= \langle x_1, ..., x_n \rangle$ Simplification.

For $\vec{y} = \vec{x} + 12\vec{i}$ and $\vec{z} = \vec{x'} + 12\vec{j}$ where $\vec{x}, \vec{x'} \in [0,11]^n$, we therefore have the bi-implication: $\vec{y} \sim_O \vec{z} \longleftrightarrow normO(\vec{y}) = normO(\vec{z})$. Therefore, *normO* is a normalization for octave equivalence.

□

Theorem 6. *$normP\langle x_1, ..., x_n \rangle = sort\langle x_1, ..., x_n \rangle$ is a normalization for P-equivalence.*

Proof. The proof of this property follows trivially from the fact that sorting a vector of integers in ascending order produces a sorted multiset. Since sorting algorithms form permutations, $\vec{x} \sim_P normP(\vec{x})$ holds. If two vectors are permutations of the same multiset, they must have the same value when those elements are sorted in ascending order [1]. Therefore, we have that $\vec{x} \sim \vec{y} \longleftrightarrow normP(\vec{x}) = normP(\vec{y})$, making *normP* a normalization for P-equivalence. □

1. Note that there is more than one permutation that will sort a vectors with fields that have duplicate values. However, this does not matter, since all of the possible sorted permutations will have the same vector value. For example, there are two permutations that will sort $\langle 1,0,0 \rangle$, but both will produce the same value: $\langle 0,0,1 \rangle$.

Theorem 7. *normOP = normP · normO is a normalization for OP-equivalence.*

Proof. First, the property that $x \sim_{OP} normOP(\vec{x})$ is easy to show by transitivity.

$\vec{x} \sim_O normO(\vec{x}) \sim_P normP(normO(\vec{x}))$

We can observe a similar, more general relationship using the definitions of O- and P-equivalence.

$\vec{x} \sim_O \vec{x} + 12\vec{i} \sim_P \sigma(\vec{x} + 12\vec{i}),\ \vec{i} \in \mathbb{Z}^n, \sigma \in S_n$

The middle term can be removed by transitivity, giving the following definition of OP-equivalence:

$\vec{x} \sim_{OP} \sigma(\vec{x} + 12\vec{i}),\ \vec{i} \in \mathbb{Z}^n, \sigma \in S_n$

Using transitivity, properties of the division algorithm described in the proof for *normO*, and properties of permutations, we can assert the following:

$\forall \vec{y} \in \mathbb{Z}^n,\ \exists \sigma \in S_n,\ \exists! \vec{i} \in \mathbb{Z}^n,\ \vec{x} \in [0,11]^n,\ \vec{x} = sort(\vec{x}) \mid \vec{y} = \sigma(12\vec{i} + \vec{x})$

This allows the previous definition of OP-equivalence to be re-written.

$\sigma(12\vec{i} + \vec{x}) \sim_{OP} \sigma'(12\vec{j} + \vec{x}),\ \vec{x} \in [0,11]^n,\ \vec{x} = sort(\vec{x}),\ \vec{i}, \vec{j} \in \mathbb{Z}^n,\ \sigma, \sigma' \in S_n$

Note that $\vec{x}$ is the representative point for an OP-equivalence class. We can now show that, for a particular, sorted $\vec{x} \in [0,11]^n$, an arbitrary $\vec{i} \in \mathbb{Z}^n$, and an arbitrary $\sigma \in S_n$, all vectors of the form $\sigma(\vec{x} + 12\vec{i})$ will be mapped to $\vec{x}$ by *normOP*.

$normOP(\sigma(12\vec{i} + \vec{x}))$	
$= sort(normO(\sigma(12\vec{i} + \vec{x})))$	Definition of *normOP*.
$= sort(normO(12(\sigma(\vec{i})) + \sigma(\vec{x})))$	Property: $\sigma(12\vec{i} + \vec{x}) = 12\sigma(\vec{i}) + \sigma(\vec{x})$
$= sort(\sigma(\vec{x}))$	Application of *normO*.
$= sort(\vec{x}))$	Property: $sort(\sigma(\vec{x})) = sort(\vec{x})$
$= \vec{x}$	$\vec{x} \in [0,11]^n$ is already sorted.

For two vectors, $\vec{y} = \sigma(12\vec{i} + \vec{x})$ and $\vec{z} = \sigma'(12\vec{j} + \vec{x'})$, we have the bi-implication: $\vec{y} \sim_{OP} \vec{z} \longleftrightarrow normOP(x) = normOP(x')$. Therefore, *normOP* is a normalization for OP-equivalence. □

Note that $normO \cdot normP$ is not a normalization for OP-equivalence. For example, consider $\langle 0,4,7\rangle \sim_{OP} \langle 60,4,7\rangle$. The two vectors, $\langle 0,4,7\rangle$ and $\langle 60,4,7\rangle$, should normalize to the same value (as they would with $normOP = normP \cdot normO$). However, $normO(normP(\langle 0,4,7\rangle)) = \langle 0,4,7\rangle$ while $normO(normP(\langle 60,4,7\rangle)) = \langle 4,7,0\rangle$.

Theorem 8. *$normOT = normO \cdot normT$ is a normalization for OT-equivalence.*

Proof. Recall that $normT(\langle x_1,...,x_n\rangle) = \langle x_1 - x_1,...,x_n - x1\rangle$. We know that $x \sim_{OT} normOT\vec{x}$ because of the following relationship:

$\vec{x} \sim_T normT(\vec{x}) \sim_O normO(normT(\vec{x}))$

Similarly, we observe a more general relationship.

$\vec{x} \sim_O \vec{x} + 12\vec{i} \sim_T \vec{x} + 12\vec{i} + k,\ \vec{i} \in \mathbb{Z}^n,\ k \in \mathbb{Z}$

Because of transitivity, the middle term can be removed.

$\vec{x} \sim_{OT} \vec{x} + 12\vec{i} + k,\ \vec{i} \in \mathbb{Z}^n,\ k \in \mathbb{Z}$

Using properties of the division algorithm and addition, we can make the following assertion.

$\forall \vec{y} \in \mathbb{Z}^n,\ \exists! \vec{i} \in \mathbb{Z}^n,\ k \in [0,11],\ \vec{x} \in \langle 0\rangle +\!\!+ [0,11]^{n-1},\ \vec{x} + k \in [0,11]^n \mid \vec{y} = \vec{x} + 12\vec{i} + k$

The definition of OT-equivalence can then be rewritten as:

$\vec{x} + 12\vec{i} + k \sim_{OT} \vec{x} + 12\vec{j} + k',\ \vec{x} \in \langle 0\rangle +\!\!+ [0,11]^{n-1},\ \vec{i},\vec{j} \in \mathbb{Z}^n,\ k,k' \in \mathbb{Z}$

Note that $\vec{x}$ is the representative point for an OT-equivalence class: a vector in $[0,11]^n$ whose first element is zero. For a particular such $\vec{x}$, an arbitrary $\vec{i}$, and an arbitrary k, we can now show that all vectors of the form $\vec{x} + 12\vec{i} + k$ will normalize to $\vec{x}$.

$normOT(\vec{x}+12\vec{i}+k)$	
$= normO(normT(\vec{x}+12\vec{i}+k))$	Definition of *normOT*.
$= normO((\vec{x}+12\vec{i}+k)-(x_1+12i_1+k))$	Definition of *normT*.
$= normO(\vec{x}-x_1+12\vec{i}-12i_1+k-k)$	Properties of addition.
$= normO(\vec{x}-x_1+12(\vec{i}-i_1))$	Properties of addition and multiplication.
$= normO(\vec{x}-x_1)$	Property of O-equivalence.
$= normO(\vec{x}+0)$	x_1 is zero by definition .
$= \vec{x}$	$\vec{x}$ is already within $[0,11]^n$

For $\vec{y} = \vec{x} + 12\vec{i} + k$ and $\vec{z} = \vec{x'} + 12\vec{j} + k'$, we then have the bi-implication: $\vec{y} \sim_{OT} \vec{z} \longleftrightarrow normOT(\vec{y}) = normOT(\vec{z})$. Therefore, *normOT* is a normalization for OT-equivalence. □

Note that $normT \cdot normO$ is not a normalization for octave equivalence. Consider the vectors $\langle 1,0,0\rangle$ and $\langle 0,11,11\rangle$, which are OT-equivalent and normalize to the same value using *normOT* as defined above. The transformations that relate these two vectors are:

$\langle 1,0,0\rangle \sim_O \langle 1,12,12\rangle \sim_T \langle 0,11,11\rangle$

However, $normT(normO(\langle 1,0,0\rangle)) = \langle -1,0,0\rangle$ and $normT(normO(\langle 0,11,11\rangle)) = \langle 0,11,11\rangle$.

Theorem 9. *$normPC = normC \cdot normP$ is a normalization for PC-equivalence.*

Proof. This proof directly follows from that of *normP* and *normC* (which is a normalization due to its use in defining C-equivalence in Chapter 3. The *normP* operation turns a vector into its sorted multiset representation, and *normC* removes all adjacent duplicate elements. Because all duplicate elements will be adjacent after calling *normP* on a vector, the subsequent *normC* will reduce a vector of pitches to a sorted set of pitches. Chords sharing the same set of pitches will be correctly normalized to the same value. Chords with different sets of pitches will normalize to different values. □

Theorem 10. *normOPC = normC · normOP is a normalization for PC-equivalence.*

Proof. This proof follows directly from the proofs for *normOP* and *normPC*. The *normOP* function converts a chord into a multiset of pitch classes sorted in ascending order. All duplicate pitch classes will be adjacent and will be removed by *normC*. Chords sharing the same set of pitch classes will be correctly normalized to the same value. Chords with different sets of pitch classes will normalize to different values. □

Theorem 11. *normPT = normT · sort is a normalization for PT-equivalence.*

Proof. Recall that $normT(\langle x_1,...,x_n\rangle) = \langle x_1 - x_1,...,x_n - x1\rangle$. We know that $x \sim_{PT} normPT\vec{x}$ because of the following relationship:

$\vec{x} \sim_P normP(\vec{x}) \sim_T normT(normP(\vec{x}))$

Similarly, we observe a more general relationship.

$\vec{x} \sim_P \sigma(\vec{x}) \sim_T \sigma(\sigma(\vec{x})) + k,\ \sigma \in S_n,\ k \in \mathbb{Z}^n$

Using transitivity, the middle term can be removed.

$\vec{x} \sim_{PT} \sigma(\vec{x}) + k,\ \sigma \in S_n,\ k \in \mathbb{Z}^n$

Properties of permutations and addition allow the following assertion.

$\forall \vec{y} \in \mathbb{Z}^n,\ \exists \sigma \in S_n,\ \exists!\ k \in \mathbb{Z},\ \vec{x} \in \langle 0\rangle +\!\!+\ \mathbb{Z}^n,\ \vec{x} = sort(\vec{x})\ \mid \vec{y} = \sigma(\vec{x}) + k$

This allows the definition of PT to be rewritten.

$\sigma(\vec{x}) + k \sim_{PT} \sigma'(\vec{x}) + k',\ \vec{x} = sort(\vec{x}), \vec{x} \in \langle 0\rangle +\!\!+\ \mathbb{Z}^n,\ \sigma, \sigma' \in S_n,\ k, k' \in \mathbb{Z}^n.$

$\vec{x}$ is a representative point for a PT-equivalence class: a sorted vector whose first element is zero. Now, for a particular $\vec{x}$, an arbitrary $\sigma \in S_n$, and an arbitrary $k \in \mathbb{Z}^n$, we will show that $normPT(\sigma(\vec{x}) + k) = \vec{x}$, where $minimum(\vec{x})$ returns the smallest element of $\vec{x}$. To do this, we make use of two observations: (1) the first field of $(sort(\vec{x}))$ is $minimum(\vec{x})$ and (2) adding a constant to a vector does not change the set of permutations that sort it.

$$
\begin{aligned}
& normPT(\sigma(\vec{x})+k) & \\
&= normT(normP(\sigma(\vec{x})+k)) && \text{Definition of } normPT. \\
&= normT(sort(\sigma(\vec{x})+k)) && \text{Definition of } normP. \\
&= normT(sort(\vec{x})+k) && \text{Property: } sort \cdot \sigma = sort. \\
&= sort(\vec{x})+k-(minimum(\vec{x})+k) && \text{Definition of } normT. \\
&= sort(\vec{x})-minimum(\vec{x})+k-k && \text{Properties of addition.} \\
&= sort(\vec{x})-minimum(\vec{x}) && \text{Simplification.} \\
&= sort(\vec{x}) && x_1 \text{ is zero.} \\
&= \vec{x} && \vec{x} \text{ is already sorted.}
\end{aligned}
$$

Therefore, for $\vec{y} = \sigma(\vec{x}) + k$ and $\vec{z} = \sigma(\vec{x'}) + k'$, we have the bi-implication: $\vec{y} \sim_{PT} \vec{z} \longleftrightarrow normPT(\vec{y}) = sort(\vec{x}) - minimum(\vec{x}) = normPT(\vec{z})$, making *normPT* a normalization for PT-equivalence.

□

Theorem 12. *normOPTC = normOPT · normOPC is a normalization for OPTC-equivalence.*

Corollary 3. *$optcEq(\vec{x},\vec{y}) = optEq(normOPC(\vec{x}), normOPC(\vec{y}))$ correctly tests for OPTC-equivalence.*

Proof. The property $\vec{x} \sim_{OPTC} normOPTC(\vec{x})$ follows from transitivity and the normalization properties for *normOPT* and *normOPC*. We define the notion of the set of all possible cardinality changes, S_C, where a given $c(\vec{x}) \in S_C$ produces a C-equivalent chord to $\vec{x}$. Properties of addition, permutations, and sets, and OPT-equivalence allow any vector to be rewritten as a cardinality operation, transposition, permutation, and octave shift of a set of pitch classes that is also its own OPT-normalization.

$$\forall y \in \mathbb{Z}^n,\ \exists \sigma \in S_n,\ c \in S_C,\ \exists!\ \vec{i} \in \mathbb{Z}^n,\ k \in [0,11],\ \vec{x} \in \mathbb{Z}^m,$$

$$\vec{x} = normOPT(\vec{x}),\ \vec{x} = normOPC(\vec{x}),\ \vec{x}+k \in [0,11]^m,\ m \leqslant n \mid \vec{y} = \sigma(c(\vec{x})) + 12\vec{i} + k$$

Using transitivity and the property above, we can re-write the definition of OPTC-equivalence as follows.

$\sigma(c(\vec{x}))+12\vec{i}+k \sim_{OPTC} \sigma'(c'(\vec{x}))+12\vec{j}+k',\ \vec{x}=normOPT(\vec{x}),\ \vec{x}=normOPC(\vec{x}),$

$\vec{i},\vec{j}\in\mathbb{Z}^n,\ \sigma,\sigma'\in S_n,\ c,c'\in S_C,\ \vec{x}+k\in[0,11]^m,\ \vec{x}+k'\in[0,11]^m$

$\vec{x}$ is a representative point for an OPTC-equivalence class. We can now show that $\vec{y}\sim_{OPTC}\vec{z}\longleftrightarrow normOPTC(\vec{y})=normOPTC(\vec{z})$. For a particular set of pitch classes, $\vec{x}\in[0,11]^n$ where $\vec{x}=normOPT(\vec{x})$, an arbitrary σ, an arbitrary k, and an arbitrary and $\vec{i}$:

$normOPTC(\sigma(c(\vec{x}))+12\vec{i}+k)$	
$=normOPT(normOPC(\sigma(c(\vec{x}))+12\vec{i}+k)$	Definition of *normOPTC*.
$=normOPT(normOPC(\vec{x})+k)$	Property of OPC-equivalence.
$=normOPT(normOP(\vec{x})+k)$	$\vec{x}$ is already set.
$=normOPT(\vec{x}+k)$	*normOPT* already calls *normOP*.
$=normOPT(\vec{x})$	Property of OPT-equivalence.
$=\vec{x}$	Definition of $\vec{x}$.

For two vectors, $\vec{y}=\sigma(c(\vec{x}))+12\vec{i}+k$ and $\vec{z}=\sigma'(c'(\vec{x'}))+12\vec{i}+k'$, we have the bi-implication: $\vec{y}\sim_{OPTC}\vec{z}\longleftrightarrow normOPTC(\vec{y})=normOPT(\vec{x})=normOPTC(\vec{z})$. Therefore, *normOPTC* is a normalization for OPTC-equivalence.

□

A.2 Group Operators

Chapter 3 defines four Abelian groups based on the O, P, T, and I relations: G_0, G_P, G_T, and G_I respectively. This section presents proof of the properties of closure and associativity and the presence of identity and inverse elements for each group. Recall that the group definitions are:

- $G_O = (\{o\ \vec{i} \mid \vec{i} \in \mathbb{Z}^n\}, \cdot)$
- $G_P = (\{p\ \sigma \mid \sigma \in S^n\}, \cdot)$.
- $G_T = (\{t\ k \mid k \in \mathbb{Z}\}, \cdot)$
- $G_I = (\{i\ k \mid k \in \{1, -1\}\}, \cdot)$

For notational simplicity in this section, instead of writing $p\ \sigma\ \vec{x}$ to indicate a permutation, the symmetric group will be used directly: $\sigma(\vec{x})$, $\sigma \in S_n$.

Theorem 13. *G_O is an Abelian group.*

Proof.

Closure: for any two members of G_O, $(o\ \vec{i_1})$ and $(o\ \vec{i_2})$:

$(o\ \vec{i_1} \cdot o\ \vec{i_2})\ \vec{x}$

$= o\ \vec{i_1}\ (o\ \vec{i_2}\ \vec{x})$

$= (\vec{x} + 12\vec{i_2}) + 12\vec{i_1}$

$= \vec{x} + 12(\vec{i_2} + \vec{i_1})$

$= o\ (\vec{i_2} + \vec{i_1})\ \vec{x}$, which is another member of G_O.

Identity: $o\,\vec{i} \cdot o\,0^n = o\,0^n \cdot o\,\vec{i} = o\,\vec{i}$

Inverse: $o\,\vec{i} \cdot o\,(-\vec{i}) = o\,(-\vec{i}) \cdot o\,\vec{i} = o\,0^n$

Associativity: $o\,\vec{i} \cdot (o\,\vec{j} \cdot o\,\vec{k})$

$= o\,\vec{i}\,(o\,(\vec{j} + \vec{k}))$

$= o\,(\vec{i} + (\vec{j} + \vec{k}))$

$= o\,((\vec{i} + \vec{j}) + \vec{k})$

$= o\,(\vec{i} + \vec{j}) \cdot o\,\vec{k}$

$= (o\,\vec{i} \cdot o\,\vec{j}) \cdot o\,\vec{k}$

Commutitivity:

$$(o\ \vec{i_1} \cdot o\ \vec{i_2})\ \vec{x}$$
$$= \vec{x} + 12(\vec{i_2} + \vec{i_1})$$
$$= \vec{x} + 12(\vec{i_1} + \vec{i_2})$$
$$= (o\ \vec{i_2} \cdot o\ \vec{i_1})\ \vec{x}$$

□

Theorem 14. *G_T is an Abelian group.*

Proof.

Closure: for any two members of G_T, $(t\ k_1)$ and $(t\ k_2)$:

$(t\ k_1 \cdot t\ k_2)\ \vec{x}$

$= (\vec{x} + k_2 1^n) + k_1 1^n$

$= \vec{x} + (k_2 1^n + k_1 1^n)$

$= t\ (k_2 + k_1)\ \vec{x}$, which is another member of G_T.

Identity: $t\,k \cdot t\,0 = t\,0 \cdot t\,k = t\,k$

Inverse: $t\,k \cdot t\,(-k) = t\,(-k) \cdot t\,k = t\,0$

Associativity: $t\,a \cdot (t\,b \cdot t\,c)$

$= t\,a \cdot t\,(b + c)$

$= t\,(a + (b + c))$

$= t\,((a + b) + c)$

$= t\,(a + b) \cdot t\,c$

$= (t\,a \cdot t\,b) \cdot t\,c$

Commutativity:

$(t\ k_1 \cdot t\ k_2)\ \vec{x}$

$= \vec{x} + k_2 1^n + k_1 1^n$

$= \vec{x} + k_1 1^n + k_2 1^n$

$= (t\ k_2 \cdot t\ k_1)\ \vec{x}$

□

Theorem 15. *G_I is an Abelian group.*

Proof. G_I has two members: $(i\ 1)$ and $(i\ (-1))$.

Closure, associativity, and commutativity: these properties follow from basic properties of multiplication of vectors by 1 and (-1).

Identity: $i\,1$

$$i\,(-1)\cdot i\,1 = i\,1\cdot i\,(-1) = i\,(-1)$$

$$i\,1\cdot i\,1 = i\,1$$

Inverse:

$$i\,1\cdot i\,1 = i\,1$$

$$i\,(-1)\cdot i\,(-1) = i\,1$$

□

Theorem 16. *For vectors of length n:*

$$\forall f = r_1\cdot\ldots\cdot r_m,\ r_i\in\{G_O,S_n,G_T\},\ \exists o_f\cdot\sigma_f\cdot t_f = f, o_f\in G_O,\sigma_f\in S_n,t_f\in G_T$$

Proof. For every pair of operators in $\{G_O,S_n,G_T\}$, there is a way to reorder them when they are from different groups and combine them when they are from the same group. These transformations follow from basic algebra and group properties.

- $o\,\vec{i}\cdot o\,\vec{j} = o\,(\vec{i}+\vec{j}),\ \vec{i},\vec{j}\in\mathbb{Z}^n$
- $\sigma\cdot o\,\vec{i} = o\,(\sigma(\vec{i}))\cdot\sigma,\ \vec{i}\in\mathbb{Z}^n,\ \sigma\in S_n$
- $o\,\vec{i}\cdot t\,k = t\,k\cdot o\,\vec{i},\ \vec{i}\in\mathbb{Z}^n,\ k\in\mathbb{Z}$
- $\forall\sigma_1,\sigma_2\in S_n,\ \exists\sigma_3\in S_n \mid \sigma_3 = \sigma_1\cdot\sigma_2$
- $\sigma\cdot t\,k = t\,k\cdot\sigma,\ k\in\mathbb{Z},\ \sigma\in S_n$
- $t\,k_1\cdot t\,k_2 = t\,(k_1+k_2),\ k_1,k_2\in\mathbb{Z}$
- $\forall f\in\{G_O,S_n,G_T\},\ f = f\cdot o\,0^n$

- $\forall f \in \{G_O, S_n, G_T\},\ f = f \cdot_t 0$
- $\forall f \in \{G_O, S_n, G_T\},\ f = f \cdot \sigma_{id}$ where $\sigma_{id}(\vec{x}) = \vec{x}$

Using these rules, operators can be iteratively reordered and combined. This will ultimately allow any function using some number of operators from G_O, S_n, and G_T to be transformed into an equivalent function that uses only one operator from each group (some of which may be identity operators). □

Appendix B

Haskell Source Code

This appendix includes code to describe portions of the Kulitta's implementation that has not been included in the main text. The full implementation of the context-sensitive monad for PTGGs along with a sample PTGG implementation are included, along with code illustrating the details of the classical and jazz foreground algorithms described in Chapter 6.

B.1 Modally Context-Sensitive PTGG Implementation

The following constant determines the behavior of the choose function. If set to True, then ID rules MUST be present in the rule set to succeed. If set to False, then if no relevant rules are found for a symbol, the symbol will be left unchanged (rather than throwing an error).

$forceIDs = False$

Construction of the grammar's symbols and structure

A CType is the Roman numeral.

data $CType = I \mid II \mid III \mid IV \mid V \mid VI \mid VII$

deriving $(Eq, Show, Ord, Enum, Read)$

An MType is a degree of modulation (relative 2nd, 3rd, ..., 7th). There is no M1 since that would indicate remaining in the home key.

data *MType* = *M2* | *M3* | *M4* | *M5* | *M6* | *M7*

deriving (*Eq*, *Show*, *Ord*, *Enum*)

A Chord is "something" with a duration. This "something" can be a *CType*, but it can also be a different datatype. This polymorphic definition of a *Chord* allows for PTGGs to handle more than one alphabet of chord symbols.

data *Chord a* = *Chord Dur a*

deriving (*Eq*, *Show*)

The Term data structure has five constructors: a chord, which acts as a nonterminal but can also be a terminal; a "sentence," or sequence of Terms; a modulation applied to a Term; a variable; and a Let statement that uses variables that are represented as strings.

data *Term a* = *NT* (*Chord a*) | *S* [*Term a*] | *Mod MType* (*Term a*) |

Let String (*Term a*) (*Term a*) | *Var String*

deriving (*Eq*, *Show*)

data *Rule a* = (*a*, *Prob*) :-> *RuleFun a*

type *RuleFun a* = *Ctxt* → *Dur* → *Term a*

type *Prob* = *Double*

Lowercase versions of the Roman numerals are used as a shorthand for chords. As in Chapter 3, the lowercase letters are a Haskell requirement and do not indicate the quality of the chords.

i, *ii*, *iii*, *iv*, *v*, *vi*, *vii* :: *Dur* → *Term CType*

[*i*, *ii*, *iii*, *iv*, *v*, *vi*, *vii*] = *map* (λ*c t* → *NT* (*Chord t c*)) $ *enumFrom I*

c ct t = *NT* (*Chord t ct*)

rf fs t = *map* ($*t*) *fs*

Modal context will be represented using Euterpea's *Mode* datatype, which has two constructors: *Major* and *Minor*.

type *Ctxt* = *Mode*

type *CtxtFun* = *Ctxt* → *MType* → *Ctxt*

We also need a function to update the context when modulations occur. The approach for determining the mode of a modulation is related to the list of Roman numeral modes in Table 3.2. Since this implementation only considers two modes, modulation *Mn* will have the mode of the triad on the n^{th} scale degree, where diminished triads are considered to be minor. For example, in the key of C-major, M2 would use the triad formed on the 2^{nd} scale degree, D-F-A (a minor triad), to determine the new mode: minor.

```
defCtxtFun :: CtxtFun
defCtxtFun Major m = [Minor,Minor,Major,Major,Minor,Minor] !!fromEnum m
defCtxtFun Minor m = [Minor,Major,Minor,Minor,Major,Major] !!fromEnum m
```

Although this implementation only addresses two modes, it could easily be extended to handle more modes by altering the type for *Ctxt* and redefining *defCtxtFun* accordingly.

B.1.1 Monad Implementation

The monad implementation remains relatively unchanged from the one described in Chapter 4. The only new additions are the set/get functions for the modal context, *setCtxt* and *getCtxt* respectively.

```
newtype Prog a = Prog ((StdGen,Ctxt) → ((StdGen,Ctxt),a))

instance Monad Prog where
  return a = Prog (λs → (s,a))
    Prog p0 >>=f1 = Prog $ λs0 →
    let (s1,a1) = p0 s0
        Prog p1 = f1 a1
    in p1 s1

getRand :: (Random a) ⇒ (a,a) → Prog a
getRand ran = Prog (λ(g,c) → let (r,g') = randomR ran g
  in ((g',c),r))
```

```
getCtxt :: Prog Ctxt
getCtxt = Prog (λ(g,c) → ((g,c),c))
setCtxt :: Ctxt → Prog ()
setCtxt c' = Prog (λ(g,c) → ((g,c'),()))
runP :: Prog a → StdGen → Ctxt → a
runP (Prog f) g c = snd (f (g,c))
update :: (Eq a) ⇒ [Rule a] → Term a → Prog (Term a)
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 c ← getCtxt
               setCtxt (defCtxtFun c m)   -- into mod
               s' ← update rules s
               setCtxt c   -- out of mod
               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')
applyRule :: (Eq a) ⇒ [Rule a] → Chord a → Prog (Term a)
applyRule rules t@(Chord d c) =
  let rs = filter (λ((c',p) :-> rf) → c' == c) rules
  in do r ← getRand (0.0, 1.0)
        x ← getCtxt
        return ((choose' t rs r) x d)
```

The *choose* function is modified from its original definition to use the *forceIDs* function.

This change is needed to support PTGGs build from modified versions of Rohrmeier's grammar as described in chapter 7. If *forceIDs* = *True* then the original definition of *choose* is used and PTGGs will be strictly required to include rules of the form $A \rightarrow A$ for all chords (otherwise an error message will be thrown). If *forcIDS* = *False*, then this requirement does not exist and chords may remain unchanged if no rules exist for them.

```
choose' :: Chord a → [Rule a] → Prob → (RuleFun a)
choose' t rs p =
  if null rs ∧ ¬ forceIDs then λc d → NT $ t   -- auto ID rule
  else choose rs p

choose :: [Rule a] → Prob → (RuleFun a)
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')
```

The iteration function, *iter*, remains unchanged, and *gen* simply takes an additional argument.

```
iter :: Monad m ⇒ (a → m a) → a → m [a]
iter f a = do a' ← f a
              as ← iter f a'
              return (a' : as)

type Seed = Int

gen :: (Eq a) ⇒ [Rule a] → Int → Seed → Ctxt → Term a → Term a
gen rules i s c t = runP (iter (update rules) t) (mkStdGen s) c !! i
```

Another interface to *gen* is provided for use with *StdGen* directly.

```
gen' :: (Eq a) ⇒ [Rule a] → Int → StdGen → Ctxt → Term a → Term a
gen' rules i s c t = runP (iter (update rules) t) s c !! i
```

The expand function eliminates Lets and Vars from a generated Term a. It allows for nested Let expressions for variables with the same name with lexical scoping.

```
expand :: [(String, Term a)] → Term a → Term a
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
```

Finally, it is often useful to "flatten" *Term* values before operating on them or printing them

```
flatten xs = let xs' = flattenRec xs in if xs' == xs then xs else flatten xs' where
  flattenRec t = case t of
    Let x a exp → Let x (flattenRec a) (flattenRec exp)
    Var x       → Var x
    Mod m t     → Mod m (flattenRec t)
    S []        → S []
    S xs        → S $ concatMap stripS $ map flattenRec xs
    NT v        → NT v where
  stripS :: Term a → [Term a]
  stripS (S xs) = concatMap stripS xs
  stripS xs = [xs]
```

B.1.2 Example Rule Set

This is the Haskell implementation of the rule set shown in Table 4.1.

Rules for *Let*

```
ruleL1, ruleL2 :: CType → RuleFun CType
ruleL1 ct ctxt t = Let "x" (c ct (t / 2)) (S [Var "x", Var "x"])
```

```
ruleL2 ct ctxt t = Let "x" (c ct (t / 4)) (S [Var "x", v (t / 2), Var "x"])

rulesL :: [Rule CType]
rulesL = concatMap (λct → [(ct, 0.1) :-> λt → ruleL1 ct t,
  (ct, 0.1) :-> λt → ruleL2 ct t]) (enumFrom I)
```

Rules for I

```
ruleI1 = (I, 0.15) :-> λctxt t →
  if ctxt == Major then S [ii (t / 4), v (t / 4), i (t / 2)]
  else S [iv (t / 4), v (t / 4), i (t / 2)]
ruleI2 = (I, 0.15) :-> λctxt t → S [i (t / 4), iv (t / 4), v (t / 4), i (t / 4)]
ruleI3 = (I, 0.15) :-> λctxt t → S [v (t / 2), i (t / 2)]
ruleI4 = (I, 0.15) :-> λctxt t →
  if ctxt == Major then S [i (t / 4), ii (t / 4), v (t / 4), i (t / 4)]
  else S [i (t / 4), iv (t / 4), v (t / 4), i (t / 4)]
ruleI5 = (I, 0.2) :-> λctxt t → if t ⩽ hn then i t else S [i (t / 2), i (t / 2)]
rulesI = [ruleI1, ruleI2, ruleI3, ruleI4, ruleI5]
```

Rules for II

```
ruleII1 = (II, 0.4) :-> λctxt → if ctxt == Major then ii else iv
ruleII1b = (II, 0.4) :-> λctxt t →
  if ctxt == Major then if t > qn then ii t else Mod M2 $ i t
  else Mod M2 $ i t
ruleII2 = (II, 0.2) :-> λctxt t →
  if ctxt == Major then S [vi (t / 2), ii (t / 2)]
  else S [vi (t / 2), iv (t / 2)]
rulesII = [ruleII1, ruleII1b, ruleII2]
```

Rules for III

$$ruleIII1 = (III, 0.9) :-> \lambda ctxt \rightarrow iii$$

$$ruleIII2 = (III, 0.1) :-> \lambda ctxt \rightarrow (Mod\ M3 \circ i)$$

$$rulesIII = [ruleIII1, ruleIII2]$$

Rules for IV

$$ruleIV1 = (IV, 0.9) :-> \lambda ctxt \rightarrow iv$$

$$ruleIV2 = (IV, 0.1) :-> \lambda ctxt \rightarrow (Mod\ M4 \circ i)$$

$$rulesIV = [ruleIV1, ruleIV2]$$

Rules for V

$$ruleV1 = (V, 0.15) :-> \lambda ctxt\ t \rightarrow S\ [iv\ (t/2), v\ (t/2)]$$

$$ruleV2 = (V, 0.10) :-> \lambda ctxt\ t \rightarrow S\ [iii\ (t/2), vi\ (t/2)]$$

$$ruleV3 = (V, 0.10) :-> \lambda ctxt\ t \rightarrow S\ [i\ (t/4), iii\ (t/4), vi\ (t/4), v\ (t/4)]$$

$$ruleV4 = (V, 0.10) :-> \lambda ctxt\ t \rightarrow S\ [v\ (t/4), vi\ (t/4), vii\ (t/4), v\ (t/4)]$$

$$ruleV5 = (V, 0.10) :-> \lambda ctxt\ t \rightarrow S\ [v\ (t/2), vi\ (t/2)]$$

$$ruleV6 = (V, 0.10) :-> \lambda ctxt \rightarrow iii$$

$$ruleV7 = (V, 0.10) :-> \lambda ctxt\ t \rightarrow S\ [v\ (t/2), v\ (t/2)]$$

$$ruleV8 = (V, 0.05) :-> \lambda ctxt\ t \rightarrow S\ [vii\ (t/2), v\ (t/2)]$$

$$ruleV9 = (V, 0.10) :-> \lambda ctxt \rightarrow v$$

$$ruleV10 = (V, 0.10) :-> \lambda ctxt \rightarrow (Mod\ M5 \circ i)$$

$$rulesV = [ruleV1, ruleV2, ruleV3, ruleV4, ruleV5,$$
$$ruleV6, ruleV7, ruleV8, ruleV9, ruleV10]$$

Rules for VI

$$ruleVI1 = (VI, 0.7) :-> \lambda ctxt \rightarrow vi$$

$$ruleVI2 = (VI, 0.3) :-> \lambda ctxt \rightarrow (Mod\ M6 \circ i)$$

$$rulesVI = [ruleVI1, ruleVI2]$$

Rules for VII

$$ruleVII1 = (VII, 0.5) :-> \lambda ctxt\ t \rightarrow \textbf{if}\ t > qn\ \textbf{then}\ vii\ t\ \textbf{else}\ Mod\ M7\ \$\ i\ t$$

$$ruleVII2 = (VII, 0.5) :-> \lambda ctxt\ t \rightarrow S\ [i\ (t/2), iii\ (t/2)]$$

$$rulesVII = [ruleVII1, ruleVII2]$$

The rule set defined below is broken up into two steps to allow easier testing of the probability sums.

$$ruleSet'\ useLets = normalize\ \$\ concat$$

$$[rulesI, rulesII, rulesIII, rulesIV, rulesV, rulesVI, rulesVII,$$

$$\textbf{if}\ useLets\ \textbf{then}\ rulesL\ \textbf{else}\ [\,]]$$

$$ruleSet\ d\ useLets = map\ (toRelDur\ d)\ \$\ ruleSet'\ useLets$$

B.1.3 Rule Utility Functions

Rule probabilities need to sum to 1.0 for rules with the same lefthand side. Since there is an option to include or exclude the **let** rules, the probabilities should be normalized (forced to sum to 1.0) before the rules are used.

$$lhs\ ((c,p) :-> rf) = c$$

$$prob\ ((c,p) :-> rf) = p$$

$$normalize :: (Eq\ a) \Rightarrow [Rule\ a] \rightarrow [Rule\ a]$$

$$normalize\ [\,] = [\,]$$

$$normalize\ (r@((l,p) :-> rf) : rs) =$$

$$\textbf{let}\ rset\ = r : filter\ ((l ==) \circ lhs)\ rs$$

```
    rset' = filter ((l /=) . lhs) rs
    psum = sum $ map prob rset
  in map (\((l',p') :-> c') -> ((l',p' / psum) :-> c')) rset ++ normalize rset'
```

Rules can be "wrapped" to produce rules that only operate on chords with at least a certain duration.

```
toRelDur :: Dur -> Rule a -> Rule a
toRelDur d ((c,p) :-> f) =
  let dmin ctxt t = minDur $ expand [] $ f ctxt t
  in ((c,p) :-> \ctxt t -> if dmin ctxt t < d then NT $ Chord t c else f ctxt t)

minDur :: Term a -> Dur
minDur (S s) = minimum $ map minDur s
minDur (Mod m t) = minDur t
minDur (NT (Chord d x)) = d
minDur _ = error "(minDur) String is not fully interpreted."

relRuleSet d rs = map (toRelDur d) $ rs
```

B.2 Post-Processing

Kulitta must perform post-processing (namely type conversion) on a PTGG's results in order to process the chords through a chord space or add musical foregrounds. Types involved are:

```
type Key = (AbsPitch,Mode)
type RChord = (Key,Dur,CType)
type TChord = (Key,Dur,AbsChord)
type TNote = (Key,Dur,AbsPitch)
type Voice = [TNote]
tnK (k,d,p) = k
```

$tnD\ (k,d,p) = d$

$tmP\ (k,d,p) = p$

$newP\ (k,d,p)\ p' = (k,d,p')$

The general interface between *Term* and the types presented here is the following:

Input		Function		Output
Seeds, PTGG	⇒	*gen*	⇒	Term
Term	⇒	*toAbsChords*	⇒	[*TChord*] with a basic triad mapping
Term	⇒	*toChords*	⇒	list version of Term
[*TChord*]	⇒	*toVoices*	⇒	[*Voice*], notes listed by voice first
[*Voice*]	⇒	*vsToMusic*	⇒	*Music Pitch*

The *toChords* function converts a Term to a list representation of the progression. This eliminates nested structures like modulations, meaning that chords must now carry information on their relative key and mode. A default of C-major or C-minor (depending on the mode given) is assumed.

```
toChords :: Term CType → Mode → [RChord]
toChords (NT (Chord d c)) m = [((0,m),d,c)]
toChords (S ts) m = concatMap (λt → toChords t m) ts
toChords (Mod mt x) m =
  let (amt,mt') = getMods mt m
    f = map (λ((k,m),d,c) → ((k+amt,mt'),d,c))
  in f (toChords x mt')
toChords x m =
  error ("(toChords) Unable to handle expression: " ++ showTerm x)
```

Lists of *RChord* can be converted directly to Euterpea's *Music Pitch*.

```
tChordsToMusic :: [TChord] → Music Pitch
tChordsToMusic = line ∘ map f where
```

$f\ ((k,m),d,as) = chord\ \$\ map\ (\lambda a \rightarrow note\ d\ (pitch\ a))\ as$

Conversion to the *TChord* type is done by turning the Roman numerals into basic triads.

$toAbsChord :: RChord \rightarrow TChord$

$toAbsChord\ ((k,m),d,c) = ((k,m),d,t\ k\ \$\ toAs\ c\ m)$

$toAbsChords :: Term\ CType \rightarrow Mode \rightarrow [TChord]$

$toAbsChords\ ts\ m = map\ toAbsChord\ \$\ toChords\ ts\ m$

Conversion of an invidiual chord involves determining the current scale and choosing scale indices from it based on the particular Roman numeral.

$toAs :: CType \rightarrow Mode \rightarrow [AbsPitch]$

$toAs\ ct\ m =$

$\quad$ **let** $s = getScale\ m +\!\!+ map\ (+12)\ s$

$\quad\quad i = head\ \$\ findIndices\ (== ct)\ [I,II,III,IV,V,VI,VII]$

$\quad$ **in** $map\ (s!!)\ \$\ map\ (+i)\ [0,2,4]$

$getScale :: Mode \rightarrow [AbsPitch]$

$getScale\ Major = [0,2,4,5,7,9,11]$

$getScale\ Minor = [0,2,3,5,7,8,10]$

$getMods\ mt\ m =$

$\quad$ **let** $mts = [M2,M3,M4,M5,M6,M7]$

$\quad\quad i = (head\ \$\ findIndices\ (== mt)\ mts) + 1$

$\quad$ **in** $(getScale\ m\ !!\ i, relPat\ m\ !!\ i)$ **where**

$\quad relPat :: Mode \rightarrow [Mode]$

$\quad relPat\ Major = [Major,Minor,Minor,Major,Major,Minor,Minor]$

$\quad relPat\ Minor = [Minor,Minor,Major,Minor,Minor,Major,Major]$

The *ctTrans* and *atTrans* functions allow transposition of progressions represented as lists of *RChord* and *TChord* respectively.

```
atTrans :: AbsPitch → [(Key,Dur,AbsChord)] → [(Key,Dur,AbsChord)]
atTrans a = map (λ((k,m),d,c) → ((fixK k a m,m),d,t (a `mod` 12) c))
ctTrans :: AbsPitch → [(Key,Dur,CType)] → [(Key,Dur,CType)]
ctTrans a = map (λ((k,m),d,c) → ((fixK k a m,m),d,c))
fixK k a Major = (k+a) `mod` 12
fixK k a Minor = ((k+a) `mod` 12) + 12
```

To add foregrounds, a [*TChord*] must be turned on its side to lists notes by voice rather than as chords. Values of type [*Voice*] can be easily converted to *Music Pitch*, with or without specifying the instrument to be used for each voice.

```
toVoices :: [TChord] → [Voice]
toVoices ts =
  let (ks,ds,ps) = unzip3 ts
  in map (λv → zip3 ks ds v) $ transpose ps
toNotes :: Voice → Music Pitch
toNotes = line ∘ map (λ(k,d,p) → note d (pitch p))
vsToMusic :: [Voice] → Music Pitch
vsToMusic = chord ∘ map toNotes
vsToMusicI :: [InstrumentName] → [Voice] → Music Pitch
vsToMusicI is = chord ∘ zipWith (λi m → instrument i m) is ∘ map toNotes
```

B.2.1 Constraint Satisfaction

The following two functions are used to convert *Let* expressions to lists of constraints as described in Chapter 5.

```
mkCons :: (Eq a) ⇒ Term CType → Predicate [a]
mkCons t xs = toCons (findInds [] t) xs where
  toCons :: (Eq a) ⇒ [[(Int,Int)]] → [a] → Bool
```

```
toCons [] xs = True
toCons (c:cs) xs =
  let f (i,j) = take (j+1-i) $ drop i xs
  in (and $ map (f (head c) ==) $ map f $ tail c) ∧ toCons cs xs
```

The *findInds* function below returns the *Let*-imposed constraints of a *Term* as lists of index ranges (pairs of integers).

```
findInds :: [(String, Term CType)] → Term CType → [[(Int, Int)]]
findInds e t = case t of
  Mod m t    → findInds e t
  S []       → []
  S (s:ss)   →
    let vs1 = findInds e s
        len = length $ toAbsChords (expand e s) Major
        vs2 = findInds e (S ss)
    in vs1 ++ map (add' len) vs2 where
        add' y = map (λ(a,b) → (a+y, b+y))
  Let x a exp →
    let e' = (x, expand e a) : e
        xCases = findIndsSub x e' exp
    in xCases : findInds e' a ++ findInds e' exp
  _          → []
findIndsSub :: String → [(String, Term CType)] → Term CType → [(Int, Int)]
findIndsSub x e t = case t of
  Let x' a exp →
    let a' = expand e a   -- to figure out sequence length
    in if x' == x then []   -- x has been redefined
       else findIndsSub x e a ++ findIndsSub x ((x', a') : e) exp
```

```
Var x'      → if x ≠ x' then [] else
  let v = maybe (error (x ++ " is undefined")) id $ lookup x e
    len = length $ toAbsChords v Major
  in [(0, len - 1)]
Mod m t'    → findIndsSub x e t'
S []        → []
S (s : ss)  →
  let len = length $ toAbsChords (expand e s) Major
  in findIndsSub x e s ++ add' (len) (findIndsSub x e (S ss)) where
  add' y = map (λ(a, b) → (a + y, b + y))
NT v        → []
```

B.3 Foreground Algorithms

Kulitta has two types of foreground algorithms: classical and jazz. For each style, there are multiple ways to add foreground features.

B.3.1 Classical Foregrounds

The classical foreground algorithm uses a collection of constants that can be user-specified (a default collection are also provided). *ntLimC* is the number of halfsteps away a neighboring tone can be and *ptLimC* is a similar value for passing tones. *pHalfC* is the probability of dividing a note's duration in half evenly (vs. a potentially asymmetrical division where the new note is an eighth note) and *pTieC* is the probability of tieing identical, sequential pitches into a single note. *rootBassThreshC* is the probability of forcing the bass voice to be the root of a chord and *noCPLThreshC* is the "voice-smoothness" parameter to favor movement within a fixed number of halfsteps.

```
data CConstants = CConstants {
  ntLimC :: Int,
  ptLimC :: Int,
  pHalfC :: Double,
  pTieC :: Double,
  rootBassThreshC :: Double,
  noCPLThreshC :: Int}
defConsts = CConstants 2 3 0.5 0.5 0.8 7
```

Adding passing and neighboring tones requires knowing which pitch classes are acceptable choices. Kulitta defines "acceptable" in this case as pitch classes that are shared by the scales of the two adjacent chordal tones.

```
allPs :: TNote → TNote → [AbsPitch]
allPs t1 t2 =
  let (o1,o2) = (tnP t1 `div` 12, tnP t2 `div` 12)
      [oMin,oMax] = sort [o1,o2]
      offs = map (12*) [oMin − 1,oMin,oMax,oMax + 1]
      (s1,s2) = (baseScale $ tnK t1, baseScale $ tnK t2)
  in nub $ concatMap (λo → t o [s | s ← s1, elem s s2]) offs where
  baseScale :: Key → [AbsPitch]
  baseScale (k,m) = normOP $ t k (getScale m)
```

Several foreground operations are defined using the type *ForeFun*.

```
type ForeFun = StdGen → TNote → TNote → (StdGen, Maybe AbsPitch)
```

A passing tone is a non-chordal tone between two chordal tones. Usually a passing tone and the notes on either side of it follow the scale directly, but Kulitta's definition is broader and allows passing tones to be up to *lim* number of half steps away from the lower and upper chordal tones.

```
pickPT :: AbsPitch → ForeFun
pickPT lim g t1 t2 =
  let [pMin,pMax] = sort [tnP t1, tnP t2]
      f x = x > pMin ∧ x < pMax ∧ (x − pMin ⩽ lim ∨ pMax − x ⩽ lim)
      ps' = [x | x ← allPs t1 t2, f x]
      (iNew, g') = randomR (0, length ps' − 1) g
  in if pMin == pMax ∨ null ps' then (g, Nothing) else (g', Just $ ps' !! iNew)
```

A neighboring tone is a non-chordal tone that is either above two chordal tones or below them, creating either an up-and-down or down-and-up motion. Kulitta's definition of a neighboring tone is similar to that of a passing tone, but placing the non-chordal tone up to *lim* half steps outside the chordal tones.

```
pickNT :: AbsPitch → ForeFun
pickNT lim g t1 t2 =
  let [pMin,pMax] = sort [tnP t1, tnP t2]
      f x = (x < pMin ∧ pMin − x ⩽ lim) ∨ (x > pMax ∧ x − pMax ⩽ lim)
      ps' = [x | x ← allPs t1 t2, f x]
      (iNew, g') = randomR (0, length ps' − 1) g
  in if pMin == pMax ∨ null ps' then (g, Nothing) else (g', Just $ ps' !! iNew)
```

Anticipations (*anticip*) and repetitions (*rept*) involve repeating chordal tones.

```
anticip, rept, doNothing :: ForeFun
anticip g t1 t2 = (g, Just $ tnP t2)
rept g t1 t2 = (g, Just $ tnP t1)
```

Finally, not all chordal tones need additional notes inserted between them. Kulitta also includes a "do nothing" foreground function to allow some pairs chordal tones to be stochastically left unchanged.

```
doNothing g t1 t2 = (g, Nothing)
```

The various foreground functions above are grouped and given probabilities of applica-

tion for each voice in a 4-voice chorale. The particular probabilities shown below are the author's choice.

$$f_1 = pickPT \circ ptLimC$$

$$f_2 = pickNT \circ ntLimC$$

$$[f_3, f_4, f5] = map\ const\ [anticip, rept, doNothing]$$

$$allFFs :: CConstants \rightarrow [[(Double, ForeFun)]]$$

$$allFFs\ c =$$

$$[[(0.3, f_1\ c), (0.1, f_2\ c), (0.6, f5\ c)],\quad \text{-- S (sopranno)}$$

$$[(0.3, f_1\ c), (0.7, f5\ c)],\quad \text{-- A (alto)}$$

$$[(0.1, f_1\ c), (0.9, f5\ c)]]\quad \text{-- T (tennor)}$$

$$++\ repeat\ [(1.0, f5\ c)]\quad \text{-- B (bass) and lower}$$

When adding new notes between chordal tones, duration must be borrowed from one of the chordal notes. Kulitta borrows time from the first chordal tone to do this. Exactly how much time is given to the new note is a stochastic choice: either the chordal tone's duration is divided in half, or an eighth note is subtracted from it to give to the new tone.

$$splitP :: CConstants \rightarrow StdGen \rightarrow AbsPitch \rightarrow TNote \rightarrow (StdGen, [TNote])$$

$$splitP\ consts\ g\ newP\ t =$$

$$\textbf{let}\ (r, g') = randomR\ (0, 1.0 :: Double)\ g$$

$$dNew = \textbf{if}\ r < pHalfC\ consts\ \textbf{then}\ tnD\ t\ /\ 2\ \textbf{else}\ en$$

$$\textbf{in}\ (g', [(tnK\ t, tnD\ t - dNew, tnP\ t), (tnK\ t, dNew, newP)])$$

Foregrounds are added to voices from highest to lowest, left to right. A foreground is added to one voice completely before moving on to the next voice. The *addFgToVoice* function adds a foreground to a single voice, and *addFG* performs foreground addition over a list of voices sorted from highest to lowest.

```
addFgToVoice :: CConstants → [(Double, ForeFun)] → StdGen → [TNote]
                → (StdGen, [TNote])
addFgToVoice c foreFuns g (t1 : t2 : ts) =
  let (j, g1) = randomR (0, 1.0) g
      fFun = chooseFF j foreFuns
      (g2, t1') = applyForeFun c g1 t1 t2 fFun
      (g3, tRest) = addFgToVoice c foreFuns g2 (t2 : ts)
  in (g3, t1' ++ tRest) where
  chooseFF j [x] = snd x
  chooseFF j ((p, x) : t) = if j < p then x else chooseFF (j - p) t
  chooseFF j [] = error "(chooseFF) Nothing to choose from!"
  applyForeFun c g t1 t2 fFun =
    let (g1, newP) = fFun g t1 t2
    in case newP of Nothing → (g1, [t1])
                    Just x  → splitP c g1 x t1
addFgToVoice c foreFuns g x = (g, x)
addFG :: CConstants → StdGen → [[TNote]] → (StdGen, [[TNote]])
addFG c g vs = let (g', vs') = fgRec c g 0 vs in tieRec c g' vs' where
  fgRec c g i vs = if i ⩾ length vs ∨ i < 0 then (g, vs) else
    let (g', v') = addFgToVoice c (allFFs c !! i) g (vs !! i)
        vs' = take i vs ++ [v'] ++ drop (i + 1) vs
    in fgRec c g' (i + 1) vs'
  tieRec c g [] = (g, [])
  tieRec c g (v : vs) =
    let (g1, v') = stochTie c g v
        (g2, vs') = tieRec c g1 vs
    in (g2, v' : vs')
```

A final post-processing step in foreground generation is the creation of ties, which are notes held across beats. This is best left until the very end, otherwise ties can create rhythmic abnormalities when adding other foreground features. Two ajacent notes are stochastically tied if they share the same pitch.

```
stochTie :: CConstants → StdGen → [TNote] → (StdGen, [TNote])
stochTie consts g (t1 : t2 : ts) =
  let (r, g1) = randomR (0, 1.0 :: Double) g
      (g2, (t2' : ts')) = stochTie consts g1 (t2 : ts)
      (d1, d2') = (tnD t1, tnD t2')
  in if tnP t1 == tnP t2' ∧ r < pTieC consts
     then (g2, (tnK t1, d1 + d2', tnP t1) : ts')
     else (g2, t1 : t2' : ts')
stochTie consts g ts = (g, ts)
```

There are two steps to adding a classical foreground to an abstract chord progression represented as Roman numerals: (1) traversing an appropriate chord space and (2) adding melodic elements. These steps are separated and presented with different type interfaces.

From a *Term CType*, a classical foreground can be added by using just the classicalFG function.

```
classicalFG :: StdGen → Key → Term CType →
                    (StdGen, (Music Pitch, Music Pitch))
classicalFG g (k, m) t =
  let consts = sort $ findInds [] t
      rChords = ctTrans k $ toChords (expand [] t) m
  in classicalFGR g (k, m) rChords consts
```

However, there are some instances where more control is desirable, such as if we are working with Let statments or perhaps want to supply a progression manually rather than using Term. The following functions allow adding a foreground to different intermediate

types.

```
classicalFGR :: StdGen → Key → [RChord] → Constraints →
                    (StdGen, (Music Pitch, Music Pitch))
classicalFGR g (k,m) rcs consts =
  let (g1, csChords) = classicalCS g (k,m) rcs consts
  in classicalFG' g1 csChords
classicalFG' :: StdGen → [TChord] → (StdGen, (Music Pitch, Music Pitch))
classicalFG' g aChords' =
  let (g4, csFG) = addFG defConsts g $ reverse $ toVoices aChords'
      is = [Bassoon, EnglishHorn, Clarinet, Oboe, SopranoSax]
      fgM = vsToMusicI is $ reverse csFG
      csM = vsToMusicI is $ toVoices aChords'
  in (g4, (csM, fgM))
```

Similarly, there are instances when we may want to use a classical chord space, but not add a classical foreground. This can be useful for mixing styles.

```
classicalCS :: StdGen → Key → [RChord] → Constraints → (StdGen, [TChord])
classicalCS g (k,m) rcs consts = classicalCS2 g (k,m)
                                   (atTrans k $ map toAbsChord rcs) consts
classicalCS2 :: StdGen → Key → [TChord] → Constraints → (StdGen, [TChord])
classicalCS2 g (k,m) aChords consts =
  let justChords = map (λ(a,b,c) → c) aChords
      (g1, g2) = split g
      (g3, eqs) = classBass 0.8 g2 $ map (eqClass satbOP opcEq) justChords
      csChords = greedyLet (noCPL 7) nearFall consts eqs g3
      aChords' = zipWith (λ(a,b,c) d → (a,b,d)) aChords csChords
  in (g3, aChords')
```

The classicalCS2 function uses a stochastic filter over equivalence classes called *classBass*.

This filter enforces that the bass holds the root with a certain probability (the "thresh" value). If the constraints can't be met, the bass is allowed to deviate from this rule for the sake of producing a result.

```
classBass :: Double → StdGen → [EqClass AbsChord] →
                (StdGen, [EqClass AbsChord])
classBass thresh g [] = (g, [])
classBass thresh g (e : es) =
  let (r, g') = randomR (0, 1.0 :: Double) g
      e' = if r > thresh then e else filter rootFilter e
      e'' = if null e' then e else e'
      (g'', es') = classBass thresh g es
  in (g'', e'' : es') where
  rootFilter :: Predicate AbsChord
  rootFilter x = or $ map (opcEq x) [[0,0,4,7],[0,0,3,7],[0,0,3,6]]
```

The code so far has only made use of the greedy approaches to constraint satisfaction. As an alternative, the following version handles constraint satisfaction differently. Two MIDI files are produced, one without melodic elements and one with them. The benefit of this alternate approach is that constraints will be 100% satisfied if a solution is found. However, existence of a solution is not guaranteed and the runtime will be quite long if solutions are sparse.

```
classicalFG2 :: StdGen → Key → Term CType → FilePath → FilePath → IO ()
classicalFG2 g (k, m) t fn1 fn2 = do
  let aChords = atTrans k $ toAbsChords (expand [] t) m
      justChords = map (λ(a, b, c) → c) aChords
      (g1, g2) = split g
      qSpace = satbOP' g1
      ecs = map (eqClass qSpace opcEq) justChords
```

```
    cons = findInds [] t
  (x, csChords) ← findSoln2 cons (progL 10) ecs
  let aChords' = zipWith (λ(a,b,c) d → (a,b,d)) aChords csChords
      (g4, csFG) = addFG defConsts g2 $ reverse $ toVoices aChords'
      is = [Bassoon, EnglishHorn, Clarinet, Oboe]
      fgM = vsToMusicI is $ reverse csFG
      csM = vsToMusicI is $ toVoices aChords'
  writeMidi fn1 fgM
  writeMidi fn2 csM
```

B.3.2 Jazz Foregrounds

Jazz requires keeping track of modes in a slightly more complicated way than was the case for the classical algorithms described in the previous section. First, we need to find the modes for Roman numerals interpreted in a particular key/mode. The type JTriple is actually a synonym for TChord, but it is used for clarity to indicate that the pitch information represents a mode rather than a chord.

```
majorModes = allModes
minorModes = drop 5 allModes ++ (take 5 allModes)
chordMode :: CType → Key → AbsMode
chordMode ct (k,m) =
  let pModes = if m == Major then majorModes else minorModes
      ctMode = pModes !! fromEnum ct
      ck = pModes !! 0 !! fromEnum ct
  in t (k+ck) ctMode
toJTriple :: (Key, Dur, CType) → (Key, Dur, AbsMode)
toJTriple (km,d,c) = (km,d, chordMode c km)
```

Simple Jazz Foreground

This approach, called *jazzFG1* (or *jazzFG1T* to interface more directly with the grammar monad defined in Chapter 4), creates jazz chords and a stochastic bassline. **Let** instantiation only takes place at the level of Roman numerals. As a result, progressions such as **let** $x = A$ **in** $x\ x$ will only exhibit *abstract* repetition, while exact instantiations of the chord may differ.

```
jazzFG1 :: StdGen → [(Key, Dur, CType)] → (StdGen, Music Pitch)
jazzFG1 g chords =
  let [gJ, gR, gOPC, gB] = take 4 $ splitN g
      jts = map toJTriple chords
      ms = map (λ(a, b, c) → ([], c)) jts
      qJ = modeSpace' alg1Temps
      chordsJ = greedyProg qJ modeEq (const True) nearFallJ gJ ms
      qOPC = makeRange' alg1Rans // opcEq
      es = map (convOPC qOPC bassRoot) chordsJ
      chordsOPC = greedyProg' (const True) nearFall gOPC es
      chordsOPC' = zipWith (λ(a, b, c) x → (a, b, x)) jts chordsOPC
      jVoices = dtrans $ map toJNote chordsOPC'
      (gRet, bassLine) = stochBass gB $ head jVoices
  in (gRet, instrument AcousticBass bassLine :=:
      jnToMusic (repeat AcousticGrandPiano) (tail jVoices))
alg1Temps = [[0, 2, 4, 6], [0, 1, 2, 4, 6]]
alg1Rans = (34, 45) : take 4 (repeat (50, 64))
bassRoot (chrd, m) = (minimum chrd `mod` 12) == head (normO m)
splitN g = let (g1, g2) = split g in g1 : splitN g2
convOPC :: QSpace AbsChord → Predicate JChord → JChord →
           EqClass AbsChord
```

```
convOPC q pj (c,m) = filter (λx → pj (x,m)) $ eqClass q opcEq c

stochBass :: StdGen → [JNote] → (StdGen,Music Pitch)

stochBass g [] = (g,rest 0)

stochBass g ((km,d,p):t) =

  let (g',pat) = pickPattern g d p

      (g'',t') = stochBass g' t

  in (g'',pat:+:t')

pickPattern g d p =

  let (r,g') = randomR (0,length pats − 1) g

      f d p = note d (pitch p)

      pats = [f d p,

        if d ⩾ hn then f qn p:+:f (d − qn) p else f d p,

        if d ⩾ hn then f (d − en) p:+:f en p else f d p]

  in (g',pats !! r)

jazzFG1T :: StdGen → Key → Term CType → (StdGen,Music Pitch)

jazzFG1T g (k,m) t = jazzFG1 g $ ctTrans k $ toChords (expand [] t) m
```

Bossa Nova

This approach interprets Roman numerals through three separate chord spaces as described in Chapter 6 in order to cut down the task's combinatorics. As a result, this type of foreground can usually be generated much more quickly than the musically simpler one in the previous section—an illustration of the fact that musical simplicity does not always imply computational simplicity.

```
jazzFG2 :: StdGen → [(Key, Dur, CType)] → (StdGen, Music Pitch)
jazzFG2 g chords =
  let gs@[gJC, gJB, gJL, gRC, gRB, gRL, gOPC_C, gOPC_B, gOPC_L, gL] =
    take 10 $ splitN g
    jts = map toJTriple chords
    ms = map (λ(a, b, c) → ([], c)) jts
    qs@[qJC, qJB, qJL] =
      map modeSpace' [alg2TempsC, alg2TempsB, alg2TempsL]
    [chordsJ, bassJ, leadJ] = zipWith (λq gx → greedyProg q modeEq
      (const True) nearFallJ gx ms) qs $ take 3 gs
    qOPC_C = filter alg2FilterC (makeRange' alg2RansC) // opcEq
    qOPC_B = makeRange alg2RansB // opcEq
    qOPC_L = makeRange' alg2RansL // opcEq
    esC = map (convOPC qOPC_C (const True)) chordsJ
    esB = map (convOPC qOPC_B bassRoot2) bassJ
    esL = map (convOPC qOPC_L (const True)) leadJ
    chordsOPC = greedyProg' (const True) nearFall gOPC_C esC
    bassOPC = greedyProg' (noCPL 7) nearFall gOPC_B esB
    leadOPC = greedyProg' (noCPL 7) nearFall gOPC_L esL
    [cc, bc, lc] = map (zipWith (λ(a, b, c) x → (a, b, x)) jts)
      [chordsOPC, bassOPC, leadOPC]
    cm = bossaChords cc
    bm = bossaBass bc
    (gRet, lm) = bossaLead gL lc
  in (gRet, chord [instrument AcousticBass bm,
      instrument AcousticGrandPiano cm,
      instrument Flute lm])
```

Each part (bass, chords, lead) uses a different set of chord templates as well as different ranges and constraints for their respective chord spaces.

$$alg2TempsC = [[0,2,4,6],[1,2,4,6]] \quad \text{-- for chords}$$

$$alg2TempsB = [[0,4]] \quad \text{-- for bass}$$

$$alg2TempsL = [[0],[2],[4]] \quad \text{-- for lead}$$

$$alg2RansB = [(34,49),(34,49)]$$

$$alg2RansC = take\ 4\ \$\ repeat\ (50,64)$$

$$alg2RansL = [(65,80)]$$

$$bassRoot2\ ([b_1,b_2],m) = normO\ [b_1,b_2] == normO\ [m\ !!\ 0, m\ !!\ 4]$$

$$bassRoot2\ \_ = error\ \texttt{"(bassRoot2) Bad arguments."}$$

$$alg2FilterC\ x = sorted\ x \wedge pianoChord\ x$$

$$toTN2'\ (k,d,[p])\ b\ l = TNote2\ k\ d\ b\ l\ p$$

$$toTN2'\ \_\ \_\ \_ = error\ \texttt{"(toTN2') Bad arguments"}$$

$$tn2M\ (TNote2\ k\ d\ b\ l\ p) = note\ d\ (pitch\ p)$$

Conversion to *Music Pitch* also takes place independently for each voice. Instead of going through an intermediate type like *Voice* as done for the classical foregrounds, simple bossa nova features are added directly at the *Music* level.

The bass foreground will start with pairs of notes as a chord. How these two pitches are handled depends on the duration of the chord. For a whole note, a standard bossa nova bass pattern is used, but shorter durations can only use subsets of that pattern. Chords longer than a whole note are partitioned into whole note sections.

```
bossaBass :: [TChord] → Music Pitch
bossaBass [] = rest 0
bossaBass ((km,d,c@[p1,p2]):t) =
  if d > wn then bossaBass ((km,wn,c):(km,d-wn,c):t) else
  if d == wn then f1 p1 p2 :+: bossaBass t else
  if d == hn then f2 p1 p2 :+: bossaBass t else f3 p1 d :+: bossaBass t where
  f1 b1 b2 = f2 b1 b2 :+: f2 b2 b1
  f2 b1 b2 = f3 b1 (qn+en) :+: f3 b2 en
  f3 b1 d = note d (pitch b1)
bossaBass _ = error "(bossaBass) Bad input"
bossaChords :: [TChord] → Music Pitch
bossaChords [] = rest 0
bossaChords ((km,d,c):t) =
  if d > wn then bossaChords ((km,wn,c):(km,d-wn,c):t) else
  if d == wn then f1 c :+: bossaChords t else f2 d c :+: bossaChords t where
  f1 c = let c' = f2 en c in rest qn :+: c' :+: rest qn :+: c' :+: rest qn
  f2 d c = chord $ map (λp → note d $ pitch p) c
bossaLead :: StdGen → [TChord] → (StdGen,Music Pitch)
bossaLead g ts =
  let ls = take (length ts - 1) (repeat False) ++ [True]
      v = zipWith3 toTN2' ts (repeat 0) ls
      (g',v') = addFgToVoice jConsts (foreFunsJ defConsts) g v
  in (g', line $ map tn2M v') where
  foreFunsJ c = [(0.5,f1 c),(0.5,f2 c)] :: [(Double,ForeFun)]
  jConsts = CConstants 2 3 0.3 0.5 0.8 7
```

Finally, the foreground is given an easy interface to the *Term* type.

```
jazzFG2T :: StdGen → Key → Term CType → (StdGen, Music Pitch)
jazzFG2T g (k,m) t = jazzFG2 g $ ctTrans k $ toChords (expand [] t) m
```

Jazz Chords

In addition to the foreground algorithms already presented, for the purpose of mixing styles it is also useful to have a simpler mapping from Roman numerals to the *AbsChord* type without altering the original chords' durations or adding any additional melodic elements.

```
jazzChords :: StdGen → [(Key, Dur, CType)] → Constraints →
                    (StdGen, [(Key, Dur, AbsChord)])
jazzChords g chords consts =
  let [gJ, gOPC, g'] = take 3 $ splitN g
      jts = map toJTriple chords
      ms = map (λ(a,b,c) → ([],c)) jts
      qJ = modeSpace' alg1Temps
      chordsJ = greedyLet (const True) nearFallJ consts
                          (map (eqClass qJ modeEq) ms) gJ
      qOPC = makeRange' alg1Rans // opcEq
      es = map (convOPC qOPC bassRoot) chordsJ
      chordsOPC = greedyProg' (const True) nearFall gOPC es
  in (g', zipWith newP jts chordsOPC)
```

Bibliography

[1] Andres Garay Acevedo. Fugue composition wtih counterpoint melody generation using genetic algorithms. In *Computer Music Modeling and Retrieval*, volume 3310 of *Lecture Notes in Computer Science*, pages 96–106. 2005.

[2] J. L. Alty. Navigating through compositional space: The creativity corridor. *Leonardo*, 28(3):215–219, 1995.

[3] Stephen Baumann. A simplified attributed graph grammar for high-level music recognition. In *International Conference on Document Analysis and Recognition*, volume 2, pages 1080–1083, 1995.

[4] Matthew I. Bellgard and Chi-Ping Tsang. Harmonizing music the Boltzmann way. *Connection Science*, 6(2):281–297, 1994.

[5] Matthew I. Bellgard and Chi-Ping 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.

[6] Thomas Bonte, Nicolas Froment, and Werner Schweer. Musescore, 2014.

[7] Steven Brown, Michael J. Martinez, and Lawrence 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*, pages 2791–2803, 2006.

[8] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Compututing Intelligence and AI in Games*, 4(1):1–43, 2012.

[9] Peter Bühlmann and Abraham J. Wyner. Varable length Markov chains. *The Annals of Statistics*, 27(2):480–513, 1999.

[10] Michael D. Buhrmester, Tracy Kwang, and Samuel D. Gosling. Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality data? *Perspectives on Psychological Science*, pages 3–5, 2011.

[11] Clifton Callender, Ian Quinn, and Dimitri Tymoczko. Generalized voice-leading spaces. *Science Magazine*, 320(5874):346–348, 2008.

[12] Murray Campbell, A. Joseph Hoane Jr., and Feng-Hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83.

[13] Parag Chordia, Avinash Sastry, and Sertan Senturk. Predictive tabla modeling using variable-length Markov and hidden Markov models. *Journal of New Music Research*, 40(2):105–118, 2011.

[14] Alexander Clark. Efficient, correct, unsupervised learning of context-sensitive languages. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, pages 28–37, July 2010.

[15] Alexander Clark, Rémi Eyraud, and Amaury Habrard. A polynomial algorithm for the inference of context free languages. In *Proceedings of International Colloquium on Grammatical Inference*, pages 29–42, 2008.

[16] Bradley J. Clement. Learning harmonic progression using Markov models. In *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1998.

[17] Michael Collins. The inside-outside algorithm, 2014.

[18] Darrell Conklin. Melodic analysis with segment classes. *Machine Learning*, 65(2-3):349–360, December 2006.

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

[20] David Cope. On the algorithmic representation of musical style. In *Understanding Music with AI*, pages 354–363. MIT Press, 1992.

[21] David Cope. Facing the music: Perspectives on machine-composed music. *Mind*, pages 79–87, 1999.

[22] David Cope. *Computer Models of Creativity*. The MIT Press, 2005.

[23] David Cope. 5000 works in bach style, 2014.

[24] David S. Dummit and Richard M. Foote. *Abstract Algebra*. Prentice Hall, New Jersey, 1999.

[25] Kemal Ebcioğlu. An expert system for Schenkerian synthesis of chorales in the style of J.S. Bach. In *Proceedings of the International Computer Music Conference*, pages 135–140, 1984.

[26] Douglas Eck and Jürgen Schmidhuber. Learning the long-term structure of the blues. In *In Proceedings of the International Conference on Artificial Neural Networks*, pages 284–289, 2002.

[27] David Ferrucci et al. Building Watson: An overview of the DeepQA project. *AI Magazine*, pages 59–79, 2010.

[28] Michael J. Fischer. Grammars with macro-like productions. In *Proceedings of the 9th Annual Symposium on Switching and Automata Theory (Swat 1968)*, SWAT '68, pages 131–142. IEEE Computer Society, 1968.

[29] Judy A. Franklin. Recurrent neural networks and pitch representations for music tasks. In *Forida AI Research Symposium*, 2004.

[30] Judy A. Franklin. Recurrent neural networks for music computation. *INFORMS Journal on Computing*, 18(3):321–338, 2006.

[31] Peter Gannon. *Band-in-a-Box: Intelligent Music Accompaniment Software for Your Multimedia Computer*. PG Music Inc., 2002.

[32] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[33] Michael Gogins. Score generation in voice-leading and chord spaces. In *Proceedings of the International Computer Music Conference*, pages 593–600, 2006.

[34] William P. Headden, III, David McClosky, and Eugene Charniak. Evaluating unsupervised part-of-speech tagging for grammar induction. In *Proceedings of the 22Nd International Conference on Computational Linguistics*, volume 1 of *COLING '08*, pages 329–336, 2008.

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

[36] Paul Hudak. Euterpea, 2014.

[37] Paul Hudak and Jonathan Berger. A model of performance, interaction, and improvisation. In *Proceedings of International Computer Music Conference*, pages 541–548, 1995.

[38] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[39] David Huron. *Sweet Anticipation Music and the Psychology of Expectation*. A Bradford Book, 2008.

[40] Amazon Inc. Amazon Mechanical Turk, 2014.

[41] Kevin Tang Jon Gillick and Robert M. Keller. Learning jazz grammars. In *Proceedings of the Sound and Music Computing Conference*, pages 125–130, 2009.

[42] Robert M. Keller and David R. Morrison. A grammatical approach to automatic improvisation. In *Sound and Music Computing Conference*, pages 330–337, 2007.

[43] Phillip B. Kirlin and Paul E. Utgoff. A framework for automated Schenkerian analysis. In *International Conference on Music Information and Retrieval*, pages 363–368, 2008.

[44] Christopher Konopka. Csound, 2014.

[45] Hendrik Vincent Koops, Jose Pedro Magalhaes, and W. Bas de Haas. A functional approach to automatic melody harmonisation. In *Proceedings of ACM Workshop on Functional Art, Music, Modeling, and Design*. ACM Press DL, September 2013.

[46] Vipin Kumar. Algorithms for constraint satisfaction problems: a survey. *AI Magazine*, 13(1):32–44, 1992.

[47] K. Lari and S.J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech ad Language*, 4:35–56, 1990.

[48] Fred Lerdahl and Ray S. Jackendoff. *A Generative Theory of Tonal Music*. The MIT Press, 1996.

[49] Mark Levine. *The Jazz Theory Book*. Sher Music, 1995.

[50] Jose Pedro Magalhaes and W. Bas de Haas. Functional modelling of musical harmony: an experience report. In *Proceedings of the 16th ACM SIGPLAN international conference on functional programming*, pages 156–162, 2011.

[51] Matteo Mainetti. Symmetric operations on equivalence relations. *Annals of Combinatorics*, 7:325–348, 2003.

[52] Guerino Mazzola. *The Topos of Music: Geometric Logic of Concepts, Theory, and Performance*. Birkhuser, 2002.

[53] James McCartney. Rethinking the computer music language: Supercollider. *Comput. Music J.*, 26(4):61–68, December 2002.

[54] Jon McCormack. Grammar based music composition. *Complex Systems*, 96:320–336, 1996.

[55] Ruslan Mitkov. *The Oxford Handbook of Computational Linguistics*. The Oxford Prress, 2003.

[56] Robert D. Morris. Voice-leading spaces. *Music Theory Spectrum*, pages 175–208, 1998.

[57] Tomasz Oliwa and Markus Wagner. Composing music with neural networks and probabilistic finite-state machines. In *Proceedings of the Conference on Appications of Evolutionary Computing*, pages 503–508, 2008.

[58] Steve Pease. *FL Studio Power! The Comprehensive Guide*. Cengage Learning PTR, 2009.

[59] Simon Peyton Jones. The Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[60] Tim Place and Trond Lossius. Jamoma: A modular standard for structuring patches in Max. In *Proceedings of the International Computer Music Conference*, pages 143–146, 2006.

[61] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990.

[62] Donya Quick and Paul Hudak. Computing with chord spaces. In *Proceedings of the International Computer Music Conference*, pages 433–440, 2012.

[63] Donya Quick and Paul Hudak. Grammar-based automated music composition in haskell. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling, and design*, pages 59–70, 2013.

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

[65] Ian Quinn and Panayotis Mavromatis. Voice-leading prototypes and harmonic function in two chorale corpora. In *Proceedings of the International Conference on Mathematics and Computation in Music*, pages 230–240, 2011.

[66] Christopher Raphael and Joshua Stoddard. Functional harmonic analysis using probabilistic models. *Computer Music Journal*, 28(3):45–52, 2004.

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

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

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

[70] Dana Ron, Yoram Singer, and Saftali Tishby. The power of amnesia: learning probabilistic automata with variable memory length. *Machine Learning*, 25:117–149, 1996.

[71] Kenneth H. Rosen. *Discrete Mathematics and Its Applications (7th Ed.)*. McGraw-Hill, Inc., New York, NY, USA, 2012.

[72] Joel Ross, Lilly Irani, M. Six Silberman, Andrew Zaldivar, and Bill Tomlinson. Who are the crowdworkers?: shifting demographics in Mechanical Turk. In *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, pages 2863–2872, 2010.

[73] Heinrich Schenker. *Harmony*. University of Chicago Press, OCLC 280916, 1954.

[74] Stephen W Smoliar. A computer aid for schenkerian analysis. In *Proceedings of the 1979 Annual ACM Conference*, pages 110–115, 1979.

[75] Johan Sundberg, Anders Friberg, and Lars Frydn. Rules for automated performance of ensemble music. *Contemporary Music Review*, 3(1):89–109, 1989.

[76] David Temperley. Modeling common-practice rhythm. *Music Perception*, 27(5):335–376, 2010.

[77] David Temperly. *The Cognition of Basic Musical Structure*. The MIT Press, 2004.

[78] David Temperly. *Music and Probability*. The MIT Press, 2010.

[79] Alan M. Turing. Computing machinery and intelligence. *Mind*, 49:433–460.

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

[81] Karel van der Toorn, Bob Becking, and Pieter W. van der Horst. *Dictionary of deities and demons in the Bible*. Leiden, 1999.

[82] Christopher White. An alphabet-reduction algorithm for chordal n-grams. In *Proceedings of the International Conference on Mathematics and Computation in Music*, pages 201–212, 2013.

[83] Christopher W. White. *Some Statistical Properties of Tonality, 1650-1900*. PhD thesis, Yale University, 2013.

[84] Gerhard Widmer and Werner Goebl. Computational models of expressive music performance: The state of the art. *Journal of New Music Research*, 33(3):203–216, 2004.

[85] Terry Winograd. Linguistics and the computer analysis of tonal harmony. *Journal of Music Theory*, 12(1):2–49, 1968.

[86] Peter Worth and Susan Stepney. Growing music: musical interpretations of l-systems. *Applications on Evolutionary Computing*, pages 535–540, 2005.

[87] Liangrong Yi and Judy Goldsmith. Automatic generation of four-part harmony. In *UAI Applications Workshop*, 2007.

[88] Jason Yust. The geometry of melodic, harmonic, and metrical hierarchy. In *Mathematics and Computation in Music*, volume 38, pages 180–192, 2009.