Technical University of Crete

Department of Electronic and Computer Engineering

Computer Aided Music Composition using Inductive Logic Programming

By:

Emmanuel-Theofanis Chourdakis

Submitted in July 2011 in partial fulfilment of the requirements for the Diploma degree in Electronic and Computer Engineering.

Thesis Committee:
Assistant Professor Michail G. Lagoudakis (Supervisor)
Associate Professor Alexandros Potamianos
Professor Minos Garofalakis

Πολυτεχνειο Κρητης

Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

Συνθεση Μουσικης Υποβοηθουμενη απο Υπολογιστη μεσω Επαγωγικου Λογικου Προγραμματισμου

Εμμανουήλ-Θεοφάνης Χουρδάκης

Υποβλήθηκε τον Ιούλιο του 2011 προς μερική εκπλήρωση των υποχρεώσεων για το Δίπλωμα Ηλεκτρονικού Μηχανικού και Μηχανικού Υπολογιστών

Εξεταστικη Επιτροπη

Επικουρος Καθηγητης Μιχαηλ Γ. Λαγουδακης (Επιβλεπων)
Αναπληρωτης Καθηγητης Αλεξανδρος Ποταμιανος
Καθηγητης Μινως Γαροφαλακης

Abstract

Music is organized sound and comes as the result of combining timbre and rhythm through a process known as composition. Computer-Aided (Music) Composition refers to using computers to compose music similarly to the way engineers use Computer-Aided Design (CAD) systems to design buildings or integrated chips. There is a plethora of Computer-Aided Algorithmic Composition (CAAC) systems with various characteristics that allow composers to work on their music at various levels of detail through formal (algorithmic) specifications of the desired outcome. In this thesis we describe a method for computer-aided music composition based on the CAAC system Strasheela and Inductive Logic Programming (ILP). We use an ILP learning algorithm to learn musical rules from examples (existing music pieces) in the form of first-order logic Horn clauses. We then transform these rules to constraints and form a Constraint Satisfaction Problem (CSP) which is solved by Strasheela to produce a new music composition that adheres to the input rules. The proposed method can be used in a variety of ways, allowing composers to form different combinations of automatically-learned and hand-written rules to yield a wide range of different music compositions.

Περίληψη

Η μουσική είναι οργανωμένος ήχος και προκύπτει ως αποτέλεσμα του συνδυασμού ηχοχρώματος και ρυθμού μέσα από μια διαδικασία γνωστή ως σύνθεση. Ο όρος Σύνθεση (Μουσικής) Υποβοηθούμενη από Υπολογιστή (Computer-Aided – Music - Composition) αναφέρεται στη χρήση υπολογιστών για σύνθεση μουσικής παρόμοια με τον τρόπο που οι μηχανικοί χρησιμοποιούν υπολογιστές για να σχεδιάσουν κτίρια ή ολοκληρωμένα κυκλώματα με τη βοήθεια συστημάτων CAD. Υπάρχουν πολλά συστήματα για Αλγοριθμική Σύνθεση Υποβοηθούμενη από Υπολογιστή (Computer-Aided Algorithmic Composition - CAAC) με διάφορα χαρακτηριστικά που επιτρέπουν στους συνθέτες να επεξεργασθούν τη μουσική τους σε διάφορα επίπεδα λεπτομέρειας μέσω τυπικών (αλγοριθμικών) προδιαγραφών για το επιθυμητό αποτέλεσμα. Στην εργασία αυτή περιγράφουμε μια μέθοδο για μουσική σύνθεση με τη βοήθεια υπολογιστή που βασίζεται στο CAAC σύστημα Strasheela και σε Επαγωγικό Λογικό Προγραμματισμό (Inductive Logic Programming - ILP). Χρησιμοποιούμε έναν αλγόριθμο μάθησης ILP για να μάθουμε μουσικούς κανόνες από παραδείγματα (υπάρχοντα μουσικά κομμάτια) στη μορφή κανόνων Horn λογικής πρώτης τάξης. Στη συνέχεια, μετατρέπουμε αυτούς τους κανόνες σε περιορισμούς και διατυπώνουμε ένα πρόβλημα ικανοποίησης περιορισμών (Constraint Satisfaction Problem - CSP) που επιλύεται από το Strasheela για να παράγει μια νέα σύνθεση που τηρεί τους κανόνες εισόδου. Η προτεινόμενη μέθοδος μπορεί να χρησιμοποιηθεί με διάφορους τρόπους, επιτρέποντας στους συνθέτες να διατυπώσουν διάφορους συνδυασμούς από αυτόματα παραγόμενους και ιδιόχειρα γραμμένους κανόνες για να παράγουν ένα ευρύ φάσμα διαφορετικών μουσικών συνθέσεων.

Contents

List of Figures

Chapter 1

Introduction

1.1 Music, Mathematics, and Composition

Every human on earth is familiar to some extent with music. One may not be able to give a correct definition of music, but most people can understand that it is something subjective and relates directly to our mood. Most people can relate emotions to music. A song can be joyful, a song can be sad. Music can be entertaining by itself, can be used to ephasize other forms of art (i.e. cinema), can even be used for emotion manipulation and propaganda. It clearly relates to our state of mind and some would argue that it even affects our physical health.

A definition for music has been the subject of a long debate. The word itself comes from the greek work "mousike" which reffered to the arts and sciences governed by Muses, the ancient Greek Goddesses of Art and Sciences. Clearly, the word meant much more than the sound-related form of art we refer to today as *music*. In ancient Rome, it meant both poetry as well as instrumental music and in the middle ages, it had the same importance as arithmetics, geometry and astronomy. Only in the fifth century, the concept of music as we know it started appearing, when Boethius split the concept of music into three major kinds: *Musica Universalis*, *Musica Humana*, and *Musica Instrumentalis*. The first two did not relate to sound itself but rather to the order of the universe and the mathematical proportions of universal bodies like planets and stars (*Musica Universalis*), and the porportions of the human body (*Musica Humana*). Only the last one (*Musica Instrumentalis*), which was considered the lesser of three, refered to music as something sung or generated by instruments. A modern and popular definition is given by Edgard Varèse: "Music is Organized Sound", meaning that "certain timbres and rhythm can be grouped together" [1].

The relationship between music mathematics, goes even further. In the teachings of Pythagoras, music was inseperable from numbers, which were thought to be the key to the whole spiritual and physical universe. "So, the system of musical sounds and rhythms, being ordered by numbers exemplified the harmony of the cosmos and corresponded to it" [2]. The first musicians used numbers and nature-derived mathematical properties to construct their first systems. Ptolemy, the leading astronomer of the time, believed that mathematical laws "underlie the systems of both music intervals and heavenly bodies" and that certain notes "correspond with particular planets, their distances from each other and their movements".

Algorithmic techniques for composing music come as a natural consequence. Later, in the fourteenth and fifteenth century, we see formal methods in music composition, like the isorhythmic technique. We also have the birth of the Baroque fugue and the Classical sonata. There is also an interest on ratios in compositions (i.e. the golden ratio) and even Mozart has been thought of having used algorithmic-like techniques in his compositions at least once.

With the end of World War II, composers like Arnold Schoenberg, Iannis Xenakis, and György Ligeti introduced their own composition techniques governed by complex, even algorithmic, procedures. Also, the development of software algorithms in other fields have led to compositions "seeded" by fields outside music (i.e. Chaos Theory).

Computer-Based Algorithmic Composition made its debut in 1956, when Lejaren Hiller and Leonard Isaacson programmed an Illiac computer to do counterpoint, resulting in the famous *Illiac Suite for a String Quartet* [3].

Up to now, there has been a large amount on work on computer-based composition. We can find compositions based on statistical, mathematical systems, logic programming, neural networks, genetic algorithms, etc.

1.2 Thesis Contribution

Inductive Logic Programming (ILP) is a much-studied subfield of Machine Learning that utilizes Logic Programming. ILP algorithms have as their objective the derivation of valid hypotheses, given some background knowledge and a set of examples.

Constraint Satisfaction Problems (CSPs) are problems that consist of a set of variables, domains of values for these variables, and a set of costraints that limit the possible value assignments over subsets of variables. A solution to a CSP problem is an assignment of values to all variables that satisfies all constraints.

In this thesis, we use an Inductive Logic Programming algorithm in order to learn musical rules, from given examples (existing music compositions), in the form of first-order logic Horn clauses, transform these rules to constraints, and feed them to the CSP solver of Strasheela, a Computer-Aided Composition System, with the goal of generating new compositions that adhere to these rules. We then show how this method can be used for computer-aided music composition by modifying these rules, or combining them with hand-written rules.

1.3 Thesis Overview

In **Chapter 2**, the necessary background knowledge for understanding this work is given. This chapter includes an introduction to Music Representation and to Computer-Aided Algorithmic Composition systems, an introduction to First Order Logic, Logic Programming, and Inductive Logic Programming, and finally, some preliminaries on Constraint Satisfaction Problems and Constraint Programming, as well as their relation to the First-Order Logic.

Chapter 3 describes the problem we study in this thesis, and also gives an overview of existing related work.

Chapter 4 includes a detailed account of our approach, as well as illustrative examples for all steps of the method.

Chapter 5 demonstrates the application of our method to music composition under various settings and provides detailed examples.

Chapter 6 discusses the problem further, gives some food for thought, and also provides some ideas for future work.

Finally, **Appendix A** provides usage instructions for our software implementation.

Chapter 2

Background

2.1 Music Representation

By the term "Music Representation" we mean the form of a musical piece or of a subset of it, which is written on a physical medium or stored into a digital storage medium. It is essential for any music representation form to be concise and contain all information required for a complete and correct reproduction of the music stored.

2.1.1 Human Music Representation

The most common way of representing music by and for humans is the music sheet. "It is a handwritten or printed form of music notation that uses modern musical symbols"[4]. This notation can be displayed on a physical medium, such as a book, or on a computer screen. It serves the purpose of concisely and correcly recording a musical piece in order to enable a trained user, to reproduce what is written in it. It comes in various forms, depending on various parameters, like the kind of music recorded (for example, Byzantine Music has different notation style from Western Classical Music), the musical instrument it refers to (for example, percussions and piano sheet differ), as well as its purpose (for example, in Jazz Music the sheet serves mostly as general instructions, while a classical piece performance requires exact reproduction). A general feature of sheet music is that it relies on the training of the user for the correct interpretation and reproduction of the information stored. This generally comes after years of training, and strongly depends on the natural capabilities of the user.

2.1.2 Computer Music Representation

On the other hand, the representation of music on computers serves a totally different purpose. Human sheet music targets the user and needs to provide a correct and efficient way for him to reproduce what is written. This often comes with a lack of implied information. For example, there is no need for the human sheet music to contain the precise velocity that the piano performer strikes the piano keys with in a crescendo. In fact, such information would make it harder for the performer to read the sheet and would require extreme controlling skills of his fingers. In its place, the "crescendo" symbol would let the performer's mind to "fill-in the gaps" on how much force should apply to each finger.

Such implication of information is generally avoided in computer music, where we usually provide the entirety of the information. The exact note, the exact time, the exact duration and the exact velocity is information known to the computer at every moment. Such information is provided in an explicit way. The most popular music representation protocol is MIDI[1] which stores the exact information needed for a complete reproduction of a wide set of music pieces.

However, there is an exception to the above statement, where information must not be explicitly provided. This is when we deal with music stored into the computer, but for the sole purpose of reproducing human sheet music. There are various formats for storing music sheets and various software packages that allow the reproduction of the represented music by the computer. In this kind of

[1] Musical Instrument Digital Interface

software, usually the computer tries to emulate the way the human mind "fills-in the gaps". For example, given the *crescendo* symbol, the computer would gradually increase the velocity that the notes are played with.

2.1.3 Micro and Macro Scale

Up to now, we have talked about representing music at a microscopic scale. That means we need to provide detailed information for every note being played (for example, at least the pitch, duration, and position of a note within a music piece). This is a very convenient representation, if we want to have complete control over music composition and we have exactly on our mind how every instrument will sound at every moment of play. However, sometimes this microscopic scale of representation can be sometimes very tiring for the composer.

Another approach is to see the composition at a macroscopic scale. The exact positions and durations of the exact notes may not be of primary concern. We see the composition as a greater "plan", taking a rather top-down approach, by constructing an abstract skeleton and completing the details later. For example, if we wanted to compose a "fugue" (a strictly structured form) we would start by defining its tonic key, the number of voices and its subject, establishing first its general structure, and then completing the details, rather than starting to compose it note-by-note.

2.2 Computer-Aided Algorithmic Composition

Algorithmic composition in music is "the process of using some formal process to make music with minimal human intervention"[5]. Algorithmic processes have been used widely for a very long time, especially in western music. The term "Algorithm" may sound alien to music, but even basic western music studies, like harmony or counterpoint, take a somewhat algorithmic approach. For example, the 4-voice harmonization of a given bassline has strict rules which must be followed, together with some degrees of freedom. More complicated composition examples, include for example the baroque "fugue" mentioned above, the classical sonata, or the more recent form of twelve-note serialism compositions.

Usually, when we speak of algorithmic composition though, we mean more complex methods of composition based on, for example, fractals, L-systems, generative grammars, neural networks, statistical models, etc. An example of this is Ioannis Xenakis' use of Maxwell and Boltzmann's "Kinetic theory of Gases" for *Pithoprakta* [3].

Computer-Aided Algorithmic Composition (CAAC) is music composition based on algorithms, where all steps of creation are supported by software. It may help the reader understand the term better, if we relate it to Computer-Aided *Design*, that is widely used e.g. for architectural or electronics design. Similarily, the composer becomes a "designer". He designs several aspects of the composition, without really getting into the trivial details, if he does not want to, and freeing his mind to work on the music at a greater, more abstract, level (or, in some situations, deeper level). The computer *aids* the composer in his effort.

Lejaren Hiller and Leonard Issacson were generally aknowledged as the first to use a computer for algorithmic composition on their famous *Illiac Suite*, a

string quartet split into four movements, composed after Hiller's idea to change some chemistry-targeted code to do counterpoint [6].

The idea of algorithmic composition using computers became more popular over the decades and, together with the increasing processing power and storage size, we have seen various nice works, both inside and outside academia. The reader can get a taste of what computer-generated composition is about, by directing his web browser at *WolframTones* [7].

Categories of CAAC systems Ariza in his text *"Navigating the landscape of Computer-Aided Algorithmic Composition Systems: A definition, seven descriptors and a lexicon of systems and research."*[8] categorizes the various CAAC systems according to 7 distinct descriptors: Scale, Process Model, Idiom-Affinity, Extensibility, Event Production, Event Production, Sound Source and User Environment. There are systems that focus on micro (e.g. Supercollider, AC Toolbox) or macro-scale (e.g. CGMusic), realtime (e.g. Algorithmic Composer, Lexicon-Sonate), or non-realtime (e.g. AthenaCL), that restrict to a single musical style, or to more than one, that are extendable or non extendable, that process predetermined music, or music generation facilities (e.g. algorithms, lists, stochastic processes, etc.), that produce the sound themselves, or configure an external program for sound synthesis and also use various ways of interacting with the user.

CAAC vs Automated Composition Both automated composition and computer-aided composition rely on the same technical means. The main difference between automated composition and its computer-aided counterpart is mainly the goals of each. While in automated composition we would want to have as minimum human intervention as possible, leaving the composition, if possible in its entirety, to the computer, in CAAC the composer remains active and the computer just helps him along the process, but does not complete the composition by itself.

Usually, automated composition is a research subject on the modeling of conventional music and the validation of the process is usually being done by some kind of Turing Test, where a human is being asked to discriminate between a man-made and a computer-made piece [9].

2.3 First-Order Logic

First-Order Logic (FOL), is a declarative, context-free representation language that allows representation of knowledge and reasoning in an unambiguous and concise way. It is declarative, as opposed to the procedural languages (like C++, Java, or Lisp), meaning that one can do inference without having to provide specialized procedures. It is context-free, as it can be generated by a context-free grammar. And it is unambiguous, as every element of the language has a unique meaning. It also has the expressive power to allow for reasoning. Its statements can be composed of simpler statements, and unlike Propositional Logic, which requires every primitive statement to have its own symbol, we can fully and concisely represent those simpler statements by a combination of objects and relations between them. Any (simple or complex) FOL statements can be either *true* or *false*.

2.3.1 Expressive Power of First-Order Logic

The elements First-Order Logic deals with are *objects*, *relations*, and *functions*. Examples of these elements are shown below:

- Objects: notes, voices, pauses, cleffs, tempo, "c5", "C Minor Pentatonic", "sharp", barline, 65, etc.
- Relations: "a third up from", "relative minor of", "has the same tonality as", etc. Relations can be unary relations (or *properties*) like "quarter note", or $n-$ary relations, like "has a tonality of".
- Functions[2]: "belongs to voice", "has composer", etc.

For example, the sentence:

> The C Major scale, has the C note as it is tonic note.

includes the following elements:

- Objects: "The C Major scale", "the C note".
- Function: "has as its tonic note"

Note that "has as its tonic note" is a function, since every scale has exactly one tonic note. Also note that we can only use "The C Major scale" as a single object, and not as a relation to the C scale, for example with the property: "the scale is major" (at least not directly). In order to do that we would need to use *Higher-Order Logic*; something that is out of the scope of this text.

FOL does not limit its expressive power to single objects. Continuing on the subject of tonality, we can express things like:

> Every scale has a tonic note.

That is, FOL can express things about some or all of the objects as well.

2.3.2 Terms, Variables, Sentences, Quantifiers

Terms are logical expressions that refer to objects [10]. A constant symbol can be a term, and an application[3] of constant symbols to another constant symbol can be a term as well. Examples of *terms*:

- $C5$
- $Cmajor$
- $FifthOctave(C)$

An *Atomic Sentence* is the application of terms to a predicate. Examples of *Atomic Sentences*:

- $Major(C)$

[2] A Function is a Relation that has "output arguments" so that every set of "input arguments" correspond to exactly one output argument. For example, each musical piece has exactly one composer, every distinct note belongs to exactly one voice, etc.

[3] An application of Y and Z to X would be Y and Z in parentheses '(' and ')' separated with the comma symbol ',' and next to X, that is $X(Y, Z)$.

- $HasDuration(Note1, HalfNote)$

Sentences can be combined, using *Connectives* ($\Rightarrow, \wedge, \vee$, and $\Leftrightarrow$), into complex sentences. For example, below is a valid complex sentence:

$$HasDuration(Note1, HalfNote) \wedge HasPitch(Note1, FifthOctave(C))$$

Quantifiers are the ways of expressing in FOL that we want a sentence to hold for *some* or for *all* the objects. The universal quantifier ($\forall$) will make the sentence following it hold for every possible substitution of the variable that it quantifies, *true*. The existential quantifier ($\exists$) on the other hand will make the sentence *true*, for some (at least one) of the possible substitutions. For example, if we want to express the fact that there exists a pitch, such that every other pitch is higher or equal, we would write in FOL:

$$\exists p \forall v \; Pitch(p) \wedge Pitch(v) \Rightarrow IsHigherOrEqual(v, p)$$

And, if the only objects we have, are the pitches $C5$, $D5$ and $E5$ the above statement would hold true, if we substituted p with $C5$ and v with any of $C5$, $D5$, or $E5$.

The symbols p and v are called *variables*. They are also *terms*, so they can be used as arguments to a relation or function. A term that has no variables is called a *ground term*.

2.3.3 Literals, Clauses, Horn Clauses

Literals can be simple sentences (for example, $HasPitch(Note1, FifthOctave(C))$) or their negation. A *Clause* is a disjunction of literals. A clause can be represented as the set of literals that participate in the disjunction. A clause can be seen as the implication of the head (one of the literals) from the body (the negation of the rest of the literals)[4]. For example:

$$\exists p \forall v \; \underbrace{\neg Pitch(p) \vee \neg Pitch(v) \vee IsHigherOrEqual(v, p)}_{\text{a clause}}$$

is equivalent to:

$$\exists p \forall v \; \neg (Pitch(p) \wedge Pitch(v)) \vee IsHigherOrEqual(v, p)$$

which is equivalent to:

$$\exists p \forall v \; \underbrace{Pitch(p) \wedge Pitch(v)}_{\text{Body}} \Rightarrow \underbrace{IsHigherOrEqual(v, p)}_{\text{Head}}$$

We can also write this clause as follows:

$$\exists p \forall v \; \underbrace{IsHigherOrEqual(v, p)}_{\text{Head}} \leftarrow \underbrace{Pitch(p) \wedge Pitch(v)}_{\text{Body}}$$

A *Horn clause* is a clause that has exactly one non-negative literal (the head) and all the remaining literals (the body – if any) are negative. The clause above is a Horn clause.

[4] Recall that, given p, q, $\neg p \vee q \equiv p \Rightarrow q$.

2.3.4 Models and Interpretations

A sentence in FOL without a meaning is useless to us, unless we give it some meaning. The meaning of $IsHigherOrEqual(v, p)$ it that it represents two pitches and that its second argument is higher than or equal to the first. That is, $IsHigherOrEqual(v, p)$ holds *true*, if pitch p is higher or equal to pitch v.

Before we define *interpretation*, we need to define *domain*:

Definition 1. *A **domain** D is a set of objects that can be applied to relation and functions symbols.*

An interpretation is defined as follows:

Definition 2. *An **interpretation** I in First-Order Logic consists of a non-empty domain D and a mapping for function and predicate symbols. Every $n-$ary function symbol is mapped to a function from D^n to D, and every $n-$ary predicate symbol is mapped to a function from D^n to Boolean values true or $false$ [11].*

Finally, give the definition of a *model* for FOL.

Definition 3. *A **model** M of a first-order logic formula F is an interpretation I for which the formula holds true.*

2.3.5 FOL Syntax and PROLOG Representation

The syntax of FOL can be described using BNF grammar notation, as shown in Figure 2.1. In this grammar, the constant symbols begin with an upper-case letter and the variables with a lower-case letter. This is somewhat different from the PROLOG-like notation we use in this text, where we begin constant symbols with a lower-case letter and variables with an upper-case letter.

The sentence above:

> Every scale has a tonic note.

would be given in FOL syntax as:

$$\forall x \; Scale(x) \Rightarrow HasATonicNote(x)$$

While in PROLOG, we would write the same sentence as:

$$hasATonicNote(X) \leftarrow scale(X).$$

The reader may have noted some characteristics of PROLOG notation and some differences with typical FOL notation:

- Constant symbols in PROLOG begin with a lower-case letter, while variables begin with an upper-case letter.
- There is no quantifier. In a PROLOG statement, the universal quantifier, is implied.
- There is an ending dot ".". Every PROLOG statement ends with a dot.

There are more differences not shown in the above example:

- Every PROLOG statement is a Horn clause.
- The logical connectives $\wedge$ and $\vee$ are replaced by the comma (",") and the semicolon (";") respectively.
- Numbers are valid terms, as are lists. Lists are LISP-like, they consist of a Head and a Tail. The Tail can be bound to another list or *nil* (the empty list symbol []).
- The equality symbol in PROLOG is the unification symbol. The "="(A,B) relation is true if A can be unified with B (i.e. there is a possible substitution of the variables in A and B that makes A and B equal).
- Some relation predicates in PROLOG can be used as "infix" operators, for example, the number addition "+", subtraction "-", or the inequality relation "<".

For a more complete view in PROLOG syntax, see [12]. From now on in this text, when we refer to First-Order Logic, we will mean its PROLOG representation.

Now that we have explained the required terminology, we can define the *logic program*.

Definition 4. *A **Logic Program** is a conjunction of logical clauses.*

2.3.6 PROLOG Queries and Production Systems

The useful thing about PROLOG is that we can make queries about the state of our world. But first, we need to describle our world.

Facts and Rules

A *fact* is a unit clause, that describes facts about our world. For example:

$$pitch(c).$$

describes the fact that "c" is a pitch. Facts, are not necessarily ground. For example, the fact below is valid:

$$something(X).$$

although not very useful.

A *rule* is a Horn clause that describes some rule in our world. For example:

$$melodicInterval(X, Y) \leftarrow pitch(X), pitch(Y), next(X, Y).$$

means that a melodic interval exists between X and Y, when they are both pitches and they appear next to each other.

Queries

Now that we have a way to describe our world, we can make questions about its state. In PROLOG, we can make questions like "Is 'c' a pitch?" or "What are the pitches?". In the first question, PROLOG would answer with a "yes" and in

the second with "c,c#,d,...". The first question can be written as the PROLOG query:

$$pitch(c).$$

and the second as:

$$pitch(X).$$

which would return every possible substitution of X for which $pitch(X)$ holds (for example c, since $pitch(c)$ holds).

Rule Production and Production Systems

Production systems are usually software that apply a set of rules to a set of facts that reside in their knowledge base in order to produce new facts. For example, if we have the following rule in our knowledge base:

"All men are mortal"

and we provide the system with the following statement:

"Socrates is a man"

The production system, will produce the following fact:

"Socrates is mortal"

Similarily, if we have the following two rules into our system:

An interval of 7 semitones is a "Perfect Fifth".

and

"Unisons", "Octaves" and "Perfect Fifths" are "Perfect Consonances".

and we provide that we have a "c5" and a "g5" note, the system would immediately know that we have a perfect consonance.

Production systems typically utilize a reasoning technique called "Forward Chaining". For the above example, suppose we have rules in the form:

$$LHS \Rightarrow RHS.$$

where LHS are the conditions we check. When these are met, we say that the rule is "fired" and the facts in RHS are inserted into the knowledge base.

In the above example, we have the following rules:

$$\begin{aligned} note(X), note(Y) &\Rightarrow distance(X - Y) \\ distance(0) &\Rightarrow unison \\ distance(7) &\Rightarrow perfectFifth \\ distance(12) &\Rightarrow octave \\ perfectFifth &\Rightarrow perfectConsonance \\ octave &\Rightarrow perfectConsonance \end{aligned}$$

When we first insert the facts $note(60)$ ($C5$) and $note(67)$ ($G5$) into our knowledge base, the first rule is fired, and we have the facts $distance(0)$ and $distance(7)$ inserted into our database. Because of the insertion of $distance(7)$,

$$
\begin{array}{rcl}
Sentence & \rightarrow & AtomicSentence \\
& | & (Sentence\ Connective\ Sentence) \\
& | & Quantifier\ Variable, \ldots\ Sentence \\
& | & \neg Sentence \\
\\
AtomicSentence & \rightarrow & Predicate(Term, \ldots) \\
& | & Term = Term \\
\\
Term & \rightarrow & Function(Term, \ldots) \\
& | & Constant \\
& | & Variable \\
\\
Connective & \rightarrow & \Rightarrow \mid \wedge \mid \vee \mid \Leftrightarrow \\
\\
Quantifier & \rightarrow & \forall \mid \exists \\
\\
Constant & \rightarrow & C5 \mid PerfectConsonance \mid Sixteenth \\
\\
Variable & \rightarrow & voice \mid position \mid v \ldots \\
\\
Predicate & \rightarrow & Note \mid MelodicInterval \mid Pitch \mid \ldots \\
\\
Function & \rightarrow & Composer \mid NumberOfVoices \mid \ldots
\end{array}
$$

Figure 2.1: The syntax of First-Order Logic with equality in BNF.

the fact $perfectFifth$ is also appended in our knowledge base by the hird rule. Because of the insertion of $perfectFifth$, the fifth rule also fires and we have a $perfectConsonance$ to our knowledge base.

If we want to show that, given a set of facts KB, after the insertion of fact F, we produce the facts A_i, we can also write it as:

$$KB \wedge F \vdash A_i$$

and we say that KB and F *entail* A_i.

2.4 Inductive Logic Programming

Logic programming is the use of logic as a declarative representation language and a theorem prover or model-generator for the purpose of solving a problem. Inductive logic programming (ILP) is the "marriage" between logic programming and machine learning. Given a set of logical facts as the already known

background knowledge base, a set of rules that apply, and a set of positive and negative examples, an Inductive Logic Programming system will find a hypothesis that is valid throughout all the positive examples and in none of the negative examples.

In order to present the problem of induction we need to define *entailment*.

Definition 5. *A* clause F_1 syntactically entails F_2 *($F_1 \vdash F_2$) if and only if* F_2 *can be deduced from* F_1.

For short, the problem of induction can be stated as follows:

Definition 6. *Given a set* $\mathcal{O}$ *of observations and a consistent background knowledge* $\mathcal{B}$*, we need to find a hypothesis (a set of clauses)* $\mathcal{H}$ *such that:*

$$\mathcal{B} \wedge \mathcal{H} \vdash \mathcal{O}$$

For the example with Socrates in the section above, if we have the two relations *man* and *mortal*, and the positive examples below:

$$\begin{aligned} man(socrates) \\ man(plato) \\ man(aristeides) \\ mortal(socrates) \\ mortal(plato) \\ mortal(aristeides) \\ mortal(alcmene) \\ mortal(casiopeia) \end{aligned}$$

and the negative examples:

$$\begin{aligned} \neg man(alcmene) \\ \neg man(cassiopeia) \end{aligned}$$

and we want to find a hypothesis about the *mortal* relation, an Inductive Logic Programming system would probably yield the following answer:

$$mortal(X) \leftarrow man(X).$$

That means, every man is a mortal, but every mortal must not necessarily be a man (in our examples Alcmene and Cassiopeia are women).

Known ILP systems include Golem [13], Progol [14], Aleph [15], etc. Learning in these systems is accomplished in various ways. For example, Progol uses inverse entailment, while Golem uses Recursive Least General Generalizations (RLGG). In our work, we use the ILP algorithm PAL [16], which uses least general generalizations (LGG), therefore we will give a brief overview of what generality and LGG are.

Generality

We provide some necessary definitions for the better understanding of the concept of Generality.

Definition 7. *Given* formulas *A and B, we say that $A \vDash B$ (or A **semantically entails** B) if and only if every* model *of A is also a* model *of B.*

Definition 8. *We say that a formula A is **more general** than formula B if and only if $A \vDash B$ and $B \nvDash A$.*

For example, the clause $note(1, P, Pitch_1, D_1), note(2, P, Pitch_2, D_1)$ is more general than $note(1, 1, 62, D_1'), note(2, 1, 64, D_1')$ since we can entail the second clause from the first, by doing the subtitutions $P/1$,$Pitch_1/62$,$Pitch_2/64$ and D_1/D_1', but there is no subtitution we can do to the second clause in order to entail the first. If the second was possible, we would say that the two clauses were *equivalent*.

Definition 9. *We say that a formula A is **equivalent** to formula B if and only if $A \vDash B$ and $B \vDash A$.*

For example, the clauses:

$$note(1, P, Pitch_1, D_1), note(2, P, Pitch_2, D_1)$$

and

$$note(1, P', Pitch_1', D_1'), note(2, P', Pitch_2', D_1')$$

are equivalent. Now that we have defined *equivalence* we can define *redundancy*:

Definition 10. *A clause C is **redundant** to program $P \wedge C$ if and only if $P \wedge C$ is **equivalent** to P.*

For example, if we have

$$P \equiv note(Voice, Position, Pitch)$$

and

$$C \equiv note(Voice', Position', Pitch')$$

then C is rendundant to $P \wedge C$. Redundancy can be applied to literals as well:

Definition 11. *A literal l ls **logically redundant** within the clause $C \vee l$ in the program $P \wedge (C \vee l)$, if and only if $P \wedge (C \vee l)$ is **equivalent** to $P \wedge C$.*

For example, $l \equiv note(2, P', Pitch')$ in $C \vee l \equiv note(1, P, Pitch) \vee note(2, P', Pitch')$ is redundant in $P \wedge (C \vee l)$, if $P \equiv note(V, P, Pitch)$.

We will now see the notion of θ-subsumption:

Definition 12. *We say that a clause C_1 θ-subsumes clause C_2, if there exists a substitution θ such that $C_1\theta \subseteq C_2$.*

With the help of θ-subsumption we can define the **more general** relation:

Definition 13. *A clause C_1 is **more general** than C_2 if and only if $C_1\theta$-subsumes C_2.*

LGG of terms:

1. $lgg(t, t) = t$.
2. $lgg(f(s_1, \ldots, s_n), f(t_1, \ldots, t_n)) = f(lgg(s_1, t_1), \ldots, lgg(s_n, t_n))$.
3. $lgg(f(s_1, \ldots, s_m), g(t_1, \ldots, t_n)) = V$, where $f \neq g$ and V is a new variable which represents $lgg(f(s_1, \ldots, s_m), g(t_1, \ldots, t_n))$.
4. $lgg(s, t) = V$, where $s \neq t$ and at least one of s and t is a variable; in which case, V is a new variable which represents $lgg(s, t)$.

LGG of atoms:

1. $lgg(p(s_1, \ldots, s_n), p(t_1, \ldots, t_n)) = p(lgg(s_1, t_1), \ldots, lgg(s_n, t_n))$, if atoms have the same predicate symbol p.
2. $lgg(p(s_1, \ldots, s_n), q(t_1, \ldots, t_n))$, is undefined if $p \neq q$

LGG of literals:

1. If L_1 and L_2 are atoms, then the LGG is defined as above.
2. If L_1 and L_2 are both negative literals, and $L_1 = \neg A_1$ and $L_2 = \neg A_2$ then $lgg(L_1, L_2) = lgg(\neg A_1, \neg A_2) = \neg lgg(A_1, A_2)$.
3. If L_1 is a positive literal, and L_2 is a negative literal, or vice-versa, then $lgg(L_1, L_2)$ is undefined.

LGG of clauses:

1. Let there be two clauses C_1 and C_2 such that $C_1 = \{L_1, \ldots L_n\}$ and $C_2 = \{K_1, \ldots, K_m\}$, then:

   $lgg(C_1, C_2) = \{L_{ij} = lgg(L_i, K_j) | L_i \in C_1, K_j \in C_2,$ and $lgg(L_i, K_j)$ is defined$\}$

Figure 2.2: The lgg operator, defining the LGGs between the two operands.

Least General Generalizations

The definition of a Least General Generalization (abbreviated LGG, also called anti-unification) is the following:

Definition 14. *Given clauses C_1 and C_2, a clause C is the Least General (or Most Specific) Generalization of C_1 and C_2, denoted as $C = lgg(C_1, C_2)$, if and only if C θ-subsumes both C_1 and C_2 and there is no more specific clause C' that $\theta-$subsumes them.*

In order to represent *least-general generalizations*, we will use the lgg operator, defined as shown in Fig. 2.2 (also in [17]); some examples are given in Fig. 2.3.

Examples on LGG of terms:

1. $lgg([a,b,c],[a,c,d]) = [a,X,Y]$
2. $lgg(f(a,a),f(b,b)) = f(lgg(a,b),lgg(a,b)) = f(V,V)$ where V stands for $lgg(a,b)$.

Examples on LGG of literals:

1. $lgg(parent(ann,mary),parent(ann,tom)) = parent(ann,X)$.
2. $lgg(parent(ann,mary),\neg parent(ann,tom))$ is undefined.
3. $lgg(parent(ann,X),daughter(mary,ann))$ is undefined.

Example on LGG of clauses:

- if $C_1 = daughter(mary,ann) \leftarrow female(mary),parent(ann,mary)$ and $C_2 = daughter(eve,tom) \leftarrow female(eve),parent(tom,eve)$ then:

  $$lgg(C_1,C_2) = daughter(X,Y) \leftarrow female(X),parent(Y,X)$$

  where X stands for $lgg(mary,eve)$ and Y stands for $lgg(ann,tom)$

Figure 2.3: Examples of the lgg operator.

2.5 Working with Constraints

Constraints stand for limitations over the form of a solution to a problem. These problems generally require a combinatorial approach and also good heuristics and a good search strategy to be solved in reasonable time. The mathematical problem that applies constraints over a set of objects and requires in case of a solution, the state of these objects to meet the constraints, is called a *Constraint Satisfaction Problem.*

2.5.1 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) can be formally presented as a triplet $\langle X,D,C\rangle$ where X is a set of variables, D is a set of domains of values for the variables, and C is as set of constraints. Every variable X_i in the set of X has a non-empty domain D_i in D. A *state* of the problem, is a substitution of some variables in X with values from the corresponding domains in D (also called an "assignment"). Every constraint refers to a subset of X and defines their allowed combinations of values. An assignment is consistent, if it does not violate any of the constraints in C. It is also called a *legal assignment.* A *complete assignment* is an assignment where every variable X_i in X takes a value and a *solution* is a complete assignment that is also legal. Some CSP problems ask for a solution (a legal complete assignment) that additionaly optimizes (maximizes or minimizes) an objective function defined over the variables of the problem.

An example: Tone row

For an example, we will try to keep as close to the main subject of this thesis: music. A tone-row (or series) is a sequence of twelve tone names (pitch classes) of the chromatic scale in which each pitch class occurs exactly once. This problem could be modeled as a CSP using a triplet:

$$\langle X, D, C \rangle$$

where:

$$\begin{aligned} X &= \{X_1, \ldots X_{12}\} \\ D &= \{\{c, c\#, d, d\#, e, \ldots b\}, \ldots \{c, c\#, d, d\#, e, \ldots b\}\} \\ C &= \{\forall i, j, i \neq j : X_i \neq X_j\} \end{aligned}$$

X being the set of variables (X_i is the pitch class in of the note in position i), D the set of domains (pitches), and C the set of constraints. An assignment in this problem would be:

$$\{X_1 = c, X_2 = c, X_3 = c\}$$

Which is not full since not all variables participate. Also it is not legal since the constraints $X_1 \neq X_2, X_1 \neq X_3$ and $X_2 \neq X_3$ are violated. A legal assignment would be:

$$\{X_1 = c, X_2 = c\#, X_3 = d\}$$

since the above constraints are all met. A legal and full assignment would be:

$$\{X_1 = c, X_2 = c\#, X_3 = d, X_4 = d\#, \ldots, X_{12} = b\}$$

which, in case we do not have an objective function to optimize, would be our solution.

Musical CSPs Musical CSPs are a special class of CSPs that specialize their variables, domains, and constraint libraries in order to produce a musical composition. An example of a musical CSP is the above tone-row problem, part of the dodecaphonic tecnique introduced by Schoenberg et al. We have notes for variables and pitches in our domains. A software package that deals specifically with this class of problems, is STRASHEELA, where our work is based on.

Constraint Programming Constraint programming is a form of declarative programming where the programmer writes programs as a set of CSP problems, and then the reasoner of the programming language tries to find solutions to these problems.

2.5.2 Constraint Satisfaction in First-Order Logic

The First-Order Logic satisfiability problem can be translated to a constraint satisfaction problem. This is useful if we want to use Logic Programs in Constraint Programming. We define a constant object domain D in our CSP. Each FOL constant is assigned a member in D. Each function symbol is assigned a function $D \times D \times \ldots D \rightarrow D$. Each relation symbol is assigned a function

$D \times D \times \ldots D \rightarrow \{true, false\}$. The logic connectives behave as expected ($\wedge$,$\vee$), and the quantifiers ($\forall$,$\exists$) range over the domain D. For example, if we have the predicates $note/1$ and $interval/1$, the constants $c5, g5, perfectFifth$, and the function symbol $harmonicInterval$, we would transform our theory to a CSP as follows:

1. Our domain D has size 3 since we have 3 constants:

$$D = \{0, 1, 2\}$$

2. We assign a member of the domain D to each of the constants. We can represent this as a cell array with all the constants, where cells can take values from the domain $\{0, 1, 2\}$:

$c5$	$g5$	$perfectFifth$
$\{0, 1, 2\}$	$\{0, 1, 2\}$	$\{0, 1, 2\}$

3. To each of the unary relations $note/1$ and $interval/1$ we assign a relation in D. We can represent it as an array of three cells, where each cell corresponds to the respective member of D and it can take $true$ or $false$ as its values:

0	1	2
$\{true, false\}$	$\{true, false\}$	$\{true, false\}$

4. To the function $harmonicInterval$ we assign a function from D to D. We can represent it as an array of four cells where each cell can take one of the values in $\{0, 1, 2\}$:

0	1	2
$\{0, 1, 2\}$	$\{0, 1, 2\}$	$\{0, 1, 2\}$

5. Each cell shown above represents a *variable* in our CSP. The domains of the variables are $\{0, 1, 2\}$ and $\{true, false\}$. The constraints are our FOL formulas that constrain the values of the cells above. This way, a solution of the translated CSP corresponds directly to an original FOL *model*.

2.6 Strasheela

Strasheela is a CAAC system created by Anders Torsten. It is written in the Oz Programming Language and features a special case of CSP solver that deals specifically with musical CSPs. Its representation is very powerful and does not limit the user to a predetermined music theory. Its music representation follows an object-oriented approach, with musical elements being objects with methods for accessing and constraining their properties. Constraints are given using the Oz syntax. Domains are integers and variables can be any kind of object in its music representation. It also features ordering methods and heuristics for efficient search while solving musical CSPs. It can produce PDF[5], MIDI, and CSOUND files of the solutions it finds.

In our work, we use the Strasheela CAAC system to solve our CSPs and produce printouts and midi files for the generated pieces.

[5]Portable Document Format.

Chapter 3

Problem Statement

3.1 Human Intuition in Composition

Modern CAAC software typically relies on the assumption that the composer will have knowledge of both the tools being used, and of the theory behind composition. For example, if a student wants to do harmonization on a given melody line, he needs to learn the rules that define the harmonization procedure. He is taught of this procedure over the years of study. And that is only for the German Baroque style of chorale that he is being taught. If he wants to study, for example, Jazz Harmony, he would probably start anew, by learning again the rules that define jazz music.

This is very typical of students. But, let us now suppose that we have a very adept student with exceptional observation capability. If you give him a set of intervals, he would be able to tell immediately if they are consonant or dissonant, the type of dissonance, the scales they may appear into, the chords, etc.

If you now show him, completed counterpoint exercises, he would be able to recognise the intervals, observe the motions between them, and would be able to generalize what he sees into a set of rules. This means, we would not have to give him formally the rules, but just show him correctly completed counterpoint exercises, and he would be able to do counterpoint on his own from now on.

3.2 Machine "Intuition" in Composition

Research in Machine Learning and Logic Programming has given computers the two above key-abilities to some extend. Given a set of values, we can form detailed hypotheses about relations between them and we can use these relations to generalize and extract patterns that we could later use in order to generate similar sets of values. This would be extremely useful in music composition.

If we wanted to write a chorale, we would not want to concern ourselves about how to provide the exact harmonization rules (given that we know them), we would just give already harmonized examples, and the system would be aware of how we would want to do harmonization from now on.

T. Anders in his work *"Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System"*[18] describes Strasheela, a very powerful and expressive CAAC system which features opensource prototype implementation that makes it ideal for research as well as for production use.

Strasheela is programmed with a very textual programming language, acts on both micro and macro scale, and is able to produce MIDI files as well as sound files with the use of csound [19]. It is available for both realtime (using extensions [20]) and offline use and its programming language (Oz) allows for distributive usage.

Maybe the most useful property of Strasheela, aside from its easy programmability, is that it shows no limitations on the musical theory that the user can utilize. One can act on almost every music theory known, tonal, atonal, or microtonal or even define new theories himself.

Such expressive power though stems from a very expressive representation, which proves to be somewhat difficult for the novice user to utilize. It also requires a somewhat functional way of thinking (In a way it is like writing in a functional programming language). For example, if we have a single voice and

we want to constraint all the intervals to a fixed value, we would write ([18], p. 115):

$$\begin{aligned} \textbf{let} \quad & myPitches \stackrel{def}{=} map(getNotes(myVoice), getPitch) \\ \textbf{in} \quad & \bigwedge map2Neighbours(myPitches, limitInterval_{pitches}) \end{aligned}$$

which is somewhat confusing for the user unfamiliar with the functional paradigm. We would like to be able to provide the rules in a more human-friendly way. In fact, from the above, as well from Ander's proposed Future Work ([18], p. 206):

> "It would be interesting to integrate learning techniques into STRASHEELA in such a way that the user would mix manually-defined compositional knowledge with explicitly represented knowledge deduced from existing pieces. If the knowledge learnt is represented in the form of rules, then this knowledge can be freely mixed with hand-written rules."

Later, in the same section, he informs us of the work conducted on an inductive logic programming algorithm, named *PAL* [16], which is able to induce patterns, given sets of examples of ground facts, as well as a prior knowledge base. In their work, they also describe a way to induce patterns in order to do counterpoint.

The above hints have motivated the work described in this thesis, which turns out to be a computer-aided composition method that relies on the power of Inductive Logic Programming. It allows us, to utilize the expressive power of a CAAC system, like STRASHEELA, with the convenience of a *by-example* pattern design methodology that results in musical compositions. Our goal is to make use of the ILP work done by *Morales et al.* (since they already describe how their algorithm can be used in music) to extract FOL patterns from existing music scores, transform the original FOL patterns to constraints, formulate the problem so that it is compatible with the functional representation used in STRASHEELA, and finally to use STRASHEELA itself for composing music by solving the resulting CSP problem.

For example the rule given above, instead of having to think in a functional way, we could just provide an example of a series of notes (3 consecutive notes would be sufficient) that have the desired constant melodic interval.

3.3 Related Work

Various research efforts have been devoted to learning musical rules of some kind. Most of the works train some kind of statistical model and most attempts have been done mainly for Harmony and mostly by using real sound data. Little has been done for musical textures (melody, harmony and rhythm) in general.

Works based on a symbolic approach have been utilizing mostly conventional algorithms, expert systems, and generative grammars. One of the notable cases is "CHORAL: An Expert System for Harmonizing Chorales in the style of J.S. Bach" [21] by Kemal Ebcioglu. The system does harmonization based on about 350 rules using first-order logic. Another notable case by Hilde et al. is "HARMONET: A Neural Net for Harmonizing Chorales in the Style of J.S Bach"

[22], where they propose a system that combines conventional agorithms with neural networks in order to learn harmonization in the style of J.S. Bach that both respects the harmony rules and also is aesthetically pleasing. However, both these systems have a pre-defined structural approach and do not learn new harmonization rules but restrict themselves to the style of J.S. Bach.

A really interesting work is that of David Cope's "Experiments in Music Intelligence" that has led to astonishing results. EMI literally cuts the works of famous composers into pieces while respecting each composer's unique "signature" and reassembles them in such a way that is interesting, aesthetically pleasing, and conforms to the composer's original style. However, EMI itself is not truly creative, in that it is not able to produce new music from scratch.

Morales et al. in their paper "Learning Musical Rules" [23] where they describe the learning of musical rules with the use of PAL (where we have based our work on), envision coupling PAL with ESCAMOL, an algorithmic tool for composition which incorporates generative grammars used in their interactive music composition system SICIB [24].

Chapter 4

Our Approach

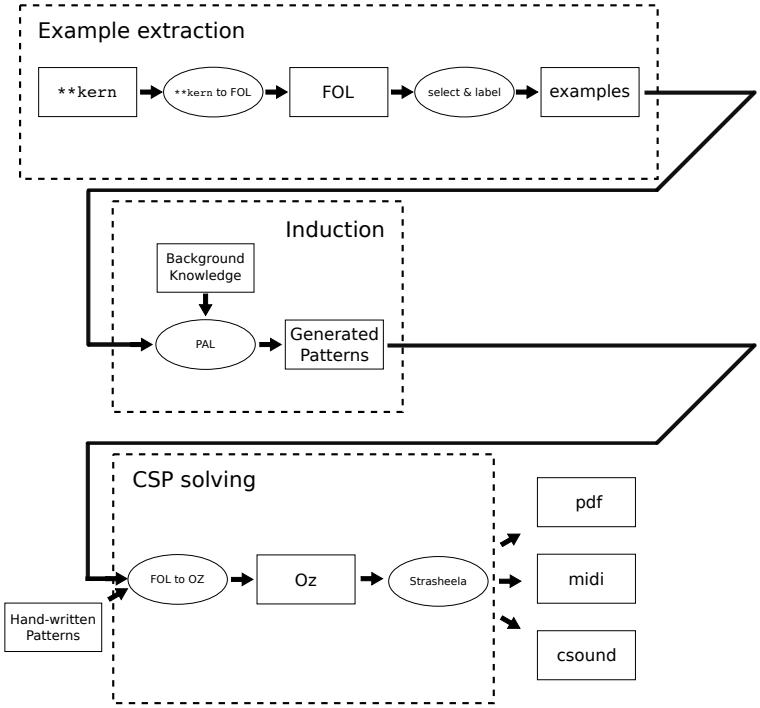


Figure 4.1: The graphical diagram of the proposed process.

In this part we will give a detailed view of our approach to computer-aided composition. We will show how one can deduce compositional rules from simple examples or complete musical pieces. We will then use these rules, mixed with our own if desired, in order to compose new musical pieces. A graphical diagram of the proposed process is given in Fig. 4.1.

4.1 Representations and Transformations

We first present the three types of representation that we use. The first is the `**kern` representation, a purely textual form used to provide examples to our system. The second is the FOL representation used to provide these examples as well as background knowledge to our ILP algorithm. Finally, the third is the Oz representation used to provide compositional constraints to Strasheela. We will also show how we translate from one form to another throughout the process.

4.1.1 The `**kern` Representation

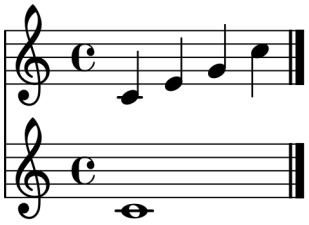
We use the `**kern` representation mainly because it is a direct, human-readable form, easy for the computer to parse (by using `perl` or `awk` for example), and it is also able to represent performance details for both the human and the computer. It is also the representation language of the Humdrum toolkit, a

software package for music research, which provides a great amount of software for music cognition, from harmonic analysis to generating pitch histograms [25]. Furthermore, `**kern` is used by the online database KernScores [26], which is a huge database of songs in `**kern` format with results from various analyses.

The `**kern` representation uses a purely ASCII data format. A `**kern` file contains several *spines* (columns, separated by a *tab* character) and *rows*. The data found at a specific *spine* and *row* is called a *record*. Each spine has a *label* (beginning with `**`) which describes the kind of data stored in its rows and a sublabel (beginning with `*`) which specifies the data further. Comments are also allowed, globally (ignored throughout the whole file when parsing) or locally in a spine (ignored only for that spine).

Usually, the row number represents position in the score (i.e. time in beats in sheet music) and the spine number represents the voice (i.e. staff in sheet music). Records typically represent *notes* or *pauses*, although they are not limited to representing only those. A record (depending on the label of its spine) can represent anything from chord analyses to lyrics, to arbitary data.

For example, the one-measure score below:



translates directly to:

```
**kern **kern
1c     4c
.      4e
.      4g
.      4cc
==     ==
```

where the dot is called a *null* token, and implies that the record on the above row still holds. The notes in `**kern` format begin with a number, which represents the inverse of their duration, and a letter from a to g representing the respective note. The letter c represents the middle c. If we want to represent the c an octave above, we write two c's. Three octaves above, three c's, etc. If we want to represent c an octave below, we enter a capital C. Two octaves below, two capital C's, etc. Accidentals are denoted with '#' (sharp) and '$-$' (flat) and alter the pitch by $+1$ and -1 repsectively. We can also have double (or more) sharps '##' or flats '$--$' which alter the pitch by 2 (or more). A pause is represented by the letter 'r'. The number at the beginning of the record represents the denominator of its duration (for example, for a duration of $\frac{1}{4}$ - a quarter note - the number at the beginning would be 4). A single = represents a single bar whereas a double == a double bar. When no additional info is given on the meter or staff, a meter of $\frac{4}{4}$ and a clef of G are implied.

Our examples are extracted from music pieces stored in the `**kern` representation. This allows for a relatively easy translation of the records to first-order

predicates. The HUMDRUM toolkit makes this parsing easy. The procedure of translation, begins by numbering the various records relative to their position in the score. For the above example, a new spine would be added indicating the positions:

```
**kern **kern **pos
1c     4c     1
.      4e     2
.      4g     3
.      4cc    4
==     ==     4
```

Note that the `**pos` value does not increase at the row of the double barline '=='. This is because the double barline is not an active object, as it does not participate in the presence (or absence) of sound, but rather serves as a way of separating measures.

After numbering the rows, it is easy to produce the necessary predicates to transform the representation to FOL.

4.1.2 FOL Predicate Representation

Since our ILP algorithm is implemented in PROLOG, the most natural way is to provide its input as first-order predicates in the style of PROLOG.

The most useful predicate we use is that of $note/4$[1]. This usually has the form $note(Voice, Position, Pitch, Duration)$ where $Voice$ is a variable that represents the voice the note belongs to, $Position$ represents the position in that voice, relative to the beginning of the score, $Pitch$ represents its pitch, and $Duration$ is the denominator of the duration, as in the `**kern` representation. All of the $Voice, Position, Pitch, Duration$ are integer-valued.

Our examples currently consist solely of *note* predicates. For this reason, the $note/4$ predicate is called an *input* predicate.

Translation from `**kern`

The translation from `**kern` to first-order predicates, is straightforward and has a one-to-one correspondence. In $note(Voice, Position, Pitch, Duration)$, $Voice$ becomes the number of spine (out of the spines that have a `**kern` label). $Position$ becomes the value of the corresponding record in the `**pos` spine. $Pitch$ is translated as follows: c, d, e, f, g, a, b are translated to $0, 2, 4, 5, 7, 9, 11$ respectively and C, D, E, F, G, A, B to $-12, -10, -8, -7, -5, -3, -1$ respectively. For any additional lower-case letter, we add 12 to this number. For any additional capital letter, we subtract 12 from this number. Finally, we add 60 (which is the MIDI number for middle c) to this number and use it as our $Pitch$. $Duration$ is translated as follows: When we parse the `**kern` file, we identify a minimum duration $\frac{1}{M}$ (for example eighth-notes) over all notes, which serves as our basic time unit. $Duration$ becomes the amount of base units that are required to complete the original note duration (i.e. if $M = 8$ and the record is written as $4c$, then $Duration$ becomes 2 since $2 \times \frac{1}{8} = \frac{1}{4}$.

[1] *note* is the name of the predicate and 4 is its arity.

The example in the previous subsection (4.1.1) using PROLOG unit clauses (supposed we have parsed it with a minimum duration of $\frac{1}{4}$) becomes:

$$note(2, 1, 60, 4).$$
$$note(1, 1, 60, 1).$$
$$note(1, 2, 64, 1).$$
$$note(1, 3, 67, 1).$$
$$note(1, 4, 72, 1).$$

4.1.3 Representation of Examples

After we have extracted the $note/4$ predicates, we must choose those subsets that will be contained in our examples. For example, if we are doing chord analysis, it may be best to extract examples from *slices* at every position and we may want to omit duration since it is irrelevant. From our little music sample, we would generate 4 examples represented as sets:

$$E_1 = \{note(2, 1, 60), note(1, 1, 60)\}$$
$$E_2 = \{note(2, 1, 60), note(1, 1, 64)\}$$
$$E_3 = \{note(2, 1, 60), note(1, 1, 67)\}$$
$$E_4 = \{note(2, 1, 60), note(1, 1, 72)\}$$

Here, the predicate $note/3$, represents just the pitch of the note, without duration.

After the example generation, we need to *characterize* them. That means, we need to provide each of the examples with a label. The examples will then be grouped by labels (examples having the same label will be in the same group) so that a rule is generated for each unique label.

4.1.4 Representation of Background Knowledge

In order to do something useful with the input predicates, we need a way to describe the relations between the components used in the rules. These relations are represented either as first-order predicates (like the $next/3$ and $over/3$ predicates given below) or as Horn clauses of the form:

$$H_1 \leftarrow B_1, B_2, \ldots, B_N.$$

where B_i can be either instances of predicates or unbound literals (literals with an unbound variable in their arguments) describing relations. And they're can be either bound or unbound.

For example, in some of our test cases, we use the $oblique\_motion/4$ predicate, that describes an oblique motion[2] occuring between two voices $V1, V2$

[2]We have an oblique motion, when the note in one of the voices stays at the same pitch, and the pitches of the notes in the other voices change.

and two positions $P1, P2$ and is defined as such:

$$\begin{aligned} oblique\_motion(V_1, V_2, P_1, P_2) \quad \leftarrow \quad & \\ & note(V_1, P_1, Pitch_{1,1}), \\ & note(V_1, P_2, Pitch_{1,2}), \\ & note(V_2, P_1, Pitch_{2,1}), \\ & note(V_2, P_2, Pitch_{2,2}), \\ & next(V_1, P_1, P_2), \\ & next(V_2, P_1, P_2), \\ & over(V_1, V_2, P_1), \\ & over(V_1, V_2, P_2), \\ & (Pitch_{1,1} = Pitch_{1,2}, \\ & Pitch_{2,1} \neq Pitch_{2,2}; \\ & Pitch_{1,1} \neq Pitch_{1,2}, \\ & Pitch_{2,1} = Pitch_{2,2}). \end{aligned}$$

$next(V_1, P_1, P_2)$ yields *true*, when on voice V_1, position P2 is the next position after P1 or, in other words, the note at position P_1 is followed by a note to P_2 (produced by our example generator, see below subsection) and $over(V_1, V_2, P_1)$ yields *true*, if the note on voice V_2 is over the one on V_1 at position P_1.

The above is an example pattern. There is a large number of patterns we have defined in the background database and the user is free to use them or discard them completely and provide the database with a complete music theory of his own.

4.1.5 Representation of Induced Rules

The rules induced, are descibed as Horn Clauses of the form:

$$\begin{aligned} H \quad \leftarrow \quad & D_1, D_2, \ldots, D_N, \\ & BK_1, BK_2, \ldots, BK_M, \\ & PN_1, PN_2, \ldots, PN_K. \end{aligned}$$

where,

- H is the *head* of the rule.
- $D_i, i = 1, \ldots, N$ are *input predicates*. These are predicates extracted from the score (but not derived from the background knowledge) and are the main components in the induction of rules. The main input predicates are $note(Voice, Position, Pitch, Duration)$ and $note(Voice, Position, Pitch)$.
- $BK_i, i = 1, \ldots, M$ are ground predicates derived from the input predicates and the background knowledge. They usually describe *properties* of our input predicates. An example would be $cmajor(Voice, Position)$ which states that the note in voice $Voice$ and position $Position$ belongs to the cmajor scale.

- $PN_i, i = 1, \ldots, K$ are ground predicates of the other rules induced by our ILP algorithm. They usually describe *actions*. An example would be a predicate $oblique\_motion(Voice_1, Voice_2, Position_1, Position_2)$ which holds true whenever there is an oblique botion between $Voice_1$, $Voice_2$, $Position_1$ and $Position_2$.

4.1.6 The Oz Representation

PAL outputs Prolog horn clauses. In order to add these as constraints understandable by our CSP solver, we must convert them to Oz syntax. Prolog predicates and horn clauses can be translated to Oz *procedures* of the form:

```
proc {Name Argument1 Argument2 ... ArgumentN}
 Statement1 Statement2 ... StatementN
end
```

where `Name` is the label of the predicate, but with an uppercase first letter (for example, *note* becomes `Note`) and `Argument1 ... ArgumentN` are the arguments of the predicate, separated by a space character. When applied to logic (or constrain) programming, the language tries to execute every `Statement`; if one of them fails, then the search fails this branch.

We can notice that this Oz representation has an almost one-to-one correspondence to a Prolog horn clause, apart from the different syntax. One difference is that, everything that is not a variable, must be put inside the body (within `proc {...}` and `end`). This means that all the possible substitutions of the arguments must be calculated *inside* the body.

For now, we will focus on the translation of Horn clauses where all the arguments are variables (which is the case of our induced patterns). For the predicate $note(Voice, Position, Pitch, Duration)$, the head of the corresponding Oz procedure becomes {`Note Voice Position Pitch Duration`}.

The $note/4$ predicate implies that at voice $Voice$ and position $Position$ there is a note with a pitch of $Pitch$ and a duration of $Duration$. When we transform our problem to a CSP, $note(Voice, Position, Pitch, Duration)$ means that we want to constraint the pitch to $Pitch$ and the duration to $Duration$ of the variable at position $(Voice, Position)$ in the score. So, the `Note` procedure becomes[3]:

```
proc {Note Voice Position Pitch Duration}
 {{Nth {Nth Score Position} Voice} getPitch($)} =: Pitch
 {{Nth {Nth Score Position} Voice} getDuration($)} =: Duration
end
```

In the case of clauses, the translation process is much more straightforward. The head is translated like above, while the body becomes the translated heads of the literals, separated by a space character in case of a conjunction or by a `dis` construct in case of a disjunction. For example, the clause:

[3]We refer the reader to the "Strasheela Reference Documentation" [27] for `getPitch` and `getDuration` and to the "Mozart Documentation" website [28] for an introduction to Oz syntax.

$$
\begin{aligned}
oblique\_motion(V_1, V_2, P_1, P_2) \quad \leftarrow \\
& note(V_1, P_1, Pitch_{1,1}), \\
& note(V_1, P_2, Pitch_{1,2}), \\
& note(V_2, P_1, Pitch_{2,1}), \\
& note(V_2, P_2, Pitch_{2,2}), \\
& next(V_1, P_1, P_2), \\
& next(V_2, P_1, P_2), \\
& over(V_1, V_2, P_1), \\
& over(V_1, V_2, P_2), \\
& (Pitch_{1,1} = Pitch_{1,2}, \\
& Pitch_{2,1} \neq Pitch_{2,2}; \\
& Pitch_{1,1} \neq Pitch_{1,2}, \\
& Pitch_{2,1} = Pitch_{2,2}).
\end{aligned}
$$

becomes in Oz:

```
proc {Oblique_Motion V1 V2 P1 P2}
    {Note V1 P1 Pitch11}
    {Note V1 P2 Pitch12}
    {Note V2 P1 Pitch21}
    {Note V1 P2 Pitch22}
    {Next V1 P1 P2}
    {Next V2 P1 P2}
    {Over V1 V2 P1}
    {Over V1 V2 P2}
    dis
      Pitch11 =: Pitch12
      Pitch21 \=: Pitch22
    []
      Pitch11 \=: Pitch12
      Pitch21 =: Pitch22
    end
end
```

where the `Next` and `Over` procedures refer to the translated $next/3$ and $over/3$ predicates. Since the *predicates* have different meaning in logic programming than in their respective procedures in Oz, in our implementation they have to be declared manually in both Oz and Prolog. On the other hand, since generated clauses have just conjunctions of literals in their body, the translation of learned rules is carried out automatically.

4.2 Rule Induction

After we have represented a musical score in FOL form, we need to select the appropriate sets of symbols in order to induce the patterns we will later use for

the generation of musical pieces. To make this clear, assume that we would like to generalize the first species Fuxian counterpoint rule

> "from one perfect consonance to another perfect consonance one must proceed in contrary or oblique motion"

we should feed our algorithm with real examples of the correct usage of this rule in a musical score.

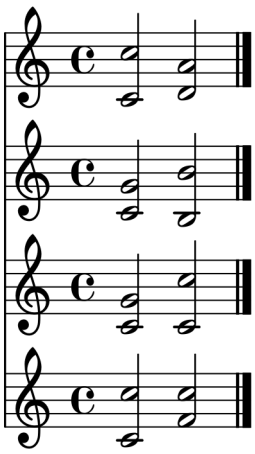


Figure 4.2: First species Fuxian counterpoint rule examples.

4.2.1 Example Sets

Example sets are sets that contain all the ground unit clauses that take part in an example. They are *instantiations* of the concept we want to learn. For example, in the third example of the counterpoint rule shown above (Fig. 4.2), if we have the harmonic transition from c-g to c-cc, where we have $note(1, 1, 60, \_)$ for c, $note(2, 1, 67, \_)$ for the g above, $note(1, 2, 60, \_)$ for the c in the second position, and $note(2, 2, 72, \_)$ for the c an octave above, the set describing the example would be[4] :

$$E_1 = \{note(1, 1, 60, \_), note(2, 1, 67, \_), note(1, 2, 60, \_), note(2, 2, 72, \_)\}$$

A second example for this rule could, e.g. be c-cc to f-cc (Fig. 4.2, staff 4):

$$E_2 = \{note(1, 2, 60, \_), note(2, 2, 72, \_), note(1, 3, 65, \_), note(2, 3, 72, \_)\}$$

As we can see, we only use the *note/4* predicates (referred to as *input predicates*).

[4]We usually characterize the example sets with a label, in order to be able to generate patterns, specific to same-label examples. We have ommited labels here for simplicity.

4.2.2 The PAL Algorithm

PAL [16] is an ILP algorithm that induces patterns from examples (given as sets of ground atoms) and a given knowledge base (facts and other patterns). Contrary to other ILP systems (such as GOLEM), it does not need any sort of prior knowledge about the learned pattern (for example, GOLEM requires declaration of the type of arguments and the arity of the head of a learned rule in advance) and can induce new patterns from scratch.

Patterns generated by PAL are Horn Clauses. In general, PAL can utilize both positive and negative examples, but because of the nature of our application[5], we utilize only positive examples.

It must be noted that in order to supress the number of literals generated in the body of a rule (and avoid a useless literal explosion) we apply the following two constraints:

1. Only *Variables* that appear in the head of the resulting pattern, can appear in the body of the pattern.
2. Every literal in a pattern is labeled and generalization takes place only between literals with the same labels.

The first constraint is quite straightforward. In our case, only variables that represent voice, position or pitch can be part of the arguments of our learned patterns. The second is also found in the original PAL article [16]. Suppose we have the following ground clauses:

$$\begin{aligned} chord(2) &\leftarrow note(1,2,60), note(2,2,64), note(2,3,67). \\ chord(3) &\leftarrow note(1,3,67), note(2,3,71), note(2,3,80). \end{aligned}$$

Least general generalizations, as defined in Chapter 2 (Background) section would produce every possible *lgg* from the combinations of the literals of the two clauses. There are 10 possible *lgg*s (1 from the head and 9 from the combinations of the *note* predicates) for these two clauses. If we had 4 note predicates to each body, we would produce $1 + 16$ more literals. For larger number of literals, the number of possible *lgg*s increases dramatically (quadratic growth).

In order to restrict the number of literals, we use the labelling technique used in the original PAL. For every predicate in an example, we label its literals, the same way we labeled examples. Labeling the literals in the above clauses, they become:

$$\begin{aligned} chord(2) &\leftarrow note(1_1,2_2,60_3), note(2_4,2_5,64_6), note(2_7,3_8,67_9). \\ chord(3) &\leftarrow note(1_1,3_2,67_3), note(2_4,3_5,71_6), note(2_7,3_8,80_9). \end{aligned}$$

Now we introduce CLGG, which is nothing more than an LGG for the literals that have the same labels in their arguments. The possible *LGG*s would now be just 6, instead of 9:

$$\begin{aligned} &lgg(note(1_1,2_2,60_3), note(1_1,3_2,67_3)) \\ &lgg(note(2_4,2_5,64_6), note(2_4,3_5,71_6)) \\ &lgg(note(2_7,3_8,67_9), note(2_7,3_8,80_9)) \end{aligned}$$

[5] A composer, typically thinks of what is correct, not of what is wrong.

thus dramatically reducing the number of literals generated.

In order to understand how the original PAL algorithm works, we will split the algorithm in four steps:

1. **Construct the head** of the pattern to be learned. This is being done for every positive example by constructing an arbitary predicate which takes as arguments all the arguments of all the ground clauses in the examples. For the example set:

$$E_1 = \{note(1,1,48), note(1,2,60)\}$$

we will construct a new temporary predicate name pn and with all the arguments of the *note* predicates as its own arguments, thus constructing the head of our rule:

$$pn(\underbrace{1,1,48}_{\text{Arguments of } note(1,1,48)}, \underbrace{1,2,60}_{\text{Arguments of } note(1,2,60)}).$$

2. **Construct the body**. This is being done for every positive example. Using rule production, find the literals produced from our knowledge base and our examples, append them to a conjunction of the ground clauses in the example, and use them as a body for the head generated in the first step, to construct a new clause.

If we have the following knowledge base:

$$\begin{aligned} distance(X,Y,Out) &\leftarrow X > Y, Out = X - Y. \\ distance(X,Y,Out) &\leftarrow X < Y, Out = Y - X. \\ distance(X,X,0). & \\ melodic\_interval(V_1,P_1,P_2,Out) &\leftarrow note(V_1,P_1,Pitch_1), \\ & \quad note(V_2,P_2,Pitch_2), \\ & \quad distance(Pitch_1,Pitch_2,Out). \end{aligned}$$

using rule production from the facts in the example set E_1 above we also produce the literal $melodic\_interval(1,1,2,12)$. We append it to a conjunction of the (labelled but we omit the labels here in favor of readability) *note* predicates found in the example, and use the resulting conjunction as the body of the head given above.

The clause constructed is:

$$\begin{aligned} pn(1,1,48,1,2,60) &\leftarrow note(1,1,48), \\ & \quad note(1,2,60), \\ & \quad melodic\_interval(1,1,2,12). \end{aligned}$$

3. **Generalize**. Suppose, we also had the positive example:

$$E_2 = \{note(1,2,60), note(1,3,72)\}$$

The constructed clauses would be:

$$\begin{aligned} pn(1,1,48,1,2,60) \leftarrow\ & note(1,1,48), \\ & note(1,2,60), \\ & melodic\_interval(1,1,2,12). \\ pn(1,2,60,1,3,72) \leftarrow\ & note(1,2,60), \\ & note(1,3,72), \\ & melodic\_interval(1,1,2,12). \end{aligned}$$

and their CLGG would be:

$$\begin{aligned} pn(1,X,Y,1,Z,W) \leftarrow\ & note(1,X,Y), \\ & note(1,Z,W), \\ & melodic\_interval(1,X,Z,12). \end{aligned}$$

which is a learned pattern that describes both E_1 and E_2 and X,Y,Z,W are variables representing the Least General Generalizations:

$$\begin{aligned} X &= lgg(1,2) \\ Y &= lgg(48,60) \\ Z &= lgg(2,3) \\ W &= lgg(60,72) \end{aligned}$$

If this pattern covers any examples in the set of negative examples, reject the generalization, save the example that led to the construction of pn to a list DL and continue with the other positive examples.

4. **Refine**. In case we end up with examples stored in the list DL mentioned above, run the algorithm again, with the examples in DL as the positive examples. This will give a refined version of the pattern we rejected, that covers these positive examples only, and none of the negative ones. This way, the pattern will have many bodies for the same head.

The generic PAL alorithm is given in Fig. 4.3 and the version we use in Fig. 4.4. In our version, we don't use negative examples, so the step of refinement, is redundant. Note that we also remove unnecessary arguments from the head of NC. PAL generated patterns, have some interesting properties:

- Only *Variables* of the same type as the arguments in our input (for example the *note/4* predicates happen to appear in the head). This way we can control what type of variables we need to constraint. This will become clear in Subsection 4.3.
- Only the *Variables* that appear as arguments in the head can appear in literals in the body. This keeps the patterns compact, as well as removes redundant constraints. If we believe that some type of constraints between other types of variables must exist, we can provide them relative to the input arguments [6].

[6] For example, if we want to constraint the interval I between two notes $note(1,X,Pitch_1,1)$ and $note(1,Y,Pitch_2,1)$, instead of constraining it as $interval(1,X,Y,H), H < 5$ we can use $constraint\_interval\_lessthan(1,X,Y,5)$ which only contains variables X and Y both of which appear in the *input predicates*.

4.2.3 PAL Induction Examples

Below, we will give two examples. The first example which shows the strength of PAL in the learning of new concepts and the second shows a detailed execution of the algorithm.

Example of new concept learning

PAL's most useful property is that it can produce new patterns almost from scratch, without any prior information on the produced pattern. This is useful, if we want to learn new concepts, previously unkown to us.

For example, suppose we only have the knowledge base:

$$\begin{aligned} distance(X, Y, Out) &\leftarrow X > Y, Out = X - Y. \\ distance(X, Y, Out) &\leftarrow X < Y, Out = Y - X. \\ distance(X, X, 0). & \\ melodic\_interval(V_1, P_1, P_2, Out) &\leftarrow note(V_1, P_1, Pitch_1), \\ & \quad note(V_2, P_2, Pitch_2), \\ & \quad distance(Pitch_1, Pitch_2, Out). \end{aligned}$$

And we want to learn a concept based on the intervals of these three notes:

$$\begin{aligned} &note(1, 1, 48). \\ &note(1, 2, 60). \\ &note(1, 3, 72). \end{aligned}$$

So we split these notes into two examples:

$$\begin{aligned} E_1 &= \{note(1, 1, 48), note(1, 2, 60)\} \\ E_2 &= \{note(1, 2, 60), note(1, 3, 72)\} \end{aligned}$$

If we provide these to PAL, it will induce a new pattern *pn*:

$$pn(V, P_1, P_2) \leftarrow note(V, P_1, Pitch_1), note(V, P_2, Pitch_2), melodic\_interval(V, P_1, P_2, 12).$$

where we can freely rename *pn* to *octave* and realize that the algorithm learned the concept of the octave.

Example of execution

Suppose we have extracted the examples E_1 and E_2 shown above, that together consist our set of positive examples $\mathcal{E}^+$. We also have in our knowledge base

KB the pattern *oblique_motion*/4 given as[7]:

$$\begin{aligned} oblique\_motion(V_1, V_2, P_1, P_2) \leftarrow\ & note(V_1, P_1, Pitch_{1,1}, \_), \\ & note(V_1, P_2, Pitch_{1,2}, \_), \\ & note(V_2, P_1, Pitch_{2,1}, \_), \\ & note(V_2, P_2, Pitch_{2,2}, \_), \\ & next(V_1, P_1, P_2), \\ & next(V_2, P_1, P_2), \\ & (Pitch_{1,1} = Pitch_{1,2}, Pitch_{2,1} \neq Pitch_{2,2}; \\ & Pitch_{2,1} = Pitch_{2,2}, Pitch_{1,1} \neq Pitch_{1,2}) \end{aligned}$$

That means, *oblique_motion*/4 is *true*, whenever the four predicates *note*/4 and the two predicates *next*/3 are true, two of the Pitches of the consecutive notes are the same and the other two are different.

Below we give all the steps of running PAL with input:

$$\mathcal{E}^+ = \{E_3, E_4\}$$

where:

$$E_3 = \{note(1,1,60,1), note(2,1,67,1), note(1,2,60,1), note(2,2,72,1)\}$$

$$E_4 = \{note(1,2,60,1), note(2,2,72,1), note(1,3,65,1), note(2,3,72,1)\}$$

1. **Select** E_3 and **label** its literals:

$$\{note(1_1, 1_2, 60_3, 1_4), note(2_5, 1_6, 67_7, 1_8), note(1_9, 2_{10}, 60_{11}, 1_{12}), note(2_{13}, 2_{14}, 72_{15}, 1_{16})\}$$

2. **Construct** NC.

   - C_1 is constructed by all the arguments in the *note*/4 predicates and a predicate name *pn*.

     $$C_1 \equiv pn(1_1, 1_2, 60_3, 1_4, 2_5, 1_6, 67_7, 1_8, 1_9, 2_{10}, 60_{11}, 1_{12}, 2_{13}, 2_{14}, 72_{15}, 1_{16})$$

   - Using the only pattern *oblique_motion* and the atoms in the example, we get the pattern instance:

     $$A_{1,1} \equiv oblique\_motion(1_1, 2_5, 1_2, 1_6)$$

   - Finally:

     $$\begin{aligned} NC \equiv pn(1_1, 1_2, 60_3, 1_4, 2_5, 1_6, 67_7, 1_8, 1_9, & \\ 2_{10}, 60_{11}, 1_{12}, 2_{13}, 2_{14}, 72_{15}, 1_{16}) \leftarrow\ & note(1_1, 1_2, 60_3, 1_4) \\ & note(2_5, 1_6, 67_7, 1_8) \\ & note(1_9, 2_{10}, 60_{11}, 1_{12}) \\ & note(2_{13}, 2_{14}, 72_{15}, 1_{16}). \end{aligned}$$

[7] A clause with a disjunction ';' is used like two Horn clauses with the same head and one operand of the disjunction each time.

3. **Enter** Inner loop. There's still one other example.

4. **Select** E_4

5. **Construct** C'_j

   (a) Similar to NC, C'_j is constructed as below, with the same predicate name PN:

$$\begin{aligned} C'_j \equiv pn(1_1, 2_2, 60_3, 1_4, 2_5, 2_6, 72_7, 1_8, 1_9, \\ 3_{10}, 65_{11}, 1_{12}, 2_{13}, 3_{14}, 72_{15}, 1_{16}) \quad \leftarrow \quad & note(1_1, 2_2, 60_3, 1_4) \\ & note(2_5, 2_6, 72_7, 1_8) \\ & note(1_9, 3_{10}, 65_{11}, 1_{12}) \\ & note(2_{13}, 3_{14}, 72_{15}, 1_{16}). \end{aligned}$$

   (b) **Generalize** NC and C'_j and replace NC with the generalization.

   - The head H of the generalized clause is the generalization of the heads of NC and C'_j.
   - Generalizations of literals in the body are being done only between the literals with the same labels[8]. For example $note(1_1, 1_2, 60_3, 1_4)$ with $note(1_1, 2_2, 60_3, 1_4)$, $note(2_5, 1_6, 67_7, 1_8)$ with $note(2_5, 2_6, 72_7, 1_8)$ etc.

   (c) The execution of loop will yield the **clause**:

$$\begin{aligned} pn(1_1, A, 60_3, 1_4, 2_5, A, B, 1_8, 1_9, \\ C, D, 1_{12}, 2_{13}, C, 72_{15}, 1_{16}) \quad \leftarrow \quad & \\ & note(1_1, A, 60_3, 1_4), \\ & note(2_5, A, B, 1_8), \\ & note(1_9, C, D, 1_{12}), \\ & note(2_{13}, C, 72_{15}, 1_{16}), \\ & oblique\_motion(1_1, 2_5, B, C). \end{aligned}$$

   (d) **Remove** constants and duplicates of the generalization variables from the head and **output** the resulting pattern.

$$\begin{aligned} pn(A, C, D) \quad \leftarrow \quad & note(1_1, A, 60_3, 1_4), \\ & note(2_5, A, B, 1_8), \\ & note(1_9, C, D, 1_{12}), \\ & note(2_{13}, C, 72_{15}, 1_{16}), \\ & oblique\_motion(1_1, 2_5, A, C). \end{aligned}$$

   In the pattern generated above, there is no useful direct meaning, however it states the following:

[8] Note that a generalization Variable represents two unlabeled literals. So for example C will represent $lgg(2_{10}, 3_{10})$ and $lgg(2_{14}, 3_{14})$ alike.

> "There is an oblique movement at the first two voices, between positions A and C where the first note of the first voice must be a middle c (midi number 60) and the second note of the second voice must be a c5 (midi number 72)"

This is a bit too restrictive in order to generate a piece. In order to produce more general and useful patterns PAL needs more and diverse examples.

4.2.4 PAL Limitations

PAL's most useful property is that it can produce new patterns almost from scratch, without requiring any prior information on the produced pattern. This is useful if we want to learn new, previously unknown concepts, which is essential for our work. However, PAL comes with some limitations. The expressivity of the patterns allowed is somewhat limited. For example, it is impossible to learn recursive patterns. Another issue is the number of examples. Too few examples can lead to overspecification (as in our agorithm execution example). On the other hand, given many examples a single noise-altered (for example, piano performance mistakes) or irrelevant example may lead to overgeneralization (the pattern would be generalized further in order to cover the mistake). Finally, PAL produced patterns as Horn clauses, where the body members are atomic sentences; disjunction cannot be used in any of the conjuncts. This limits the compactness of generated patterns.

4.3 Rule Application and Music Generation

After the induction of rules, we need a way to use them for music generation. In our approach, we compose a music piece by composing its score. Our score is a *grid* of variables corresponding to notes with domain $< Pitch, Duration >$, where $Pitch$ is an integer giving the MIDI number of the note symbol and $Duration$ is an integer giving the duration of a note, relative to the duration of a *beat*. The score can be initially fully unconstrained (create the score from scratch) or partially constrained (for example, if we want to complete the melody given a *cantus firmus*).

4.3.1 CSP Formulation

A CSP is as a triplet $\langle X, D, C \rangle$ where X is the set of variables, D a domain of values, and C the set of constraints. In our problem, we define the set of variables $X = \{X_{ij} : i = 1, \ldots, V, j = 1, \ldots, P\}$ where X_{ij} stands for the note at voice i and position j in the score for V voices and P positions with domain $D = Pitch \times Duration$, where $Pitch$ is the domain of pitches (MIDI numbers) and $Duration$ is the domain of durations (integers as well). Our (initial) CSP is defined as:

$$\langle \{X_{11}, X_{21}, \ldots, X_{N1}, X_{12}, \ldots X_{NM}\}, Pitch \times Duration, C \rangle$$

where C is initially empty, but constraints are added by applying the learned patterns to variables in X. An example of a score where consecutive notes, as

well as simultaneous notes depend on one another, is given in Fig 4.5. Nodes represent variables, whereas edges represent (binary) constraints.

Recall that the only variables appearing in our induced rules are *Voice*, *Position*, *Pitch*, and *Duration*. Given our CSP formulation, only *Pitch* and *Duration* are subject to constraints. The other two $Voice$ and $Position$ are used to index the variables X_{ij} and just describe the location of each note in the score. If we want to apply an induced rule as a constraint to the entire score we have to introduce constraints for all possible X_{ij}. Arguments in the Head with domains $Voice$ and $Position$ serve this purpose well. Suppose we have the pattern:

$$\begin{aligned} pn(V_1, P_1, Pitch_1, Duration_1, \\ P_2, Pitch_2, Duration_2) \quad \leftarrow \quad & note(V_1, P_1, Pitch_1, Duration_2), \\ & note(V_1, P_2, Pitch_2, Duration_2), \\ & next(V_1, P_1, P_2), \\ & constraint\_pitch(Pitch_1, equals, Pitch_2). \end{aligned}$$

which implies that every note in the same voice is constrained to the same pitch. We want this pattern to hold true in every combination of voices and positions possible, or if we momentarily exceed PROLOG's notation and complete it using quantifiers[9]:

$$\begin{aligned} \forall V_1 \\ \forall P_1 \\ \forall P_2 \\ \exists Pitch_1 \\ \exists Duration_1 \\ \exists Pitch_2 \\ \exists Duration_2 \\ pn(V_1, P_1, Pitch_1, Duration_1, \\ P_2, Pitch_2, Duration_2) \quad \leftarrow \quad & note(V_1, P_1, Pitch_1, Duration_2), \\ & note(V_1, P_2, Pitch_2, Duration_2), \\ & next(V_1, P_1, P_2), \\ & constrain\_pitch(Pitch_1, equals, Pitch_2). \end{aligned}$$

This way, the predicates *note*/4 and *constrain_pitch*/3 will pass as constraints to every triplet $\langle V, P1, P2 \rangle$ of voices and consecutive positions. After the constraints are added to the constraint set of our CSP, the solver will find a solution that holds for all these constraints, possibly with some exceptions as discussed below.

[9]Note that we may use PROLOG notation, but we do not write a PROLOG program. In reality, the language we use for the application of rules, allows for logic quantifiers.

4.3.2 Constraint Relaxation

There are situations, when learning rules from a musical piece, where these rules cannot hold simultaneously when applied as constraints over the entirety of a new composition. Such a situation will render the CSP solver unable to find a solution, therefore we need to relax the problem and apply the resulting constraints selectively over the new composition. To solve such a relaxed version of the problem, we introduce "choice points" when applying the constraints. A "choice point" gives us the liberty to decide whether to place a constraint over some variables or not. If we place a constraint and the solver is unable to find a solution under this decision, then we roll-back to this choice point, we lift this constraint, and we continue by placing the other constraints (that may possibly apply to the same variables). The additional freedom introduced by the choice points gives rise to various strategies on placing constraints, such as periodic, fixed, context-sensitive, or even random.

4.3.3 Constraints Application Example

Below is a (simplified[10]) example of the application of constraints and execution of the CSP solver for two patterns on a score of one voice and three positions. We'll represent our score as:

$$\langle Pitch_1, Pitch_2, Pitch_3 \rangle$$

where $Pitch_i$ is the domain of the pitch for the i-th note, and is initialized as $\{60, \ldots, 72\}$ (all the pitches between middle c and an octave above). Duration is irrelevant in this example.

We'll use the following two patterns:

$$\begin{aligned}
&pn_1(V, P_1, Pitch_1, Duration_1, \\
&\qquad P_2, Pitch_2, Duration_2) \quad \leftarrow \quad &&note(V, P_1, Pitch_1, Duration_1), \\
&&&note(V, P_2, Pitch_2, Duration_2), \\
&&&next(V, P_1, P_2), \\
&&&Pitch_1 < Pitch_2. \\
&pn_2(V, P_1, Pitch_1, Duration_1, \\
&\qquad P_2, Pitch_2, Duration_2) \quad \leftarrow \quad &&note(V, P_1, Pitch_1, Duration_1), \\
&&&note(V, P_2, Pitch_2, Duration_2), \\
&&&next(V, P_1, P_2), \\
&&&Pitch_1 > Pitch_2.
\end{aligned}$$

which conflict with each other. It is obvious that, if they are added to our constraint set both at the same time, our CSP solver will be unable to find a solution. The following actions take place:

1. For every two consecutive notes, a constraint (pn_1) to their pitches is added. For every pair of notes, the second note will have a higher pitch.

[10] Simplified, for clarity reasons. In reality, a choice point will be created for the assignment of the first group of constraints (the "<" constraints) as well.)

A choice point is created in order to remove the constraint in case the Solver fails to find a solution. Our score becomes:

$$\langle\{60,\ldots,72\},\{61,\ldots,72\},\{62,\ldots,72\}\rangle$$

2. For every two consecutive notes, a constraint (pn_2) to their pitches is added. For every pair of notes, the second note will have a lower pitch. A choice point is created in order to remove the constraint in case of failure. Since the first group (the "pn_1" constraints) were applied as well, our score becomes.

$$\langle\emptyset,\emptyset,\emptyset\rangle$$

3. Since the domains are empty, no solution can be found, so our CSP solver terminates the search in this branch and backs up. We then go back to a choice point and choose to omit one of the rules. Since the "pn_2" constraints were added last, we go back to that choice point and continue by omitting them. The score becomes once again:

$$\langle\{60,\ldots,72\},\{61,\ldots,72\},\{62,\ldots,72\}\rangle$$

and our solver returns the (random) solution:

$$\langle 60,65,70\rangle$$

Note that the example above relies on a specific strategy for distributing choice points. This strategy needs not be immutable. It can be changed as we see fit. In this example, we inserted a choice point, wherever we decided to apply a pattern to every voice and every position simultaneously. We could utilize a better (and more complex) strategy where we put the choice points, so that they reflect the application of patterns to any voice and any position, individually and independently.

4.3.4 Rule Modification and Handwritten Rules

After the induction of rules from examples, it is up to the user to decide how to use them in order to generate new scores. One can modify the extracted rules by relaxing several constraints or by changing others before feeding them to the CSP solver. For example, if we apply our ILP algorithm to a Bach chorale and figure out that it is written in a harmonic minor scale, we could easily remove the scale constraint to get the same piece in a new scale (defined a-priori or chosen randomly). Furthermore, one could even add a fifth voice to the four-voice chorale along with some additional and appropriate rules.

Rules need not be induced by examples only. The clarity of Horn clauses, allows us to easily build our own rules. For example, if we want to constraint every melodic interval in some voice, to, for example, 2 semitone (a whole tone), we would write:

$$\forall V \forall P$$
$$\exists P'$$
$$myrule(V,P,P')$$

where:

$$\begin{aligned} myrule(V, P, P') \quad \leftarrow \quad & note(V, P, Pitch), \\ & note(V, P', Pitch'), \\ & next(V, P, P'), \\ & distance(Pitch, Pitch', 2). \end{aligned}$$

4.4 Summary

The approach taken can be summarized in distinct phases, which include the extraction of examples, the induction of rules in the form of first-order predicates, and their application to a musical CSP solved by Strasheela. The ILP algorithm implemented and used is Morales' PAL algorithm [16]. The input to the ILP algorithm are examples automatically extracted from musical scores.

The steps taken in order to obtain the rules from the musical scores we provide as input are:

1. Obtain a piece's musical score in `**kern` representation.
2. Transform the `**kern` representation to first order predicates.
3. Generate and characterize the example sets for the target concepts.
4. Run the ILP PAL algorithm to induce generalized patterns.
5. Add, delete, modify, and combine generated and hand-written patterns.
6. Transform these patterns to constraints in a musical CSP.
7. Solve the musical CSP using Strasheela to obtain a new score.

given:

- a logic program $\mathcal{K}$
- a set of pattern definitions $\mathcal{P}$, such as $\mathcal{P} \subseteq \mathcal{K}$.
- a set of positive ($\mathcal{E}^+$) and negative ($\mathcal{E}^-$) examples each one described as a set of ground unit clauses

select an example $E_1 \in \mathcal{E}^+$
construct a new clause(NC) defined as: $NC \equiv C_1 \leftarrow E_1 \cup A_{1,1} \cup A_{1,2}, \ldots$ where:

- $\mathcal{K} \wedge E_1 \vdash A_{1,i}$ and $A_{1,i}$ is a ground instance of a pattern definition in $\mathcal{P}$
- C_1 is a head clause constructed from the arguments used in E_1 and a new predicate name PN.

set $\mathcal{E}^+ = \mathcal{E}^+ - \{E_1\}$
while $\mathcal{E}^+ \neq \emptyset$

- **select** a new example $E_j \in \mathcal{E}^+$
- **construct** a new clause (C'_j) defined as: $C'_j \equiv C_j \leftarrow E_j \cup A_{j,1} \cup Aj, 2, \ldots$ where:
  - $\mathcal{K} \wedge E_J \vdash A_{j,i}$ and $A_{j,i}$ is a ground instance of a pattern definition in $\mathcal{P}$
  - C_j is a head clause constructed from the arguments used in E_j and predicate name PN
- **set** $NC = clgg(C'_j, NC)$ where $clgg$ is an lgg between literals with the same labels.
- **if** NC covers an example in $\mathcal{E}^-$, **then** reject NC', save E_j in a disjunct list DL, and **continue**
- **set** $\mathcal{E}^+ = \mathcal{E} - \{E_j\}$

output NC
if there are examples in DL not covered by an NC, **then set** $\mathcal{E}^+ = DL$ and start the whole process again.

Figure 4.3: The complete PAL algorithm.

given:

- a logic program $\mathcal{K}$
- a set of pattern definitions $\mathcal{P}$, such as $\mathcal{P} \subseteq \mathcal{K}$.
- a set of positive ($\mathcal{E}^+$) examples each one described as a set of ground unit clauses

select and example $E_1 \in \mathcal{E}^+$
construct a new clause (NC) defined as: $NC \equiv C_1 \leftarrow E_1 \cup A_{1,1} \cup A_{1,2}, \ldots$ where:

- $\mathcal{K} \wedge E_1 \vdash A_{1,i}$ and $A_{1,i}$ is a ground instance of a pattern definition in $\mathcal{P}$
- C_1 is a head clause constructed from the arguments used in E_1 and a new predicate name PN.

set $\mathcal{E}^+ = \mathcal{E}^+ - \{E_1\}$
while $\mathcal{E}^+ \neq \emptyset$

- **select** a new example $E_j \in \mathcal{E}^+$
- **construct** a new clause (C'_j) defined as: $C'_j \equiv C_j \leftarrow E_j \cup A_{j,1} \cup A_{j,2}, \ldots$ where:
  - $\mathcal{K} \wedge E_J \vdash A_{j,i}$ and $A_{j,i}$ is a ground instance of a pattern definition in $\mathcal{P}$
  - C_j is a head clause constructed from the arguments used in E_j and predicate name PN
- **set** $NC = clgg(C'_j, NC)$ where $clgg$ is an lgg between literals with the same labels.
- **set** $\mathcal{E}^+ = \mathcal{E} - \{E_j\}$

remove constants and duplicates of variables from the head of NC
output NC

Figure 4.4: The modified version of PAL.

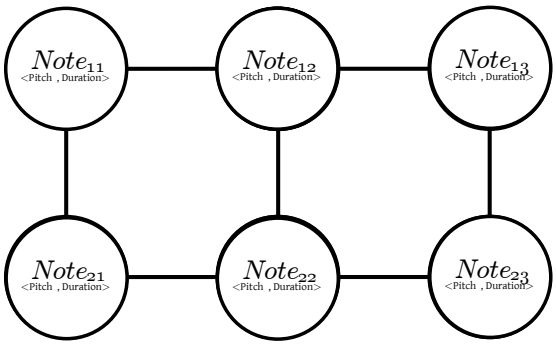


Figure 4.5: A 2×3 (2 Voices, 3 Positions) score in the form of a CSP.

Chapter 5

Application to Music Composition

5.1 A Method for Music Composition

Up to know we have defined all the methods and techniques we'll be using. Some of them may require some extra work, like providing the background knowledge we'll be using for rule production. We will describe them in this section.

The general idea is that:

1. Extract examples from existing music compositions.
2. Provide the background knowledge.
3. Run the ILP algorithm.
4. Transform the induced rules, into a CSP.
5. Modify, mix, or remove the existing rules and/or add new ones.
6. Run the CSP solver to yield a new composition.

Manual writing of rules

We can manually provide rules in FOL syntax, as described in previous chapters. In this case, no extra background knowledge must be given, aside from the definition of the patterns we will be using. In our implementation, only basic predicates, like $note/3$ and $melodic\_interval/4$ or the predicates describing motions, are provided. In this case, we just have to provide the system with the rules that define our music theory in FOL syntax. We do not need to provide any examples.

Single example set and rule extraction

We may use a single piece and rule induction to extract the rules that characterize this piece. We can then modify them by hand, perhaps relaxing some of the rules (rules act as constraints), and generate new pieces based on these modified rules or a subset of them. In this case, we have to provide the system with the required background knowledge, as well as the piece we want to extract rules from, in the form of examples. Usually, a single rule is extracted from this one example set.

Multiple example sets and rule induction

We may use a single or multiple piece(s) and rule induction to extract a variety of rules that characterize these pieces. If we want to do, for example, harmonic analysis, we may want to extract rules from all the 2×2 windows of simultaneous and consecutive notes over the score(s) (that include both harmonic intervals and their melodic transitions). We may extract one or more rules (one rule per example set). In this case, we need to provide the background knowledge as well as the example sets.

In the next section, we will provide examples of our methods. We will show one example for each case, the steps we follow, as well as the resulting composition.

5.2 Using Hand-Written rules

We will start by showing how easily one can compose new songs by providing hand-written rules in FOL syntax. Suppose we want to utilize four of the First species Fuxian counterpoint rules [16]:

1. From one **perfect consonance**[1] to another **perfect consonance**, one may progress with a **contrary or oblique motion**.
2. From a **perfect consonance** to an **imperfect consonance**, one may progress with **any motion**.
3. From an **imperfect consonance** to a **perfect consonance**, one may progress with a **contrary or oblique motion**.
4. From one **imperfect consonance** to another **imperfect consonance**, one may progress with **any motion**.

Note the terms in bold typeface. These correspond to predicates that we need to have defined in our knowledge base. Every numbered sentence above is a rule and corresponds to a specific pattern. Suppose we have the background knowledge given in Fig. 5.1 and Fig. 5.2. In order to express the first rule in FOL syntax we think as follows:

- Recognize the different positions we refer to. A harmonic consonance characterizes the harmonic interval between two notes at the same position from consecutive voices (one over the other). A motion implies melodic progression between two succesive notes in the same voice, (one next to the other). So, we are looking at 2×2 windows. over voices and positions. If a note is at (V, P) then this window also includes the notes at $(V+1, P)$, $(V, P+1)$ and $(V+1, P+1)$. Our "input" predicates are then:

$$note(V_1, P_1, P_{11})$$
$$note(V_1, P_2, P_{12})$$
$$note(V_2, P_1, P_{21})$$
$$note(V_2, P_2, P_{22})$$

  where P_{ij} are the corresponding pitches and the following apply:

$$next(P_1, P_2)$$
$$over(V_1, V_2)$$

  where $over(V, P, P')$ and $next(V, V', P)$ are already defined in the background knowledge.

  So, until now, our pattern for the four rules we seek to write has become (remember that our score is a two dimensional grid of undetermined notes, so the input *arguments* are just $Voice, Position$ of the note to be constrained):

[1] We refer to harmonic consonances.

$$
\begin{aligned}
over(V_1, V_2, P) \leftarrow\ & note(V_1, P, Pitch_1), \\
& note(V_2, P, Pitch_2), \\
& V_2 = V_1 + 1, \\
& Pitch_2 > Pitch_1. \\
next(V, P_1, P_2) \leftarrow\ & note(V, P_1, Pitch_1), \\
& note(V, P_2, Pitch_2), \\
& P_2 = P_1 + 1. \\
harmonicInterval(V_1, V_2, P, I) \leftarrow\ & note(V_1, P, Pitch_1), \\
& note(V_2, P, Pitch_2), \\
& over(V_1, V_2, P) \\
& I = Pitch_2 - Pitch_1. \\
harmonicPerfectConsonance(V_1, V_2, P) \leftarrow\ & harmonicInterval(V_1, V_2, P, 0); \\
& harmonicInterval(V_1, V_2, P, 7); \\
& harmonicInterval(V_1, V_2, P, 12). \\
harmonicImperfectConsonance(V_1, V_2, P) \leftarrow\ & harmonicInterval(V_1, V_2, P, 3); \\
& harmonicInterval(V_1, V_2, P, 4); \\
& harmonicInterval(V_1, V_2, P, 8); \\
& harmonicInterval(V_1, V_2, P, 9). \\
obliqueMotion(V_1, V_2, P_1, P_2) \leftarrow\ & note(V_1, P_1, Pitch_{11}), \\
& note(V_1, P_2, Pitch_{12}), \\
& note(V_2, P_1, Pitch_{21}), \\
& note(V_2, P_2, Pitch_{22}), \\
& next(V_1, P_1, P_2), \\
& next(V_2, P_1, P_2), \\
& over(V_1, V_2, P_1), \\
& over(V_1, V_2, P_2), \\
& (Pitch_{11} \neq Pitch_{12}, \\
& Pitch_{21} = Pitch_{22}; \\
& Pitch_{11} = Pitch_{12}, \\
& Pitch_{21} \neq Pitch_{22}).
\end{aligned}
$$

Figure 5.1: A sample background knowledge.

$$
\begin{aligned}
contraryMotion(V_1, V_2, P_1, P_2) \leftarrow\ & note(V_1, P_1, Pitch_{11}),\\
& note(V_1, P_2, Pitch_{12}),\\
& note(V_2, P_1, Pitch_{21}),\\
& note(V_2, P_2, Pitch_{22}),\\
& next(V_1, P_1, P_2),\\
& next(V_2, P_1, P_2),\\
& over(V_1, V_2, P_1),\\
& over(V_1, V_2, P_2),\\
& (Pitch_{11} > Pitch_{12},\\
& Pitch_{21} < Pitch_{22};\\
& Pitch_{11} < Pitch_{12},\\
& Pitch_{21} > Pitch_{22}).\\
directMotion(V_1, V_2, P_1, P_2) \leftarrow\ & note(V_1, P_1, Pitch_{11}),\\
& note(V_1, P_2, Pitch_{12}),\\
& note(V_2, P_1, Pitch_{21}),\\
& note(V_2, P_2, Pitch_{22}),\\
& next(V_1, P_1, P_2),\\
& next(V_2, P_1, P_2),\\
& over(V_1, V_2, P_1),\\
& over(V_1, V_2, P_2),\\
& (Pitch_{11} > Pitch_{12},\\
& Pitch_{21} > Pitch_{22};\\
& Pitch_{11} < Pitch_{12},\\
& Pitch_{21} < Pitch_{22}).\\
restrictToContraryOrObliqueMotion(V_1, V_2, P_1, P_2) \leftarrow\ & contraryMotion(V_1, V_2, P_1, P_2);\\
& obliqueMotion(V_1, V_2, P_1, P_2).\\
restrictToAnyMotion(V_1, V_2, P_1, P_2) \leftarrow\ & contraryMotion(V_1, V_2, P_1, P_2);\\
& obliqueMotion(V_1, V_2, P_1, P_2);\\
& directMotion(V_1, V_2, P_1, P_2).
\end{aligned}
$$

Figure 5.2: A sample background knowledge (cont.).

$$\begin{aligned} pn(V_1, P_1) \quad \leftarrow \quad & note(V_1, P_1, P_{11}), \\ & note(V_1, P_2, P_{12}), \\ & note(V_2, P, P_{21}), \\ & note(V_2, P_2, P_{22}), \\ & next(P_1, P_2) \\ & over(V_1, V_2) \end{aligned}$$

- Give the relations that hold for the input predicates and the restriction that need to be taken for these relations. In our example, for the first rule we want our notes at (V_1, P_1) and (V_2, P_1) to form a perfect consonance, and the same to apply to the notes at (V_1, P_2), (V_2, P_1).

$$harmonicPerfectConsonance(V_1, V_2, P_1)$$
$$harmonicPerfectConsonance(V_1, V_2, P_2)$$

and also we have a restriction that apply in this case. We must progress by moving with contrary or oblique motion, this can be described as:

$$restrictToContraryOrObliqueMotion(V_1, V_2, P_1, P_2)$$

which restricts the movement to either an oblique or contrary motion.

Given the above, the 4 rules can be written as follows:

$$
\begin{aligned}
pn_1(V,P) \leftarrow\ & note(V,P,Pitch_{11}),\\
& note(V,P_2,Pitch_{12}),\\
& note(V_2,P,Pitch_{21}),\\
& note(V_2,P_2,Pitch_{22}),\\
& next(P,P_2)\\
& over(V,V_2)\\
& harmonicPerfectConsonance(V_1,V_2,P_1)\\
& harmonicPerfectConsonance(V_1,V_2,P_2)\\
& restrictToContraryOrObliqueMotion(V_1,V_2,P_1,P_2).\\
pn_2(V,P) \leftarrow\ & note(V,P,Pitch_{11}),\\
& note(V,P_2,Pitch_{12}),\\
& note(V_2,P,Pitch_{21}),\\
& note(V_2,P_2,Pitch_{22}),\\
& next(P,P_2)\\
& over(V,V_2)\\
& harmonicPerfectConsonance(V_1,V_2,P_1)\\
& harmonicImperfectConsonance(V_1,V_2,P_2)\\
& restrictToAnyMotion(V_1,V_2,P_1,P_2).\\
pn_3(V,P) \leftarrow\ & note(V,P,Pitch_{11}),\\
& note(V,P_2,Pitch_{12}),\\
& note(V_2,P,Pitch_{21}),\\
& note(V_2,P_2,Pitch_{22}),\\
& next(P,P_2)\\
& over(V,V_2)\\
& harmonicImperfectConsonance(V_1,V_2,P_1)\\
& harmonicPerfectConsonance(V_1,V_2,P_2)\\
& restrictToContraryOrObliqueMotion(V_1,V_2,P_1,P_2).\\
pn_4(V,P) \leftarrow\ & note(V,P,Pitch_{11}),\\
& note(V,P_2,Pitch_{12}),\\
& note(V_2,P,Pitch_{21}),\\
& note(V_2,P_2,Pitch_{22}),\\
& next(P,P_2)\\
& over(V,V_2)\\
& harmonicImperfectConsonance(V_1,V_2,P_1)\\
& harmonicImperfectConsonance(V_1,V_2,P_2)\\
& restrictToAllMotions(V_1,V_2,P_1,P_2).
\end{aligned}
$$

- Translate these rules to constraints, apply them to every note in every voice in our undetermined score, and run the solver. A sample result

with unconstrained durations and an additional constraint that restricts every note to a note of the *G-major* scale is given in Fig. 5.3. Note that there are 8 notes in every voice and that the rules apply for each of the note, distinctively. For example the drawn 2×2 square consists of the first four first top-left notes (B-B-B-D); note that in this case, the second rule has been applied. Also note that the output is somewhat, upside-down. Usually we write the bass on the bottom and the treble on the top; the inverse ordering is due to our solution encoding in STRASHEELA. The end result does not conform to the general counterpoint rules of 1st species, since we do not provide them in their entirety but nevertheless it demonstrates how one can generate compositions that adhere to specific rules.

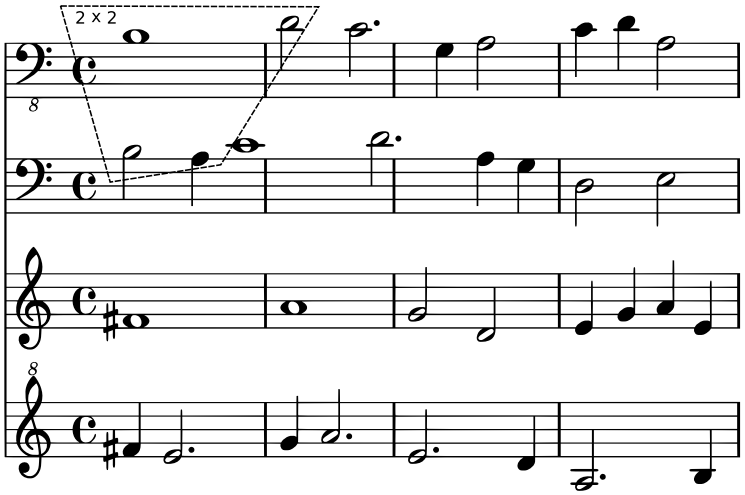


Figure 5.3: A composition that adheres to the First species Fuxian counterpoint rules.

5.3 Using Extracted Rules

In this scenario, we will show how we are able to produce new compositions from existing ones through the use of Rule Production. Suppose we have the simple melody in Fig 5.4. This melody has been taken from "PAL: A Pattern-Based First Order Inductive System" (Morales et al.) [16] as an example for counterpoint analysis. We will use this piece to extract rules and then relax them one by one to get new melodies based on this input.

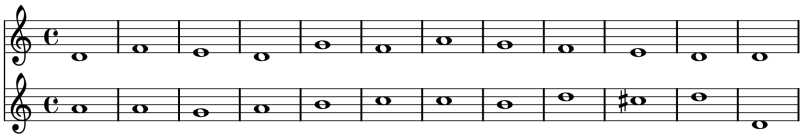


Figure 5.4: The simple 2 voice piece we will use as input.

We view this score as a solution to a CSP. Suppose we have had composed this song by applying a number of rules. The pitches and durations of the notes are nothing more than a solution to the corresponding CSP. If we somehow "discover" the rules that apply throughout the song, we should be able to produce the same song, just by finding a solution to that CSP. We can find these rules by running a rule production algorithm (for example, the PAL algorithm), with just one example set, so that induction does not take place[2]. We, then, can modify, add new, or even remove the existing rules from the description of our song. We could remove every constraint that is too specific for our tastes (for example, the *harmonicInterval* patterns that constraint an interval to a specific value) and keep only the rules about consonance or dissonance, certain melodic rules (for example, if we have a melodic step or skip) and the rules that describe motions (oblique, contrary, or direct). We then can run our solver again, and get interesting results. Below, we give an example of how one can get a new piece by removing existing constraints, providing new ones and even modifying the existing patterns.

Our input melody (Fig. 5.4), after the execution of PAL, becomes a single long pattern, of the form:

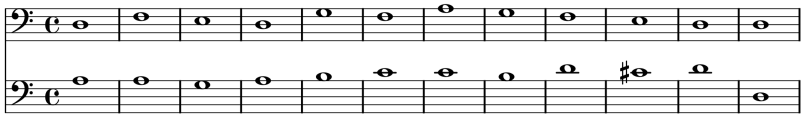
$$
\begin{aligned}
pn \quad \leftarrow \quad & note(1,1,62),\ldots,note(1,12,62),\\
& note(2,1,69),\ldots,note(2,12,62),\\
& \vdots\\
& cMajor(1,1),\ldots,cMajor(2,12)\\
& \vdots\\
& gMajor(1,1),\ldots,gMajor(2,12)\\
& \vdots\\
& contraryMotion(1,2,3,4),\\
& \vdots\\
& restrictToContraryOrObliqueMotion(1,2,11,12),\\
& melodicStep(1,1,2),\\
& \vdots\\
& melodicUpwards(2,10,11).
\end{aligned}
$$

Where the $note(Voice, Position, Pitch)$ predicates describe notes at voice $Voice$

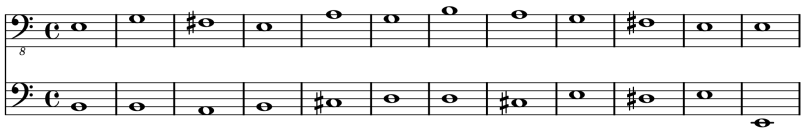
[2]Of course, we would still need to provide the system with a sufficient pattern library (background knowledge), in order to produce the rules.

and position $Position$ with a pitch $Pitch$, $cMajor(Voice, Position)$ and $gMajor(Voice, Position)$ indicate that the note at $(Voice, Position)$ can be part of a mode of the cmajor or gmajor scale respectively, $contraryMotion$ and $restrictToContraryOrObliqueMotion$ are defined as in the previous example, $melodicStep(Voice, Position_1, Position_2)$ holds when there is a melodic step[3] at $Voice$ between $Position_1$ and $Position_2$ and $melodicUpwards(Voice, Position_1, Position_2)$ is true when there is an ascending interval between $Position_1$ and $Position_2$ at voice $Voice$.

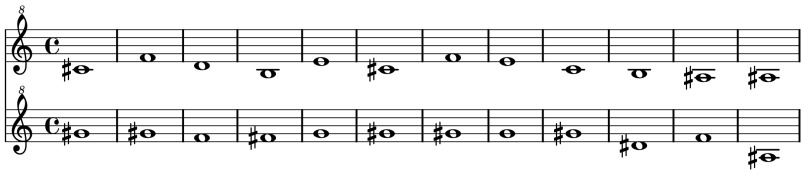
Running the solver with the entire pattern induced as a constraint, we simply regenerate the same piece. By removing the *note* predicates, we get the same piece transposed (but still on the same scale).



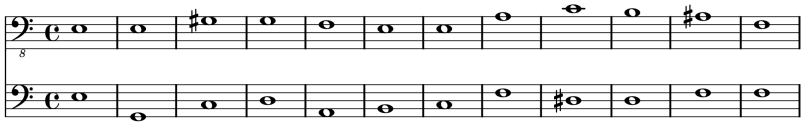
By removing the scale-referring predicates (ex. the *gmajor* and *cmajor* predicates), we get the same piece, transposed and at a different mode.



By removing the interval-related predicates (for example, the *harmonicInterval*), we get a new piece, that respects the consonances and dissonances of our theory as well as the melodic movements of our original piece.

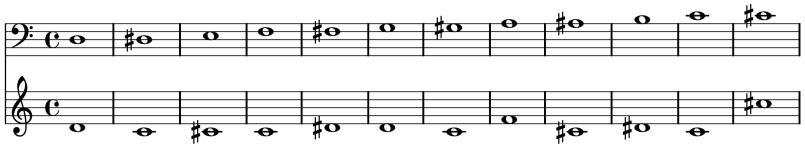


By removing every melodic movement related predicate (for example, the *melodicUpwards* or *melodicSteady* predicates), we get a different piece.

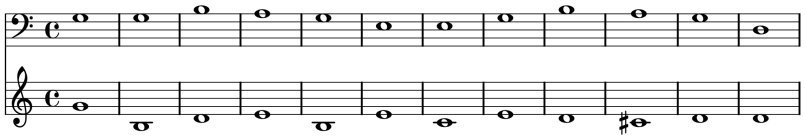


We can also get some nice results, if we input our own melody and let our system do the counterpoint. Here, we have used a chromatic scale as our *cantus firmus*.

[3] A melodic interval of at most a major third.



Here is an example where we just added the "perfect fourth" to the list of perfect consonant intervals.



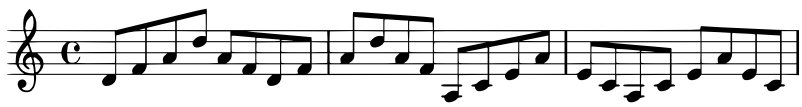
5.4 Using Induced Rules

In the previous example, we did not utilize induction at all. The pieces we could compose were limited to 11 whole-note measures and 2 voices, because of the original melody. Also, the rules extracted were applied to the same places for every piece we could compose.

We should not limit ourselves this way. We would like to compose as long pieces as we like and also apply the rules wherever we like. Below we show how this can be achieved by using induction to learn patterns that can be applied freely to our pieces.

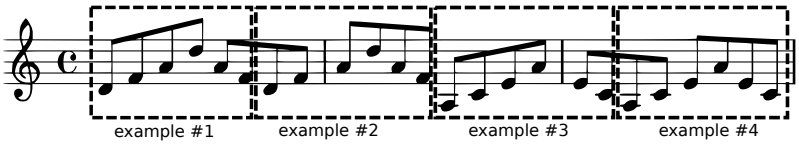
Arpeggios

One of the simplest melodic patterns are the *arppegios*, that is, the notes of a chord played in a sequence. A sequence of arppegios on a rhythm of $\frac{6}{8}$ for the chord sequence $Dm - Dm - Am - Am$[4] would be:



Suppose we wanted to learn the general pattern for such appregios. We need to provide our ILP algorithm with the examples that describe best our pattern. We can see that every measure contains one appregio, so we will use one measure for each example. The input to PAL is the above sequence of arpegios, split into examples:

[4]We got the output out of Strasheela, without checking for meter. That is why we have a meter of $\frac{4}{4}$ on our scores.



i.e. for example #1 we have:

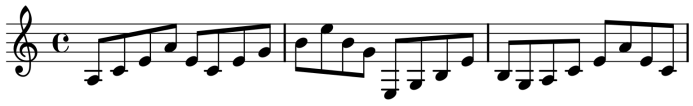
$$E_1 = \{note(1,1,62), note(1,2,65), note(1,3,69), note(1,4,74), note(1,5,69), note(1,6,65)\}$$

Our algorithm will give a single pattern as a FOL clause of the form:

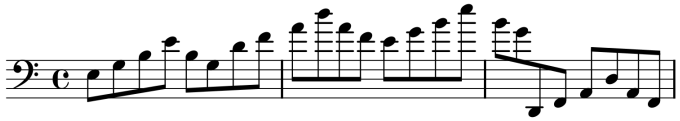
$$\begin{aligned} pn(Position_1, Pitch_1) \leftarrow\ & note(1, Position_1, Pitch_1), \\ & note(1, Position_2, Pitch_2), \\ & note(1, Position_3, Pitch_3), \\ & note(1, Position_4, Pitch_4), \\ & note(1, Position_5, Pitch_5), \\ & note(1, Position_6, Pitch_6), \\ & next(1, Position_1, Position_2), \\ & next(1, Position_2, Position_3), \\ & next(1, Position_3, Position_4), \\ & next(1, Position_4, Position_5), \\ & next(1, Position_5, Position_6), \\ & melodic\_interval(1, Position_1, Position_2, 3), \\ & melodic\_interval(1, Position_2, Position_3, 4), \\ & melodic\_interval(1, Position_3, Position_4, 5), \\ & melodic\_interval(1, Position_4, Position_5, 5), \\ & melodic\_interval(1, Position_5, Position_6, 4), \\ & melodic\_upwards(1, Position_1, Position_2), \\ & melodic\_upwards(1, Position_2, Position_3), \\ & melodic\_upwards(1, Position_3, Position_4), \\ & melodic\_downwards(1, Position_4, Position_5), \\ & melodic\_downwards(1, Position_5, Position_6), \\ & \vdots \end{aligned}$$

Which is a pattern that, when applied as a constraint, will give an arpeggio that begins at $Position_1$ and has its base pitch at $Pitch_1$. Our pattern's head has some "input" variables (variables that refer to voice or position) and we need to specify how we are going to apply it to our score.

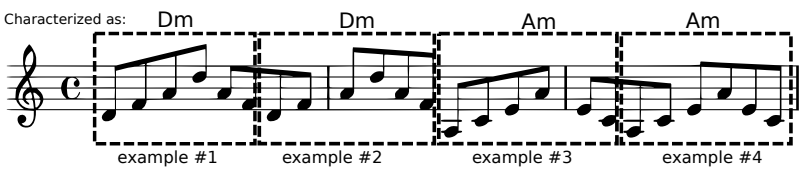
It does not make sense to apply this pattern to every single position in our score, because clearly the constraint cannot be satisfied for all sets of 6 continuous notes simultaneously. If we do so, the CSP solver will relax these constraints through the choice points and will deliver something like the score that follows:



A better idea would be to focus on measures. If we apply the rule to the first note of every measure in our score, we imply that every six succesive notes in a measure we will contain an apreggio.



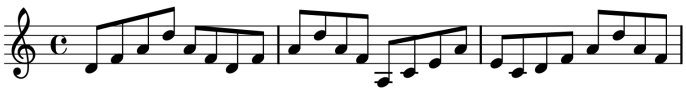
Now, we get single-measure arpreggios that begin at various pitches. If we would like to get only Am and Dm arpeggios, we would need to characterize our examples as Am or Dm before giving them as input to our ILP algorithm. We split again our melody into examples, and we characterize each one as Am or Dm:



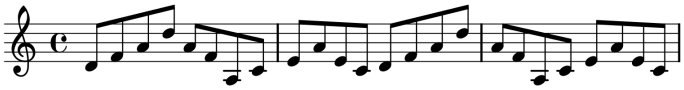
If we characterize the examples as such, we will get two different patterns, one for Dm and one for Am:

$$
\begin{aligned}
pnDm(Position_1) \leftarrow\ & note(1, Position_1, 62),\\
& note(1, Position_2, 65),\\
& note(1, Position_3, 69),\\
& note(1, Position_4, 72),\\
& note(1, Position_5, 69),\\
& next(1, Position_1, Position_2),\\
& next(1, Position_2, Position_3),\\
& next(1, Position_3, Position_4),\\
& next(1, Position_4, Position_5),\\
& next(1, Position_5, Position_6),\\
& \vdots\\
& melodic\_interval(1, Position_1, Position_2, 3),\\
& melodic\_interval(1, Position_2, Position_3, 4),\\
& melodic\_interval(1, Position_3, Position_4, 5),\\
& melodic\_interval(1, Position_4, Position_5, 5),\\
& melodic\_interval(1, Position_5, Position_6, 4),\\
& melodic\_upwards(1, Position_1, Position_2),\\
& melodic\_upwards(1, Position_2, Position_3),\\
& melodic\_upwards(1, Position_3, Position_4),\\
& melodic\_downwards(1, Position_4, Position_5),\\
& melodic\_downwards(1, Position_5, Position_6),\\
& \vdots\\
pnAm(Position_1) \leftarrow\ & note(1, Position_1, 57),\\
& note(1, Position_2, 60),\\
& note(1, Position_3, 64),\\
& note(1, Position_4, 69),\\
& note(1, Position_5, 64),\\
& next(1, Position_1, Position_2),\\
& next(1, Position_2, Position_3),\\
& next(1, Position_3, Position_4),\\
& next(1, Position_4, Position_5),\\
& next(1, Position_5, Position_6),\\
& \vdots\\
& melodic\_interval(1, Position_1, Position_2, 3),\\
& melodic\_interval(1, Position_2, Position_3, 4),\\
& melodic\_interval(1, Position_3, Position_4, 5),\\
& melodic\_interval(1, Position_4, Position_5, 5),\\
& melodic\_interval(1, Position_5, Position_6, 4),\\
& melodic\_upwards(1, Position_1, Position_2),\\
& melodic\_upwards(1, Position_2, Position_3),\\
& melodic\_upwards(1, Position_3, Position_4),\\
& melodic\_downwards(1, Position_4, Position_5),\\
& melodic\_downwards(1, Position_5, Position_6),\\
& \vdots
\end{aligned}
$$

If we randomly choose at each measure which of these patterns to apply, we will get something like this:



Of course, we can use the pattern in a more targetted way. We can specify that we want to apply a $Dm - Am - Dm - Am$ arpreggio sequence which results to the score below:



Chapter 6

Conclusion

6.1 Limitations

Music can be seen as a formal language and can be described well in first-order logic. This enables us to learn patterns and use them in order to constrain a music piece, like music theory rules do. However, this learning process has some theoretical and practical problems. Methods like "forward-chaining" for rule production (used in the ILP methods we use) have exponential time complexity in the worst case. The search methods for solving CSPs also have exponential time complexity in the worst case. Clearly, the problem of learning music rules and applying them to composition is in general NP-hard (as most interesting problems). Therefore, as also evidenced by our experience, we do not expect that we will be able to apply these ideas to lengthy compositions.

Regarding induction, we use PAL which in turn uses the concept of *Least General Generalizations*. LGG has various disadvantages: with misformed examples it can easily lead to overgeneralized rules, while with insufficient examples it can lead to overspecification. It also provides no way of knowing if a sufficient number of examples have been provided. Additionally, it is not resilient to noise; noisy data can easily lead to undesirable results due to fitting of the noise. There is also the issue of background knowledge. By increasing the information provided in the background knowledge base, we most likely increase the size of the literals. PAL has a way of constraining the amount of literals produced and we have constrained it even further, but still, it can lead to patterns that are huge in literal size. And, unfortunately the larger the pattern size, the larger number of examples it requires to avoid overspecification.

In this work, we wanted to learn the structure of rules that characterize a musical example, apply these rules as constraints over a new score and run the CSP solver to get the final composition. What we have not discussed in detail in this text is how the application of patterns in the CSP is being done. Assigning values to our CSP variables at random and/or applying constraints at random positions does not work well. While the results may be correct, they're not always pleasing to the ear. This may be difficult to be dealt with the use of FOL rules alone. While one can enhance sheet music with many performance details, an issue arises when trying to model human behavior in performance or, even worse, in composition. For example, it can be very difficult to formulate the thought process of a Jazz musician improvising. Perhaps the usage of FOL rules should be coupled with some sort of adaptive model such as a neural network as in HARMONET [22] or some kind of a statistical model.

6.2 Future Work

Strasheela **Extension** As it currently stands now, our work is implemented by a software prototype based on a mixture of programming languages (Prolog, Oz, Bash, Python together with Awk for the music representation/cognition library HumDrum [25]). This implementation makes debugging very inefficient and provides a rather awkward workflow in order to proceed from existing musical pieces to compositions. It would be ideal if our work's prototype could be rewritten in Oz, which is the language of the original Strasheela prototype, as a Strasheela extension.

Distributed Environments Memory and processing power in modern computers have exponentially grown and their almost constant connectivity nowadays have made them easily accessible and inexpensive resources. Grid computing, cloud services, and distributed systems in general have become very easy to establish. It would be nice if we could modify our implementation so that it utilizes the processing power of computers distributed across the internet. A good start would be implementing our prototype as a Strasheela extension (see above) since the Oz programming language offers various built-in methods for distributed computing.

ILP Algorithms PAL works well for our simple examples, but since it uses *lgg*, it has its limitations and greatly depends on the examples given (i.e. noisy or incorrect data in the examples can lead to overgeneralization, whereas lack of enough examples can lead to overspecification). We would like to see how other ILP algorithms perform within our problem. For example, there is Muggleton's CIGOL which uses reverse resolution or PROGOL which uses inverse entailment. A list of various ILP systems can be seen at [29]. It is possible one could adapt several of these systems in order to do rule-production in the style of PAL.

Behavioral Learning Hild et al. [22] use a neural network in conjuction with deterministic symbolic rules to control the application of these rules, so that the resulting piece is not only correct, but also aesthetically pleasing. In our work, we do structural learning of rules using ILP, but we do not do any attempt to control their application. It would be nice if we could do behavioral learning, using some kind of metric. For example, we could use some work based on the emotions induced by music (i.e. works described in [30]), to guide the composition process.

User Interface One may design a graphical user interface that allows the user to input the examples for the ILP algorithm in a graphical way (perhaps using a score as input like any notation software). Such a graphical user interface could allow the user to work on the rules as if they were distinct graphical elements and even design new rules and modify them in such a graphical way.

Music Editing Aside from a method for computer-aided music composition, our work could be used to support a nice score completion property of conventional score-writing software in the same way that word-completion is being done on word processors, thus speeding up the writing of a score. The software would gradually learn the way the score-writer writes and suggest possible completions in the same way modern word processing software does.

Appendix A

Software Usage Instructions

A.1 Software Repository

The user may check out a current snapshot of our code from GITHUB:

https://github.com/mmxgn/induction

A.2 System Information

Working versions for our prototype implementation:

- Operating system: DEBIAN GNU/LINUX 6.0.1 (squeeze).
- Shell: BASH
- Prolog: SWI-PROLOG 5.10.4.
- Python: 2.6.6.
- Awk: GAWK 1.3.7.
- HUMDRUM: Git snapshot (May 27, 2011).
- MOZART/OZ: 1.4.0
- STRASHEELA: 0.9.10

Used software package websites:

- SWI-PROLOG: http://www.swi-prolog.org/index.txt
- Python: http://www.python.org/
- HUMDRUM: http://github.com/genos/humdrum/
- MOZART/OZ: http://www.mozart-oz.org/
- STRASHEELA: http://strasheela.sourceforge.net/

A.3 Scripts

Below is a list of the scripts we developed and their usage.

- `kerntofol.sh` – BASH script
  - usage: `sh kerntofol.sh input.krn timebase`, where `input.krn` is a `**kern` file and `timebase` is the minimum time base unit we use.
  - outputs: `input.in.pl`, where `input.in.pl` is the translated `input.krn` `**kern` file to FOL predicates.
- `generate_2x2_examples.pl` – PROLOG script
  - usage: `swipl -s generate_2x2_examples.pl > examples.pl`
  - expects: a file `input.pl` containing the song in FOL predicates format.

  - outputs: `examples.pl`, where `examples.pl` is a file containing example sets of the 2×2 window form we saw in Chapter 5, using the *note*/3 input predicates.

- `generate_measure_examples.pl` – PROLOG script
  - usage: `swipl -s generate_measure_examples.pl > examples.pl`
  - expects: a file `input.pl` containing the song in FOL predicates format.
  - outputs: `examples.pl`, where `examples.pl` is a file containing example sets from the predicates in each measure, using the *note*/3 input predicates.

- `generate_measure_examples_duration.pl` – PROLOG script
  - usage: `swipl -s generate_measure_examples_duration.pl > examples.pl`
  - expects: a file `input.pl` containing the song in FOL predicates format.
  - outputs: `examples.pl`, where `examples.pl` is a file containing example sets from the predicates in each measure, using the *note*/4 input predicates.

- `generate_wholepiece_examples.pl` – PROLOG script
  - usage: `swipl -s generate_wholepiece_examples.pl > examples.pl`
  - expects: a file `input.pl` containing the song in FOL predicates format.
  - outputs: `examples.pl`, where `examples.pl` is a file containing the whole piece as a single example set, using *note*/3 input predicates.

- `generate_wholepiece_examples_duration.pl` – PROLOG script
  - usage: `swipl -s generate_wholepiece_examples_duration.pl > examples.pl`
  - expects: a file `input.pl` containing the song in FOL predicates format.
  - outputs: `examples.pl`, where `examples.pl` is a file containing the whole piece as a single example set, using *note*/4 input predicates.

- `characterize.py` – PYTHON script
  - usage: `python characterize.py`
  - expects: a file `example.pl` containing example sets.
  - outputs: `examples_new.pl`, where `examples_new.pl` is a file containing the labeled examples.

- `pal.pl` – PROLOG script
  - usage: `swipl -s pal.pl`

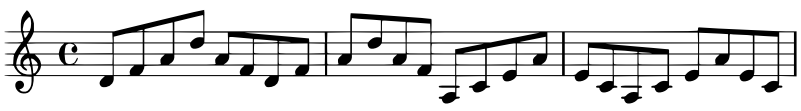
  - expects: a file `examples.pl` containing our positive example sets and a file `patterns.pl` containing the patterns already known in our knowledge base.
  - outputs: `pal_out.pl`, where `pal_out.pl` is a file containing the induced rules in PROLOG Horn clauses format.

- `genpats.py` – PYTHON script
  - usage: `python genpats.py pal_out.pl`
  - outputs: `oz_pal.oz`, `oz_loops.oz`, where `oz_pal.oz` is a file containing the induced rules, in OZ format, and `oz_loops.oz` is a file describing how these will be applied to our score.

- `solver.oz` – OZ script
  - usage: `oz solver.oz` – runs CSP solver.
  - expects: `oz_patterns.oz`, `oz_pal.oz`, `oz_loops.oz`.

A.4 Workflow Example

Suppose we have the following melody in `**kern` format, in `arp.krn`:



and we want to generate a new piece by learning a concept for each measure (6 notes here). The steps we follow are:

1. Convert to FOL:

```
$ sh kerntofol3.sh arp.krn 8
$ cp arp.in.pl input.pl
```

2. Generate examples from each measure.

```
$ swipl -s generate_measure_examples.pl > examples.pl
```

3. Characterize examples:

```
$ python characterize.py
$ cp examples_new.pl examples.pl
```

4. Run PAL:

```
$ swipl -s pal.pl
```

5. Convert to OZ representation:

```
$ python genpats.py pal_out.pl
```

6. Run CSP solver.

```
$ oz solver.oz
```

7. Feed to Oz. Press CTRL+'.'+'B'. At the explorer window, double click the green box (solution).

.

Bibliography

[1] "Edgard Varèse." http://en.wikipedia.org/wiki/Edgard_Var%C3%A8se, 2011. [Online; accessed 21-July-2011].

[2] john a. maurer iv, "History of algorithmic composition," tech. rep., Stanford University, 1999. https://ccrma.stanford.edu/~blackrse/algorithm.html.

[3] M. Edwards, "Algorithmic Composition: Computational Thinking in Music," *Communications of the ACM*, vol. 54, no. 7, pp. 58–67, 2011.

[4] "Sheet music — Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Sheet_music, 2011. [Online; accessed 22-July-2011].

[5] A. Alpern, "Techniques for algorithmic composition of music," tech. rep., Hampshire College, 1995.

[6] C. Wamser and C. Wamser, "Lejaren A. Hiller, Jr.: A Memorial Tribute to a Chemist-Composer," *Journal of Chemical Education*, vol. 73, no. 7, p. 601, 1996.

[7] "Wolframtones: An experiment in a new kind of music." http://tones.wolfram.com/.

[8] C. Ariza, "Navigating the landscape of computer-aided algorithmic composition systems: a definition, seven descriptors, and a lexicon of systems and research," in *International Computer Music Conference*, pp. 765–772, 2005.

[9] T. Anders, "Composing music by composing rules: Computer aided composition employing constraint logic programming," tech. rep., School of Music & Sonic Arts, Queen's University Belfast, 2003.

[10] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd ed., 2002.

[11] A. Sakharov, "Interpretation — mathworld – a wolfram web resource, created by Eric W. Weisstein." http://mathworld.wolfram.com/Interpretation.html.

[12] W. F. Clocksin and C. S. Mellish, *Programming in Prolog: Using the ISO standard*. Springer, 2003.

[13] S. Muggleton and C. Feng, "Efficient induction of logic programs," *Inductive logic programming*, vol. 38, pp. 281–298, 1992.

[14] S. H. Muggleton, "Inverse Entailment and Progol," *New Generation Computing*, vol. 13, no. 3, pp. 245–286, 1995.

[15] "The aleph manual." http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html.

[16] E. F. Morales, "PAL: A pattern-based first-order inductive system," *Machine Learning*, vol. 26, pp. 227–252, Feb – March 1997.

[17] S. Dzeroski and N. Lavrac, eds., *Relational Data mining*. Springer, 2001.

[18] T. Anders, *Composing music by composing rules: Design and usage of a generic music constraint system*. PhD thesis, School of Music & Sonic Arts, Queen's University Belfast, 2007.

[19] "Csounds.com." http://www.csounds.com/.

[20] T. Anders and E. R. Miranda, "Constraint-Based Composition in Real-time," in *International Computer Music Conference*, 2008.

[21] K. Ebcioğlu, "An expert system for harmonizing chorales in the style of J. S. Bach," *The Journal of Logic Programming*, vol. 8, pp. 145–185, 1990.

[22] H. Hild, J. Feulner, and W. Menzel, "HARMONET: A neural net for harmonizing chorales in the style of J. S. Bach," in *Neural Information Processing Systems*, pp. 267–267, Morgan Kaufmann, 1993.

[23] E. Morales and R. Morales, "Learning musical rules," in *IJCAI-95 International Workshop on Artificial Intelligence and Music, 14th International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Canada*, 1995.

[24] R. Morales-Manzanares, E. F. Morales, R. Dannenberg, and J. Berger, "SICIB: An Interactive Music Composition System Using Body Movements," *Computer Music Journal*, vol. 25, pp. 62–79, July 2001.

[25] "The humdrum toolkit: Software for music research." http://musicog.ohio-state.edu/Humdrum/.

[26] "Kernscores." http://kern.ccarh.org/.

[27] "Strasheela reference documentation." http://strasheela.sourceforge.net/strasheela/doc/StrasheelaReference.html.

[28] "Mozart documentation." http://www.mozart-oz.org/documentation/.

[29] "Inductive logic programming — Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Inductive_logic_programming, 2011. [Online; accessed 22-July-2011].

[30] "2010: Audio music mood classification." http://www.music-ir.org/mirex/wiki/2010:Audio_Music_Mood_Classification.