Modeling music theories with the logic programming library core.logic

Martin Forsgren

Department of Computing Science
Umeå University, Sweden
id09mfn@cs.umu.se

Abstract. This paper examines the suitability of using the programming language Clojure and its logic programming library core.logic for modeling music theories. This is done by modeling some harmonization rules, which are then used to find accompanying chords to a melody, and conversely to find a suitable melody to a given chord progression. The implementation shows that core.logic is indeed suitable for modeling music theories and useful for generating shorter snippets of music, but too slow to generate full pieces. Adding support for custom search strategies to core.logic and using a search strategy optimized for music problems is suggested as a way to improve the performance.

1 Introduction

Computers have been used to create and model music for almost as long as there have been computers. The Illiac suite, by many regarded as the first computer generated composition, was created in the 50s [1], and rules to describe composition have been used long before computers existed [2].

Tonal music, that includes most western music, is especially well suited to model with computers since it has been mathematically formalized and there are many works describing rules for various aspects of tonal music e.g. harmonization and counterpoint [1].

Constraint programming is a method often used to model music theories which makes it possible to write music rules declaratively and makes it easy to combine the rules in different ways [2].

Using constraint programming is often more computationally expensive than using custom algorithms written in procedural languages but has the advantage of being more flexible, since the rules can be specified separately and then freely be combined, and less time-consuming, since specially designed algorithms does not need to be developed [2].

The core.logic logic programming library for Clojure has support for constraint programming and is modeled after miniKanren [3] and its extensions cKanren[4] and αKanren[5]. The miniKanren system is designed with focus on pure relations and finite failure[1], this design makes it possible to express things

[1] Finite failure means that programs that can not produce an answer should fail in a finite amount of time. Even if it is not possible to guarantee finite termination

that would diverge in many other logic programming systems and to use all of a relations arguments both as input and output arguments [3].

1.1 Research aims

The goal of this paper is to investigate the suitability of using the programming language Clojure and its logic programming library core.logic for modeling musical theories. This will be done by trying to model some theories for harmony and from that generate music that complies with the established rules.

1.2 Motivation

The design of core.logic differs from many other logic programming systems and it would be interesting to see if this has an effect on how well core.logic can be used to model music theories. There also exists a fairly popular[2] library for sound synthesis and music programming for Clojure called Overtone. Using core.logic to model music theories could be a good additional tool for people wanting to create music with Clojure.

1.3 Related work

A number of music constraint programming systems exist [2], as well as systems for automatic harmonization [1].

1.4 Structure of the paper

Section 2 explains theory needed for the rest of the paper. Section 2.1 will give a brief explanation of constraint logic programming in general, Section 3.1 gives an overview on core.logic and Section 2.2 goes on to explain some music theory needed for harmonization. Section 3 describes an implementation of the music theories using core.logic. Section 4 shows how the implemented theories can be used to generate accompanying chords to a given melody and to generate a suitable melody to a given chord progression. Section 5 discusses the implementation and core.logic's suitability for modeling music theories. Finally Section 6 gives some suggestions for future research.

2 Theory

2.1 Constraint Logic programming

A constraint logic program consists of a set of logic variables and constraints on those variables, this is sometimes called a knowledge base.

in general (the Entscheidungsproblem), it is possible to make a large portion of programs have this property.

[2] To give a rough picture there were 482 subscribers to the Overtone mailing list 2012-05-22, `https://groups.google.com/forum/?fromgroups#!members/overtone`

The knowledge base can be queried by running it through a logic engine which searches through the knowledge base and returns all assignments to the query variables that is consistent with the constraints. It is similar to how a relational database system is given a relational statement (often in the form of a SQL-query) and returns all data that satisfy the statement.

In contrast to regular variables, logic variables does not always contain a certain value, they instead contain information about possible values for the variable. When a logic variable is created nothing is known about it, so it is assumed that it can take on all values, but as constraints and relations between variables are added more knowledge is gained about them and the possible values they can take on gets fewer.

Programming with relations Logic programming uses relations instead of functions. For example + can be seen as a relation between three numbers: the two numbers that are added and their sum. The + relation can be used just like + would be used in any programming language - if we have $1 + 2 = z$ the logic engine can conclude that z must be three - but in contrast to ordinary programming languages all of the the parameters to the relation can be used both for input and output, for example $1 + y = 4$ gives $y = 3$. Sometimes several answers are possible, if x and y are positive integers $x + y = 2$ is satisfied both by $x = 1, y = 1$ and $x = 2, y = 0$ and $x = 0, y = 2$.

This ability to use any parameter as either input or output makes it possible to "run a program backwards", for example a interpreter written relationally can not only be used to generate a output value given a program, but can also, given a desired output, generate a program that when run gives that output [6].

2.2 Music Theories

Scales A scale is an ordered set of pitches, most often spanning a single octave. Each pitch in a scale is called a degree [7].

The most common type of scales in western music are diatonic scales, for example both the major and minor scales are diatonic [7].

A diatonic scale is a scale that has seven degrees separated by five whole tones and two semitones and where the semitones are spaced so that they are two or three whole tones apart [7].

The degrees of the diatonic scale are called tonic (I), super tonic (II) mediant (III), subdominant (IV), dominant (V), submediant or superdominant (VI) and leading tone or subtonic (VII) [8][7].

Intervals There are six possible simple intervals between the degrees in a diatonic scale. An interval between notes of the same degree is called unison, an interval one degree apart is called second, an interval two degrees is called a third, and so on, see Figure 1.

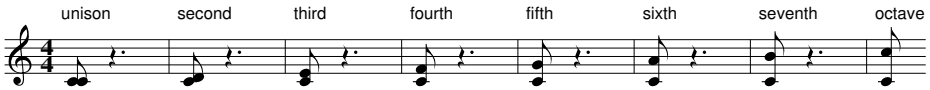

Fig. 1. Intervals in a diatonic scale

Chords A chord is a harmonic set of notes played together. The most common type of chord are the triad. A triad, as the name suggests, consists of three notes. They are called the root, the third – since it is a third apart from the root – and the fifth.

The most important chords are the tonic, subdominant and dominant, also called the primary triads (Figure 2).

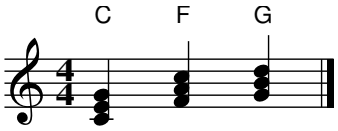

Fig. 2. The primary triads in the C-major scale

Harmonization Much can be said about harmony in music and many rules and guidelines can be found in the numerous books and treatises that have been written on the subject [1], but in this paper only a few basic harmonization rules will be used as an example.

When all notes are equal in length such as in the four part choral music modeled by Tsang and Aitken in [9] it is possible to always keep simultaneous notes in harmony, that is in the same chord, however when a melody consisting of for example quarter notes is accompanied by whole note chords, some non-harmonic notes must be allowed in the melody to not make the composition too restricted.

Two common non-harmonic note types are passing notes and auxiliary notes, also called neighbor notes or returning notes.

A passing note is a note that lies between two chordal notes that are a third apart and and an auxiliary note is a note that lies between two chordal notes of the same pitch and are either a step above or below in pitch [10] (Figure 3).

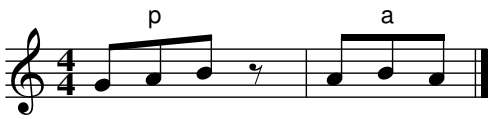

Fig. 3. A passing note (p) and an auxiliary note (a).

3 Implementation

The harmonization rules implemented are a subset of the rules described in an example[3] on the homepage of Torsten Anders constraint based music composition system Strasheela [11].

The rules were chosen as an example because they were simple but still result in music that sounds fairly good according to the author. What sounds good is of course highly subjective, the important thing here however is not to make good music but to see how one can declare rules that the generated music will follow. Rules can always be changed and added to make better sounding music, for some definition of better.

In addition to core.logic, the implementation uses the music programming library Overtone for creating sound and the composition library Leipzig which integrates with Overtone and provides functions for manipulating notes represented as maps.

Besides the relations described below some non-relational preprocessing functions for adding neighbor notes and dividing a phrase into bars were defined, these will not be described in detail here but they can be found in the complete source code[4] for the interested.

3.1 A short introduction to core.logic

The core constructs of core.logic are `==` (pronounced unify), `conde`, `fresh`[5]

The `==` relation states that two values should be the same, `fresh` introduces new logic variables and `conde` is used for logical disjunction (or) and can therefore be used to get several alternative answers.

To run a core.logic program `run*` or `run` is used. A call to `run*` looks like this:

```
(run* [query-variable(s)]
  relations)
```

[3] `http://strasheela.sourceforge.net/strasheela/doc/Example-AllIntervalSeries.html`, fetched 2013-03-30

[4] The complete source code can be found on `http://github.com/dentrado/musiklogik`

[5] fresh is called exist in [3]

and returns a list of all possible values the query variables can have (an empty list if no values are possible). `run` works just like `run*` but takes a number as its first argument and returns at most that many answers.

In addition to `==`, `fresh` and `conde`, core.logic contains many useful relations e.g. for working with sequences and for constraint programming over finite domains, residing in the namespace `core.logic/fd`.

Relations in core.logic and miniKanren are often called goals and conventionally have names ending with o.

Figure 4 shows some short example programs, the results of the programs are shown as a comment: `;=> (result)`.

```
(run* [q]
  (== q "foo"))
;=> ("foo")

(run* [q]
  (== q "foo")
  (== q "bar"))
;=> ()

(run* [q]
  (fresh [x y]
    (== x 1)
    (== x y)
    (== q y)))
;=> (1)

(run* [q]
  (conde
    [(== q "foo")]
    [(== q 1) (== q 2)]
    [(== q 3)]))
;=> ("foo" 3)

(run* [q]
  (fresh [x]
    (appendo x q '(1 2 3))))
;=> ((1 2 3) (2 3) (3) ())

(run* [q]
  (fd/in q (fd/interval 0 5)))
;=> (0 1 2 3 4 5)

(run* [q]
  (fd/in q (fd/interval 0 100))
  (fresh [x y]
    (fd/in x y (fd/interval 0 5))
    (fd/* x y q)))
;=> (0 1 2 3 4 5 6 8 9 10 12 15 16 20 25)
```

Fig. 4. core.logic examples

3.2 Music representation

Some music theories express how a note relates to previous and following notes, while others express relations between simultaneous notes. Thus, how and what of this contextual information is represented, greatly affects how easy it is to express different music theories [2].

Pitches are represented as integers, a natural choice since core.logic at the time of writing only supports constraint programming for integers. 0 will be the

first note in the scale and 7 will be one octave higher. Only diatonic scales are supported, and pitches not in the scale can't be expressed. This also happens to be the way Leipzig represent pitches.

Notes are represented as maps (hash tables), both to simplify interfacing with Leipzig and because it makes it easy to add additional context information to the notes. The note maps most importantly contain the note's pitch and duration but they also contain the previous and next note to make it easier to express constraints specifying how a note relates to its neighbors, see Figure 5.

Chords are simply represented as vectors containing three pitches, since only triads are used and no duration or context information is needed in this case, see Figure 6.

```
{:pitch 1, :duration 1, :time 1          ; A passing note
 :next {:pitch 2, :duration 1, :time 2},
 :prev {:pitch 0, :duration 1, :time 0}}
```

Fig. 5. Notes are represented as a maps

```
[0 2 4] ; The tonic chord
```

Fig. 6. Chords are represented as vectors containing three pitches

3.3 Relations

Pitch relations Intervals between pitches are modeled as the difference between two pitch values, after the values have been normalized to the first octave. Wrap around is also taken into account so there is for example an interval of 1, a second, between 6 and 0, see Figure 7.

Note relations A passing note is modeled as a note whose pitch is a second above its predecessor and a second below its successor, or conversely a second below its predecessor and a second above its successor, see Figure 8.

An auxiliary note is modeled as a note which is either a second below or a second above both its predecessor and successor, which should have the same pitch, see Figure 8.

```
(def domain (fd/interval 14))

(defn normalizeo
  "normalize a pitch to be between 0-7"
  [p p-normalized]
  (fd/in p domain)
  (fd/in p-normalized (fd/interval 0 7))
  (conde
    [(fd/< p 7) (== p p-normalized)]
    [(fd/>= p 7) (fd/- p 7 p-normalized)]))

(defn intervalo [num pitch1 pitch2]
  (fresh [p1norm p2norm]
    (fd/in num pitch1 pitch2 p1norm p2norm domain)
    (normalizeo pitch1 p1norm)
    (normalizeo pitch2 p2norm)
    (conde
        [(fd/- p2norm p1norm num)]
        [(fd/eq (= (- (+ 7 p2norm) p1norm) num))] ; octave wraparound
        )))
```

Fig. 7. Pitch relations

```
(defn aux-noteo [note]
  (fresh [p1 p2]
    (fd/in p1 p2 domain)
    (featurec note {:prev {:pitch p2}, :pitch p1, :next {:pitch p2}})
    (conde
      [(secondo p2 p1)]    ; (< prev next)
      [(secondo p1 p2)])))  ; (> prev next)

(defn passing-noteo [note]
  (fresh [p1 p2 p3]
    (fd/in p1 p2 p3 domain)
    (featurec note {:prev {:pitch p1}, :pitch p2, :next {:pitch p3}})
    (conde
      [(secondo p1 p2) (secondo p2 p3)]    ; (< prev next)
      [(secondo p3 p2) (secondo p2 p1)])))  ; (> prev next)
```

Fig. 8. Note relations

Chord relations A chord is a triad if it's a third between the first and second note and a fifth between the first and third note. A note is said to be in a chord if its normalized pitch is part of the chord.

A chord can accompany a bar if all tones in the bar are legal notes, that is a note that is either in the chord or is a passing or auxiliary note between two notes in the chord, see Figure 9.

```
(defne triado [chord]
  ([[a b c]] (thirdo a b) (fiftho a c)))

(defn in-chordeo [tone chord]
  (fresh [p p-norm]
    (featurec tone {:pitch p})
    (fd/in p p-norm domain) ;p-norm
    (normalizeo p p-norm)
    (membero p-norm chord)))

(defn legal-noteo
  [note chord]
  (conde
    [(in-chordeo note chord)]
    [(fresh [prev next]
       (featurec note {:prev prev :next next})
       (in-chordeo prev chord)
       (in-chordeo next chord)
       (conde [(passing-noteo note)] [(aux-noteo note)]))]))

(defn accompanyo [[bar chord]]
  (everyg #(legal-noteo % chord) bar))
```

Fig. 9. Chord relations

4 Results

The relations defined above can be used to find accompanying chords to a given melody or to find a melody to a given chord progression or even to generate a piece of music completely from scratch (the last however is slow and does not give particularly good results).

In the figures below all examples are in C major, but the generated music can be played back in any diatonic scale.

4.1 Finding accompanying chords to a melody

As an example accompanying chords will be generated for the first four bars of the German folk song "Horch was kommt von draussen rein".

All chords are constrained to be triads, furthermore the first and last chord must be the same, this is done because it sounds better, but it also reduces the search space significantly. Finally the four chords are constrained to accompany one bar each (Figure 10).

```
(def horch (phrase
             [1 1 1 1, 1 1 2, 1 1 2, 1 1 2]
             [0 1 2 3, 4 5 4, 3 1 6, 4 2 7]))
(def melody (add-neighbours horch))
(def mel-bars (bars melody))

(def find-chords-example
  (let [chords (lvars 4)]
    (run 4 [q]
      (== q chords)
      (== (first chords) (last chords))
      (everyg triado chords)
      (everyg accompanyo (zip mel-bars chords)))))
```

Fig. 10. Example of finding accompanying chords to a melody

These rules gives several possibilities, three examples of generated chord progressions can be seen in Figure 11.

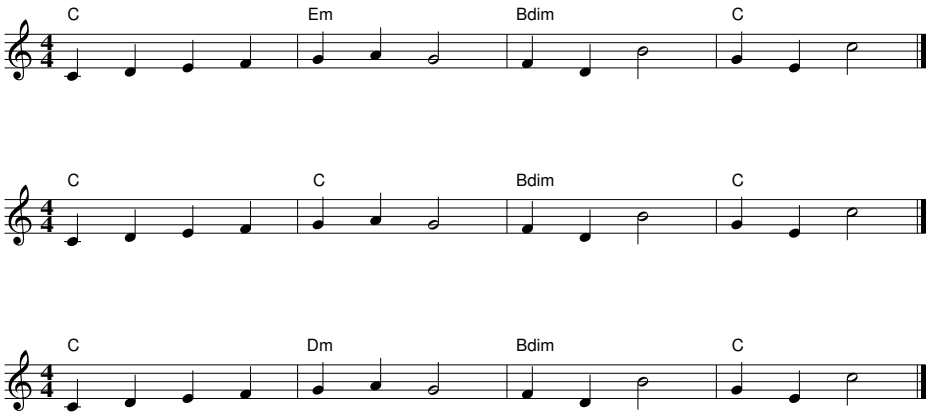

Fig. 11. Three different chord progressions generated for the first four bars of the German folk song "Horch was kommt von draussen rein"

4.2 Finding a suitable melody to a chord progression

The constraints can also be used to, given a chord progression and note lengths, find a melody that harmonizes with the chords. Two extra constraints are added to make the melody more pleasing: the first and last notes should be the tonic, and every bar should contain a unique sequence of pitches[6]. Figure 12 shows the code needed and Figure 13 shows a generated melody which follows the given rules.

```
(def find-melody-example
  (let [chords [[0 2 4] [3 5 7] [4 6 8] [3 5 7]
                [0 2 4] [4 6 8] [3 5 7] [0 2 4]]
        lmelody (lvars (+ 4 8 3 3, 4 8 3 3))
        melody (add-neighbours
                (phrase
                 (concat [1 1 1 1] (repeat 8 1/2) [1 1 2] [1 1 2]
                         [1 1 1 1] (repeat 8 1/2) [1 1 2] [1 1 2])
                 lmelody))
        mel-bars (bars melody)]
    (run 1 [c m]
      (== c chords) (== m melody)
      (== (first lmelody) 0)                ; Start with the tonic.
      (== (last lmelody) 0)                 ; End with the tonic.
      (everyg #(fd/in % domain) lmelody)    ; Constrain the pitches.
      (everyg #(fd/distinct (map :pitch %)) ; Make the melody more interesting by
              mel-bars)                     ; making notes within a bar distinct.
      (everyg accompanyo (zip mel-bars chords)) ; harmonize melody with chords
      )))
```

Fig. 12. Example of finding accompanying chords to a melody

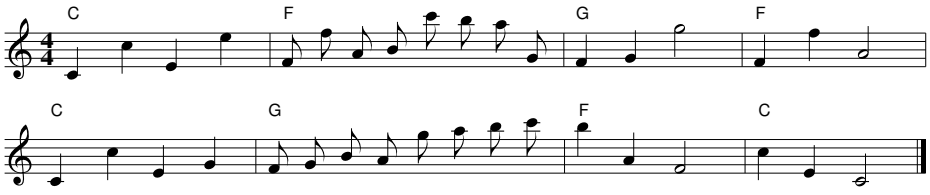

Fig. 13. A melody generated to fit the given chord progression and note lengths

[6] Repeating notes are quite common in music, but the uniqueness constraint was a easy way to get more variation to the generated melody

5 Discussion

The quality of the generated music, as expected with such simplistic rules, isn't great and some of the accompanying chords choices are a bit unusual, for example a G7 would probably be chosen as the third chord by most musicians, but the rules only handle triads and Bdim is basically a G7 without the root. The Dm in the third example is possible since passing notes are allowed anywhere so the first and last note in the second bar are passing notes between the F and A in the Dm chord.

As shown in the results expressing music theories as relations makes for a very adaptable system, it is possible to leave out any part and the logic engine will fill in the blanks. This flexibility would be hard to achieve with non-relational programming - An algorithm for finding harmonizing chords to a melody written in an ordinary procedural language could not easily be used to find a harmonizing melody to given chords.

Constraint programming also makes it possible to define rules in a declarative way, and makes it easy to add new rules without having to change anything else.

There is one drawback though, performance, finding solutions is quite slow – finding the 8 bar melody above takes around 50 seconds on an Intel i5-3570K. Therefore core.logic is probably not suited to generate complete musical pieces in one run, but as long as it is used to generate shorter phrases it could be a useful tool for composing music.

One way to improve the performance could be with custom search strategies optimized for music problems. A good search strategy can improve the performance dramatically [11]. However, unlike Strasheela, core.logic does not yet support configurable search strategies.

6 Future work

Extending core.logic with customizable search strategies would make it more suited both for use in music and many other areas where high performance is needed and domain specific knowledge can be used to improve the search strategy.

Claire Alvis is working on bringing customizable search strategies to cKanren[7], her work could perhaps be used as a starting point.

References

[1] Pachet, F., Roy, P.: Musical harmonization with constraints: A survey. Constraints **6**(1) (2001) 7–19

[2] Anders, T., Miranda, E.R.: Constraint programming systems for modeling music theories and composition. ACM Comput. Surv. **43**(4) (October 2011) 30:1–30:38

[7] `https://groups.google.com/forum/\#!msg/minikanren/zNpXENRA8tc/VIhcJZqNGU0J`, fetched 2013-03-30

[3] Byrd, W.E.: Relational programming in miniKanren | Techniques, applications, and implementations. PhD thesis, Indiana University (2010)
[4] Alvis, C.E., Willcock, J.J., Carter, K.M., Byrd, W.E., Friedman, D.P.: minikanren with constraints. In: Proc. Workshop on Scheme and Functional Programming, Portland, Oregon. (2011)
[5] Byrd, W.E., Friedman, D.P.: αkanren a fresh name in nominal logic programming. In: Proceedings of the 2007 Workshop on Scheme and Functional Programming, Universite Laval Technical Report DIUL-RT-0701, pp. 79-90. (2007)
[6] Byrd, W.E., Holk, E., Friedman, D.P.: minikanren, live and untagged: Quine generation via relational interpreters. Proceedings of the 2012 Workshop on Scheme and Functional Programming (2012)
[7] Loy, D.G.: Musimathics: The mathematical foundations of music, Volume 1. MIT Press, Cambridge, Mass. London (2006)
[8] Hewitt, M.: Harmony for computer musicians. Course Technology PTR/Cengage Learning, Australia United States (2010)
[9] Tsang, C., Aitken, M.: Harmonizing music as a discipline in constraint logic programming. In: Proceedings ICMC'91. (1991)
[10] Hewitt, M.: Music theory for computer musicians. Course Technology, CENGAGE Learning, Boston, MA (2008)
[11] Anders, T.: Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System. PhD thesis, Queen's University (2007)