Functional Generation of Harmony and Melody

José Pedro Magalhães
Department of Computer Science,
University of Oxford
jpm@cs.ox.ac.uk

Hendrik Vincent Koops
Department of Information and Computing Sciences,
Utrecht University
h.v.koops@uu.nl

Abstract

We present FCOMP, a system for automatic generation of harmony and accompanying melody. Building on previous work on functional modelling of musical harmony, FCOMP first creates a foundational harmony by generating random (but user-guided) values of a datatype that encodes the rules of tonal harmony. Then, a melody that fits to the harmony is generated in a compositional sequence: generate all "possible" melodies, filter them to remove obvious bad choices, pick one candidate note per chord, and then embellish the resulting melodic line.

At this very early stage, we aim to define a solid system as a foundation that can be used to further improve upon. We care especially about modularity, so that each individual part of the pipeline can be easily improved, and ease of adaptation, so that users can quickly adapt the generated music to their liking. The resulting system generates simple but harmonious music, and serves as a good case study on how functional programming enables quick and clean prototyping of new ideas, even in the realm of automatic music composition.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming; H.5.5 [*Information Interfaces and Presentation*]: Sound and Music Computing

Keywords Automatic composition, automatic harmonisation, harmony, HarmTrace, Haskell, melody

1. Introduction

> *Composition consists in two things only. The first is the ordering and disposing of several sounds... in such a manner that their succession pleases the ear. This is what the Ancients called melody. The second is the rendering audible of two or more simultaneous sounds in such a manner that their combination is pleasant. This is what we call harmony, and it alone merits the name of composition.*
>
> Jean-Benjamin de La Borde
> *Essai Sur La Musique Ancienne Et Moderne (La Borde 1780)*

Music is an art form with a very long history, predating even literacy. Musical composition, the process of creating new music, is a subject of study for centuries, from D'Arezzo (1026) to Schönberg (1967), to name but two. As any other form of fine art, music embodies human nature, creativity, and aesthetics. Given the artistic nature of music, it is perhaps unsurprising that *automated* music composition, or music composed by algorithmic means, is such a challenging topic. Music is composed of many aspects, all intertwined: melody, harmony, rhythm, form, repetition, instrumentation, tempo, dynamics, etc. Good music considers all these aspects individually, and addresses their combination.

In this paper we present FCOMP, a system that generates chord sequences (harmony) and accompanying melodies. It can be a seen as a simplified, or foundational, automatic music composition system, at least in the historical sense of La Borde (1780). FCOMP (a combination of the words "functional" and "composition") deals only with harmony and melody generation, leaving all other aspects of music unaddressed (at least for now, but see Section 7). FCOMP should thus be seen as a foundational tool; its output is not meant to be music comparable to that composed by humans. Instead, we see it as an exercise in functional modelling of music. It showcases the benefits of using Haskell, a pure, statically-typed functional programming language (Peyton Jones 2003): FCOMP is highly modular, easy to adapt and improve, and uses advanced functional programming techniques (such as indexed types and generic programming) to model the high-level concepts of music theory in a natural and effective way. Haskell's algebraic datatypes behave similarly to context free grammars, which can be used to model the language of well-formed chord sequences. Furthermore, functional composition without explicit mutable state provides a composable way for defining a pipeline of independent processes, making the global algorithm easier to understand and adapt.

Figure 1 shows a diagram of the components of FCOMP which can be easily adapted or exchanged. There are two steps in the harmony generation phase (Section 4), and 4 in the melody generation phase (Section 5). The compositional style of FCOMP makes it easy both to understand and to modify the system, which is ideal for a research and educational tool. Specifically, our contributions are:

- FCOMP, a system for automatic generation of harmony and melody, available online at `http://hackage.haskell.org/package/FComp`;
- Another example of an application of a model of harmony encoded as a Haskell datatype (Magalhães and De Haas 2011);
- A composable pipeline for generating melodies that fit into a given harmony, with each step being easy to modify or improve;
- As a side-effect of working on FCOMP, we have also developed a generic data generation function with constraints (which we use when generating harmony sequences), and made improvements to the `instant-generics` generic programming library, extending previous work (Magalhães and Jeuring 2011) to also support datatypes with indices of kinds other than $\star$.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FARM '14, September 6, 2014, Gothenburg, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3039-8/14/09. . . $15.00.
http://dx.doi.org/10.1145/2633638.2633645

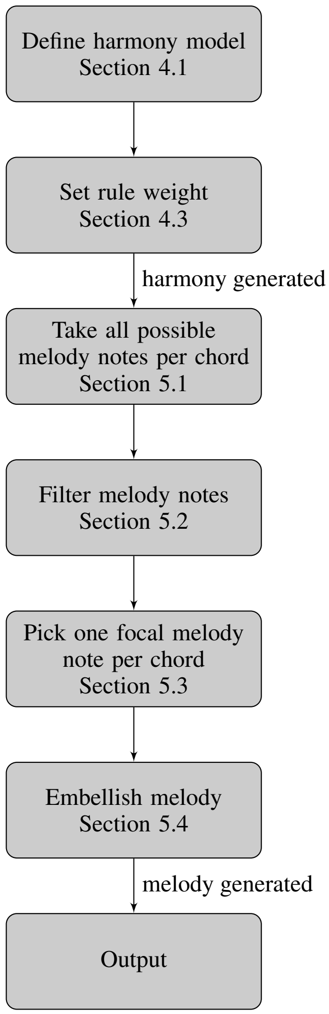

Figure 1. Architecture of FComp

1.1 Notation

We use Greek letters for type variables, apart from κ, which is reserved for kind variables. A colour version of this paper is available from the first author's webpage.

1.2 How to use FComp

FComp is available online at `http://hackage.haskell.org/package/FComp`. After downloading and installing the code, FComp can be executed to generate random songs in MIDI format. They will also be played-back automatically. The interested user can peruse and edit the source code to change the behaviour of the generation process, as documented throughout this paper.

1.3 Roadmap

The rest of this paper is organised as follows. We begin by reviewing related work and positioning FComp in Section 2. Then, we provide a brief introduction to music theory in Section 3, and proceed to describe how FComp generates harmony (Section 4) and accompanying melody (Section 5). We then show and discuss a number of examples in Section 6, list directions for future work in Section 7, and conclude in Section 8.

2. Related work

The field of computer generated music concerns music composed by, or aided by, a computer program. Although large parts of today's popular and electronic music is created with aid of a computer (digital effects, editing, etc), computer generated music refers specifically to music where the composition itself is created by a computer. This section gives a brief overview of historic and current research in this field.

Music, in contrast to other forms of art, has been subjected to rigorous formalisation for a long time, arguably since the invention of the tonal system. Formalisation and algorithms go hand in hand, and composers have therefore been using algorithms to assist in the creation of new music for hundreds of years. It is only natural that composers started to use the computers with these algorithms when they became more common. The *Illiac Suite* (Hiller and Isaacson 1959) is generally agreed to be the first piece of music composed by a computer. Hiller and Isaacson used the Illiac computer at the University of Illinois, Urbana-Champaign in an experiment to program, in binary, algorithms that created a piece in four parts, each exploring different musical ideas. Their work paved the way for composers such as John Cage, Iannis Xenakis, Gottfried Michael Koenig, and many more, who all used computers in various ways to generate music.

In general, there are two types of systems in computer generated music: stochastic and determinate systems. More recent systems are usually of a third type, which is a hybrid of stochastic and determinate techniques.

Stochastic systems Stochastic (or aleatoric) systems use random procedures (for example Markov chains or random number generators) to create a new piece on every run. An early example of a composer using stochastic systems to create music is Iannis Xenakis, a pioneer of *stochastic music*. In his piece *Analogiques*, Xenakis used Markov chains to generate musical content. Markov-based models are interesting because of their ability to generate new musical material in a learned style. Later examples are works by John Cage and Cope (1996). A more recent example of the use of a Markov-based system is the Continuator (Pachet 2003), which is able to learn and generate music in any learned style. One advantage of such an approach is that the system can be used as an instrument: it supports musical interaction and is therefore capable of creating improvisations and continuations of a piece on the fly. These types of systems differ with FComp in that the musical rules to compose new pieces need to be learned from a corpus of example music. The absence of explicit, symbolic music information such as harmonic structure makes these systems harder to interpret and analyse. Systems that do use explicit symbolic rules to create a piece are generally called determinate systems.

Determinate systems Determinate systems are rule-based models that produce the same output every time on every run. Most of the research in this area deals with tonal music, since this type of music has a long history of strict formalisation. Rule-based systems have been used by Ebcioğlu (1988) to create four-part chorales and by Steedman (1984) to create jazz chord sequences, for example. Rohrmeier (2007, 2011) models tonal harmony as an elaborate recursive context-free grammar (CFG). The HarmTrace harmony model (De Haas et al. 2013) implements and extends the ideas of Rohrmeier, but differs from Rohrmeier's grammar in several aspects. Rohrmeier's model is more elaborate, as it includes phrasing and modulating into any key at any point in a sequence. However, from an implementation perspective, this would generate too many ambiguous solutions for a single sequence of chords. Therefore, in HarmTrace, modulation and phrasing are not implemented as context-free rules in the way Rohrmeier formulates them. Another example of rule-based systems are *Lindenmayer systems* (L-systems, Lindenmayer 1968), a parallel rewriting system and a type of formal grammar which is capable of generating a complex hierarchy from a simple input. L-systems are used by Prusinkiewicz (1986), for example, to generate a string of symbols to be interpreted as a sequence of notes.

Hybrid systems Modern techniques in algorithmic music, like FCOMP, are often a hybrid of stochastic and determinate systems. Purely rule-based systems are generally too predictable to create interesting new music, but allow for thorough analysis of the generated output. A purely stochastic system is capable of generating new, unexplored musical ideas and is therefore better capable for modelling creativity, but its output is not easily explained. Hybrid models usually try to combine these two approaches: the deterministic part creates a context in which a musical piece can be explored, and a stochastic part that tries to creatively find interesting patterns within this context. An example of such a hybrid system is that of Koops et al. (2013), which is capable of harmonising a given melody through the aid of a functional model of tonal harmony. The given input melody is analysed and appropriate chords are selected for each note. These chords are then checked by a harmony model, that will select a good sequence to match the melody. Because there is not a single best chord sequence to harmonise a melody, the sequence is chosen randomly from a pool of good candidates. Another example comes from Quick and Hudak (2013), who use a generative grammar in which each rule is associated with a probability for generating pieces similar to classical chorales.

3. A brief introduction to music theory

This section provides a brief introduction to music theory, harmony, melody, and composition topics needed to better understand FCOMP. For a thorough introduction, we refer the reader to Laitz (2008). To ease the understanding of musical concepts for programmers, we accompany our description with illustrative code snippets of how to encode each musical concept in Haskell. Since FCOMP follows a line of previous work (Magalhães and De Haas 2011; De Haas et al. 2013; Koops et al. 2013), this section restates some parts of earlier work adapted for the current context.

Notes Individual sounds with a fixed frequency are represented by a *note* in Western tonal music. Notes are commonly referred to by one of the elements in the list of semitones $[C, C_\sharp \approx D_\flat, D, D_\sharp \approx E_\flat, E, F, F_\sharp \approx G_\flat, G, G_\sharp \approx A_\flat, A, A_\sharp \approx B_\flat, B]$. We can encode this notion of *note* in Haskell as a parametrised datatype:

```
data Note α            = Note Accidental α
data Accidental        = ♮ | ♯ | ♭
data DiatonicNatural   = C | D | E | F | G | A | B
type NoteNatural       = Note DiatonicNatural
```

Notes can be raised or lowered by a semitone, which is denoted by an accidental: the former is denoted by ♯ and the latter is denoted by ♭. An note that is neither raised nor lowered is denoted by ♮. We make Note a parametrised datatype because we will later have notes labelled with datatypes other than DiatonicNatural.

Enharmonic Equivalence Most musical instruments since the 18^{th} century are tuned in equal temperament. In this system of tuning, every pair of adjacent notes has an identical frequency ratio, which results in the perceived distance between an interval being constant for every equal interval in the system, wherever it may appear. In equal temperament, a note with the accidental ♯ is *enharmonic equivalent* to that note one DiatonicNatural higher with the accidental ♭.[1] To given an example, *Note C* ♯ and *Note D* ♭ sound the same, but are "spelled" differently. For simplicity we abbreviate the syntax of notes by writing $C_\sharp$ instead of *Note* ♯ *C*, for example, and C instead of *Note* ♮ *C* (naturals are the default accidental). Enharmonic equivalence is denoted with $D_\sharp \approx E_\flat$.

In musical set theory, *pitch classes* are defined by two equivalence relations. Pitches belong to the same pitch class if they have some relation of compositional or analytical interest, such as the octave relation (Roeder 2013). The second relation is the enharmonic equivalence relation, which means that all the pitches played on the same key of a regular piano keyboard are in the same set. From these two equivalence relations there are just 12 pitch classes, corresponding to the notes of the chromatic scale, often numbered from 0 to 11. The choice of which pitch class to call 0 is a matter of convention; we call C 0 (in which case $C_\sharp \approx D_\flat$ is 1, D is 2, etc.). The function *toSemitone* returns the pitch class of a Note:

```
toSemitone :: (Enum α) ⇒ Note α → Int
toSemitone (Note acc p) = (sem + toPitchClass acc) `mod` 12
  where sem = [0,2,4,5,7,9,11] !! fromEnum p
toPitchClass :: Accidental → Int
toPitchClass ♮ =  0
toPitchClass ♯ =  1
toPitchClass ♭ = −1
```

Similarly, we can define a function *toNote* that returns the note corresponding to a pitch class:

```
toNote :: Int → NoteNatural
toNote i = let roots = [Note ♮ C, Note ♯ C, Note ♮ D
                       ,Note ♭ E, Note ♮ E
                       ,Note ♮ F, Note ♯ F, Note ♮ G
                       ,Note ♭ A, Note ♮ A
                       ,Note ♭ B, Note ♮ B]
           in roots !! (i `mod` 12)
```

Semitone distance	Name
0	unison
1	minor second (semitone)
2	major second
3	minor third
4	major third
5	perfect fourth
6	diatonic tritone
7	perfect fifth
8	minor sixth
9	major sixth
10	minor seventh
11	major seventh
12	octave

Table 1. A list of intervals and their names

Intervals The distance between two notes is called an *interval* and can be either melodic or harmonic. A melodic interval consists of two notes that sound consecutively, whereas a harmonic interval denotes two notes that sound together. Intervals are commonly classified as a combination between their quality and a number. The quality is one of major, minor, or perfect, and the number is one of unison, second, third, etc. Intervals can be seen as the building blocks of a melody, in that a series of (melodic) intervals creates a melodic sequence. Common names for the intervals are shown in Table 1; enharmonic equivalent names have been omitted. A representation of equivalent intervals on a staff can be found in Figure 2.

[1] This does not hold for all note combinations (in particular for B and C, and E and F), but further detail is unnecessary here.

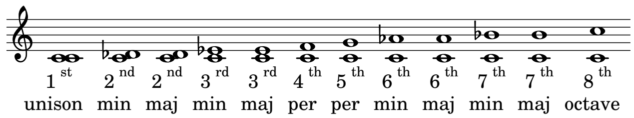

Figure 2. Intervals and their difference in staff positions.

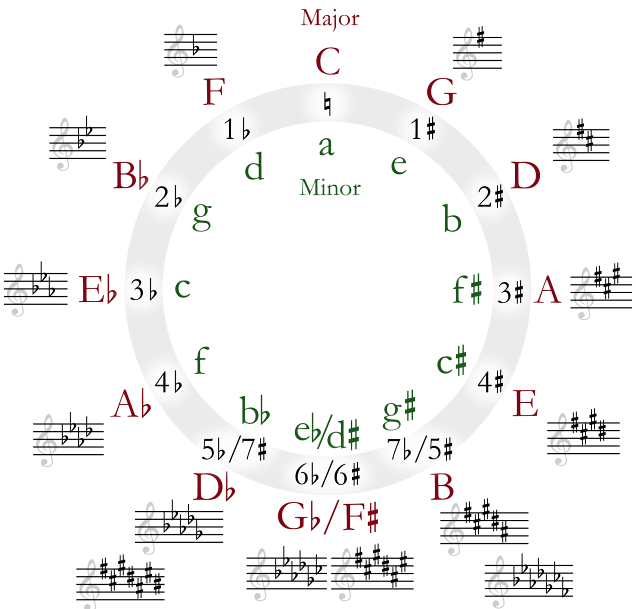

Figure 3. The circle of fifths.[2]

Scales and Scale Degrees A *scale* is a sequence of intervals in a specific ascending or descending order. Scales are divided into categories based on the qualities of intervals they contain, being major, minor, or other. The notes of these scales are denoted by *scale degrees*, and can be be identified by Roman numerals (I, II, III, IV, V, VI, and VII), the first of which is also referred to as the *tonic*. This scale degree representation can be implemented in Haskell as a DiatonicDegree together with the earlier introduced parametrised Note in the following way:

```
data DiatonicDegree = I | II | III | IV | V | VI | VII
type NoteDegree     = Note DiatonicDegree
```

Keys Commonly, the notes of a scale belong to a certain *key*, which defines the context in which the character of a musical piece can be explored. The definition of a key is rather complex and historically not without disagreement, but can be seen as the notes of a scale, its chords, and the use of specific chord progressions. The key is named after the tonic, which is usually the focal point of a piece. A key is defined by the number of accidentals on the scale; Figure 3 shows which keys are associated with which specific accidentals. We can represent keys in Haskell as follows:

```
data Key = Key { keyRoot  :: Note DiatonicNatural
               , keyMode :: Mode }
data Mode = Maj_Mode | Min_Mode
```

Chords and Quality Combining two or more harmonic intervals creates a *chord*. The simplest chords are called *triads*, consisting of one harmonic interval of a third and one harmonic interval of a fifth on a common root note. The three notes of this chord are called root, third, and fifth. The *quality* of the chord is called *minor* if the third is minor, and, similarly, if the third is major, the chord is called *major*. The chord is called *diminished* if the third is minor and the fifth is lowered one semitone. A *dominant seventh* chord is a major chord with an added seventh interval.

Chords can be labelled unambiguously (Harte et al. 2005) by giving the following parts:

1. The chord root, which is either an absolute note like a *C* or a scale degree;
2. The quality of the chord, for example major or minor; and
3. Optional added or removed intervals, for example a seventh. For simplicity we will not use these added intervals in our implementation.

We encode this simplified labelling of chords in Haskell as follows:

```
data Chord α = Chord { chordRoot    :: Note α
                     , chordQuality :: Quality }
data Quality  = Maj | Min | Dom7 | Dim
```

This datatype represents a chord built from a *chordRoot*, which can be encoded as a NoteNatural or a NoteDegree. This way we can represent chords built from an absolute root note, as well as chords built from scale degrees. Quality defines if the chord is major, minor, dominant, or diminished; we limit ourselves to these four choices for simplicity. As with Notes, we define convenience type synonyms for chords encoded with degrees or labels:

```
type ChordDegree  = Chord DiatonicDegree
type ChordNatural = Chord DiatonicNatural
```

Table 2 shows all the scale degrees and their corresponding chords in major and minor keys. We use $r{:}q$ as a syntactic shorthand for *Chord r q*.

Harmony *Functional harmony* (Whittall 2013) is a theory of tonal harmony by Riemann (1893) that describes the common harmony practice from the 18^{th} until the 20^{th} century. The functional harmony theory states that each chord within a key can be reduced to one of three harmonic functions—tonic, dominant, or subdominant. The tonic affirms the key, the subdominant builds tension, and the dominant builds maximum tension. These rules are expressed in the grammar of FComp (see Section 4).

Composition The process of composing an original piece of music differs greatly between styles and historical periods. In very broad terms it compasses the structuring and ordering of sounds, but this can be done in an infinite number of interesting ways.

Although far from being a solved problem, the kind of western tonal music that this research is concerned with generally uses rather specific rules to create a musical piece. These rules prescribe a system in which harmony and melody are manipulated within the boundaries of a chosen key and scale. An example of such a rule is that a piece should always end in a *cadence*, which is a specific sequence of (at least two) chords that create a sense of resolution and indicates whether the piece is to continue or has concluded. In our case, we use a compositional technique in which chords act as building blocks for a melody. By creating a sequence of chords (a *harmony*) generated from a harmony model, we create

[2] Image taken from `http://en.wikipedia.org/wiki/File:Circle_of_fifths_deluxe_4.svg`.

Scale degree	*I*	*II*	*III*	*IV*	*V*	*VI*	*VII*
Major key	*I*:*Maj*	*II*:*Min*	*III*:*Min*	*IV*:*Maj*	*V*:*Maj*	*VI*:*Min*	*VII*:*Dim*
Minor key	*I*:*Min*	*II*:*Dim*	*III*:*Maj*	*IV*:*Min*	*V*:*Min*	*VI*:*Maj*	*VII*:*Dim*

Table 2. Scale degrees and chords arising in major and minor keys.

a restriction on how we can build a *melody*. A correct sequence of chords is relatively easy to generate, but creating a pleasant melody is much harder. Good melodies usually have easily discernible recurring patterns and events at several temporal levels. Typically, a subsequently altered, repeated, or sequenced succession of notes throughout a musical piece can be found in a melody, something which is generally considered to be the sign of a great composer, if done in an interesting and appealing way.

4. Generating harmony

Our system is concerned with the generation of two basic ingredients of tonal music: *harmony* and *melody*. These two elements are intertwined; a specific harmony sequence restricts the freedom in generating an accompanying melody, and a standalone melody often induces certain harmony progressions. In FCOMP, we begin by generating a harmony sequence, and then create a melody that fits the harmony. We choose this approach as a matter of convenience; while harmony often follows strict rules that are amenable to hierarchical modelling, rules for writing "correct" melodies are more subtle and hard to specify formally. As such, we use the harmony to restrict the freedom of choice in the melody. This section deals with the problem of generating valid harmony sequences; Section 5 looks at the generation of fitting melodies.

4.1 Representing harmony structure hierarchically

Like De Haas et al. (2013) in their HARMTRACE system, we use a model of tonal harmony as a family of Haskell datatypes. However, our model is significantly different from those in HARMTRACE. Since our main concern is harmony *generation*, and not *recognition*, we do not have to worry much about ambiguity in the model, for example. Furthermore, we encode only very basic harmony rules, eliding the complexity of chains of secondary dominants, tritone substitutions, etc; even a very simple harmony allows for the creation of musically-interesting melodies, so we do not need extra complexity at this stage. Fortunately, the design of HARMTRACE makes it easy to define new models, and we can reuse lots of code for our simplified model due to the use of generic programming techniques (Magalhães and De Haas 2011).

For ease of presentation, we show the model as a parametrised context-free grammar. In reality, the model consists simply of Haskell generalised algebraic datatypes (GADTs, Schrijvers et al. 2009); the translation from this notation to actual GADTs is straightforward, as shown in Section 4.2.

In the rules below, we use the variable $\mathfrak{M} \in \{\mathrm{Maj}, \mathrm{Min}\}$ when the rule is applicable both to pieces in minor and major modes. Superscripts denote chord quality: a major chord (no superscript), minor (*m*), dominant seventh (7), and diminished (0). Pieces consist of sequences (lists) of phrases. A phrase can either be in tonic-dominant-tonic form, or dominant-tonic:

1 $Piece_{\mathfrak{M}} \rightarrow [Phrase_{\mathfrak{M}}]$
2 $Phrase_{\mathfrak{M}} \rightarrow Ton_{\mathfrak{M}}\ Dom_{\mathfrak{M}}\ Ton_{\mathfrak{M}}$
3 $\qquad\qquad \mid \quad Dom_{\mathfrak{M}}\ Ton_{\mathfrak{M}}$

The tonic consists only of the *I* chord, in major or minor depending on the mode of the piece:

4 $Ton_{\mathrm{Maj}} \rightarrow I_{\mathrm{Maj}}$
5 $Ton_{\mathrm{Min}} \rightarrow I^{m}_{\mathrm{Min}}$

We allow more freedom in the dominant. A dominant can either expand to a dominant or major chord built on the fifth scale degree, to a diminished chord built on the seventh, be preceded by a subdominant, or even be prepared by a secondary dominant (a dominant chord on the second scale degree):

6 $Dom_{\mathfrak{M}} \rightarrow V^{7}_{\mathfrak{M}}$
7 $\qquad \mid V_{\mathfrak{M}}$
8 $\qquad \mid VII^{0}_{\mathfrak{M}}$
9 $\qquad \mid Sub_{\mathfrak{M}}\ Dom_{\mathfrak{M}}$
10 $\qquad \mid II^{7}_{\mathfrak{M}}\ V^{7}_{\mathfrak{M}}$

The subdominant, in major mode, can either be realised by a *II*:Min chord or a *IV*:Maj, this one optionally preceded by a *III*:Min. In minor mode, for simplicity, we build a subdominant only from a *IV*:Min chord:

11 $Sub_{\mathrm{Maj}} \rightarrow II^{m}_{\mathrm{Maj}}$
12 $\qquad \mid IV_{\mathrm{Maj}}$
13 $\qquad \mid III^{m}_{\mathrm{Maj}}\ IV_{\mathrm{Maj}}$
14 $Sub_{\mathrm{Min}} \rightarrow IV^{m}_{\mathrm{Min}}$

Finally, scale degrees map to actual ChordNaturals, when given a specific key. We show only a few of these rules as an example (choosing *C* major as key):

15 $I_{\mathrm{Maj}} \rightarrow C{:}Maj$
16 $I^{m}_{\mathrm{Min}} \rightarrow C{:}Min$
17 $V^{7}_{\mathfrak{M}} \rightarrow G{:}Dom^{7}$
18 $VII^{0}_{\mathfrak{M}} \rightarrow B{:}Dim$

We have shown a deliberately simplified model of harmony, which will suffice for our purposes of generation of simple melodies. However, due to our use of generic programming techniques, the model is very easy to extend. All that is necessary is to add or remove rules; the rest of the code adapts automatically to the new rules.

4.2 Concrete representation as GADTs

The rules of the previous section are implemented as Haskell GADTs. We show the encoding of a piece and phrases (specifications 1–3):

```
data Piece = ∀μ :: Mode.Piece [Phrase μ]
data Phrase (μ :: Mode) where
  Phrase_IVI :: Ton μ → Dom μ → Ton μ → Phrase μ
  Phrase_VI  ::          Dom μ → Ton μ → Phrase μ
```

Each of the constructors of datatypes corresponds to one specification. The type index μ is used to keep track of which rules are applicable in major or minor mode. For convenience, it is existentially-quantified at the Piece level. We use datatype promotion (Yorgey et al. 2012) so that we can reuse the constructors introduced in Section 3 as types; this is not essential, but removes code duplication, and makes the datatypes more correct, as their indices cannot be instantiated with types of the wrong kinds.

Tonics and dominants (specifications 4–10) are encoded as follows:

```
data Ton (μ :: Mode) where
  Ton_Maj :: SD 'I 'Maj → Ton 'Maj_Mode
```

$$Ton_{Min} :: \mathsf{SD}\ `I\ `Min \rightarrow \mathsf{Ton}\ `Min_{Mode}$$

$$\textbf{data}\ \mathsf{Dom}\ (\mu :: \mathsf{Mode})\ \textbf{where}$$
$$Dom_1 :: \mathsf{SD}\ `V\ \ `Dom^7 \rightarrow \mathsf{Dom}\ \mu$$
$$Dom_2 :: \mathsf{SD}\ `V\ \ `Maj\ \ \rightarrow \mathsf{Dom}\ \mu$$
$$Dom_3 :: \mathsf{SD}\ `VII\ `Dim\ \ \rightarrow \mathsf{Dom}\ \mu$$
$$Dom_4 :: \mathsf{SDom}\ \mu \rightarrow \mathsf{Dom}\ \mu \rightarrow \mathsf{Dom}\ \mu$$
$$Dom_5 :: \mathsf{SD}\ `II\ `Dom^7 \rightarrow \mathsf{SD}\ `V\ `Dom^7 \rightarrow \mathsf{Dom}\ \mu$$

In the constructors of Ton, we can see that Ton_{Maj}, corresponding to specification 4, is only available in the major mode, while Ton_{Min} is only available in minor mode. In contrast, all the constructors of Dom are mode-agnostic.

Scale degrees are at the bottom of the hierarchy, and simply contain a ChordDegree (as defined in Section 3):

$$\textbf{data}\ \mathsf{SD}\ (\delta :: \mathsf{DiatonicDegree})\ (\gamma :: \mathsf{Quality})\ \textbf{where}$$
$$SurfaceChord :: \mathsf{ChordDegree} \rightarrow \mathsf{SD}\ \delta\ \gamma$$

Scale degrees are indexed at the type level over their degree and quality. The constructor *SurfaceChord* ignores the indices, but all chords are generated in a type-based fashion that ensures that in a $SurfaceChord\ c :: \mathsf{SD}\ \delta\ \gamma$, the c has the appropriate degree and quality (see Section 4.3).

4.3 Generic data generation with constraints

Having a model of harmony as a family of Haskell datatypes, the task of generating possible harmony sequences comes down to generating random values. Furthermore, because our datatypes use indices to keep track of which scale degrees are allowed where, we can guarantee that the generated chord sequences will be harmonically correct according to the model: we just have to guarantee that *SurfaceChord*s we produce have a degree and quality matching those given at the type level. As such, there is no need to filter generated sequences for validity.

However, not all harmonically *valid* chord sequences are musically *interesting*. For our purposes, we prefer longer sequences, with a richer harmonic structure. If we devise a truly random data generator, it is likely that many sequences will be simple $V{:}Maj$-$I{:}Maj$ sequences, for example. We would like to have some control over which rules should appear more or less frequently, in order to be able to direct the generation into specific choices, while keeping an element of randomness. At the same time, we want to keep our data generation code *generic*. Since FCOMP is designed to be easily adaptable, and changing the model means changing datatype definitions, we want to have as little type-specific code as possible, so that fewer code changes need to be made for every change to the model.

Our solution is a generic data generation program that can be parametrised over weights for each constructor. Generic data generation is a relatively straightforward task; adding constructor weights makes it slightly more involved. The details of this generic program are outside the scope of this paper; in this section we'll show simply how it can be used to produce chord sequences. The interface to the generator is a function *gen* which produces values in QuickCheck's Gen monad (Claessen and Hughes 2000) according to some user-provided FrequencyTable, which is a list of constructor names together with their desired weight:

$$\textbf{type}\ \mathsf{FrequencyTable} = [\,(\mathsf{String},\mathsf{Int})\,]$$
$$gen\ ::\ (\mathsf{Representable}\ \alpha, \mathsf{Generate}\ (\mathsf{Rep}\ \alpha)) \Rightarrow \mathsf{FrequencyTable} \rightarrow \mathsf{Gen}\ \alpha$$

The Representable instances are required by the generic programming library we use, and are obtained using Template Haskell (Sheard and Peyton Jones 2002) with no added complexity for the user. The Generate type class implements the generic function. The only non-generic part of our generator is the case for SD, where we enforce that the generated *SurfaceChord* has the degree and quality given by the type-level indices:

$$gen_{SD} :: \mathsf{Gen}\ (\mathsf{SD}\ \delta\ \gamma)$$
$$gen_{SD} = return \circ SurfaceChord\ \$\ Chord\ (Note\ \natural\ d)\ q$$
$$\quad \textbf{where}\ d = toDegree\ \ (Proxy :: \mathsf{Proxy}\ \delta)$$
$$\qquad\quad q = toQuality\ (Proxy :: \mathsf{Proxy}\ \gamma)$$
$$\textbf{data}\ \mathsf{Proxy}\ (\alpha :: \kappa) = Proxy$$

The classes ToDegree and ToQuality perform a type-to-value mapping, ensuring we build a chord with degree and quality matching the indices:

$$\textbf{class}\ \mathsf{ToDegree}\ (\delta :: \mathsf{DiatonicDegree})\ \textbf{where}$$
$$toDegree :: \mathsf{Proxy}\ \delta \rightarrow \mathsf{DiatonicDegree}$$
$$\textbf{instance}\ \mathsf{ToDegree}\ `I\ \textbf{where}$$
$$toDegree\ \_ = I$$
...

$$\textbf{class}\ \mathsf{ToQuality}\ (\gamma :: \mathsf{Quality})\ \textbf{where}$$
$$toQuality :: \mathsf{Proxy}\ \gamma \rightarrow \mathsf{Quality}$$
$$\textbf{instance}\ \mathsf{ToQuality}\ `Maj\ \textbf{where}$$
$$toQuality\ \_ = Maj$$
...

This is a typical way of handling pseudo-dependently typed programming with singleton values in Haskell (Eisenberg and Weirich 2012).

4.4 Examples

We can now show an example of actual harmony generation. Below we build a generator that favours the production of Dom_4 and Dom_5 constructors. Omitted constructors are given a default weight of 1. We use a function *printOnKey* to display the generated chords (transforming the Chord NoteDegree of the model into Chord NoteNatural):

$$testGen :: \mathsf{Gen}\ (\mathsf{Phrase}\ `Maj_{Mode})$$
$$testGen = gen\ [\,(\texttt{"Dom4"},3),(\texttt{"Dom5"},4)\,]$$
$$example :: \mathsf{IO}\ ()$$
$$example = \textbf{let}\ k = Key\ (Note\ \natural\ C)\ Maj_{Mode}\ \textbf{in}\ sample'\ testGen \gg\!= mapM\_\ (printOnKey\ k)$$
$$printOnKey :: \mathsf{Key} \rightarrow \mathsf{Phrase}\ `Maj_{Mode} \rightarrow \mathsf{IO}\ \mathsf{String}$$

We can now observe a sample of generated chords in an interactive compiler session:

$$>example$$
$$[C{:}Maj, D{:}Dom^7, G{:}Dom^7, C{:}Maj]$$
$$[C{:}Maj, G{:}Dom^7, C{:}Maj]$$
$$[C{:}Maj, E{:}Min, F{:}Maj, G{:}Maj, C{:}Maj]$$
$$[C{:}Maj, E{:}Min, F{:}Maj, D{:}Dom^7, G{:}Dom^7, C{:}Maj]$$
$$[C{:}Maj, D{:}Min, E{:}Min, F{:}Maj, D{:}Dom^7, G{:}Dom^7, C{:}Maj]$$

These chords can also be seen on a staff in Figure 4. We can observe that the Dom_4 constructor was used in the last three values, and Dom_5 in the last two, as expected from the weights given.

This example serves only as a simple demonstration of the power of our generic generator of chord sequences. Not only is the harmony model used easy to adapt, it is also easy to guide the generation into specific harmony rules. Since we support diverse harmony models, we could model different styles of harmony, if we wanted to generate jazz style music, or Bach chorales. Adding or changing a model requires recompilation, but adapting the weights for a specific model does not. Carefully chosen weights can also be used to forbid entirely certain rules (by assigning a weight of zero),

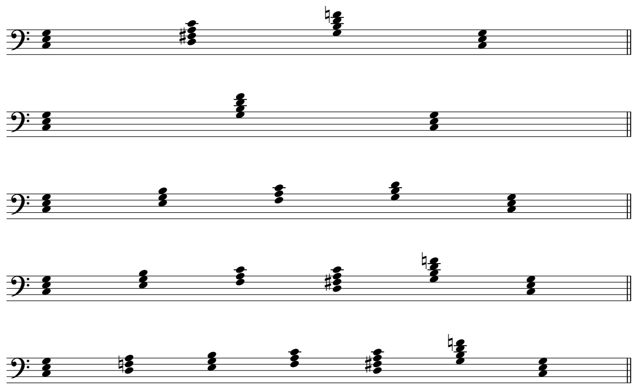

Figure 4. An example of generated chords.

or make others certain (by assigning a very large weight). This type of freedom allows for *creative* experimentation in music generation, and from a rather musical point of view; we are not adjusting computational parameters, we are adjusting musical rules! We expect this to make FCOMP easier to use by musicology experts, who are more familiar with music theory than with programming languages.

5. Generating melody

Having generated a harmonic basis, we can proceed to generating a melody that fits into the harmony. FCOMP achieves this in 4 steps:

1. Generate a list of candidate melody notes per chord;
2. Refine the candidates by filtering out obviously bad candidates;
3. Pick one focal candidate melody note per chord;
4. Embellish the candidate notes to produce a final melody.

These four steps combine naturally using plain monadic bind:

```
generateMelody :: Key → State MyState Song
generateMelody k =
        genCandidates >>= refine >>= pickOne >>= embellish
    >>= return ∘ Song k
```

We describe each of these steps in a subsection of its own.

5.1 Candidate melody notes per chord

In order to generate a melody, we first need to define the concept of song, which is a list of chords paired with melody notes, in a specific key:

```
data Song = Song Key [(ChordNatural, [MelodyNote])]
```

Up until this stage we haven't had to consider notes in different octave positions. However, when producing a melody, the octave of each note matters, as we want to avoid the discontinuities that would be caused by having all the notes in the same octave (number 3). As such, we introduce the type of MelodyNote, which is simply a NoteNatural together with its octave:

```
data MelodyNote = MelodyNote {mnRoot    :: NoteNatural
                             ,mnOctave  :: Octave}
type Octave = Int
```

The process of generating melodies will frequently require a source of randomness, and may need to look up the original chords and indeed even their harmony relationships. As such, we embed the whole melody generation code in a state monad, keeping a number of relevant information in the state:

```
data MyState = MyState {genState    :: StdGen
                       ,keyState    :: Key
                       ,pieceState  :: Piece
                       ,chordsState :: [ChordDegree]}
```

With this infrastructure in place, we are ready to proceed to the first step, which consists of generating a list of candidate melody notes per chord. In order to favour consonance, and keeping with simplicity, FCOMP only considers the notes of the chord to be initial candidates. This is a simple enumeration of the notes in the generated chords, followed by a trivial embedding into MelodyNote. We begin by assigning all notes to the same octave, and look into continuity problems later. The type of the function that performs this generation step, *genCandidates*, is shown here; its code is elided as it is not particularly insightful:

```
genCandidates :: State MyState [(ChordNatural, [MelodyNote])]
genCandidates = ...
```

5.2 Filter initial candidates

In the second phase, we reduce the number of candidate melody notes by filtering out some undesirable notes. As an example, we show how to ensure that:

- The first melody note is always one of *I*, *III*, or *V*;
- The last melody note is *I*, if *I* is in the final chord, or *V* otherwise.

These rules help enforce basic melody writing principles. With the first rule, we make sure that the first melody note helps reinforce the key of the piece, by choosing one of *I*, *III*, or *V*, the notes in the tonic chord. The second rule guarantees that the melody feels stable at the end, by picking the root note of the key whenever possible, or the fifth scale degree otherwise (as then we will be in a half cadence).

The Haskell code to perform this filtering is shown below. To handle the first melody note, it builds the notes of the tonic chord (*first*), and then computes the intersection of this list with the current candidates for the first note (*firstNotes*). The last note is handled by *final*:

```
refine :: [(ChordNatural, [MelodyNote])]
       → State MyState [(ChordNatural, [MelodyNote])]
refine ((cl, mns) : cs) =
  do k ← gets keyState
     let indices    = case keyMode k of
                        Maj_Mode → [0,4,7]
                        Min_Mode → [0,3,7]
         first      = map (makeNote k) indices
         firstNotes = let wanted = first `intersect` mns
                      in if null wanted then mns else wanted
         lastNote ns = let (a, [b]) = splitAt (length ns − 1) ns
                       in a ++ [final b]
         final (c, n) = let n′ = if makeNote k 0 ∈ n
                                  then [makeNote k 0]
                                  else [makeNote k 7]
                        in (c, n′)
     return $ ((cl, firstNotes) : lastNote cs)

makeNote :: Key → Int → MelodyNote
makeNote k i = let ki = toSemitone (keyRoot k)
               in MelodyNote (toNote (i + ki)) 3
```

5.3 Pick one focal candidate per chord

The third step of our algorithm chooses one candidate per chord out of the list of candidates under consideration so far. We pick a note randomly (function *choose*), but we also ensure that sequences such as *VII-I* and *I-VII* are done with the interval of a second, and not a seventh. Since we know that, up to this stage, all melody notes were on octave number 3, this just requires an appropriate transposition by an octave up or down (function *resolve*):

```
pickOne :: [(ChordNatural,[MelodyNote])]
        → State MyState [(ChordNatural,MelodyNote)]
pickOne cs =
  do s ← get
     let g      = genState s
         rs     = randoms g
         k      = keyState s
         result = map choose (zip cs rs)
         choose ((cl,mns),r) = (cl,mns !! (r `mod` length mns))
         resolve ((c1,n1) : (c2,n2) : cns)
           | n1 ≡ makeNote k 0 ∧ n2 ≡ makeNote k 11
           = (c1,n1) : (c2,octaveDown n2) : resolve cns
           | n1 ≡ makeNote k 11 ∧ n2 ≡ makeNote k 0
           = (c1,n1) : (c2,octaveUp n2) : resolve cns
           | otherwise = (c1,n1) : resolve ((c2,n2) : cns)
     return (resolve result)

octaveDown, octaveUp :: MelodyNote → MelodyNote
octaveDown (MelodyNote r n) = MelodyNote r (n − 1)
octaveUp   (MelodyNote r n) = MelodyNote r (n + 1)
```

5.4 Embellish

The last step of melody generation in FComp is to embellish the notes chosen for each chord. From the previous step we get one MelodyNote per ChordNatural; in this step, we return a list of MelodyNotes per chord, as we might want to have multiple melody notes per chord. Unlike previously, this list does not represent a set of candidates; now it represents a linear sequence of notes.

Embellishing a melody is a process of creativity and invention. The possibilities are limitless; in fact, of the four steps of generating a melody in FComp, we suspect this last one to be the most complex and important. For now, we present only two simple forms of embellishment:

- If two consecutive melody notes are the same, we randomly pick a small melodic variation between the two notes;
- Otherwise, connect two consecutive melody notes with a melodic line taken from a scale.

Since our embellishment techniques always look at two consecutive notes, we first perform a traversal of the chords and call *connectNotes* with every two consecutive notes (also passing a fresh StdGen):

```
embellish :: [(ChordNatural,MelodyNote)]
          → State MyState [(ChordNatural,[MelodyNote])]
embellish ((c,mn) : cs) = do g ← gets genState
                             k ← gets keyState
                             return $ go k (c,mn,g) cs
  where
    go k (c1,n1,_) []           = [(c1,[n1])]
    go k (c1,n1,g) ((c2,n2) : cs) =
      let (_,g') = next g
      in    (c1,connectNotes g k c1 n1 n2)
          : go k (c2,n2,g') cs
```

The function *connectNotes* is the main workhorse of this stage. It takes a generator, the key, current chord and melody note, the next melody note, and it returns a list of notes representing the new melodic line for this chord. In case the two notes are the same (we will use the C note as the reference for our example), we choose uniformly between one of four embellishment choices:

1. Do nothing, leaving the repetition unchanged;
2. Transform *C-C* into *C-D-E-C*;
3. Transform *C-C* into *C-B-C*;
4. Transform *C-C* into *C-E-D-C*.

The Haskell code responsible for this embellishment is shown here:

```
connectNotes :: StdGen → Key → ChordNatural
             → MelodyNote → MelodyNote → [MelodyNote]
connectNotes g k c n1 n2 | n1 ≡ n2 =
  let scale            = scaleFromChord c
      inScale line     = if n1 ∈ scale then line else [n1]
      f123−132 [c,d,e] = [c,e,d]
  in case fst (randomR (0,3 :: Int) g) of
      0 → [n1]
      1 → inScale ∘ take 3 ∘ dropWhile (≢ n1) $         scale
      2 → inScale ∘ take 2 ∘ dropWhile (≢ n1) $ reverse scale
      3 → inScale ∘ f123−132 ∘ take 3 ∘ dropWhile (≢ n1) $ scale
```

When two consecutive notes are different, we connect them by using notes taken from a scale on the current key. This makes essential use of the Ord instance for MelodyNote:

```
connectNotes g k c n1 n2 =
  let scale = scaleFromKey k
      line  = if n1 < n2
              then takeWhile (<n2) ∘ dropWhile (⩽ n1)
                     $ scale
              else  takeWhile (n2<) ∘ dropWhile (n1 ⩽)
                     $ reverse scale
  in n1 : line

scaleFromChord :: ChordNatural → [MelodyNote]
scaleFromChord = ...

scaleFromKey :: Key → [MelodyNote]
scaleFromKey = ...
```

The consequence of using this strategy when embellishing two non-adjacent notes is that we will have very continuous melodies. While discontinuity has to be used carefully in a melody, total absence of discontinuity is not common. We discuss how to further improve our melodies in Section 7.

6. Examples

Having seen the internals of FComp, we are ready to show some sample results. This section analyses three pieces generated by FComp, describing their harmony and melody. We show two pieces in major mode (Section 6.1 and Section 6.2) and one piece in minor mode (Section 6.3), all of them in a different key. In Section 6.4 we discuss common properties of the generated songs. Since FComp does not currently assign any specific rhythmic values, we abstract away from rhythm and meter in our rendering of the scores.

6.1 Piece 1

The first generated piece, shown in Figure 5, is in the key of C major. A sequence of five chords is generated with a melody containing the notes C, D, E, F, and G.

Harmony The chord sequence opens with a $C{:}Maj$ major chord, followed by $E{:}Min$, $F{:}Maj$, $D{:}Dom^7$, $G{:}Dom^7$, ending in $C{:}Maj$. In scale degree notation this progression is $I{:}Maj$-$III{:}Min$-$IV{:}Maj$-$II{:}Dom^7$-$V{:}Dom^7$-$I{:}Maj$.

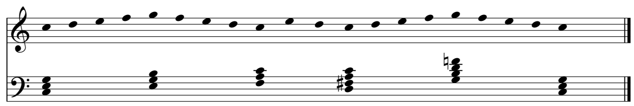

Figure 5. Piece 1 in *C* major.

The sequence opens with the *C*:*Maj* because of a constraint in the grammar, as can be seen in specification 4. Pieces consist of sequences of phrases, and a phrase either starts with the tonic or dominant. If it starts with the tonic, as it does in this case, it consists only of the *I* chord, which is a *C*:*Maj* in the case of the key of *C* major. In Figure 6 we show the harmony tree corresponding to the generated sequence of chords. We label each node with the name of the rule subscripted with the specification number (from Section 4).

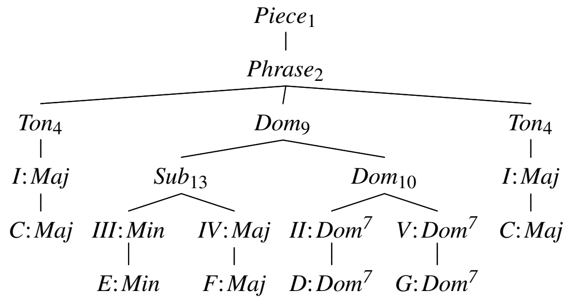

Figure 6. The harmony tree for Piece 1.

The final three chords form a classic sequence: the $II{:}Dom^7$-$V{:}Dom^7$-$I{:}Maj$ progression. This is a widely used sequence, most notably in jazz harmony, but also as a closing part of a sequence called a *cadence*. The chords of the progression successively descend in intervals of a fifth; this establishes tonality, but in an harmonic interesting way by introducing chromaticism (adding non-key notes).

Melody The melody consists of twenty notes spanning from *C* to *G*. In this example, a repetition is created: the melody between chord one to three is repeated between chord three and six. Because each of them is harmonised with different chords, they have their own character. From the first chord to the second the notes are connected through a series of ascending notes. The same pattern, appears in reverse from the second to the third chord. In between chord three and four, a jump of a third is created due to the use of the fourth rule of the *embellish* function as described in Section 5.4. The restatement of an embellishment in the melody shows that FComp is capable of generating a piece with repetition, something which is considered to be important in musical form.

6.2 Piece 2

This second piece, shown in Figure 7, is in the key of *G* major. It consists of a sequence of seven chords with a melody containing the notes between a *C* and a *G* an octave higher.

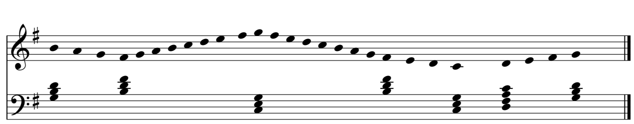

Figure 7. Piece 2 in *G* major.

Harmony The harmony sequence of this piece, just like the previous example, opens with a *I* chord, which is a *G*:*Maj* chord in the key of *G* major. After the *G*:*Maj* chord, a sequence of *A*:*Min*, *C*:*Maj*, *A*:*Min*, *C*:*Maj*, *D*:*Maj* follows, before ending at a *G*:*Maj* chord. In scale degree notation this progression is *I*:*Maj*-*II*:*Min*-*IV*:*Maj*-*II*:*Min*-*IV*:*Maj*-*V*:*Maj*-*I*:*Maj*. Just like in the last example, *Phrase* is expanded to *Ton Dom Ton*; the entire harmony tree corresponding to the generated sequence of chords can be found in Figure 8. It is a sequence of repeating *II*-*IV* before ending in a perfect cadence, which is the most direct means of establishing the tonic, and the end of a piece. This cadence is strengthened by the preceding *IV*, creating a very strong sense of conclusion.

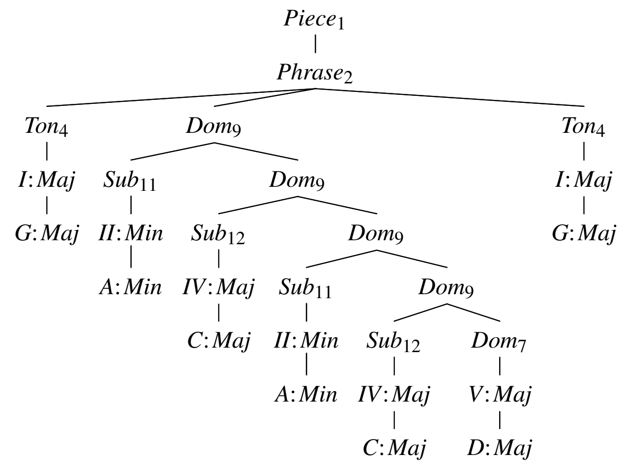

Figure 8. The harmony tree for Piece 2.

Melody This melody is rather continuous, without any jumps. This is because none of the chord melody notes are the same. In this case, the embellishment function connects two notes with a melodic line taken from a scale. This creates a sense of an ending of a larger piece, for example the final line of a concerto in which the musician can affirm the key in a final virtuous line.

6.3 Piece 3

The third and last piece is in the key of *E* minor and can be found in Figure 9. A sequence of six chords is generated with a melody spanning six notes, from *C* until *A*.

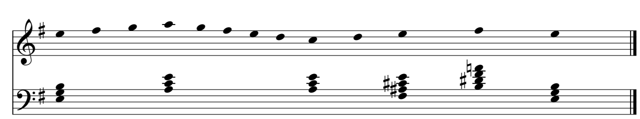

Figure 9. Piece 3 in *E* minor.

Harmony This harmony sequence begins with a *E*:*Min* chord, and is followed by a sequence of *A*:*Min*, *A*:*Min*, $F_\sharp{:}Dom^7$, $B{:}Dom^7$, ending with a *E*:*Min* chord. In scale degree notation, this corresponds to $I{:}Min$-$IV{:}Min$-$IV{:}Min$-$II{:}Dom^7$-$V{:}Dom^7$-$I{:}Min$. The harmony tree of this sequence can be found in Figure 10. The ending of this sequence contains same type of cadence as in piece 1, but this time in minor mode. Just like in major, this sequence affirms the key in a strong way, and is considered a good way to close a musical piece.

Melody Writing melodies in a minor key is slightly trickier than in a major key. Without going into much detail, in a minor key, if a melody is ascending, the sixth and seventh note are commonly raised by one semitone for aesthetic reasons. FComp currently does not take this into account, which means the *C* and *D* of the third chord are unaltered. Fortunately, in this case, that does not

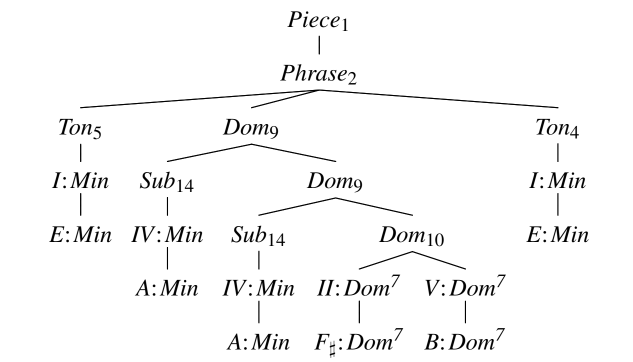

Figure 10. The harmony tree for Piece 3.

create an unpleasant melody. Just like in the previous example, the melody is continuous, with scales connecting the chords. The third to last note, *E*, is the seventh of the chord $F_\sharp$:*Maj*, and as such should resolve downwards to $D_\sharp$, but it doesn't. In a future version of FCOMP, dominant sevenths should be resolved properly, as the melodic minor scale should be taken into account.

6.4 Discussion

FCOMP is a first step towards a fully functional automatic composition system. These three examples show that FCOMP is capable of generating well-formed chord sequences with melodies that are musically valid in a context of western tonal music. Although the melodies generated at this stage are simple, the examples show that with simple rules and a firm harmonic foundation humble but pleasant melodies can be created. The examples also show room for improvement on several stages of the composition process. The next section discusses some of these possible future work directions.

7. Future work

As mentioned before, generation of music to a convincing standard is a very hard task, and we cannot expect to solve it with a simple system. As such, we see FCOMP as a first step, a foundation for future work on music generation using Haskell. We now explore some of the many possible future work directions for FCOMP.

7.1 Voice leading and counterpoint

While FCOMP generates well-formed harmony sequences, it does not deal with the performance of the harmony. Currently, chords are simply output in root position, with all voices in total homophony. The problem of distributing the notes of a chord into separate melodic lines is known as "voice leading". This is an area of active research in computational musicology (see, for example, Allan and Williams (2005)). A natural extension of FCOMP is to incorporate voice leading. This also opens the door to introduce counterpoint (i.e. a relationship between independent melodic lines). Counterpoint is a well studied and formalised aspect of music, which we believe makes its implementation easier to grasp.

7.2 Handle repetition

Repetition plays a crucial role in music (Margulis 2014). Melodies often contain a hierarchy of repeated themes and rhythmic patterns, from small melody parts repeated in different ways, to longer phrases appearing in a variation or transposition, to structure and form at the highest level, such the sonata form of exposition, development, and recapitulation. Currently FCOMP has no explicit knowledge of repetition; while repeated harmony sequences may arise naturally (like in Figure 7), the melody will only include repetition by chance. This hierarchical nature of melody seems to lend itself well for modelling in a similar nature to our harmony model, possibly with greater room for stochastic processes to model creativity. We hope to include explicit handling of repetition in FCOMP through the use of a model of musical form, similarly to the graph grammars of Quick and Hudak (2013).

7.3 Improve embellishment

The current way we handle embellishment is rather primitive. For starters, many other forms of ornamentation of repeated notes could be thought of, and care could be taken to ensure that some forms of embellishment are more prevalent than others, to prevent the melody from sounding too random. Furthermore, embellishing non-consecutive notes by simple scale connection is too naive, and leads to long sequences of scales, which sound boring. Jumps could be introduced, taking care to ensure a good balance between continuity and discontinuity, and between upwards and downwards movement.

7.4 Rhythm, form, instrumentation, dynamics

At present, FCOMP deals only with harmony and melody generation. But a piece of music consists of much more; at the very least, rhythm has to be addressed. For better results, however, the large-scale structure of the piece has to be considered (the musical form). Finally, finer details such as the instrumentation and dynamics should be considered too. Although intertwined (the instrument choice affects the melody, for example, as not all instruments have the same range), these aspects can be added incrementally. Our main priority is to add rhythmical knowledge to FCOMP.

8. Conclusion

In this paper we introduced FCOMP, a system for automatic generation of harmony and accompanying melody in a functional setting, designed to be simple and easy to understand and improve. FCOMP uses advanced functional programming techniques for simplicity, as these help remove code duplication and enforce semantic constraints, helping to prevent errors. We've seen how our system can generate simple but pleasing pieces, and how it can be modified to support different harmonies and melody styles. We hope to continue working in FCOMP in the future, as we believe it offers a great opportunity for research not only in automated composition in Haskell, but also in advanced functional programming techniques in practice.

Acknowledgments

The first author is supported by EPSRC grant number EP/J010995/1. We thank Bas de Haas and anonymous reviewers for comments on a draft version of this paper.

References

Moray Allan and Christopher K. I. Williams. Harmonising chorales by probabilistic inference. In *Advances in Neural Information Processing Systems 17*, pages 25–32, 2005. ISBN 9780262195348.

Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.

David Cope. *Experiments in musical intelligence*, volume 12. AR Editions Madison, WI, 1996.

Guido D'Arezzo. *Micrologus*. Rome: Desclee, Lefebvre et S. Edit. Pont., 1904, 1026. URL `http://imslp.org/wiki/Micrologus_(D'Arezzo,_Guido)`.

Kemal Ebcioğlu. An expert system for harmonizing four-part chorales. *Computer Music Journal*, pages 43–51, 1988.

Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130. ACM, 2012. doi:10.1145/2364506.2364522.

W. Bas de Haas, José Pedro Magalhães, Frans Wiering, and Remco C. Veltkamp. HarmTrace: Automatic functional harmonic analysis. *Computer Music Journal*, 37:4:37–53, 2013. doi:10.1162/COMJ_a_00209.

Christopher Harte, Mark B. Sandler, Samer A. Abdallah, and Emilia Gómez. Symbolic representation of musical chords: A proposed syntax for text annotations. In *Proceedings of the 6th International Society for Music Information Retrieval Conference*, pages 66–71, 2005.

Lejaren Arthur Hiller and Leonard Maxwell Isaacson. *Experimental music: composition with an electronic computer*. McGraw-Hill, 1959.

Hendrik Vincent Koops, José Pedro Magalhães, and W. Bas de Haas. A functional approach to automatic melody harmonisation. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, pages 47–58. ACM, 2013. doi:10.1145/2505341.2505343.

Jean-Benjamin de La Borde. *Essai sur la musique ancienne et moderne. Tome premier*. Essai Sur La Musique Ancienne Et Moderne. A Paris: de l'imprimerie de Ph. D. Pierres; et se vend chez Eugène Onfroy, MDCCLXXX, 1780. URL https://archive.org/details/essaisurlamusiv100labo.

Steven G. Laitz. *The complete musician: an integrated approach to tonal theory, analysis, and listening*. Oxford University Press, 2008. ISBN 9780195301083.

Aristid Lindenmayer. Mathematical models for cellular interactions in development I. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

José Pedro Magalhães and W. Bas de Haas. Functional modelling of musical harmony: an experience report. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 156–162. ACM, 2011. doi:10.1145/2034773.2034797.

José Pedro Magalhães and Johan Jeuring. Generic programming for indexed datatypes. In *Proceedings of the 7th ACM SIGPLAN Workshop on Generic Programming*, WGP '11, pages 37–46. ACM, 2011. doi:10.1145/2036918.2036924.

Elizabeth Hellmuth Margulis. *On Repeat: How Music Plays the Mind*. Oxford University Press, 2014. ISBN 9780199990825.

Francois Pachet. The continuator: Musical interaction with style. *Journal of New Music Research*, 32(3):333–341, 2003.

Simon Peyton Jones, editor. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. Journal of Functional Programming Special Issue 13(1).

Przemyslaw Prusinkiewicz. Score generation with L-systems. In *Proceedings of the 1986 International Computer Music Conference*, pages 455–457. Ann Arbor, MI: MPublishing, University of Michigan Library, 1986.

Donya Quick and Paul Hudak. Grammar-based automated music composition in Haskell. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, FARM '13, pages 59–70. ACM, 2013. doi:10.1145/2505341.2505345.

Hugo Riemann. *Vereinfachte Harmonielehre; oder, die Lehre von den tonalen Funktionen der Akkorde*. Augener, 1893.

John Roeder. Pitch class. In *Oxford Music Online*. Oxford University Press, 2013. URL http://www.oxfordmusiconline.com/subscriber/article/grove/music/21855. Accessed May 22.

Martin Rohrmeier. A generative grammar approach to diatonic harmonic structure. In *Proceedings of the 4th Sound and Music Computing Conference*, pages 97–100, 2007.

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

Arnold Schönberg. *Fundamentals of Musical Composition*. Faber & Faber, Incorporated, 1967. ISBN 9780571196586. URL http://books.google.co.uk/books?id=N-lCPgAACAAJ.

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 341–352. ACM, 2009. doi:10.1145/1596550.1596599.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, volume 37 of *Haskell '02*, pages 1–16. ACM, December 2002. doi:10.1145/581690.581691.

Mark J. Steedman. A generative grammar for jazz chord sequences. *Music Perception*, pages 52–77, 1984.

Arnold Whittall. Functional harmony. In *The Oxford Companion to Music*. Oxford University Press, 2013. URL http://www.oxfordmusiconline.com/subscriber/article/opr/t114/e2730. Accessed May 22.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012. doi:10.1145/2103786.2103795.