Experience Report: Functional Modelling of Musical Harmony

José Pedro Magalhães  W. Bas de Haas
Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{jpm,bash}@cs.uu.nl

Abstract

Music theory has been essential in composing and performing music for centuries. Within Western tonal music, from the early Baroque on to modern-day jazz and pop music, the function of chords within a chord sequence can be explained by harmony theory. Although Western tonal harmony theory is a thoroughly studied area, formalising this theory is a hard problem.

We present a formalisation of the rules of tonal harmony as a Haskell (generalized) algebraic datatype. Given a sequence of chord labels, the harmonic function of a chord in its tonal context is automatically derived. For this, we use several advanced functional programming techniques, such as type-level computations, datatype-generic programming, and error-correcting parsers. As a detailed example, we show how our model can be used to improve content-based retrieval of jazz songs.

We explain why Haskell is the perfect match for these tasks, and compare our implementation to an earlier solution in Java. We also point out shortcomings of the language and libraries that limit our work, and discuss future developments which may ameliorate our solution.

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

General Terms Experimentation, Languages

1. Introduction

The deep connection between mathematics and music has been known at least since the times of Plato (Mountford 1923). In the realm of tonal harmony in particular, when studying the relationships between sequential chords, we notice order and regularity; some combinations sound pleasing while others sound peculiar. These observations led music theorists to develop ways to analyse the function of a chord in its tonal context (e.g. Riemann 1893). Among the first to formalize these theories were Lerdahl and Jackendoff (1996), whose work inspired the formal grammar of Rohrmeier (2007, 2011). This grammar was implemented by De Haas et al. (2009) and used for modelling harmonic similarity. Models of tonal harmony are useful because they explain the role or function that a musical chord has within a piece of music. For instance, the same musical chord often has different functions depending on the context in which it occurs.

We present HARMTRACE (Harmony Analysis and Retrieval of Music with Type-level Representations of Abstract Chords Entities), an adaptation of the Java approach of De Haas et al. to a functional setting. Exploring the connection between a context-free grammar and an algebraic datatype, we represent different musical harmonies as values of a datatype. Unlike in previous work, we encode all the musical restrictions in the type itself; strong static typing guarantees that well-typed values represent harmonical sequences. Furthermore, a strongly-typed model gives us higher expressiveness and results in simpler code: through techniques such as datatype-generic programming and type-level computation, most of the code is automatically derived from the types. In a way, the types are the code: most of the code (that would otherwise have to be written manually) follows directly from the types.

A formal model of musical harmony can be used to improve many other typical music processing tasks. Content-based Music Information Retrieval (MIR, Downie 2003), for instance, is a rapidly expanding area within multimedia research which aims at keeping large repositories of digital music maintainable and accessible. Within MIR the notion of similarity is crucial: songs that are similar in one or more features to a given relevant song are likely to be relevant as well. The majority of approaches to notation-based music retrieval focus on melodic similarity. Using our harmony model, we present a method that allows the retrieval of music based on harmonic similarity. We show that comparing a harmonic analysis in tree form, which explains the functions of chords within a sequence, using a generic edit-distance function predicts harmonic similarity better than the edit-distance between the original textual sequences of chords. Chord labels which do not “fit” in our model are automatically corrected at the parsing stage.

Contribution In this paper we present a new functional model of Western tonal harmony in Haskell and explain why Haskell is particularly suited for modelling harmony. We show how our model can be used to perform automatic harmony analysis of sequences of textual chord labels, and that such an analysis improves the task of retrieving harmonically similar pieces. Along the way, we explain how several features of Haskell, such as type-level computations, error-correcting parsers and generic programming, are essential to our approach. All the code of HARMTRACE and the results used for comparison are available upon request.

The rest of this report is structured as follows: we first introduce basic concepts of harmony in Section 2, and then explain how we encode them in Haskell in Section 3. In Section 4 we show applications of our model, which we evaluate in Section 5. We conclude in Section 6 by discussing the limitations of our system and pointing out directions for future development.
is important to stress that formal modelling of tonal harmony is a difficult task, since the rules of harmony are highly ambiguous and often formulated imprecisely.

3. Encoding harmony as a datatype

We now discuss how we formalize general harmony theory as a datatype. Throughout the rest of the paper we elide most of the musical details and concentrate on a small but representative subset of the rules. The general idea is that we convert an input sequence of chord labels, such as "$C:\maj F:\maj D:7 G:7 C:\maj$", into a value of a Haskell datatype which captures the function of chords within the sequence. Since we want to abstract from specific keys, we first translate every chord label into a scale degree. For this to be possible, we assume we know the key of every input song. For instance, our previous example is in C major, so it translates to "$I:\maj IV:\maj II:7 V:7 I:\maj$".

3.1 Naive approach

Using standard algebraic datatypes, we can encode alternatives as constructors, sequences as arguments to a constructor, and repetitions as lists. A first (and very simplified) approach could be the following:

```haskell
data Piece = Piece [Ton]

data Ton = TD T | Dom Ton | TMaj Degree

data Dom = DSD D | SDom Dom | DVMaj Degree

data SDom = SIVMaj Degree

data Degree = Deg Int Class

data Class = Maj | Min | Dom | Dim
```

We see a piece as a list of tonics. A tonic can translate to the first scale degree or branch into a dominant node and another tonic node. Similarly, a dominant can branch into a subdominant and a dominant, or simply be the fifth degree.

The leaves of our tree are the input labelled scale degrees, which consist of a root degree (an integer between 1 and 7) together with a chord class:

```haskell
goodPiece, badPiece :: Piece

goodPiece = Piece [TMaj (Deg 1 Maj)]

badPiece = Piece [TMaj (Deg 2 Maj)]
```

The problem with this representation is evident: non-sensical sequences such as $badPiece$ are allowed by the type-checker. We know that a $Tonic$ can never be the second scale degree: it is either the first degree or a $Dominant$ followed by a $Tonic$. However, since we do not constrain the $Degree$ argument in $TMaj$, we have to make sure at the value-level that we only accept $Deg 1 Maj$ as an argument. To guarantee that the model never deals with invalid sequences we would need a separate proof of code correctness.

3.2 More type information

Fortunately, we can make our model “more typed” simply by using phantom types to encode degrees and chord classes at the type level:

```haskell
data Ton = TD T | Dom Ton | TMaj Degree

data Degree = Deg Int Class
```

Now we detail precisely the root and class of the scale degree expected by $TMaj$, $Dom$ and $SDom$ are adapted in a similar way. We need type-level scale degrees and classes to use as arguments to the new $Degree$ type:

```haskell
data / data II; data III; data IV; data V; data VI; data VII;
```
data Maj; data Min; data Dom; data Dim;

It only remains to guarantee that Degrees are built correctly. An easy way to achieve this is to have a type class mediating type-to-

value conversions, and a function to build degrees:

class ToRoot δ where toRoot :: δ → Int
instance ToRoot I where toRoot _ = 1

... class ToClass γ where toClass :: γ → Class
instance ToClass Maj where toClass _ = Maj

... deg :: (ToRoot δ, ToClass γ) ⇒ δ → γ → Degree δ γ
deg r c = Deg (toRoot r) (toClass c)

If we also make sure that the constructor Deg is not exported, we can be certain that our value-level Degrees correctly reflect their type. Sequences like badPiece above are no longer possible, since the term Tmaj (deg (⊥ :: II) (⊥ :: Maj)) is not well-typed.

3.3 Secondary dominants

So far we have seen how to encode simple harmonic rules and guaranteed that well-typed pieces make “sense”. However, we also need to encode harmonic rules that account for secondary dominants. According to harmony theory, every scale degree can be preceded by the scale degree of the dominant class seven semitones up.

To encode this notion we need to compute transpositions on scale degrees. Since we encode the degree at the type-level this means we need type-level computations. For this we use GADTs (Pej-
ton Jones et al. 2009) and type families (Schrijvers et al. 2008). GADTs allow us to conveniently restrict the chord root and class for certain constructors, while type families perform the necessary transpositions for relative degrees. To support chains of secondary dominants we change the Degree type as follows:

data Degree δ γ η where
  Basedeg :: DegreeFinal δ γ → Degree δ γ (Su η)
  Consy :: Degree (V/ δ) Dom η → DegreeFinal δ γ → Degree δ γ (Su η)

data DegreeFinal δ γ = Lab Int Class

We now have two constructors for Degree δ: Basedeg is the base case, which stores a Root and a Class as before. In Consy we encode the relative dominants. Its type says we can produce a Degree δ for any root δ and class γ by having a DegreeFinal for that root and class preceded by a Degree δ of root V/ δ of the dominant class. The type family V/ δ transposes its argument degree seven semitones up:

type family V/ δ
  type instance V/ I = V
  type instance V/ V = I

To avoid infinite recursion in the parser we use a type-

level natural number in Degree δ. This parameter also serves to control the number of allowed secondary dominants:

data Su η

data Ze

data Degree δ γ = Degree δ γ (Su (Su (Su (Su Ze))))

Typically we use values between 4 and 7 for η. Its value greatly affects compilation time; see the discussion in Section 6.

We have shown a very simplified description of our model of musical harmony as a Haskell datatype. In reality, our model is larger and more detailed, albeit still far simpler than the hundreds of pages of Piston and DeVoito (1991), for instance. To provide an idea of the kind of structure our datatype models, we show an example piece as a pretty-printed value of our datatype in Figure 2. Within this short piece every chord is part of a dominant, subdominant, or tonic structure, and the IV:maj and G:7 are preceded by their secondary dominants. Because the I:7 does not resolve to IV:maj, the parser inserts the expected scale degree IV (see Section 4.1.2).

4. From chord labels to harmonic structure

We have seen how to put Haskell’s advanced type system features to good use in the definition of a model of tonal harmony. In this section we further exploit the advantages of a well-typed model while defining a generic parser from labelled scale degrees (e.g. “I:maj IV:maj II:7 V:7 I:maj”) to our datatype. We also show other operations on the model, like pretty-printing and diffing.

4.1 Parsing

From the high-level musical structure (e.g. the Ton datatype of Section 3.2) we can easily build a parser in applicative style mimicking the structure of the types:

data Parser α -- abstract

class Parse α where parse :: Parser α

instance Parse Ton where
  parse = TdT <$> parse <$> parse
  <$> Timaj <$> parse

For the purposes of this paper we keep Parser abstract; in our implementation we use the uu-parsinglib package (Swierstra 2009). We prefer uu-parsinglib over, say, parsec because our grammar is highly ambiguous and we can put error correction to good use, as we will explain later.

The instance of Parse for Ton is trivial because it follows di-
rectly from the structure of the datatype. It can even be obtained by syntactic manipulation of the datatype declaration: replace \ by \<$\>, add <$ after the constructor name, separate constructor ar-

guments by <$> and replace each argument by parse. The code is tedious to write, and since we have several similar datatypes it becomes repetitive and long.

To compound the problem, the rules of harmony are naturally ambiguous, and we often change the model in search of the best solution. Even more importantly, different musical styles can have significantly different harmony rules (e.g. baroque harmony versus jazz), so our solution should support multiple models. We solve all these problems by not writing instances like the one above. In-

stead, we use datatype-generic programming to derive a parser au-
tomatically in a type-safe way. We use the \texttt{instant-generics} package, which implements a library similar to that initially described by Chakravarty et al. (2009). Due to length considerations we cannot explain how generic programming works in this paper, but our generic parser is entirely trivial. The order of the constructors and their arguments determines the order of the alternatives and sequences; in particular, we avoid left-recursion on our datatypes, which returns a (heterogenously-typed) list of all children of a term, \texttt{build}, which rebuilds a term given a list of new children, \texttt{eqCon}, which computes equality of terms based only on their top-level constructor, and \texttt{typeOf}, which returns a unique representation for the type of a term. For performance reasons we use \texttt{typeOf} from the standard \texttt{Data.Typeable} library, while the other functions are easily implemented in \texttt{instant-generics}. However, the generic \texttt{diff} is rather slow; we discuss this problem in detail in Section 6.

### 4.1 Adhoc parsers

The only truly non-generic parser is that for \texttt{DegreeFinal}, which is also the only parser that consumes any input. It uses the type classes \texttt{ToRoot} and \texttt{ToClass} as described in Section 3.2.

Unfortunately, we are also forced to write the parser instances for GADTs such as \texttt{Degree}, since \texttt{instant-generics} does not support GADTs. Although the code remains entirely trivial, the instance heads become more complicated, since they have to reflect the type equalities introduced by the GADT. As an example, we show the parser code for \texttt{Degree}:

```haskell
instance ( Parse (DegreeFinal δ γ) , Parse (Degree δ V/δ η) ) ⇒ Parse (Degree δ γ (Su η)) where
  parse = BaseDeg<$>parse
  <$><> ConsV <$>parse <$><> parse
```

The context of the instance reflects the type of the constructors of \texttt{Degree}; \texttt{BaseDeg} introduces the Parse \texttt{(DegreeFinal δ γ)} constraint, whereas \texttt{ConsV} requires Parse \texttt{(Degree δ V/δ η)} too.

The need for type-level natural numbers becomes evident here; the instance above is “undecidable” for GHC, meaning that the rules for instance contexts become relaxed. Normally there are constraints on what can be written in the context to guarantee termination of type-checking. Undecidable instances lift these restrictions, with the side-effect that type-checking becomes undecidable in general. However, we are certain that type-checking will always terminate since we recursively decrease the type-level natural number \( η \). This means we also need a “base case instance” where we use the \texttt{empty} parser which always fails; this is acceptable because it is never used.

```haskell
instance Parse (Degree δ γ Ze) where parse = empty
```

Note how useful the type class resolution mechanism becomes: it recursively builds a parser for all possible alternatives, driven by the type argument \( η \). This also means potentially long type-checking times; fortunately our current implementation remains compilable under a minute. We discuss type-checker performance issues in more detail in Section 6.

### 4.1.2 Error correction

We cannot hope to be able to model all valid harmonic relations in our datatype. Furthermore, songs often contain mistakes or mistyped chords, or sequences of dubious harmonic validity. However, these things are often a localized problem, and most of the song still makes sense. In our solution we rely on error correction while parsing: chords which do not fit the structure are automatically deleted or preceded by inserted chords, according to heuristics computed from the grammar structure. We keep track of the number of corrections, since the ratio of corrections to number of input chords provides a measure of meaningfulness of the parse tree. For most songs, parsing proceeds with none or very few corrections. Songs with a very high error ratio denote bad input or wrong key assignment, which results in meaningless scale degrees.

### 4.2 Visualising harmonic relations

In a way similar to the generic parser of Section 4, we also have a generic pretty-printer, which produces output suitable for generation of graphical representations such as that of Figure 2. Note how chords inserted by the parser become leaves with the label \texttt{ins} attached. Similar issues with adhoc instances for GADTs arise, which we solve in the exact same way as described in Section 4.1.1.

### 4.3 Generic \texttt{diff}

A practical application of our tonal harmony model is estimating the harmonic similarity of two songs. An easy way to obtain a measure of similarity between two \texttt{Pieces} is to use a generic \texttt{diff} algorithm. Just like the parser and the pretty-printer, our generic \texttt{diff} is derived from the structure of the datatypes, and adapts automatically to any change. We have implemented it in the style of Lempink et al. (2009) for the \texttt{instant-generics} library. Our \texttt{diff} is based on four primitive generic functions: \texttt{children}, which returns a (heterogenously-typed) list of all children of a term, \texttt{build}, which rebuilds a term given a list of new children, \texttt{eqCon}, which computes equality of terms based only on their top-level constructor, and \texttt{typeOf}, which returns a unique representation for the type of a term. For performance reasons we use \texttt{typeOf} from the standard \texttt{Data.Typeable} library, while the other functions are easily implemented in \texttt{instant-generics}. However, the generic \texttt{diff} is rather slow; we discuss this problem in detail in Section 6.

### 5. Evaluation

In this section we evaluate the parsing results of our system and compare the retrieval performance of the \texttt{gdiff} similarity measure with a simple baseline \texttt{diff} on the input tokens.

#### 5.1 Datasets

We have performed our experiment with two datasets: the dataset of De Haas et al. (2009) (which we call \texttt{small}) and a larger dataset (\texttt{large}). Both datasets consist of textual chord sequences extracted from user-generated Band-in-a-Box files that were collected on the Internet. Band-in-a-Box is a software package that generates accompaniment given a chord sequence provided by the user. The small dataset contains a selection of pieces that “harmonically make sense”, while the large dataset includes many songs that are harmonically atypical. This is because the files are user-generated, and contain peculiar and unfinished pieces, wrong key assignments and other errors; it can therefore be considered “real life” data. Within both datasets there are different chord sequences that describe the same piece in different ways; these can be used to do a retrieval experiment. The \texttt{small} dataset contains 72 songs, of which 35 have no similar songs, 11 have one similar song, and 5 have two other similar songs. The \texttt{large} dataset contains 854 songs of which 485 have no similar songs; the remaining songs have between 1 and 7 similar songs. Note that songs with no similar songs are akin to noise for the retrieval task (see Section 5.3).

#### 5.2 Parsing results

The parsing results are shown in Table 1. For each dataset, we show the average time taken to parse a song and the average error ratio. The error ratio is a measure of how many corrections the parser performed. We define it as a ratio between the number of correction steps and the number of chord labels, but we remove sequences of duplicate chord labels from the input first. A ratio of 0.2, for instance, means that 20% of the significant labels of the sequence have been altered. Lower ratios indicate that the song fits our harmony model better.

On the \texttt{small} dataset, which consists of “harmonically correct” chord sequences, our model performs very well. The songs are
parsed quickly and with average error ratio below 0.07. The large datasets is more problematic. The parsing time increases considerably, mostly because the ambiguity of our model can make the error-correction process rather expensive. The error ratio also increases considerably, but in no way does the parser crash or refuse to produce a valid output. A higher error ratio is also expected, since this dataset has a lot of noisy or meaningless songs.

5.3 Matching results

To test gdiff as a similarity measure for musical harmony, we have performed a retrieval experiment. In this experiment, the task is to retrieve the similar (but not identical) songs based on the edit distance of the gdiff algorithm. The distance between all pairs of songs is calculated and for every song a ranking is constructed by sorting all other songs on the basis of their distance. To place the performance of the gdiff algorithm and the difficulty of the task in perspective, we compare with a baseline algorithm. This method uses no harmony information whatsoever; we simply tokenize the input string into a list of Degrees and perform a standard diff on that list (using the diff package). We use this method to provide a baseline case: the generic diff, having all the harmony information available, has to perform better than this. We call this simple algorithm baseline, while the generic diff of Section 4.3 is named gdiff.

For our datasets we know all the clusters of similar songs. We can therefore analyse the rankings by calculating the Mean Average Precision (MAP). The MAP is a single-figure measure between 0 and 1 quantifying the precision of the retrieved results at all recall levels [Manning et al. 2008] Chap. 8, p. 160); a higher MAP value indicates a better ranking. For the small dataset, gdiff has a MAP of 0.853, while baseline scores 0.475. In the large dataset the difference is smaller, but gdiff still outperforms baseline with a score of 0.510 against 0.395, respectively.

We tested whether the difference in MAP is significant by performing a Wilcoxon Signed-rank test. We chose the Wilcoxon Signed-rank test because the underlying distribution of the average precision over the queries is unknown and this Signed-rank test does not require the distribution to be normally distributed. The differences between the baseline and the gdiff dataset were statistically significant, \( W = 1058.5, p < 0.0001 \) on the small dataset, and also on the large dataset, with \( W = 80352, p < 0.0001 \).

5.4 Comparison with previous work

There are considerable differences between our HARMTRACE system and the context-free grammar approach of De Haas et al. (2009) (hereafter referred to as ISMIR09):

Error-correcting parsers One of the drawbacks of ISMIR09 is that a sequence of chords that does not match the context-free specification precisely will be rejected. For instance, appending one nonsensical chord to an otherwise grammatically correct sequence of symbols will still force the parser to reject the complete sequence, not returning any partial information about what it has parsed. HARMTRACE solves this rejection problem by using error correcting parsers [Swierstra 2009]. This allows us to formalize the rules of tonal harmony that we are certain of, and leave the borderline cases to the parser.

Ambiguity control Music, and harmony in particular, is intrinsically ambiguous. Hence, certain chords can have multiple meanings within a tonal context. This is reflected in both ISMIR09 and HARMTRACE. A major drawback of ISMIR09 is that it is very limited in ways of controlling the ambiguity of the grammar. ISMIR09 uses weighing to order the grammar rules by adding low weights to rules that explain rare phenomena. However, controlling conditional execution would require some form of high-level grammar generation system, since all rules are replicated for each scale degree and chord class. On the other hand, HARMTRACE supports more flexible conditional execution, through the use of GADT’s and type families. An example is the restriction of secondary dominants to chords of the Dom class (Section 3.3).

Parsing performance There are considerable differences in the parsing performance of HARMTRACE compared to ISMIR09 on both datasets. HARMTRACE takes 1.65x to parse the small dataset, while ISMIR09 takes more than 9m. When we compare parsing performance on the large dataset the differences become even more prominent: ISMIR09 rejects 89.7% of the 854 pieces and 3.9% of the dataset had to be excluded because the parsing process would not terminate (due to unconstrained ambiguities). The remaining pieces parse in 84m13s, while HARMTRACE parses the entire dataset in 5m14s. All measurements were done on the same Intel Core 2 duo E6600, 2.4 GHz, machine using GHC 7.0.2 and Java SE 1.6.0_17.

Retrieval effectiveness Both HARMTRACE as well as ISMIR09 have been evaluated on the small dataset. When we compare the retrieval effectiveness of the gdiff approach with the best performing variant of ISMIR09 (MAP of 0.859), we conclude there is no statistically significant difference (\( W = 685, p = 1.00 \), using the same test procedure as in Section 5.3). Because ISMIR09 rejects 89.7 percent of the pieces, no sensible comparison between the two approaches on the large data set can be performed.

Grammar simplicity In ISMIR09 all context-free rules were written by hand, which is not only a tedious and error-prone enterprise, but can also result in very large grammars. By using Haskell’s GADTs to represent the rules of tonal harmony, we gain more expressive power, and the grammar becomes shorter and easier to maintain. For instance, GADTs allow us to write rules that hold for every Maj chord. In their approach, this is expressed by having one rule for major I, II, III, etc.

Code repetition Our Haskell system is more concise than the Java implementation of ISMIR09. An analysis of the number of significant source lines of code reveals that ISMIR09 has 5545 lines, while HARMTRACE has 1721, less than one fourth.

6. Discussion and conclusion

We have shown how Haskell can be used to implement a model of musical harmony. Our solution outperforms a previous Java approach both in terms of speed, functionality and elegance. However, the current implementation has a number of limitations, which we now describe in detail.

Type-checker performance As mentioned in Section 4.1.1 it is easy to make the type-checker take very long to compile our code. We managed to keep the type-checking time acceptably low, but this is only because we are “helping” it. We minimized the number of type families used (four in total, all similar to V/) and we (automatically) place each instance declaration in a separate module, since this speeds up compilation considerably. Furthermore, we represent each scale degree as an independent type: type-level computations, such as transposition, are then indexed over each type. A

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Error ratio</th>
<th>Time taken (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>0.067</td>
<td>23.833</td>
</tr>
<tr>
<td>large</td>
<td>0.200</td>
<td>381.837</td>
</tr>
</tbody>
</table>

Table 1. Error ratio and parsing runtime averaged over all songs.
more concise way of representing scale degrees would be to use type-level naturals. Transposition is then simply summing modulo the total number of scale degrees. Unfortunately this makes the compilation time unacceptably high. We hope that native type-level naturals are added to GHC soon so that we can simplify our type-level computations without a performance penalty.

**Parser performance** The higher average parsing time per song on the larger datasets shown in Table 1 is caused mostly by a couple of songs taking very long. In the large dataset, only 6% of the songs take longer than one second to parse. The three slowest songs take 41s, 24s, and 15s to parse. They are long songs, and either contain chord sequences which our model does not account for or are harmonically atypical. In these pathological cases the parser combinator takes very long to compute the possible corrections. This is somewhat understandable, since our grammar is highly ambiguous and there are multiple non-trivial possible corrections. However, such long parsing times are undesirable; perhaps the number of steps to lookahead in the parser could be dynamically adjusted based on the number of possible alternatives. This would hopefully lead to shorter parsing times, albeit at the cost of potentially worse corrections.

**Matching performance** The generic diff is a powerful tool that solves the matching problem almost "for free" (Section 5.3). However, to use it we need new generic functions to be derived for every datatype. This means longer compilation times, but also more adhoc instances, since there is no suitable generic programming library supporting GADTs. These instances amount to over 200 lines of repetitive and error-prone code. Worse even, it runs very slowly; our implementation uses type-safe runtime cast, which prevents fusion of the generic representations. This prevents us from using the generic diff on datasets with thousands of songs.

Besides addressing the limitations pointed out above, we also plan to add new functionality to our system:

**Mode and key** In Section 3 we only discussed the rules for pieces in a major key. However, many songs are written in a minor key; this affects the expected scale degrees at the leaves, invalidates some alternatives and creates others. Nevertheless, a large number of rules holds for both pieces in a major and a minor key. Currently we handle this using a similar model for pieces in a minor key:

```haskell
data Piece µ = PieceMaj [TonMaj] | PieceMdo [TonMdo]
```

This however, leads to unnecessary mode duplication, since most of the harmony rules are independent of mode. A better alternative would be to index pieces by their mode:

```haskell
data MajMode, data MinMode;
data Piece µ = Piece [Ton µ]
```

The type variable $\mu$ would then be indexed with either $\text{MajMode}$ or $\text{MinMode}$, similarly to $\delta$ for degrees and $\gamma$ for chord classes. We think this would be an elegant way of expressing mode in the model.

Additionally, we currently restrict ourselves to songs in a single key, but often songs change the key throughout their development. This means that scale degree $I$ no longer maps to chord C, but to F, for instance. Indexing the model over the key and introducing rules for modulation which would change this key would be a good way of encoding key changes.

Unfortunately, such changes would make the entire model indexed over one or more type variables, which would preclude the use of generic programming altogether. We plan to build into the new type-level datatype the necessary infrastructure to be able to deal with indexed datatypes adequately.

**Other applications** We show how to use our model for improving music retrieval, but we believe other tasks can be improved similarly. For instance, algorithms for computing chord labels from audio or images (scores) often recognize a set of possible chords at each step, with different probabilities. Our model could then be used to check which chords are harmonically valid at each step, therefore introducing harmony knowledge into the algorithm. Another interesting development would be to implement a (generic) enumerator over our datatypes; this would correspond to a generator of harmonically valid sequences of chords.

Overall, we are convinced that strong static typing and generic programming are essential tools in modelling musical harmony. We hope our approach paves the way for future functional approaches to musical modelling and processing.

**References**


