What Constitutes a Musical Pattern?

Orestis Melkonian
Department of Information and Computing Sciences
Utrecht University
The Netherlands
melkon.or@gmail.com

Iris Yuping Ren
Department of Information and Computing Sciences
Utrecht University
The Netherlands
y.ren@uu.nl

Wouter Swierstra
Department of Information and Computing Sciences
Utrecht University
The Netherlands
w.s.swierstra@uu.nl

Anja Volk
Department of Information and Computing Sciences
Utrecht University
The Netherlands
a.volk@uu.nl

Abstract

There is a plethora of computational systems designed for algorithmic discovery of musical patterns, ranging from geometrical methods to machine learning based approaches. These algorithms often disagree on what constitutes a pattern, mainly due to the lack of a broadly accepted definition of musical patterns.

On the other side of the spectrum, human-annotated musical patterns also often do not reach a consensus, partly due to the subjectivity of each individual expert, but also due to the elusive definition of a musical pattern in general.

In this work, we propose a framework of music-theoretic transformations, through which one can easily define predicates which dictate when two musical patterns belong to a particular equivalence class. We exploit simple notions from category theory to assemble transformations compositionally, allowing us to define complex transformations from simple and well-understood ones.

Additionally, we provide a prototype implementation of our theoretical framework as an embedded domain-specific language in Haskell and conduct a meta-analysis on several algorithms submitted to a pattern extraction task of the the Music Information Retrieval Evaluation eXchange (MIREX) over the previous years.

CCS Concepts • **Information systems** → **Music retrieval**; • **Applied computing** → *Sound and music computing*; • **Software and its engineering** → *Domain specific languages.*

FARM '19, August 23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6811-7/19/08…$15.00
https://doi.org/10.1145/3331543.3342587

Keywords transformation, edit distance, musical patterns, contravariance, evaluation, clustering

ACM Reference Format:
Orestis Melkonian, Iris Yuping Ren, Wouter Swierstra, and Anja Volk. 2019. What Constitutes a Musical Pattern?. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design (FARM '19), August 23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3331543.3342587

1 Introduction

Musical patterns are hard to define. Patterns are ubiquitous in music. Generated by composers, accented by performers and perceived by listeners, musical patterns contribute to a better understanding and smoother communication in music. For example, in an improvisation session or a music theory classroom, one might hear descriptions such as "This pattern here was repeated three times". However, in music theory and music information retrieval (MIR), an agreed-upon definition of "musical pattern" has been elusive. One can argue that a musical pattern is "an excerpt of special importance", or "a salient fragment", or "a prominent unit", etc. In addition, in different contexts of musical corpora, experts use different terminologies such as "lick", "riff", "leitmotif", or "sequence", to refer to these patterns in music. This variability in the definition makes it difficult to design and evaluate automated computational systems that extract musical patterns.

Musical patterns are useful but hard to extract. Data-driven algorithms and rule-based algorithms have been designed to extract musical patterns. Nevertheless, a challenge in automating musical pattern discovery is the subjectivity on when musical patterns are perceived and the ambiguity in music compositions [McFee et al. 2017; Ren et al. 2018b]. This increases the complexity of formalising the pattern discovery task, which contributes to a lack of annotations datasets as well. The question is, therefore, how do we leverage the limited data and theory to design and evaluate a pattern discovery system?

MIREX and datasets. In terms of evaluation, previous research has addressed the challenge to a certain extent. Most algorithms have been tested on unassociated datasets with disparate metrics [Janssen et al. 2013]. One attempt to provide a robust benchmarking suite for algorithm evaluation is the MIREX Discovery of Repeated Themes & Sections task initiated in 2014[1]. In the task, a pattern is defined as a set of time-pitch pairs that occurs at least twice in a piece of music and the JKU-PDD dataset was introduced [Collins 2013]. Pattern discovery algorithms submitted to MIREX use different models and methods from geometry [Collins et al. 2013; Meredith 2015], information theory [Conklin 2010; Velarde et al. 2016] and machine learning[Pesek et al. 2017]. According to the evaluation metrics in this task, the state-of-the-art algorithms perform acceptably well in precision, recall, and F1-scores, although they cannot reproduce the human-annotated patterns yet. Another pattern annotation dataset which has been used for evaluating the algorithms is the MTC-ANN Dutch Folk Song dataset [Van Kranenburg et al. 2016]. Using this dataset, human annotations have been compared with algorithmically extracted patterns by their performance in a classification task [Boot et al. 2016] showing the annotated patterns perform better. Furthermore, a large disagreement between annotated and computationally extracted patterns has been shown in both the JKU-PDD and MTC-ANN dataset in [Ren et al. 2018a, 2017].

Understanding musical pattern discovery algorithms. In this paper, we encode computationally well-defined and music theoretically relevant atomic transformations and their compositions to investigate the output of pattern discovery algorithms, human annotations of patterns and random passages. This constitutes a first step into understanding the differences between algorithms and human annotations in terms of the underlying atomic transformations.

Contributions.

Executable Model We describe a framework for defining transformations of musical patterns, utilising well-known abstractions from category theory. We also provide a Haskell implementation of the model, in the form of an embedded domain-specific language (DSL).

Pattern Analysis We use our framework to define several music-theoretic pattern transformations, limiting ourselves to only those that are well-established in music theory. Nonetheless, we extend them with the notion of approximation, so as to account for ad-hoc insertion/removal of musical elements. We then analyse both the JKU-PDD and the MTC-ANN datasets, by examining the patterns discovered by experts and state-of-the-art algorithms alike.

Pattern Discovery As a nice by-product to our transformational framework, our DSL can act as a minimal query language for musical patterns. Therefore, we can utilize our DSL to derive a (naive) pattern discovery algorithm almost for free; a transformation written in our DSL will now act as the query to search against in a given music piece.

2 Musical Patterns and Modelling

Repetitions and variations. Musical patterns and variations are closely related. On one hand, variation of a musical element consists of repeating it with modifications to one or more of its attributes (e.g. pitch, duration). Conversely, patterns can be defined as variations of an initial musical element. Furthermore, variation can be local or global in music: ornaments such as trills and turns are local, form and thematic variations are global.

Prototype and transformations. Independent of the variation, there must be some original material from which the variations stem. We refer to these as the *prototype patterns*. Each prototype pattern may be related to several pattern *occurrences*; typically, there is some *transformation* necessary to map the prototype pattern to the pattern occurrence, accounting for the variation amongst pattern occurrences.

Unsurprisingly, we model these transformations as functions. For example, one simple transformation is chromatic (real) transposition, $\lambda x \rightarrow x + n$, that transposes each note by some pitch shift n. Using this transformation, we could start with a short minor blues "lick" in F, such as $[Eb4, C4, Ab4, F3, F4]$, and allow transposition to all keys by keeping the allowed pitch shifts below an octave (i.e. $0 \ldots 11$ semitones), thus deriving a key-independent version of the original lick. For instance, we can extract the same lick in the key of G by applying f two times, which gives us $[F4, D4, Bb4, G3, G4]$. It is natural to expect that chromatic transposition is an equivalence relation, except for people with absolute pitch.

3 Implementation in Haskell

In this section, we give an overview of the main techniques we used to implement our meta-analytic framework and motivate the design choices we made along the way.

We strive to find a mathematically sound solution to our problem, using first class functions to model musical transformations. Although we chose to implement everything in Haskell, we do not rely on any of Haskell's advanced type-level features (e.g. data kinds), thus any other strongly-typed functional programming language would suffice. All code is publicly available in a Github repository[2].

[1] https://www.music-ir.org/mirex/wiki/2017:Discovery_of_Repeated_Themes_%26_Sections

[2] https://github.com/omelkonian/hs-pattrans

3.1 Basic Music Datatypes

First of all, we need to define some basic music datatypes. *Time* occurrences are represented as number of crochet beats from the start of the song, while *MIDI* values are represented as plain integers conforming to the MIDI standard[3]. Several music-specific notions such as *Scale* and *Degree* are again represented by their arithmetic equivalents.

```
type Time   = Double
type MIDI   = Integer -- also represents intervals
type Degree = Integer
type Scale  = Map MIDI Degree
```

Moving on to datatypes that correspond to data entities in the datasets, a *Note* is a certain pitch occurring at a specific point in time and a *Pattern* is a sequence of notes. Since we do not currently support polyphony, we can treat a piece of music as a single huge pattern.

```
data Note = Note
  { ontime :: Time
  , midi   :: MIDI
  } deriving Eq
type Pattern    = [Note]
type MusicPiece = Pattern
```

The output of a pattern discovery algorithm is a set of *pattern groups*, which have a prototype pattern and (possibly many) occurrences of that pattern across the musical piece. We also keep some metadata (such as to which piece of music a pattern group belongs), which will be used later in visualization.

```
data PatternGroup = PatternGroup
  { prototype  :: Pattern
  , occurences :: [Pattern]
  , metadata   :: Metadata
  } deriving Eq
```

Lastly, the following properties of music elements will be needed when we later define the actual transformations. We will certainly require to retrieve different viewpoints [Conklin and Anagnostopoulou 2001] of a pattern (e.g. durations, pitches), as well as transform absolute values to relative ones. For instance, we might want to see the *intervals* between a pattern's pitch values, rather than the absolute pitches in isolation.

```
toRelative :: Num a ⇒ [a] → [a]
toRelative = fmap (−) ∘ pairs

inverse :: Num a ⇒ [a] → [a]
inverse = fmap negate

basePitch :: Pattern → MIDI
basePitch = midi ∘ head

pitch, intervals :: Pattern → [MIDI]
pitch     = fmap midi
intervals = toRelative ∘ midi

durations, rhythm, normalRhythm :: Pattern → [Time]
durations     = fmap ontime
rhythm        = toRelative ∘ durations
normalRhythm = normalizeTime ∘ rhythm
  where
    normalizeTime (t0 : ts) = 1 : map (/t0) ts
    normalizeTime []        = []
```

Note how we normalize time information; instead of looking at absolute duration values, we care about the relevant increase or decrease compared to the initial time value. This will become useful when we define time-specific transformations.

3.2 Transformations

We now define music-specific transformations between patterns, which form equivalence relations. In order to do so, we start out with the design of a minimal combinator DSL, which will help us further down the road.

The definitions of our transformations will essentially be a predicate on two elements, that returns *True* when they belong to the equivalence class in question and *False* otherwise. Since we would like to reuse the same type for viewpoints of a pattern (e.g. time) and possibly between different types, we universally quantify over the types being compared. We also provide a convenient infix notation to check for equivalence, as well as a type alias for *homogeneous* checkers of elements of the same type.

```
newtype Check a b = Check { unCheck :: a → b → Bool }
(⟺) :: a → b → Check a b → Bool
(x ⟺ y) p = unCheck p x y
type HomCheck a = Check a a
```

The simplest possible comparison is that of exact equality between elements of the same type, which can be defined using the *Eq* typeclass.

```
-- e.g. ([1,2] ⟺ [1,2]) equal
equal :: Eq a ⇒ HomCheck a
equal = Check (≡)
```

In order to combine multiple checks in conjunction, we must recognise the monoidal algebraic structure of checkers. Note that *Check* can be seen as a *Monoid* in more than one way, i.e. using either conjunction or disjunction. Here we stick to the conjunctive version (note the use of ∧), since we do not have any use-cases for disjunctive checkers yet. If such a DSL construct is needed in the future, we will provide **newtype** wrappers and the programmer would then need to manually annotate which monoid instance to use (when one uses ⋄).

[3] Here, we only consider the part of a MIDI value that represents pitch number, ignoring other features such as velocity.

```
instance Monoid (Check a b) where
  ε       = Check $ λ_ _ → True
  p ⋄ q = Check $ λx y → (x ⟺ y) p
                       ∧ (x ⟺ y) q
```

In order to define more interesting pattern relations with compositional operations like *map*, we first make the observation that the type parameters (a and b in this case) occur in a *contravariant* position, since they are arguments to a function. Moreover, we observe that there are two different arguments and we want a functorial action in both. As an example of contravariance, assume you have a checker for durations (i.e. *HomCheck Time* and you want to use that to check equivalence of two patterns (i.e. *HomCheck Pattern*). Hence, you need to have a function *HomCheck Time → HomCheck Pattern*, but you cannot define it even if you had a function *Time → Pattern*. What you, in fact, must have in your hands is a function *Pattern → Time* (notice the reversal in the argument order, aka *contravariance*).

With these observations, we arrive at the definition of a *contravariant bi-functor*:

```
class ContravariantBifunctor p where
  contraBimap :: (c → a) → (d → b) → p a b → p c d
  contraBimap f g = contra₁ f ∘ contra₂ g

  contra₁ :: (c → a) → p a b → p c b
  contra₁ f = contraBimap f id

  contra₂ :: (d → b) → p a b → p a d
  contra₂ g = contraBimap id g

instance ContravariantBifunctor Check where
  contraBimap f g p = Check $ λx y → (f x ⟺ g y) p
```

We also define specialized infix variants of the contravariant operations on checkers of equal types, where we modify one or both of the arguments.

```
(›$‹) :: (b → a) → HomCheck a → HomCheck b
f ›$‹ p = contraBimap f f p

(›$) :: (a → a) → HomCheck a → HomCheck a
(›$) = contra₁

($‹) :: (a → a) → HomCheck a → HomCheck a
($‹) = contra₂
```

Figure 1 depicts the associated contravariant operations diagramatically, where $f :: c \to a$ and $g :: d \to b$. Furthermore, Figure 2 shows the specific case of homogeneous checkers, assuming $f :: b \to a$ and $\eta :: a \to a$.

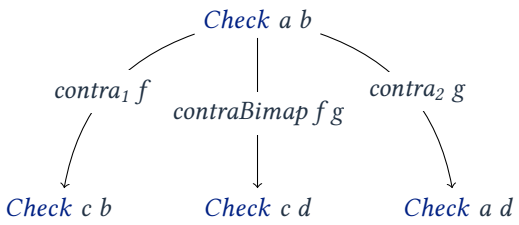

Figure 1. Contravariant operations on heterogeneous checkers.

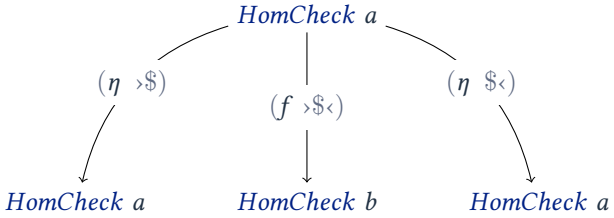

Figure 2. Contravariant operations on homogeneous checkers.

As a first example usage of a contravariant checker, we can define when a pattern is an exact repetition of another later in the song:

```
exactRepetitionOf :: HomCheck Pattern
exactRepetitionOf = rhythm ›$‹ equal
                  ⋄ pitch  ›$‹ equal
```

3.3 Approximate Equality

Before we proceed on defining our transformations, let us extend our checkers with the notion of approximation. We do so by passing a floating point number $p \in [0, 1]$, representing the approximation degree as a percentage.

```
type ApproxCheck a = Float → HomCheck a
(≈) :: ApproxCheck a → Float → HomCheck a
approxChecker ≈ p = approxChecker p
```

Our little combinator DSL is now complete and we can proceed with defining some useful checkers. First, we extend equality with approximation of percentage p, by also allowing equality when:

1. At least $p\%$ of the prototype remains in the occurrence:
$$ignored \leqslant (1 - p) * N$$
2. At least $p\%$ of the occurrence appears in the prototype:
$$added \leqslant (1 - p) * M$$

where N, M are the lengths of the prototype and occurrence respectively, *added* is the number of elements in the occurrence that do not appear in the pattern, *ignored* is the number of prototype elements that were not deleted and *delete* is an operation that respects the prototype order (unlike Haskell's \\ operator).

Note that this definition consitutes a particular form of *edit distance*[4], where we measure similarity of two list-like values by the minimum amount of modifications required to transform one into the other. This procedure, in fact, generalizes to any list we might wish to compare:

```
-- e.g. ([A,C,F,A,B] ⟺ [A,C,G,A,B]) (eq≈ 0.8)
eq≈ :: Eq a ⇒ ApproxCheck [a]
eq≈ p = Check $ λxs ys →
        (ignored ⩽ (1 − p) ∗ length xs)
    ∧   (added   ⩽ (1 − p) ∗ length ys)
  where
    (added, ignored) = del ys xs
    del :: [a] → [a] → (Int, Int)
    del ys [] = (length ys, 0)
    del [] xs = (0, length xs)
    del ys (x : xs)
        | Just (ysL, ysR) ← x `del1` ys
        =   (first (+length ysL) <$> del ysR xs)
        `min` (second (+1) <$> del ys xs)
        | otherwise
        = second (+1) <$> del ys xs
    del1 :: a → [a] → Maybe ([a], [a])
    -- try to delele and and return the surrounding lists
```

In the example illustrated in the comment above, we have $N = M = 5$ and we have *ignored* the prototype note F and *added* the occurrence note G (i.e. $ignored = 1$ and $added = 1$). Thus we can conclude that these two patterns are 80% equal, since the two required conditions are satisfied:

$$ignored = 1 \leqslant 1 = (1 - 0.8) * 5$$
$$added = 1 \leqslant 1 = (1 - 0.8) * 5$$

The intuition behind the algorithm is rather simple; for each element of the prototype:

1. Traverse the occurrence until you find the prototype element.
2. If found, consider two cases:
   a. Increase *added* by the number of bypassed elements during the search.
   b. Ignore it nonetheless, incrementing *ignored*.
3. Otherwise, increment *ignored*.
4. Continue with the rest of the prototype.

[4] https://en.wikipedia.org/wiki/Edit_distance

The worst-case time complexity of the algorithm above is $O(NM)$ where N, M are the lengths of the prototype and occurrence respectively, since we would need to traverse the whole occurrence for each element of the prototype. Nevertheless, we set a maximum look-ahead on the deletion process to facilitate faster analyses, resulting in overall runtime complexity of $O(N)$, since we would perform $O(1)$ computation for each of the N elements of the prototype[5].

Attributes built from pairs of basic elements (e.g. *rhythm* from *durations*, *intervals* from *pitch*) score rather badly on the previous definition of approximate equality, since a simple insertion on the initial pattern would create a lot of differences in the paired output. As a motivating example, assume our prototype consists of the notes A and B and the occurrence adds a passing note Bb in-between them. While *first-order* approximate equality works well when comparing their *pitch*, we get in trouble when comparing their *intervals*; we have lost all similarity between them, since the prototype has intervals $[2]$ and the occurrence $[1, 1]$.

In order to remedy this, we define a *second-order* approximate equality that is additionally allowed to merge consecutive elements in the occurrence and match them to a single entity of the prototype.

```
-- e.g. ([A,B,C,D,E] ⟺ [A,Bb,B,D,E,F,F#])
--      (intervals >$< eq≈² 0.75)
eq≈² :: (Ord a, Num a, Eq a) ⇒ ApproxCheck [a]
eq≈² p = ...
```

Apart from the extension of *del1* to allow merging, the code is identical to the first-order case.

In the example above, we ask whether the intervals of these two patterns are 80% equal. For clarity, let us first simplify the comparison by inlining the interval calculation, where intervals are represented by number of semitones upward:

$$([2, 1, 2, 2] \iff [1, 1, 3, 2, 1, 1])\ (eq\approx^2 \approx 0.75)$$

Second-order approximate equality allows us to merge the first two and the last two intervals of the occurrence, replacing them with their sum:

$$([2, 1, 2, 2] \iff [2, 3, 2, 2])\ (eq\approx^2 \approx 0.75)$$

We now have only a single *ignored* interval in the prototype and a single *added* interval in the occurrence, thus it is safe to conclude that these (interval) patterns are 75% equal:

$$ignored = 1 \leqslant 1.0 = (1 - 0.75) * 4$$
$$added = 1 \leqslant 1.5 = (1 - 0.75) * 6$$

We, furthermore, set a maximum threshold on the number of consecutive elements we are allowed to accumulate, thus inducing only a constant overhead to the previous time complexity in the first-order case.

[5] Note that the faster implementation is actually an under-approximation of the proposed algorithm, i.e. there might be false negatives in the results.

The first music-theoretic checker we introduce is known as *horizontal translation* and is the repetition of a melody later in time, i.e. their pitches and relative durations must be (approximately) equal.

```
exactOf :: ApproxCheck Pattern
exactOf p = rhythm ›$‹ eq≈² p
          ◇ pitch  ›$‹ eq≈ p
```

Similarly, we can define all the usual transformational patterns in musicology, such as *vertical translation* (i.e. real/chromatic transposition, tonal/diatonic transposition), *horizontal reflection* (i.e. inversion), *vertical reflection* (i.e. retrograde) and *time scaling* (i.e. diminution—speeding up, augmentation—slowing down). A complete list can be found at the end of this paper.

```
transpositionOf :: ApproxCheck Pattern
transpositionOf p = rhythm   ›$‹ eq≈² p
                  ◇ intervals ›$‹ eq≈² p

inversionOf :: ApproxCheck Pattern
inversionOf p = basePitch ›$‹ equal
              ◇ rhythm    ›$‹ eq≈² p
              ◇ intervals ›$‹ (inverse $‹ eq≈² p)

retrogradeOf :: ApproxCheck Pattern
retrogradeOf p = rhythm ›$‹ (reverse $‹ eq≈² p)
               ◇ pitch  ›$‹ (reverse $‹ eq≈ p)

rotationOf :: ApproxCheck Pattern
rotationOf p = rhythm   ›$‹ (reverse $‹ eq≈² p)
             ◇ intervals ›$‹ (reverse $‹ eq≈² p)

augmentationOf :: ApproxCheck Pattern
augmentationOf p = normalRhythm ›$‹ eq≈² p
                 ◇ pitch        ›$‹ eq≈ p
```

An interesting case is that of *tonal* transposition, where we are not transposing in absolute semitones but rather in scale degrees, which could possibly change the pitch structure of a pattern. As an example, constructing a triad from the first degree of C major results in a major triad (i.e. C-E-G), but transposing to the second degree would result in a minor triad (i.e. D-F-A).

This transformation is defined with respect to a given scale, which we currently choose based on a simple heuristic that considers the notes appearing in the prototype. We guess the most "fitting" scale, with respect to the number of prototype pitches that belong to the scale.

```
tonalTranspOf :: ApproxCheck Pattern
tonalTranspOf p =
    rhythm ›$‹ eq≈² p
◇   Check $ λxs ys →
      (xs ⟺ ys)
      (apply (guess xs) ›$‹ intervals ›$‹ eq≈² p)
  where
    -- guess the scale that is the most "fitting" to a pattern
    guess :: Pattern → Scale
    -- interpret a pattern with respect to a given scale
    apply :: Scale → Pattern → Pattern
```

4 Music Data

For applying our model to actual data, we use the standard JKU-PDD dataset and the MTC-ANN Dutch Folk Song dataset [Van Kranenburg et al. 2016]. The exceptionally large number of annotated patterns MTC-ANN contains makes it a perfect candidate for a classification experiment. In this section, we examine groups of patterns, random passages, and their features in this dataset.

Parsing the datasets. We use the Parsec library [Leijen and Meijer 2001] to parse the datasets, converting the music data from each dataset format to our own internal representation of musical elements. The parsing code is not of much interest, so we refer the reader to the available source code.

Annotated patterns. During the making of MTC-ANN, three experts have been asked to annotate the prominent patterns in each song which best classify the song into one of 26 tune families. *Tune family* is a concept in ethnomusicology that groups together tunes sharing the same ancestor in the process of oral transmission [Boot et al. 2016]. The dataset consists of 360 Dutch folk songs with 1657 annotated pattern occurrences. In an annotation study on what influences human judgements when categorising melodies belonging to the same tune family, repeated patterns turned out to play the most important role [Volk and Van Kranenburg 2012]. It is, therefore, reasonable to use repeated pattern discovery algorithms on this dataset.

Patterns from algorithms. We use the six pattern discovery algorithms and extract the patterns from the MTC-ANN dataset using the same setup as in [Boot et al. 2016; Ren et al. 2017]. The extracted patterns from each algorithm form a subgroup under the umbrella of the extracted pattern group. The seven algorithms were submitted to the MIREX task during 2014-2017: SIATECCompress - TLP (**SIAP**), SIATECCompress - TLF1 (**SIAF1**), SIATECCompress - TLR (**SIAR**) [Meredith 2015], **VM** & **VM2** [Velarde et al. 2016], SYMCHM (**SC**) [Pesek et al. 2017], and SIARCT-CFP (**SIACFP**) [Collins et al. 2013].

We compare annotated and extracted patterns with randomly sampled passages as a baseline in order to potentially support or refute the significance of musical patterns. In more detail, taking the annotated patterns from MTC-ANN, random passages are sampled with the following procedure:

1. For each annotated pattern, we find the corresponding song where the annotation appears.
2. Find a random starting point and take an excerpt of the same length as the pattern.
3. Repeat five times to prevent accidental results.

5 Results

5.1 Patterns Explained by Transformations

By giving a different approximation degree to each of our transformations, we get a rich spectrum of pattern equivalence classes. This allows to count how many occurrences of the datasets belong in each such class.

For each pattern group of each expert/algorithm of each song of each dataset, we start comparing against each of the following transformations in-order:

$$\begin{aligned} &\mathit{exactOf} \rightarrow \mathit{transpositionOf} \rightarrow \mathit{tonalTranspositionOf} \\ &\quad \rightarrow \mathit{inversionOf} \quad \rightarrow \mathit{augmentationOf} \\ &\quad \rightarrow \mathit{retrogradeOf} \quad \rightarrow \mathit{rotationOf} \end{aligned}$$

We perform multiple passes by varying the approximation degree amongst 100%, 80%, 60%, 40% and 20%. As a last step, we combine the results from all pattern groups belonging to the same dataset to get a better overview.

We visualize both individual and aggregated results as pie charts with Chart[6], a declarative library for generating 2D charts and plots. Here, we display only the overall results of the different datasets; for the complete set of analysis results we refer the reader to a dedicated webpage[7].

Figure 3 and Figure 4 show results from the JKU-PDD dataset; Figure 5, Figure 6 and Figure 7 show results from the MTC-ANN dataset. For better readability of the charts:

- Each color is assigned to a transformation and its approximation. For example, "exact" repetitions are colored rose; "exact" repetitions up to 80%, 60%, 40%, and 20% approximation are assigned with the rose color, too. Similarly, the blue parts are the "transposed", the normal/chromatic transposition. The tonal transpositions, "tonalTransped", are in the red regions. The pattern occurrence relations that cannot be accounted for by the transformation are black.
- Recall that our passes start with "exact" repetitions, then "transposition", and other transformations with approximation: we ordered them counterclockwise in the pie chart.

[6] https://hackage.haskell.org/package/Chart
[7] https://omelkonian.github.io/hs-pattrans/charts.html

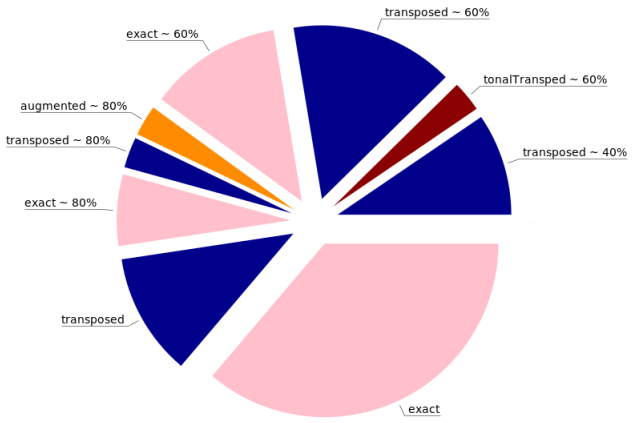

Figure 3. JKU-PDD: transformations in expert annotations. "Exact" repetition has the largest proportion in comparison to other transformations. All pattern occurrence relations can be accounted for by a few categories of transformations and approximation up to 40%. The composition of the pie chart is very different from the algorithmic counterpart (Figure 4).

Figure 3 depicts the results for the expert annotations on JKU-PDD. It is evident that our transformations are in sync with the mindset of the music expert, since we cover half of them by strict transformations and then gradually cover the rest of the spectrum by approximation. This is not the case for the algorithmic output on the same dataset in Figure 4, where only a small fragment of the pattern groups can be explained with strict transformations and, even at the presence of loose approximation, we do not get a satisfiable cover. In both cases though, a handful of transformation types dominate, namely exact repetition and tonal/chromatic transposition.

Figures 5 and 6 provide the same statistics for MTC-ANN. In comparison to the classical one, we discern a bit more variety in the types of patterns that appear (e.g. augmentation). This difference of transformations between different datasets might have stemmed from the spontaneous process of oral transmission, which is usually how folk songs are retained. This is very different from classical music, which has precise notations and undergoes a well-thought-out process of composing. Another potential reason for the difference is that there are much more patterns were annotated in MTC-ANN than in JKU-PDD. More investigations are needed for investigating the causes.

Despite the differences in what kind of transformations are there in two different datasets, the same general trends appear: a much more coherent perception of musical pattern in the expert annotations, in contrast to the freedom the algorithms allow.

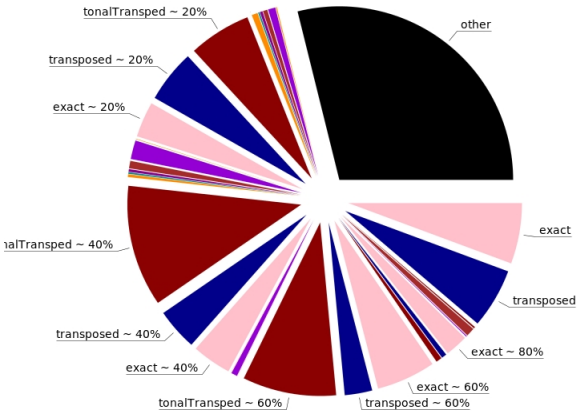

Figure 4. JKU-PDD: transformations in algorithmic output. Instead of having the "exact" repetition being in the dominant position as in Figure 3, there is a variety of transformations: the algorithms discover a more diverse set of patterns than expert annotations. A large proportion of pattern occurrence relation cannot be explained by any transformation we considered ("other" as colored in black).

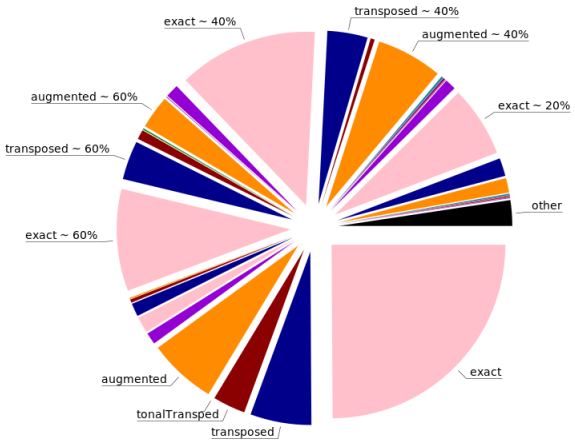

Figure 5. MTC-ANN: transformations in expert annotations. Similar to the expert annotations in JKU-PDD (Figure 3), "exact" repetition is the dominant element in the pie chart. However, in this dataset, we have a more diverse range of transformations. A small portion of the patterns cannot be accounted for by any transformation we considered ("other" as colored in black).

Here, we also include a randomized sample of the original dataset (Figure 7), to verify that random patterns do not pass through our transformation checker. Fortunately, we see a minuscule percentage of strict transformations and a general domination of low-approximation transformations such as 20% and 40%, which is reasonable since 0% approximation is valid for any set of patterns. Curiously, when comparing the "other" category (the black wedges, which consist of unmatched occurrences using transformation and approximation), we see that the proportion is less than the algorithmic counterpart but more than the expert annotations. This reminds us that, not only annotations are well-related by transformations, also the random parts from the corpus can be linked with each other, given that we allow for approximation; and there are more relations that are not explainable by our model in the algorithmic output than in the random excerpts in the corpus.

5.2 Pattern Querying and Discovery

While the checkers defined were initially designed to handle the meta-analysis of existing pattern discovery algorithms, it turns out that one gets a pattern query language for free!

Given a prototype in a song and a pattern checker (i.e. *HomCheck Pattern*), we keep a sliding window having the same size as the prototype and check the query against all such occurrences. The implementation is trivial, as shown below:

```
type WindowSize = Int
type Query a    = (HomCheck a, a)
```

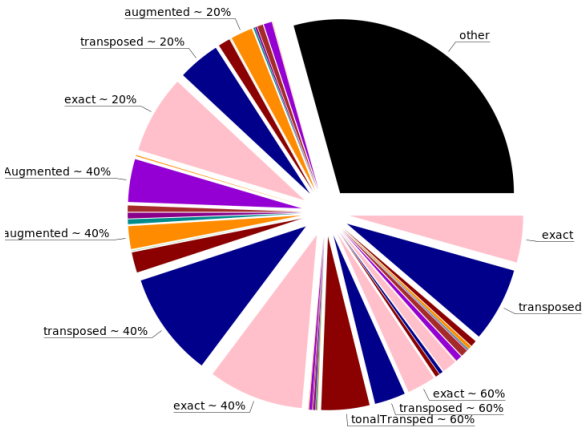

Figure 6. MTC-ANN: transformations in algorithmic output. Similar to the algorithmic output in JKU-PDD (Figure 4), we have a variety of transformations. There is also a matching trend from both JKU-PDD and MTC-ANN between the expert annotations and the algorithmic output: the algorithms discover a more diverse set of patterns and the transformation we considered are less capable of covering their occurrence relations.

```
query :: Query Pattern → MusicPiece → [Pattern]
query (checker, prototype) =
  filter (λp → (prototype ⟺ p) checker)
```

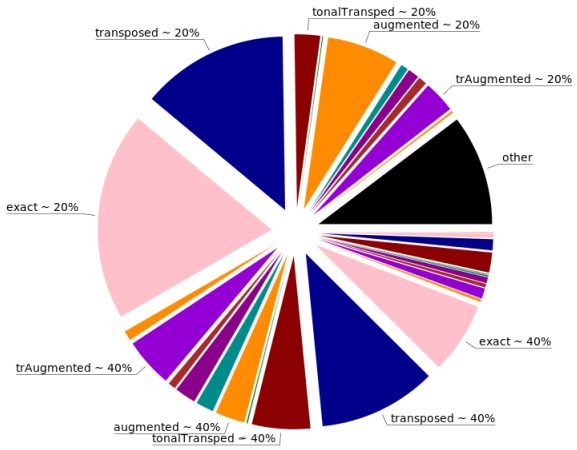

Figure 7. MTC-ANN: transformations in randomized samples. This baseline demonstrates that the proportions of transformations in expert annotations and algorithmic output have different traits than the random samples. Please see more analysis in text.

```
  ∘ slide (length prototype)
    where
      slide :: WindowSize → [a] → [[a]]
      slide n xs = [ take n (drop d xs)
                   | d ← [0 .. (length xs − n `max` 0)]]
```

We also provide a user-friendly interface, where the user specifies the transformation to check in a song from the database with respect to a certain prototype pattern. Then, the sliding window algorithm extracts all occurrences that satisfy the given checker to MIDI files.

```
data UserQuery a = ToPattern a ⇒ Check Pattern :@ a

class ToPattern a where
  toPattern :: a → MusicPiece → Pattern

(⁇) :: ToPattern a ⇒ Song → UserQuery a → IO ()
```

Notice that *UserQuery* is polymorphic over any type that is an instance of the *ToPattern* typeclass, i.e. types that, given a particular song, can designate a musical pattern.

One example of such a type is a pair of time points, indicating start and end time points in the song, as illustrated in the following example query:

```
instance ToPattern (Time, Time) where
  toPattern (startT, endT) =
    takeWhile ((⩽ endT) ∘ ontime)
  ∘ dropWhile ((< startT) ∘ ontime)

testQuery :: IO ()
testQuery = "bach" ⁇ (transpositionOf ≈ 0.8) :@ (21, 28)
```

A more flexible alternative is to immediately provide a piece of music, in which case we use the Euterpea Haskell library that provides a concise musical DSL [Hudak and Quick 2018]. In fact, we piggyback on Euterpea's MIDI export functionality to produce the extracted patterns after discovery.

```
instance ToMusic1 a ⇒ ToPattern (Music a) where
  toPattern _ = musicToPattern
   -- converts a Euterpea music value to our own datatype

testQuery2 :: IO ()
testQuery2 = "bach" ⁇ (transpositionOf ≈ 0.8)
                    :@ line [c 4 qn, e 4 hn, g 4 qn]
```

Time complexity. Although the proposed algorithm has a naive cubic time complexity and is significantly limited to checking patterns of fixed size, we estimate to perform reasonably well in practice since the size of the query N will usually be much smaller than the size of the whole music piece M.

First, note that all transformations mentioned in this paper run in quadratic time, since they consist of linear preprocessing steps which transform the musical data in some way, followed by the quadratic approximation algorithm.

In the case of the pattern discovery algorithm, we always check patterns of equal length N, so each individual check runs in $O(N^2)$. Moreover, we would need to perform $O(M)$ such checks in the average case via the sliding window method. Hence, the worst-case time complexity is $O(M^3)$.

Alas, the expected input size of a query will normally be a simple musical pattern of small constant size. Consequently, we would need to perform $O(M)$ checks, but each one would run in constant $O(1)$ time, resulting in an overall time complexity of $O(M)$.

6 Conclusion

We have presented an expressive DSL to describe (monophonic) music-theoretic transformations, based on simple notions of category theory, namely monoids and contravariant functors. One of the significant benefits of this approach is its compositionality, meaning we can express complex transformations in terms of simpler ones.

The most important such 'base' transformation is the generic notion of approximate equality of lists, that we instantiate for a multitude of music types. We, furthermore, generalise this approximation for musical elements that arise from pairing, moving to a second-order notion of equality that is allowed to combine consecutive elements. We propose a complete quadratic algorithm to perform this

approximation, but also provide a linear variant which is faster under-approximation that we use to produce our results.

Using our transformational framework, we run an analysis of two well-established pattern datasets of classical and folk music, respectively. Since the focus of this paper is on the design and implementation of the framework itself, we only present a general overview of the transformations that algorithms and experts tend to detect.

We further analyse the results of this paper to acquire detailed transformation profiles for each algorithm and the expert annotations, by checking the relative usage of each type of transformation for each set of outputs. These profiles provide the means to further compare algorithmic output against expert annotations, but also a classification of the design space of pattern discovery algorithms, helping us identify key design choices and opportunities to explore new points in this space. For instance, we are able to classify different discovery algorithms according to their transformational profile with reasonable accuracy.

As a final contribution, we present a pattern discovery algorithm that allows us to use our transformation definitions as queries to a given musical piece. Although it suffers from a naive cubic running time complexity, it proves useful in practice with queries of small size. Since the DSL is embedded in Haskell, we can re-use existing music libraries in tandem with our framework, as exemplified in our use of Euterpea musical expressions as queries for pattern discovery.

Future work. One severe limitation of our current solution is that we are limited to analysing monophonic music, but it would certainly be interesting to see how our approach scales with polyphony. Although we could easily incorporate polyphonic pieces by taking simultaneous voices as separate melodic movements, we believe that we would need to consider other types of patterns to get satisfying results. An intuitive step towards this direction would be to model the Neo-Riemannian theory of harmonic transformations [Cohn 1997] using our DSL, possibly extending it to accommodate phenomena we have not anticipated yet.

Glossary

We provide a list of musical transformations and the description we used in this paper (following the format of "full name (short name if it appears in pie chart): description using music terms—further explanation"):

- Exact repetition (exact): repeat an occurrence with exactly the same musical events—horizontal translation, transposition in time.
- Real/Chromatic transposition (transposed): move pitches by a fixed number—vertical translation.
- Tonal/Diatonic transposition (tonalTransped): move pitches in scale degree (major mode only in producing the result of this paper; future work will extend it to other modes)—vertical translation.
- Inversion: change the direction of pitch intervals—horizontal reflection (real/chromatic transposition only in producing the result of this paper, future work will extend it to tonal/diatonic including different modes).
- Retrograde: mirror pitches and durations backwards—vertical reflection.
- Augmentation (augmented): lengthen durations—time scaling, slowing down.
- Diminution: shorten durations—time scaling, speeding up.
- Rotation: change the direction of pitch intervals and mirror note durations backwards—an example composition of transformations, different from a retrograde-inversion in its different first note.

References

Peter Boot, Anja Volk, and W Bas de Haas. 2016. Evaluating the role of repeated patterns in folk song classification and compression. *Journal of New Music Research* 45, 3 (2016), 223–238.

Richard Cohn. 1997. Neo-riemannian operations, parsimonious trichords, and their" tonnetz" representations. *Journal of Music Theory* 41, 1 (1997), 1–66.

Tom Collins. 2013. Discovery of repeated themes and sections. *Retrieved 4th May, http://www. musicir. org/mirex/wiki/2013: Discovery of Repeated Themes & Sections* (2013).

Tom Collins, Andreas Arzt, Sebastian Flossmann, and Gerhard Widmer. 2013. SIARCT-CFP: Improving Precision and the Discovery of Inexact Musical Patterns in Point-Set Representations.. In *ISMIR*. 549–554.

Darrell Conklin. 2010. Discovery of distinctive patterns in music. *Intelligent Data Analysis* 14, 5 (2010), 547–554.

Darrell Conklin and Christina Anagnostopoulou. 2001. Representation and Discovery of Multiple Viewpoint Patterns.. In *ICMC*. Citeseer, 479–485.

Paul Hudak and Donya Quick. 2018. *The Haskell School of Music: From signals to Symphonies*. Cambridge University Press.

Berit Janssen, W Bas De Haas, Anja Volk, and Peter van Kranenburg. 2013. Finding repeated patterns in music: state of knowledge, challenges, perspectives. In *International Symposium on Computer Music Modeling and Retrieval*. Springer, 277–297.

Daan Leijen and Erik Meijer. 2001. Parsec: Direct style monadic parser combinators for the real world. (2001).

Brian McFee, Oriol Nieto, Morwaread M. Farbood, and Juan Pablo Bello. 2017. Evaluating Hierarchical Structure in Music Annotations. *Frontiers in Psychology* 8 (2017), 1337. https://doi.org/10.3389/fpsyg.2017.01337

David Meredith. 2015. Music analysis and point-set compression. *Journal of New Music Research* 44, 3 (2015), 245–270.

Matevž Pesek, Aleš Leonardis, and Matija Marolt. 2017. SymCHM—An Unsupervised Approach for Pattern Discovery in Symbolic Music with a Compositional Hierarchical Model. *Applied Sciences* 7, 11 (2017), 1135.

Iris Yuping Ren, Hendrik Vincent Koops, Dimitrios Bountouridis, Anja Volk, Wouter Swierstra, and Remco C Veltkamp. 2018a. Feature analysis of repeated patterns in dutch folk songs using principal component analysis. *FMA* 14, 5 (2018), 86.

Iris Yuping Ren, Hendrik Vincent Koops, Anja Volk, and Wouter Swierstra. 2018b. Investigating Musical Pattern Ambiguity in a Human Annotated Dataset. *The Proceedings of the 15th International Conference on Music Perception and Cognition and the 10th triennial conference of the European Society for the Cognitive Sciences of Music* (2018), 361–367.

Iris Yuping Ren, Vincent Koops, Anja Volk, and Wouter Swierstra. 2017. In Search Of The Consensus Among Musical Pattern Discovery Algorithms. *Proceedings of the International Society for Music Information Retrieval* (2017).

Peter van Kranenburg, Berit Janssen, and Anja Volk. 2016. The Meertens Tune Collections: The Annotated Corpus (MTC-ANN) versions 1.1 and 2.0. 1.

Gissel Velarde, David Meredith, and Tillman Weyde. 2016. A wavelet-based approach to pattern discovery in melodies. In *Computational Music Analysis*. Springer, 303–333.

Anja Volk and Peter van Kranenburg. 2012. Melodic similarity among folk songs: An annotation study on similarity-based categorization in music. *Musicae Scientiae* 16, 3 (2012), 317–339.