HarmLang: A Probabilistic Language for Music Notation, Manipulation, and Analysis

Abstract

Over the history of computers, many attempts have been made to create languages and libraries that express music, with varying levels of success. Some musical encodings are ambiguous; others too limited, or too verbose. There is nearly always a tradeoff between the clarity of an encoding and its expressive power. As an alternative to existing languages and libraries, we present HarmLang, a domain specific language embedded in Haskell for musical composition, manipulation, and probabilistic analysis designed to be intelligible to those with limited programming experience, but powerful enough to satisfy the needs of experienced programmers. Alongside the language, we provide a formal mathematical model for the types of music on which HarmLang is capable of operating.

1 Introduction

1.1 A Brief History of Computers and Music

Until the 1970s, music notation was performed almost exclusively by hand. Through a millennium of development, handwritten music on scores became a powerful, beautiful, and universally understood tool for the expression and distribution of music. However, scores suffer from limited modularity and low information density. Some notation is ambiguous or nonstandard, and human interpretation is often necessary.

Since computers became widespread, composers and arrangers turned to software like Finale and Sibelius to create scores. Digital musical formats became the standard. Devices began to support the MIDI format. Meanwhile, musical libraries and languages began to develop, with mainly technical users.

LilyPond (Nienhuys and Nieuwenhuizen 2003) is a language used mainly for score typesetting in the way LaTeXis used mainly for typesetting text. While LilyPond is Turing complete, much like latex performing musical calculations is nontrivial and not a standard use case.

Euterpea is a combinator library built in Haskell for music development (Hudak 2014). Designed for a variety of uses, Euterpea is an environment for algorithmic composition, and analysis, MIDI output, audio processing and instrument design. It aims to be a solution for any musical use.

The library is very expressive, but its power comes at a cost – it is difficult to approach for new users. Its implementation as a combinator library in Haskell lends itself to a complex system of type constructors that make the task of composition more difficult than would a language with syntactic support.

1.2 Goals

In developing HarmLang, we wanted to appeal to both musicians and programmers. Musicians may want to notate compositions, generate arrangements, and transpose and transform their music. Programmers may want to make probabilistic programs, leverage libraries, and use powerful type systems and functions.

With seemingly disparate target audiences, we had to think carefully about our syntax, so that it was basic enough for non-technical users, yet extensible enough for power users.

Based on those constraints, we had the following goals in mind as we developed HarmLang:

- Strong Haskell integration, so that the language can be used in a natural way with Haskell code and libraries.
- Extensive, but logical type system and initial basis, so that users can take advantage of a wealth of modular boilerplate code.
- Conciseness and simplicity, as well as familiarity to musicians.
- Expressiveness, with the ability to encode most Western music.
- Integration with MIDI, a widely-recognized format for encoding timed musical events. Though not human readable, MIDI is supported by a variety of electronic musical devices and programs.
- Close integration with probabilistic musical analysis.

1.3 Formal Definition of Music in HarmLang

People have attempted to formalize musical systems for nearly as long as music itself has been played. In designing HarmLang, we tried to use a model that was general enough to encode a wide variety of existing music, yet simple for users to understand and build upon. In this section, we introduce the musical representation we developed and the many design tradeoffs we encountered in creating it.

Formal definitions of PITCHCLASSes, INTERVALs, and HARMONYs are given in Figure 1. The PITCHCLASS and INTERVAL definitions are relatively straightforward and uncontroversial throughout the domain of 12 tone[3] music.

[2]Transposition is formally defined by the $\mathbb{Z}_{12}$ isomorphism, but can be informally thought of as adding an interval to a PITCHCLASS, and interpreting the result as a pitch class.

[3]In fact, these definitions can easily be generalized to arbitrary equal tempered musical schemes, such as "24 tone equal tempered", a scheme used in some forms of music originating in India, however notions of Western harmonies are tightly coupled with the 12 tone equal tempered scale (and similar systems), so it would not be

1. **Pitch Class**: There are 12 pitch classes in music. Differences between them are measured in intervals, and with the transposition[1] operation these classes form a group isomorphic to $(\mathbb{Z}_{12}, +)$.

2. **Interval**: Intervals are the differences between pitch classes[2], and thus are also elements of $\mathbb{Z}_{12}$. Intervals are similar to distances in the space of pitch classes, but are not true distances because they are not symmetric, indeed if we denote the interval from a to b as i, the interval from b to a is $12 - i$.

3. **Harmony**: Harmonies can be thought of a collection of unique pitch classes P where $|P| \geq 1$, and one element of P is denoted the root. This definition is isomorphic to a root pitch class and a set of intervals, representing the intervals from the root pitch class to each of the remaining elements in P, and this second formulation is the one used in HarmLang.

Figure 1: Formal definitions for PITCHCLASSes, INTERVALs, and HARMONYs.

The definition for a HARMONY, on the other hand, is potentially controversial, and somewhat limits the expressive power of HarmLang. In HarmLang, all chords must be rooted. However, it is sometimes valuable to think of a chord as a collection of notes without an explicit root pitch. For that reason, a rest (silence), which can be thought of as 0 notes being played cannot be represented, though HarmLang provides a way to represent silence as a `Chord` type outside of this model: see Section 3.1. Representing a HARMONY as a root PITCHCLASS and a set of intervals from that root is often more useful for the sake of analysis than the "set of notes" model. For instance, using a root and set of intervals to characterize a chord trivializes the task of transposition[4]. This property becomes extremely useful for implementing and proving probabilistic features of the language, particularly key agnosticism (Section 2.2.5).

2 Features and Abstractions

2.1 Haskell Integration

HarmLang is a domain specific language strongly embedded in Haskell, and the symbiosis is such that HarmLang cannot survive outside of Haskell. A HarmLang embedding in another language would result in code that would look *very* different, as a lot of HarmLang code is simply the application of Haskell operations on HarmLang types (which are, at the end of the day, Haskell types as well).

HarmLang has a rich type system, elucidated in Section 3.1, and several typeclasses with associated algebraic laws and rules for typeclassing of composite types, presented in Section 3.2. In addition to these core features, HarmLang has built in support for easy to use probabilistic programming abstractions, explained in Section 2.2. Finally, HarmLang supports syntax for extremely concise notation of constants that resembles traditional music notation whenever possible through QuasiQuotation. The QuasiQuotation grammar is discussed in the appendix, Figure A8.

particularly useful to do so.

[4]Transposition on a chord by some interval is the act of transposing every pitch class in the chord by said inverval.

2.2 Probabilistic Programming in HarmLang

In this section, we discuss the probabilistic elements of HarmLang. Sufficient treatment of this material is given such that it should be digestible to those without prior exposure to probabilistic programming, assuming a reasonable background in the mathematics of probability, however the authors recommend that interested parties refer to (Ramsey and Pfeffer 2002) for further information.

2.2.1 Finite Discrete Probability Models

A finite discrete probability distribution represents a probability distribution over a finite but not necessarily bounded set of discrete values. Such a distribution is used to answer queries along the lines of "what is the probability of seeing some chord", or, "what is the probability of seeing a chord that satisfies some predicate. For a complete list of `Dist` operations and explanations, please consult the appendix.

2.2.2 Conditional Probabilities

Probability distributions are a useful primitive, but we want to ask questions of the form "What is the probability of seeing chord γ given that we have just seen chords "α, β". This conditional information is referred to as the evidence, so for evidence of type e and a distribution of type d, a conditional distribution has type `e -> Dist d`.

In HarmLang, the most useful conditional distribution is of type `[Chord] -> Dist Chord`, where `[Chord]` represents a series of chords, and the `Dist` represents the probability distribution of chords that follow the series.

2.2.3 Markov Chain Models

A very natural model for representing conditional probabilities of the form `[a] -> Dist a` is a higher order Markov model, which has notes representing ordered lists of `a`, and gives transition probabilities to other `a`. In HarmLang, this is achieved through the use of the `HarmonyDistributionModel` abstraction, which builds a kth order Markov Model of chords (Markov 1906), and can be querried for chord distributions. Relevant types include:

```
buildHarmonyDistributionModel ::
  Int -> [ChordProgression] -> HDM
distAfter ::
  HDM -> ChordProgression -> Dist Chord
```

A k value for the higher order Markov model and an array of chord progressions (to be subdivided into kmers) are used to build the model, and the model can be queried for distribtions. The `HDM` could be removed so that the build function would return a conditional distribution (`ChordProgression -> Dist Chord`, but in the future `HDM` will support additional functionality so this is not desirable.

2.2.4 Problem Scale and Tractability

The space of possible chords is very large: there are possible 12 root notes, and 11 intervals from which to choose combinations, yielding by basic combinatoric mathematics a total of $12 * \mathcal{P}(\{1, ..., 11\}) = 24576$ possible chords. These large distributions are extremely slow to operate on (as some of the probabalistic algorithms run in $\theta(n^2)$ time), and they are also quite large. The problem is exacerbated exponentially when conditional probabilities over chord progressions are considered, as in the Markov chain model, because the distribution for *each possible condition* must be stored, and conditions are cartesian products of chords, thus for a model with some kmer size k, $(12 * 2^{11})^{k+1}$ possible probabilities exist. As one might imagine, this quickly becomes intractable.

In practice, only a tiny fraction of the 2^{11} possible chord types are used, and many transitions between these chords are rarely if ever used. Careful choice of a prior that does only includes a subset of the possibilities and lazy evaluation yield problems that are solveable in reasonable time.

2.2.5 The Key Agnosticism Property

The HarmonyDistributionModel supports an import domain specific optimization that shrinks the state space of conditional chord distributions by 12 and allows the data used to build a `HarmonyDistribtionModel` to be more effectively utilized. Based on the musical assumption that listeners hear *relative* pitches in a piece of music, but are much less sensitive to *absolute* pitches, we came up with the *key agnosticism property*, which is defined by the following algebraic law:

Suppose α is a conditional distribution, β is a `Interval`, and γ is a `[Chord]`. If a has the key agnosticism property, then $\alpha(\text{transpose}^5\beta\gamma) == \text{transpose}b(ac)$.

The proof of this property for `HDM`s is ommitted due to space concerns, but the utility of this property cannot be understated: this means that information gleaned from a piece in one key can be applied to a piece in another key, allowing training data used to build a `HDM` to be more effectively utilized.

3 Implementation

HarmLang is implemented as an embedded domain specific language in Haskell. The parser is written in Parsec (Leijen and Meijer 2001), and QuasiQuotation (Mainland 2007) is used to allow HarmLang code inline with Haskell programs.

3.1 Types

The HarmLang type system is designed to simultaneously be easy for those unfamiliar with Haskell to use as well as powerful and flexible in the hands of a Haskellite. To achieve a compromise, we designed a complex web of types around our musical model. See Figure 2 for a comprehensive description of the types exposed to users of HarmLang.

The primitive types are composed solely of base Haskell types, and represent primitive musical concepts. These primitives include PITCHCLASSes, INTERVALs, TIMEs, and DISTs (finite probability distributions)[6].

The composite types are algebraic data types composed of primitive types and other composite types. PITCHes are composed of a PITCHCLASS (e.g. "B") and an OCTAVE (e.g. "5"), representing a PITCHCLASS at a given OCTAVE (e.g. "B5"). NOTEs consist of a PITCH played for a duration of TIME, though they do not have a "start time" and "end time" – they simply have a length. CHORDs are defined in a way that closely resembles their description in Section 1.3. As a sum type, a CHORD can either be a root and interval set, or a STRINGs. This string can be used to represent concepts that don't fit directly into the model, such as a rest, metadata in a chord progression, or something else entirely. TIMEDCHORDs are simply CHORDs played for a duration of TIME; CHORD is to TIMEDCHORD as PITCH is to NOTE. Finally, the CHORDDISTRIBUTION type represents a distribution over chords. The string representation of chords is not supported in the distributions, meaning that the space of all chords used in distributions is finite. Please refer to Section 2.2.4 for information on the size of the space.

The list types are simply aliases to improve readability. It is much easier to understand a type signature that contains a `ChordProgression` than one that contains a `[Chord]`. In the first case, it is clear that we mean a progression of temporally related chords. In the second, there is no implied timing. The same logic applies to TIMEDCHORDPROGRESSION and NOTEPROGRESSION. Finally, we represent CHORDTYPES as `[Interval]`, and we want a clearer name to reference them.

3.2 Typeclasses

[6] It is a bit strange to think of something so complicated as a finite probability distribution, with its generic type, as a primitive. However, the typeclass rules introduced in Section 3.2.2 work well under this classification schema. Furthermore, classifying DIST as a primitive frees us entirely from thinking about its complex representation.

[7] Including a generically typed argument in a generically typed type constructor, see CHORDPROGRESSIONDISTRIBUTION for an example of this.

HarmLang primitive types

P 1. `data PitchClass = PitchClass Int`
PITCHCLASS is {`Data, Typeable, Show, Eq, Enum, Transposable`}
P 2. `data Interval = Interval Int`
INTERVAL is {`Data, Typeable, Show, Eq, Ord, Enum, Transposable`}
P 3. `data Octave = Octave Int`
OCTAVE is {`Data, Typeable, Show, Eq, Ord, Enum`}
P 4. `data Time = Time Int Int`
TIME is {`Data, Typeable, Show, Eq, Ord, Timed`}
P 5. `data Dist a = Dist [(a, Double)]`
DIST is {`Eq, Show)`}

HarmLang composite types

C1. `data Pitch = Pitch PitchClass Octave`
PITCH is {`Data, Typeable, Show, Eq, Ord, Transposable`}
C2. `data Note = Note Pitch Time`
NOTE is {`Data, Typeable, Show, Eq, Transposable, Timed`}
C3. `data Chord = Harmony PitchClass [Interval] | Other String`
CHORD is {`Data, Typeable, Show, Eq, Ord, Enum, Transposable)`}
C4. `data TimedChord = TimedChord Chord Time`
TIMEDCHORD is {`Data, Typeable, Show, Eq, Timed, Transposable)`}
C5. `type ChordDistribution = Dist Chord`
CHORDDISTRIBUTION is `Transposable`

HarmLang list types

L 1. `type ChordProgression = [Chord]`
CHORDPROGRESSION overrides {`Show`}
L 2. `type TimedChordProgression = [TimedChord]`
TIMEDCHORDPROGRESSION is {`MIDIable`} overrides {`Show`}
L 3. `type NoteProgression = [Note]`
NOTEPROGRESSION is {`MIDIable`} overrides {`Show`}
L 4. `type ChordType = [Interval]`
CHORDTYPE overrides {`Show`}

Figure 2: Haskell type declarations and typeclasses for all HarmLang types

Here we define the rules for typeclasses on composite HarmLang types. In the following definitions, the symbol for the class relation element of ($\in$) in the context A $\in$ B is used both to denote that A is a component of type B, and also to denote that type A is an instance of typeclass B. With context, this dual use is unambiguous.
We define A a component of B to mean either that A is an argument of a type constructor[7] or that recursively A is a component of an argument of a type constructor of B.

1. SHOW:
   $\forall$ A $\in$ B, A $\in$ SHOW $\Rightarrow$ B $\in$ SHOW
2. EQ:
   $\forall$ A $\in$ B, A $\in$ EQ $\Rightarrow$ B $\in$ EQ
3. ORD:
   A $\in$ C, A $\in$ ORD, $\forall$ B $\in$ C $\setminus\{$A$\}$, B $\notin$ ORD $\Rightarrow$ C $\in$ ORD
4. TRANSPOSABLE:
   A $\in$ TRANSPOSABLE, A $\in$ B $\Rightarrow$ B $\in$ TRANSPOSABLE
5. TIMED:
   A $\in$ C, A $\in$ TIMED, $\forall$ B $\in$ C $\setminus\{$A$\}$, B $\notin$ TIMED $\Rightarrow$ C $\in$ TIMED

Figure 3: HarmLang typeclass rules

Typeclasses are used in HarmLang, partly for the usual reason: to abstract functionality in a manner that allows code reuse. However, they are also used to allow those with little programming experience to write code in a natural, straightforward way.

Consider, for example, the TRANSPOSABLE typeclass. When a user learns that she can transpose a NOTE, she may also want to transpose a PITCH or even a TIMEDCHORDPROGRESSION. Fortunately, with identical syntax, she can. Even without an understanding of types and typeclasses, the HarmLang typeclasses help users to work with the language.

In sections 3.2.1 and 3.2.2, we discuss the typeclasses of HarmLang and rules for their use in composite types.

3.2.1 Introduction to HarmLang Typeclasses

1. DATA, TYPEABLE: In all cases, DATA and TYPEABLE are used together. The instancing is handled automatically with `deriving` through the {-# LANGUAGE DeriveDataTypeable #-} language extension. DATA and TYPEABLE are used to work with the PATQ and EXPQ types, which are used in the QuasiQuotation module. See (Sheard and Jones 2002) and (Mainland 2007) for more information.
2. SHOW: For code-writing simplicity, all the HarmLang types implement SHOW. In order to avoid confusing users with alternate syntaxes, we manually implemented each SHOW such that the following (slightly informal) algebraic law holds: For some HarmLang type T, if T has a corresponding INTERPRET function in the INTERPRETER module, the INTERPRET function is the inverse. Note that this inverse is one way, since there are many possible `Strings` that HarmLang interpreters map to the same value of the SHOW function. Aside from further hiding the Haskell representation of HarmLang types from inexperienced Haskellites, our methods have the added benefit that when objects are printed, they still appear in HarmLang syntax.
3. EQ: Many HarmLang types are finite and thus trivially comparable for equality. Even HarmLang types that are not finite have a computable [8] test for equality, so we implement and expose it.
4. ORD: Many HarmLang types are also well ordered. One interesting exception is the PITCHCLASS type; we could easily impose an ordering on $\mathbb{Z}_12$ but in music it is not useful to do so.
5. ENUM: In HarmLang, the ENUM typeclass is instanced for all *finitely* enumerable types. TIMEs and `Octaves` are not finite[9], and as such do not instance ENUM. The ENUM typeclass is very useful, as it enables syntactic sugar of the form:

```
allChords :: [Chord]
allChords = [ [hl| 'A[]' |] ..
  (Chord [hl| 'G#' |]
  (map Interval [0..11]) ]
```

   This feels very natural, is familiar to Haskell users, and the additional syntactic support on HarmLang primitives makes HarmLang feel more like a language and less like a library. This code example is taken from the initial basis, and represents a list uniquely containing every chord representable by HarmLang.
6. TRANSPOSABLE: Any pitched musical entity can be transposed by any interval. Transposition can be thought of as addition in $\mathbb{Z}_{12}$ on pitched entities, and is a very natural operation for those with musical backgrounds. TRANSPOSABLE provides a `transpose` function to allow this, and is defined as follows:

```
class Transposable a where
  transpose :: a -> Interval -> a
```

   (Please consult Section 5.1.4 for an explanation of the perverse argument ordering used in TRANSPOSABLE.)
   In addition, all transpose functions are required to satisfy the following algebraic law:
   $\alpha \in$ INTERVAL, $\beta \in$ TRANSPOSABLE $\Rightarrow \beta =$ `(transpose (inverse`[10] β`)) (transpose` $\beta\,\alpha$`)`
7. TIMED: Any timed entity can be queried for its time. It can also be transformed to an entity with a new time that is exactly the same in every other regard. The TIMED typeclass is defined as follows:

```
class Timed a where
  askTime :: a -> Time
  reTime :: a -> Time -> a
```

8. MIDIABLE: The MIDIABLE typeclass provides functionality to convert a HarmLang object to a MIDI repre-

[8] Computable, but usually quite efficient, except in the case of DIST, which is comparable in $\theta(n \log n)$ time

[9] In HarmLang theory, INTs and DOUBLEs are treated as though they were not finite, though this is not the case.

[10] Here `inverse` denotes the inverse operator for the group of INTERVALs. See section 3.3 for more detail.

sentation, which can be played by external software or loaded into external programs or hardware.

3.2.2 Typeclass Rules

These formal typeclass rules provide a calculus by which given only a knowledge of the typeclasses of primitive types, and recursive knowledge of the types (but not necessarily the typeclasses) of all types used in a type's type constructors, the typeclasses of a composite type may be calculated. The calculus is provided in Figure 3, and is elucidated here.

The typeclass rules only mandate that a HarmLang type *must* implement a typeclass; the lack of a rule does not imply that the HarmLang type *cannot* implement a typeclass.

The SHOW rule simply states that if all components of a type implement SHOW, so should the type itself. The EQ rule similarly states that if all components of a type implement EQ, the type shall do so as well. These rules capture the intuition that if the components of a type can be shown or tested for equality, the type should do the same, usually in a componentwise manner (though not necessarily). In, all HarmLang primitives instance both typeclasses, it therefore follows inductively by the type rules for each that all HarmLang composites shall instance both typeclasses as well. ORD is a more complicated case, as there is no standard and intuitive manner by which to order an algebraic type of more than a single type if more than one of them is ordered itself. As an example, take the complex numbers: though both the real and the imaginary portions are ordered, there is not a standard definition for the ordering of the $\mathbb{C}$, even though one *could* define an ordering. In HarmLang, only types which contain *exactly* one ordered component need be ordered. The TRANSPOSABLE rule is more flexible, it simply mandates that if at least one component of a type is TRANSPOSABLE, than so too shall be the type. This captures the intuition that the transposable components should be transposed, as in a NOTE or a CHORD in the note set representation. Finally, the TIMED rule, like the ORD rule applies to types with a single TIME component. This is because the `askTime` function of the `Time` type class doesn't generalize logically to objects with more than one timed component.

These rules capture intuition about how each typeclass should persist, and it is strongly recommended that any HarmLang library providing additional composite types follow them so the expected typeclasses are provided.

The List types do not have the same set of formal rules, as what they represent is more flexible, however they do all provide custom SHOW implementations (to allow `show` output to match HarmLang syntax, which was accomplished through the use of the {`-# LANGUAGE OverlappingInstances #-`} language extension.

A similar logical rule exists for `Enum`, but it cannot be formulated in terms of components like the above (because a component that does not instance `Enum` may or may not be infinitely enumerable). Stated in informal English, the rule is as follows: "That which has at least one enumerable component, and no infinitely enumerable components, shall itself be finitely enumerable". This can be achieved by taking the cross product of the finite set of each component, and using it to create a new mapping from integers onto the values of the type. Perhaps with more rigorous typeclasses, this rule could be formalized.

3.3 Initial Basis

The HarmLang language goes beyond the QuasiQuotation sublangage (which is really only useful for pattern matching and value entry), and provides a complex web of types, typeclasses, and associated functions. A large initial basis is also provided to perform common operations on HarmLang types. A few useful selections from the initial basis are discussed here, though the actual initial basis is much larger.

```
inverse :: Interval -> Interval.
```

The `Inverse` function is defined such that the following algebraic law holds: `Transposable` α and `Interval` β, `transpose` (`transpose` β (β)) α = α. Thus transpose returns the group inverse of any interval.

```
toChord :: PitchClass -> [PitchClass]
-> Chord
```

and

```
getNotesFromChord :: Chord
-> [PitchClass]:
```

These functions convert between the two representations of a chord discussed in Figure 1. If the list produced by `getNotesFromChord` is split into `head` and `tail`, then `toChord` is its inverse. The converse of the previous statement is also true.

```
isInversion :: Chord -> Chord -> Bool.
```

This function determines if two chords contain the same notes (but do not necessarily have the same root). The following algebraic law holds:

`isInversion` α β $\Rightarrow$ (==) [] ((

) (`getNotesFromChord` α) (`getNotesFromChord` β))

4 Evaluation

HarmLang is a DSL intended to be used for several things throughout the musical domain. We created a few example programs to show HarmLang's effectiveness at its intended goals.

4.1 Example Programs

4.1.1 Notation

While it is difficult to objectively gauge the effectiveness of HarmLang's notation without a third-party study, we would like to offer a comparison with two different musical notations with HarmLang's syntax.

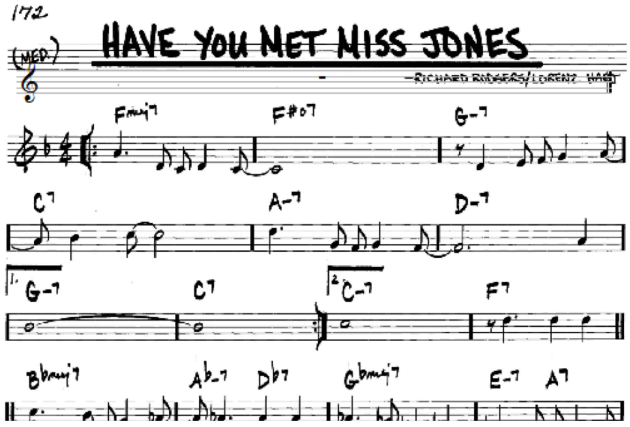

Image from The Real Book (Various 2004).

```
a1 = [hl|[FMa7:4 F#o7:4 Gm7:4 C7:4 Am7:4
Dm7:4 Gm7:4 C7:4]|]

a2 = [hl|[FMa7:4 F#o7:4 Gm7:4 C7:4 Am7:4
Dm7:4 Cm7:4 F7:4]|]

b  = [hl|[BbMa7:4 Abm7:2 Db7:2 GbMa7:4
Em7:2 A7:2 DMa7:4 Abm7:2 Db7:2 GbMa7:4
Gm7:2 C7:2]|]

a3 = [hl|[FMa7:4 F#o7:4 Gm7:4 C7:2 Bb7:2
Am7:4 D7:4 Gm7:4 C7:4 FMa7:4 Gm7:4 C7:4]|]

progression = a1 ++ a2 ++ b ++ a3
```

```
\chordmode {
  \startChords
  \startSong
  \myMark "A"
  \startPart
  f1:maj7 | fis:dim7 | g:m7 | c:7 | \myEndLine
  a:m7 | d:m7 | g:m7 | c:7 | \myEndLine
  \endPart
  \myMark "A"
  \startPart
  f:maj7 | fis:dim7 | g:m7 | c:7 | \myEndLine
  a:m7 | d:m7 | c:m7 | f:7 | \myEndLine
  \endPart
  \myMark "B"
  \startPart
  bes:maj7 | aes2:m7 des:7 | ges1:maj7 |
  e2:m7 a:7 | \myEndLine
  d1:maj7 | aes2:m7 des:7 | ges1:maj7 |
  g2:m7 c:7 | \myEndLine
  \endPart
  \myMark "A"
  \startPart
  \repeat volta 2 {
        f1:maj7 | fis:dim7 | g:m7 |
        c2:7 bes:7 | \myEndLine
        a:m7 d:7.9- | g:m7 c:7 |
  } \alternative {
        {
          f d:m7 | g:m7 c:7 |
        }
        {
          f1*2 | \myEndLine
        }
  }
  \endPart
  \endSong
  \endChords
}
```

Figure 4: The popular Jazz ballad “Have You Met Miss Jones” in score, LilyPond, and HarmLang notation.

While scores and LilyPond both serve purposes somewhat different from HarmLang, HarmLang is clearly more concise and intuitive than LilyPond and more modular than scores. For a comparison, see Figure 4.

4.1.2 Stylistic Inference

To evaluate the utility of HarmLang as a DSL for probabalistic music analysis, we crafted a program to solve the stylistic inference problem in HarmLang. We define the Stylistic Inference Problem as follows:

"Given labeled chord progressions by various artists, determine the authorship of additional instances (chord progressions)."

This problem statement is based on a similar problem in (De León et al. 2004), where Ponce de León et al "develop a system able to distinguish musical styles from a symbolic representation of melodies (digital scores) using shallow structural features, like melodic, harmonic, and rhythmic statistical descriptors." We characterized the problem in terms of chord transitions so it could be an inference problem in the probabilistic sense.

We constructed a database of over 400 jazz songs from the Vanilla Book (Patt 2014), and defined our styles as music produced by the following authors: *Jerome Kern*: 13, *Duke Ellington*: 13, *Antonio-Carlos Jobim*: 10, *George Gershwin*: 16. We then performed cross validation (Stone 1974) where we held out 3 progressions by each artist and used the remainder to train harmony distribution models. Cross validation was repeated 4 times, for a total of 48 classifications.

Using the trained models, we calculated that each model generated each chord progression, and took the artist associated with the most likely model to be the artist that composed the piece. Using this technique, we were able to achieve 54% accuracy, and expected accuracy by random guessing would be only 25%, so we conclude that our techniques worked given only a very small amount of training data, though not with very high accuracy.

The full code for the stylistic inference problem is available in the appendix, figure A10, but the majority of it is boilerplate[11] and IO. We were able to express the inference code itself in a tiny amount of code, which speaks to the success of the language in simplicity and brevity. The relevant code appears in Figure 5

HarmLang saved us both from having to write a parser for the input data and from having to write a conditional probability system, and additionally we benefited from HarmLang's domain specific optimizations, such as the Key Agnosticism Property (Section 2.2.5). We were able to express a very complicated program in a very concise amount of code, so at least for specific applications such as this, the authors feel that it is safe to say that HarmLang is a successful language.

[11] At present, it is a bit more difficult than we would like to operate on a HDM, and priors are more difficult to work with than we would like, though there is a simpler build function for those who wish to use an HDM without a prior that is much easier to use.

5 Conclusion

5.1 Further Work

Though many features of the language have been built, there are several features we have not yet fully developed.

5.1.1 Tool Support

One of the central features that is missing from HarmLang is integration with tools. One such tool would be a syntax highlighter for HarmLang.

5.1.2 Antiquotation

In the current iteration of HarmLang, we support quasiquotation (Mainland 2007) for both expressions and pattern matching, however we do not support antiquotation. In this section, we discuss the implications of supporting antiquotations, with mockup examples of what antiquotation could potentially look like, and what the advantages would be.

Expression quasiquotation is extremely useful for defining musical constants, and pattern matching can be used to match on specific instances of the HarmLang types, however antiquotation would allow much more complicated HarmLang programs to be notated concisely through pattern matching. See Figure 6 for an example of functions utilizing quasiquoted pattern matching with and without antiquotation.

At times antiquotation in expressions might be marginally useful, particularly in progressions, but it seems as though the same effect could be achieved by splitting the quasiquoted progression into 2 about the value to antiquote, making the value a single element array, and concatenating the three reslting arrays. This solution has minimal syntactic overhead, and is probably not a common use case. More complicated antiquotation, such as antiquotation of components of a HarmLang value, as in Figure 7, may be more useful.

All things considered, antiquotation would ease the creation of more complicated expression code, and more complicated pattern matching code, but would be only marginally useful in notation and probabilistic computation. While it would be "nice to have", the core of HarmLang functions very well without it and rarely does it seem like a "missing feature."

5.1.3 Probabilistic Music Generation

The initial formulation for HarmLang also included probabilistic chord progression generation. This is one area where the host language actually got in the way of our vision: as a pure functional language, random number generation is a difficult process that either involves passing around random number generators or monadic code (See Figure A11 for a brief overview of random number generation in Haskell).

The work of David Cope (Cope 2005) in probabilistic music generation was quite inspirational to the authors, and being able to do similar things in HarmLang using probabilis-

```
probProgGivenModel :: HarmonyDistributionModel -> ChordProgression -> Probability
probProgGivenModel hdm@(HDMTC thisK _ _) prog = product (map (\ (kmer, nextVal) ->
  probv (distAfter hdm kmer) nextVal )  (sliceKmersWithLastSplit thisK prog) )

 generating the given progression.
inferStyle :: [HarmonyDistributionModel] -> ChordProgression -> [Probability]
inferStyle models prog = map (\ model -> probProgGivenModel model prog) models
```

Figure 5: Stylistic Inference Problem: Relevant Code

Pattern Matching without antiquotation:

```
chordGrump :: Chord -> String
chordGrump [hl| 'Am7' |] = "I don't like that chord."
chordGrump [hl| 'B7' |] = "I don't like that chord either."
chordGrump _ = "I don't know what that chord is, but I probably don't like it."
```

Pattern Matching with antiquotation:

```
chordGrump :: Chord -> String
chordGrump [hl| '$(A, _)' |] = "I don't like chords rooted in A."
chordGrump [hl| '$(_, 7)' |] = "I just hate the sound of 7th chords."
chordGrump [hl | '$a' |] = "No, not the dreaded " ++ (show a) ++ " chord!"
                        ++ "That's my least favorite!"
```

Figure 6: Pattern Matching with and without antiquotation.

tic music generation was an attractive idea. In order to cater to the other goals of the language and allow those with little programming experience to pick up HarmLang quickly, we decided early on avoiding monadic code at all costs was necessary. The motto of mo' 'nads mo' problems, while it disallowed generation functions, allowed us to create a much more digestible language. However, it is important to note that nothing is stopping libraries and programs to create that do use sampling: it has been done in Haskell, and all of necessary types and functions to extend HarmLang in such a manner are exposed.

5.1.4 Design Mistakes in HarmLang

Overall the design of the HarmLang types proved to work very well, but some decisions were made that we later came to regret. A solid refactoring effort would fix these, though we do have some "legacy code" that would be broken by such changes.

Transposable definition Early on in the development, we defined the transpose typeclass like so:

```
class Transposable a where
  transpose :: a -> Interval -> a
```

This should immediately send up a red flag to any experienced Haskell developer: inverting the first two arguments would yield a transpose function with a *significantly* more useful partial application: indeed, partial application as it stands is near useless, and our code base is filled with `(flip transpose)` to overcome this deficiency.

MIDIable

A few mistakes were made in the design of the MIDIable interface. Relevant code from `IO.hs`:

```
class MIDIable a where
    makeTrack :: a -> Track Ticks
    outputToMidi :: a -> String -> IO ()

writeMidi::[Track Ticks]->FilePath->IO()
```

There's nothing obviously wrong with this typeclass, but it could be improved by making it more generic. If each type of progression implementing `MIDIable` implemented another typeclass of converting it to a generic form capable of representing the same types of music that all the `MIDIable` instances represent, we could implement MIDI functionality *as well as score generation* functionality, and any other similar output functions, on top of this representation. This is far more extensible, and the contract exposed for common use could be identical. The typeclass and associated functions would look something like this:

```
class Noteable a where
    toNoteCollection :: a -> [[Note]]

npToMidi  :: [[Note]] -> Track Ticks
writeMidi::Track Ticks->String->IO()

midiOut::(Noteable a)=>a->String->IO()
midiOut=writeMidi.npToMidi.toNoteCollection

scoreOut::(Noteable a)=>a->String-> IO()
--Similar implementation
```

`MIDIable` is an interface which certainly works for what it was intended to do, but a more general interface could be more useful and extensible.

Antiquotation into an expression, and equivalent code without antiquotation.

```
[hl| [Am7 Dm7 $a G7] |]
concat $ [hl| [Am7 Dm7] |] [a] [hl| [G7] |]
```

Partial type antiquotation

```
makeMajorChord :: PitchClass -> Chord
makeMajorChord root = [hl| $(Chord root M) |] --M for major, equivalent to [4 7]
```

Figure 7: Expressions with and without antiquotation.

ChordType

In HarmLang, we have the definition `type ChordType = [Interval]`. This is logical, it allows us to represent a chord type, which is really just a collection of intervals (as explained in Figure 1 and defined in 3.1) as a ChordType, which allows users to show intent with a more meaningful type name. However, in truth a chord type is not just an array of intervals: for I the set of intervals, the set of valid chord types are $\mathcal{P}(I \setminus \{0\})$ (where $\mathcal{P}$ denotes the power set). The generic `[Interval]` representation denotes an infinite set, as duplications are allowed, and 0 intervals are also allowed. Furthermore, for `ChordType` to be useful, it must be comparable. `ChordType` implements `Eq`, and the order of the intervals does not matter (because they are all relative to the root). To allow this operation to be efficient, HarmLang `ChordTypes` are assumed to be sorted. As you can probably see, there are a myriad of illegal `ChordTypes`, and it would be good Haskell to not allow the representation of these (or at least check for it).

If `ChordType` were a fully fledged algebraic data type, these properties could be enforced by obscuring the type constructor and forcing interaction through functions (though this would hurt pattern matching on chords). This would also allow `ChordTypes` to be stored and compared more efficiently as integers (note that `ChordTypes` are enumerable, implement the `Enum` typeclass, and require only $\log_2(2^{11}) = 11$ bits to store. This would be a significant performance improvement to HarmLang, which would in turn allow users to create more complicated probabalistic programs and get answers faster *without worrying about representation*. This is very in line with the goals of HarmLang, so this would overall be a nice update to the language.

Note

Taken from the `Note` definition:

```
data Note = Note Pitch Time
--TODO At some point we want to add
--the option for a rest.
```

The TODO says it all, rests (a musical concept where nothing is played for a set amount of time) need to be added. The updated type definition would be as follows:

```
data Note = Note Pitch Time | Rest Time
```

This would allow rests to be represented in `NoteProgressions`, and is absolutely necessary for the `Noteable` typeclass defined above to function properly. It would be a small change, but to implement this, we would need an extra case in every pattern match on `Note`. Other than the slightly increased code body necessary, there are no downsides to this change, we simply didn't get to it.

Interpreter

The `Interpreter` module in HarmLang allows runtime parsing of HarmLang syntax into HarmLang values. This is an extremely useful idea, particularly for interactive programs, because user entry matches HarmLang syntax, freeing program writers from having to interpret user input and providing a uniform syntax. Furthermore, unlike more complicated language like SQL, JavaScript and Python, where such things are gaping security flaws, HarmLang only allows value definition expressions and pattern matches, so unless someone actually wrote a program to interpret, say, a chord progression as code, there is no security cost to interpreting user input at runtime. The only downside of the `Interpreter` module is that it has the exact functionality of the `Read` typeclass, and is thus unnecesarily confusing to those accustomed to Haskell.

Acknowledgments

Kathleen Fisher for educating us in language design, Haskell, and the tools necessary to build an embedded language.

Norman Ramsey for his work on the Probability monad, and his lectures on the largely isomorphic Dice World finite discrete probability system that the probabilistic aspects of HarmLang were based on.

Caleb Malchik for tirelessly labor on the probabilistic parts of the HarmLang language. Caleb was instrumental in the implementation of `Dist`, and also in the program described in Section 4.1.2.

Jayme Woodgerd for her advice on programming musical systems in Haskell, and her code for Improvise, which served as a reference for MIDI code.

References

David Cope. *Computer models of musical creativity*. MIT Press Cambridge, 2005.

Pedro J Ponce De León, Carlos Pérez-Sancho, and José M Inesta. A shallow description framework for musical style recognition. In *Structural, Syntactic, and Statistical Pattern Recognition*, pages 876–884. Springer, 2004.

Paul Hudak. *The Haskell School of Music – From Signals to Symphonies*. (Version 2.6), January 2014.

Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

AA Markov. Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. izv fiz-matem obsch kazan univ (series 2) 15: 135–156. see also:"extension of the limit theorems of probability theory to a sum of variables connected in a chain". *Appendix B of R. Howard "Dynamic Probabilistic Systems*, 1, 1906.

Han-Wen Nienhuys and Jan Nieuwenhuizen. Lilypond, a system for automated music engraving. *Colloquium on Musical Informatics*, May 2003.

Ralph Patt. *The Vanilla Book*. http://www.ralphpatt.com/VBook.html, August 2014.

Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. *ACM SIGPLAN Notices*, 37(1):154–165, 2002.

Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

M. Stone. Cross-validatory choice and assessment of statistical predictions. *J. Royal Stat. Soc.*, 36(2):111–147, 1974.

Various. *The Real Book*. Hal Leonard Publishing Corp, Milwaukee, WI, sixth edition edition, 2004.

System.Random

```
class RandomGen g where
  next :: g -> (Int, g)
  split :: g -> (g, g)
  genRange :: g -> (Int, Int)
  --returns the range of the generator

getStdRandom::(StdGen->(a,StdGen))->IO a
--Oh my, a Monad!
```

There are also Probability and Sampling monads, but they are arguably even more difficult to use, as the IO monad is presumably already in use in any HarmLang program.

Figure 11: *Appendix*: Techniques for Random and Pseudo-random Number Generation in Haskell

A1 Appendix

A1.1 Grammar

HarmLang syntax is powered by Haskell Quasiquotation, which allows HarmLang code to be used in line with Haskell programs. An EBNF (De León et al. 2004) grammar for the HarmLang quasiquotation sublanguage is provided in Figure 8.

There are some interesting quirks to the HarmLang grammar, which are enumerated here. Lists are implemented as Haskell lists, and as such are homogeneous (thus one may note that the above grammar there exist programs that are grammatical but not valid due to typing restrictions).

A second interesting point is that # and b work as the successor and predecessor functions, respectively, of a given pitch class. The letters "`A`" through "`G`" can be thought of literal values in the space. Note all values in $\mathbb{Z}_{12}$ can be directly expressed as literals without the use of the successor or predecessor functions, however due to the finite cyclic nature of the group of pitch classes, all 12 pitch classes can be described with a successor or predecessor and even a single literal.

A1.2 Probabilistic Elements

This `Dist` type is a generic distribution type for probability distributions over finite discrete sets, and in HarmLang it is usually used through the `ChordDistribution` synonym, which is simply a `Dist Chord`. The operations supported on the `Dist` type are given in Figure A9.

Example probabilistic code for the probabilistic inference problem is provided in 10.

⟨*whitespacechar*⟩ = " " | "\t" ;
⟨*whitespace*⟩ = ⟨*whitespacechar*⟩, {⟨*whitespacechar*⟩} ;
⟨*optionalwhitespace*⟩ = {⟨*whitespacechar*⟩} ;

⟨*digit*⟩ = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
⟨*uint*⟩ = ⟨*digit*⟩, {⟨*digit*⟩} ;
⟨*int*⟩ = ["-"], ⟨*uint*⟩ ;

⟨*noteletter*⟩ = "A" | "B" | "C" | "D" | "E" | "F" | "G" ;
⟨*modulator*⟩ = "#" | "b" ;

⟨*pitchclass*⟩ = ⟨*noteletter*⟩ | ⟨*pitchclass*⟩, ⟨*modulator*⟩ ;

⟨*pitch*⟩ = ⟨*pitchclass*⟩, "@", ⟨*int*⟩ ;

⟨*timeunit*⟩ = ⟨*uint*⟩, "/", ⟨*uint*⟩ | ⟨*uint*⟩ ;

⟨*note*⟩ = ⟨*pitch*⟩, ":", ⟨*timeunit*⟩ ;

⟨*interval*⟩ = ⟨*int*⟩ ;

⟨*rest*⟩ = "_" | "REST" | "SILENCE" ;
⟨*begin*⟩ = "BEGIN" | "START" ;
⟨*end*⟩ = "END" ;
⟨*specialharmony*⟩ = ⟨*rest*⟩ | ⟨*begin*⟩ | ⟨*end*⟩ ;
⟨*namedchord*⟩ = "5" | "M" | "m" | "b5" | "dim" | "o" | "7" | "Ma7" | "m7" | "mMa7" | "m7b5" | "dim7" | "o7" | "9" | "b9" | "#9" | "Ma9" | "m11" | "7b6" | "13" | "Ma13" ;
⟨*chord*⟩ = ⟨*pitchclass*⟩, ("[]" | "[", {⟨*interval*⟩, ⟨*whitespace*⟩}, ⟨*interval*⟩, "]")
| ⟨*pitchclass*⟩, ("[]" | "[", {⟨*pitchclass*⟩, ⟨*whitespace*⟩}, ⟨*pitchclass*⟩, "]")
| ⟨*pitchclass*⟩, ⟨*namedchord*⟩
| ⟨*specialharmony*⟩ ;

⟨*timedchord*⟩ = ⟨*chord*⟩, ":", ⟨*timeunit*⟩ ;

⟨*parsabletype*⟩ = ⟨*interval*⟩ |⟨*pitchclass*⟩ | ⟨*pitch*⟩ | ⟨*note*⟩ | ⟨*chord*⟩ | ⟨*timedchord*⟩ ;

⟨*singleton*⟩ = ⟨*optionalwhitespace*⟩, "′", ⟨*parseabletype*⟩, "′", ⟨*optionalwhitespace*⟩ ;

⟨*progression*⟩ = ⟨*optionalwhitespace*⟩, ("[]" | "[", {⟨*parseabletype*⟩, ⟨*whitespace*⟩}, ⟨*parseabletype*⟩, "]"), ⟨*optionalwhitespace*⟩ ;

Figure 8: *Appendix*: EBNF grammar for the HarmLang quasiquotation sublanguage.

Constructors:

```
certainly :: (Eq a) => a -> Dist a
choose :: (Eq a) => Double -> Dist a -> Dist a -> Dist a
equally :: (Eq a) => [a] -> Dist a
weightedly :: (Eq a) => [(a, Double)] -> Dist a
```

Transformers:

```
pmap :: (Eq a, Eq b) => (a -> b) -> Dist a -> Dist b
-- Joint distribution where events of type b depend on events of type a.
-- Each (A, B) node holds the probability P(A ^ B) = P(A ^ B | A) * P(A).
jointdep :: (Eq a, Eq b) => Dist a -> (a -> Dist b) -> Dist (a, b)
-- Joint disribution for two independent events
joint :: (Eq a, Eq b) => Dist a -> Dist b -> Dist (a, b)
-- Filters a distribution for some predicate.
pfilter :: (Eq a) => (a -> Bool) -> Dist a -> Dist a
```

Observers:

```
-- Probability of a value in a distribution.
probv :: (Eq a) => Dist a -> a -> Double
-- Prob of seeing a value that matches a given predicate
prob :: (Eq a) => (a -> Bool) -> Dist a -> Double
-- Expectation of a function over the distribution
expected :: (Eq a) => (a -> Double) -> Dist a -> Double
-- Returns all a with nonzero probability in the distribution
support :: Dist a -> [a]
-- Gives a list of values paired with the probs, sorted
likelylist :: Dist a -> [(a, Double)]
-- Returns the most likely a in the distribution.
maxlikelihood (Eq a) -> Dist a -> a
```

Figure 9: *Appendix*: Exposed functions on the `Dist` type

```haskell
import HarmLang.Types
import HarmLang.InitialBasis
import HarmLang.ChordProgressionDatabase
import HarmLang.HarmonyDistributionModel
import HarmLang.Priors

import Data.List
import Data.Maybe

--Calculates P(progression | HarmonyDistributionModel),
--or probability of generating a progression from a generative model.
probProgGivenModel :: HarmonyDistributionModel -> ChordProgression -> Probability
probProgGivenModel hdm@(HDMTC thisK _ _) prog = product (map (\ (kmer, nextVal) -> probv
  (distAfter hdm kmer) nextVal ) (sliceKmersWithLastSplit thisK prog) )

inferStyle :: [HarmonyDistributionModel] -> ChordProgression -> [Probability]
inferStyle models prog = map (\ model -> probProgGivenModel model prog) models

--map with index
mapInd :: (a -> Int -> b) -> [a] -> [b]
mapInd f l = let
    mapIndH f [] _ = []
    mapIndH f (a:as) i = (f a i):(mapIndH f as ((+) i 1))
  in
    mapIndH f l 0

-- groups progressions in a CPD by artist, denoting the artist with a string
getByArtist :: ChordProgressionDatabase -> [(String, [TimedChordProgression])]
getByArtist cpd = (getProgressionsCategorizedByCriterion cpd "Artist")

-- extracts from a list of categories the n with the most progressions
getTopCategories :: Int -> [(String, [TimedChordProgression])] ->
  [(String, [TimedChordProgression])]
getTopCategories n = (take n) . reverse . sortGroupsBySize

artists :: [String]
artists = ["Antonio-Carlos Jobim", "Duke Ellington", "George Gershwin", "Jerome Kern"]
getTestArtists :: ChordProgressionDatabase -> [(String, [TimedChordProgression])]
getTestArtists db = filter (\ (name, cps) -> (elem name artists)) (getByArtist db)

splitTrainingTest :: Int -> [[ChordProgression]] -> ([(ChordProgression, Int)],
  [[ChordProgression]])
splitTrainingTest tSize db = (concat $ mapInd (\ l index -> map (\ q -> (q, index))
  (take tSize l)) db, map (drop tSize) db)

-- build harmony distribution models, and some cool priors to boot.
makeHdms :: [[ChordProgression]] -> [[ChordProgression]] -> [HarmonyDistributionModel]
makeHdms allData hdmData =
  let
    k = 3
    -- all transitions between chords in db are equally likely
    priorPrior = chordLimitedLaplacianPriorFromDb $ concat allData
    -- HDM of all data in db
    prior = hdmPrior $ buildHarmonyDistributionModelWithPrior k priorPrior 1.0 (concat hdmData)
  in
    map (\thisHdmData -> buildHarmonyDistributionModelWithPrior k prior 1.0 thisHdmData) hdmData

-- gives the name of each category followed by a colon followed by number of
-- progressions in that category. each entry separated by newline.
summary :: [(String, [TimedChordProgression])] -> String
summary ([]) = ""
summary ((s,l):rest) =  s ++ ": " ++ (show $ length l) ++ "\n" ++ (summary rest)

main :: IO ()
main = do
  putStrLn "Please enter the path to the database."
  path <- getLine
  cpd <- loadChordProgressionDatabase (if path == "" then "./res/progressions.txt" else path)
  let topClasses = (getTestArtists cpd)
  putStrLn $ "Top Classes:\n" ++ (summary topClasses)
  let (test, training) = splitTrainingTest 3 (map ((map toUntimedProgression) . snd) topClasses)
  let hdms = makeHdms (map ((map toUntimedProgression) . snd) (getByArtist cpd)) training

  putStrLn $ concat (map (\ (prog, classIndex) -> "Class " ++ (show classIndex) ++ ", " ++
    ("rank " ++ (show $ getRank (inferStyle hdms prog) classIndex)) ++ ", " ++
    (show $ inferStyle hdms prog) ++ "\n") test )

getRank :: (Ord n, Num n) => [n] -> Int -> Int
getRank l i = fromJust $ Data.List.elemIndex (l !! i) (reverse $ Data.List.sort l)
```

Figure 10: *Appendix*: Stylistic Inference Problem Code