Music Theory with Dependent Types (Experience Report)

ANONYMOUS AUTHOR(S)

Throughout history, music has been composed following general rules and guidelines, expressed informally through natural language and examples. The expressiveness of dependent type theory allows us to capture these rules formally, and then use them to automate analysis and synthesis of music.

In this experience report, we explore expressing a small subset of the rules of the common practice period in Agda, a functional programming language with full dependent types. We focus on the construction of species counterpoint as well as four-part harmonization of melody. We point out both the advantages of using dependent types to express music theory and some of the challenges that remain to make languages like Agda more practical as a tool for musical exploration.

CCS Concepts: • **Applied computing → Sound and music computing**; • **Software and its engineering → Functional languages**; • **Theory of computation → Type theory**.

Additional Key Words and Phrases: dependent types, counterpoint, harmony

ACM Reference Format:
Anonymous Author(s). 2020. Music Theory with Dependent Types (Experience Report). *Proc. ACM Program. Lang.* 4, ICFP, Article 1 (August 2020), 13 pages.

1 INTRODUCTION

Edgar Varèse describes music as "organized sound", and throughout history cultures have developed and applied systems of rules and guidelines to govern the music they create, most notably the Common Practice Period of Western music spanning the 17th to early 20th centuries. These rule systems are seldom absolute, and indeed deliberate breaking of the rules is often part of the aesthetic, but they roughly constrain the music they apply to and give it a common form and sound.

Artists and theoreticians have attempted to capture and codify these rule systems, informally in natural language, and typically accompanied by examples from the existing literature. The intent is both to analyze existing music and then to use these principles to guide the creation of new music, in other words for synthesis.

Starting in the 20th century, computers have become ubiquitous in music in every area, including sound synthesis, composition and production [Roads et al. 1996]. In terms of music theory, there has been a line of recent work on using functional programming for harmonic analysis [De Haas et al. 2011, 2013; Magalhães and de Haas 2011], harmonization of a melody and generation of melodies based on a harmonization [Koops et al. 2013; Magalhães and Koops 2014] and counterpoint [Szamozvancev and Gale 2017]. There is also an established Haskell library Euterpea [Hudak and Quick 2018] for general music and sound exploration.

To describe rules of basic harmonic structure, Magalhães and de Haas [2011] and their successors use dependent types, for example to index chords by major or minor mode. However Haskell currently has limited support for dependent types, and requires many extentions and tricks such as the use of singleton types [Eisenberg and Weirich 2013].

In this paper we explore what can be done in the context of music by using a programming language that offers full dependent types. We use Agda [Norell 2007] since it is fairly mature and aims to be both a functional programming language and a proof assistant. It also features a Haskell FFI so we can take advantage of existing Haskell libraries, in particular for MIDI generation.

2020. 2475-1421/2020/8-ART1 $15.00
https://doi.org/

Full dependent types allow expression of predicate logic, and it is tempting to take a standard textbook on music theory such as Piston and DeVoto [1987] or Aldwell and Cadwallader [2018] and formalize it in type theory. As a first step towards this goal, we start with the modest task of expressing a small, relatively strict rule set known as species counterpoint [Fux 1965], intended for combining interdependent melody lines to produce a pleasant-sounding result. Roughly speaking, what we do is to write a custom musical type checker in Agda. Compared to the previous work by Cong and Leo [2019], where rules are expressed directly as Agda types, our approach makes it easier to describe fine-grained rules, produce readable error messages, and add or drop rules depending on the circumstances.

We also present preliminary work on harmonizing a melody using a subset of rules based on Piston and DeVoto [1987], contrasting with existing work by Koops et al. [2013]. Since counterpoint and harmony are not separate concepts but in fact deeply intertwined, we would wish to reuse counterpoint rules to develop natural-sounding harmonizations. We show that, with our custom type checker, it is straightforward to reuse rules in different contexts. Put differently, our representation of rules satisfies modularity.

The rest of this paper structured as follows. Section 2 introduces basic musical concepts as well as their Agda representation. Sections 3 and 4 describe our formalization of counterpoint and harmony, focusing on the composable and modular aspects of the proposed approach. Lastly, Section 5 concludes the paper with future perspectives.

This is an experience report, and we highlight both the advantages and disadvantages of using Agda for music theory. On one hand the expressiveness of dependent types makes it easy and natural to describe music theory rules. On the other hand we find the emphasis on proof construction and particularly the extra work needed for decidable equality can add an extra burden which is not always welcome. However overall we feel the positives far outweigh the negatives, and in the final section we discuss possible ways to reduce the tedium.

2 BASIC MUSICAL TERMS AND THEIR REPRESENTATION IN AGDA

Pitches. The most basic concept in melody is *pitches*, which tell us how high or low a tone is. We represent pitches as natural numbers, but throughout the paper, we use a more readable notation `name octave`, where `name` ranges over `c`, `d`, `e`, etc., and `octave` denotes which octave the pitch belongs to. For instance, the middle C is represented as `c 5`.

Scale Degrees. While pitches uniquely determine the position of a tone on e.g. a piano keyboard, *scale degrees* stand for the relative position of a tone in a scale. For instance, a diatonic scale consists of seven scale degrees, which corresponds to C, D, E, F, G, A, B in the case of C major. We represent scale degrees as a datatype with seven constructors `d1` - `d7`.

Duration. For rhythm, the fundamental concept is *duration*, an unspecified unit of time during which a sound or silence lasts. We represent duration as a natural number, and use this value to calculate the absolute length when the music is played at a specific tempo. As an example, the duration constant `whole` has value `16` and corresponds to 4 seconds if the tempo is 60 beats per minute.

Intervals. An *interval* represents the difference in pitch between two notes. There are 13 kinds of interval within an octave (Figure 1), and these intervals can be classified from several different perspectives: (i) major or minor; (ii) consonant or dissonant; and (iii) perfect or imperfect. We represent intervals as natural numbers, but in this paper, we refer to them using names such as `maj3` and `per8`, which are more reader-friendly.

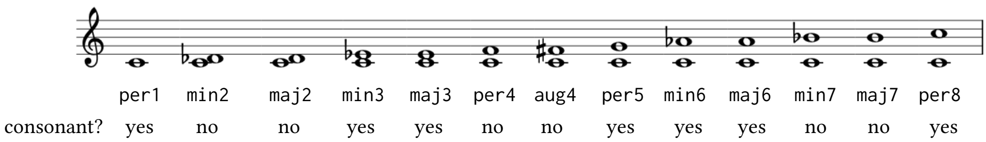

Fig. 1. 13 Kinds of Intervals

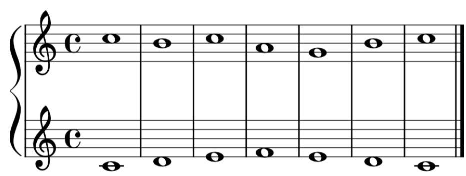

Fig. 2. First Species Counterpoint

Chords and Triads. A *chord* refers to a set of simultaneously sounding notes. In this paper we are particularly interested in *triads*, which are chords consisting of three notes separated by the 3rd intervals (i.e., `maj3` and `min3`). We define triads as a datatype with roman numeral constructors representing the scale degree of the lowest note (called the *root*). The other two notes are then members of the scale a third and a fifth above the root. For instance, the triad `I` in C major corresponds to the chord C-E-G. Triads can also be arranged in a way that a non-root note appears as the lowest note. Such arrangement is called *inversion*; if a triad is not applied inversion, it is in *root position*.

3 COUNTERPOINT

Counterpoint is a technique for combining multiple lines of melodies. Composing counterpoint is like arranging a song for choir: we start with a *cantus firmus*, which serves as the base melody, and compose a counterpoint line above or below the cantus firmus. When doing this, we must make sure that the whole music sounds harmonically pleasing, and that the individual melodic lines are distinguishable to the listener.

In this section, we present an implementation of species counterpoint, based on the formulation given by Fux [1965]. The idea is to represent "good" counterpoint as a dependent record, whose fields encode the musical content as well as proofs that the counterpoint follows certain rules. Due to the space limitation, we only show some exerpts of our implementation; see our anonymous supplementary material `Counterpoint.agda` for full details.

3.1 First Species Counterpoint

First species counterpoint is the simplest variant of counterpoint. In first species, we set one note against each note in the cantus firmus, which is required to start with a tonic (the first note of a scale, such as C in C major) and consist only of whole notes. Figure 2 shows an example of first species counterpoint. The lower line is the cantus firmus, which we excerpt from a German song called *Froschgesang* (Frog's song). The upper line is the counterpoint composed by the second author.

3.1.1 Representing Music. In our implementation, we represent each bar as a pitch-interval pair `(p , i)`, where `p` is the pitch of a cantus firmus note, and `i` is the interval between `p` and the corresponding counterpoint note. We then represent a sequence of bars as a list of pitch-interval pairs, but with one proviso: we separate the first and last bars from the middle bars. Therefore, the Frog's song in Figure 2 is represented as a compound of the following three elements:

```
first : PitchInterval -- Pitch × Interval
first = (c 5 , per8)

middle : List PitchInterval
middle = (d 5 , maj6) :: (e 5 , min6) :: (f 5 , maj3) ::
         (e 5 , min3) :: (d 5 , maj6) :: []

last : PitchInterval
last = (c 5 , per8)
```

3.1.2 *Representing Rules.* The reason behind our three-part representation of music is that different parts are subject to different rules, as we detail below.

Beginning. The beginning of the music should express perfection. As we saw in Section 2, there are three intervals that are classified as perfect: the 1st (usually called the *unison*), 5th, and 8th. The first interval of the music must then be one of these intervals. In our formalization, we implement this rule as the `checkBeginning` function, which reports an error `not158 i` when the first interval `i` is an invalid one. Since Agda does not have exceptions, we turn the error into a `Maybe` value by wrapping it around the `just` constructor.

```
data BeginningError : Set where
  not158 : PitchInterval → BeginningError

checkBeginning : PitchInterval → Maybe BeginningError
checkBeginning pi@(_ , i) =
  if ((i == per1) ∨ (i == per5) ∨ (i == per8))
  then nothing
  else just (not158 pi)
```

Intervals. The middle bars of the music should maintain consonance of intervals and independence of melodic lines. As we saw in Figure 1, consonant intervals include the unison, 3rd, 5th, 6th, and 8th, and among these, the unison is clearly an obstacle to distinguishing between the two lines of music. Therefore, the middle bars must consist only of the latter four intervals. We encode this rule as the `checkIntervals` function, which returns a list of errors corresponding to the occurrences of dissonant intervals and unisons.

```
data IntervalError : Set where
  dissonant : Interval → IntervalError
  unison    : Pitch    → IntervalError

intervalCheck : PitchInterval → Maybe IntervalError
intervalCheck (p , i) with isConsonant i | isUnison i
intervalCheck (p , i) | false | _    = just (dissonant i)
intervalCheck (p , i) | _     | true = just (unison p)
intervalCheck (p , i) | _     | _    = nothing

checkIntervals : List PitchInterval → List IntervalError
checkIntervals = mapMaybe intervalCheck
```

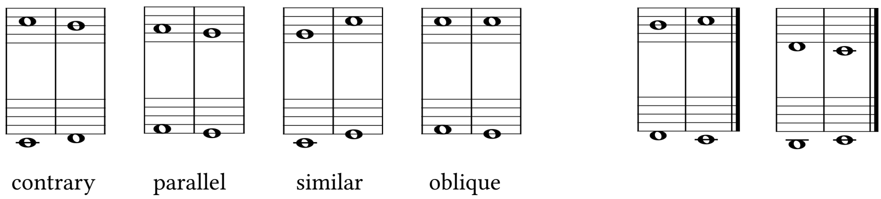

Fig. 3. Motion (left) and Cadence (right)

Motion. The independence of melodic lines is also affected by *motion*, i.e., the way one interval moves to another interval. As Figure 3 left shows, there are four kinds of motion: *contrary* (two lines go in different directions), *parallel* (two lines go in the same direction by the same distance), *similar* (two lines go in the same direction by different distances), and *oblique* (one line plays the same note). It is known that approaching a perfect interval by parallel or similar motion tends to fuse the melodic lines into one. To rule out such motion, we define the `checkMotion` function, which inspects all pairs of two adjacent intervals (using an auxiliary function `pairs`) and returns a list of erroneous patterns.

```
data MotionError : Set where
  parallel : PitchInterval → PitchInterval → MotionError
  similar  : PitchInterval → PitchInterval → MotionError

motionCheck : PitchInterval → PitchInterval → Maybe MotionError
motionCheck i1 i2 with motion i1 i2 | isPerfect (proj₂ i2)
motionCheck i1 i2 | contrary | _     = nothing
motionCheck i1 i2 | oblique  | _     = nothing
motionCheck i1 i2 | parallel | false = nothing
motionCheck i1 i2 | parallel | true  = just (parallel i1 i2)
motionCheck i1 i2 | similar  | false = nothing
motionCheck i1 i2 | similar  | true  = just (similar i1 i2)

checkMotion : List PitchInterval → List MotionError
checkMotion = mapMaybe (uncurry motionCheck) ∘ pairs
```

Ending. The ending of the music should express relaxation. There is a consensus that the unison and 8th are the most stable intervals, hence the music must end with either of these intervals. The last interval should also be approached by a *cadence*, a progression that gives rise to a sense of resolution (Figure 3 right). This in turn suggests that a valid ending requires the middle bars to be non-empty. We encode these rules as the `checkEnding` function, which, upon finding an invalid ending, reports one of the three possible errors.

```
data EndingError : Set where
  not18    : PitchInterval       → EndingError
  not27    : PitchInterval       → EndingError
  tooShort : List PitchInterval  → EndingError

endingCheck : PitchInterval → PitchInterval → Maybe EndingError
endingCheck pi1@(pitch p , i) (pitch q , interval 0)  =
  if ((p + 1 ≡ᵇ q) ∧ (i == min3)) then nothing else just (not27 pi1)
```

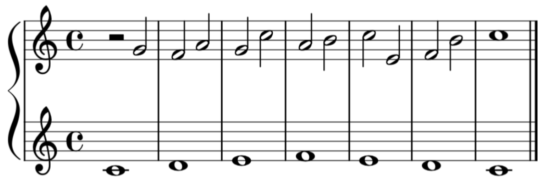

Fig. 4. Second Species Counterpoint

```
endingCheck pi1@(pitch p , i) (pitch q , interval 12) =
  if ((q + 2 ≡^b p) ∧ (i == maj6) ∨ (p + 1 ≡^b q) ∧ (i == min10))
  then nothing
  else just (not27 pi1)
endingCheck pi1               pi2                    =
  just (not18 pi2)

checkEnding : List PitchInterval ⟶ PitchInterval ⟶ Maybe EndingError
checkEnding []        _ = just (tooShort [])
checkEnding (p :: []) q = endingCheck p q
checkEnding (p :: ps) q = checkEnding ps q
```

3.1.3 Putting Things Together. Using the encoding of music and rules we have seen so far, we define `FirstSpecies`, a record type inhabited by correct first species counterpoint. The first three fields of this record type represent the first, middle, and last bars, respectively. The last four fields stand for the proofs that the music satisfies all the required properties.

```
record FirstSpecies : Set where
  constructor firstSpecies
  field
    firstBar    : PitchInterval
    middleBars  : List PitchInterval
    lastBar     : PitchInterval
    beginningOk : checkBeginning firstBar ≡ nothing
    intervalsOk : checkIntervals middleBars ≡ []
    motionOk    : checkMotion (firstBar :: middleBars) ≡ []
    endingOk    : checkEnding middleBars lastBar ≡ nothing
```

With this record type, we can show that the counterpoint in Figure 2 is correct, since the music is an inhabitant of `FirstSpecies`.

```
fs : FirstSpecies
fs = firstSpecies first middle last refl refl refl refl
```

3.2 Second Species Counterpoint

We next discuss a more complex variant of counterpoint, called the second species. In second species, we set *two* half notes against every cantus firmus note. This gives rise to the distinction between *strong* beats (the first interval in a bar) and *weak beats* (the second interval). Figure 4 is an example of two-against-one counterpoint, again composed for the Frog's song.

3.2.1 Representing Music. In second species, the first and last bars may have a different structure from middle bars. More specifically, it is encouraged to begin the counterpoint line with a half rest and end with a whole note. Therefore, in our implementation, we reuse the `PitchInterval` type for the first and last bars, and define a new type `PitchInterval2` for middle bars.

```
first2 : PitchInterval
first2 = (c 5 , per5)

middle2 : List PitchInterval2 -- Pitch × Interval × Interval
middle2 =
  (d 5 , min3 , per5) :: (e 5 , min3 , min6) :: (f 5 , maj3 , aug4) ::
  (e 5 , min6 , min3) :: (d 5 , min3 , maj6) :: []

last2 : PitchInterval
last2 = (c 5 , per8)
```

3.2.2 Representing Rules. The rules for second species counterpoint can be obtained by tweaking those for first species and adding a few new ones. Here we go through the rules without showing the corresponding Agda functions, as they are largely similar to what we defined for first species.

Beginning. The beginning of the music may be either the 5th or 8th, but *not* the unison, as it prevents the listener from recognizing the beginning of the counterpoint line.

Strong Beats. Strong beats in middle bars are constrained by the same rules as in first species: they must all be consonant, non-unison intervals.

Weak Beats. Weak beats are allowed to be dissonant if they are created by a *passing tone*, i.e., a note in the middle of two step-wise motions in the same direction (as in bars 4-5 of Figure 4). They may also be the unison if they are left by step in the opposite direction from their approach (as in bars 5-6 of Figure 4).

Motion. Parallel and similar motion towards a perfect interval is prohibited across bars.

Ending. The last interval must be the unison or 8th, preceded by an appropriate interval that constitutes a cadence structure.

3.2.3 Putting Things Together. Now we define `SecondSpecies`, a record type inhabited by correct second species counterpoint. As in `FirstSpecies`, we have three fields holding the musical content, followed by five fields carrying the proofs of the required properties discussed above[1].

```
record SecondSpecies : Set where
  constructor secondSpecies
  field
    firstBar      : PitchInterval
    middleBars    : List PitchInterval2
    lastBar       : PitchInterval
    beginningOk   : checkBeginning2 firstBar ≡ nothing
    strongBeatsOk : checkStrongBeats middleBars ≡ []
    weakBeatsOk   : checkWeakBeats middleBars (secondPitch lastBar) ≡ []
    motionOk      : checkMotion2 (firstBar ::
                                  (expandPitchIntervals2 middleBars)) ≡ []
    endingOk      : checkEnding2 middleBars lastBar ≡ nothing
```

[1]The auxiliary function `secondPitch` in `weakBeatsOk` extracts the counterpoint note of a given interval, and `expandPitchIntervals2` turns a list of `PitchInterval2`s into a list of `PitchInterval`s.

Using SecondSpecies, we can show that the second species counterpoint in Figure 4 is correct.

```
ss : SecondSpecies
ss = SecondSpecies first2 middle2 last2 refl refl refl refl refl
```

3.3 Comparison with Previous Work

The type-theoretical formalization of counterpoint has first been attempted by Szamozvancev and Gale [2017]. They encode the rule for intervals by declaring a type class `ValidHarmInterval` that classifies valid intervals, and then specifying which instances are invalid using GHC's support for custom type errors. They also implement the restriction on motion based on a similar idea.

The availability of the custom type error feature is an advantage of using Haskell: it allows the rule writer to leverage Haskell's type checker. We believe that extending Agda with such support would be beneficial in music formalization and for many other purposes.

More recently, Cong and Leo [2019] give an implementation of first species counterpoint using dependent types in Agda. Their idea is to represent correct counterpoint as a list-like datatype, where the base cases enforce a valid ending and the inductive case guarantees correct uses of intervals and motion (by means of implicit arguments representing proofs).

Our representation of counterpoint improves on Cong and Leo's in two ways. First, while Cong and Leo rely on the Agda type checker to report errors, we use our own type checker (implemented as the `checkXXX` functions) and type errors (defined as the `XXXError` datatypes). This allows us to produce more user-friendly error messages. Suppose we have replaced the second interval of Figure 2 by an octave (8th) of D, breaking the rule that perfect intervals cannot be approached by parallel motion. In our implementation, this causes an error in the `motionOk` field, with the following message:

```
(parallel (c 5 , per8) (d 5 , per8) :: []) != [] of type (List MotionError)
```

In contrast, Cong and Leo would just report the existence of an unsolved metavariable `_13`, meaning that Agda failed to construct a proof required by the counterpoint constructor. Thus, the user would see the following message in the Agda buffer:

```
_13 : motionOk (c 5 , per8) (d 5 , per8)
```

For a non-expert user, it may be difficult to connect this message to the corresponding musical error.

The second improvement from Cong and Leo [2019] is that, instead of incorporating all the rules into a counterpoint datatype, we define each rule as a separate function and combining them using a record type. This makes it easy to switch between strict and relaxed rule sets: we just need to add or remove certain fields. It also allows us to reuse some of the rules in a context other than counterpoint, as we will see in the next section.

4 HARMONY

Harmony refers to simultaneously sounding tones (chords) and its study is primarily concerned with the progression of chords throughout a piece of music. It is in essence the of dual of counterpoint. In this section we formalize a small part of the classic text Piston and DeVoto [1987], in particular part of Chapter 9 which is concerned with techniques for harmonizing a given melody.

4.1 Harmonic Progression

The primary Example 9.1 in Piston's text (Figure 5 left) shows a sample four part SATB harmonization in C major. The melody is the series of highest notes, i.e., the soprano (S) line, while the other three parts alto (A), tenor (T), and bass (B) are the melodies below it. Roman numerals denote the names of the triads at each vertical slice. Note that even though there are four voices there are only

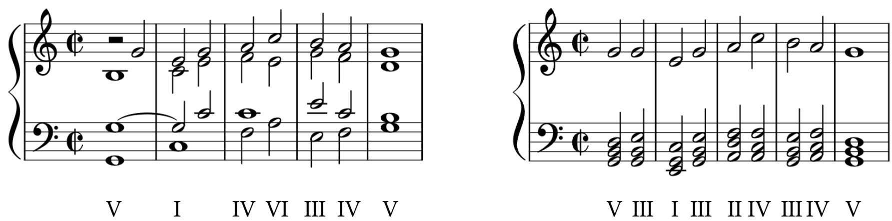

Fig. 5. Harmonization by Piston (left) and FHARM (right)

three distinct pitches sounding at each vertical slice (where tones an octave apart are considered to have the same relative pitch). As is typical the root of the triad is always doubled. Also note that all triads are in root position (meaning the root is in the bass voice), as Piston restricts to this case initially in the chapter (he does relax it later). The goal of a harmonization is both to create an aesthetically pleasing chord progression as well as fluid melodies in each voice (called voice leading) and independence of each music line (in other words good counterpoint). In this chapter Piston gives advice on how to achieve these goals.

We would like to formalize some of this advice and apply it to automate harmonization of melodies. We start with selection of the sequence of triads, in other words the harmonic progression. Earlier in the text (on page 23) Piston presents a Table of Usual Root Progressions, noting which triads usually follow a given one and which are sometimes used or rare. It is easy to encode this in Agda using the `Triad` datatype from Section 2.

```
record NextTriad : Set where
  constructor nextTriad
  field
    usual : List Triad; sometimes : List Triad; rare : List Triad

rootProgression : Triad → NextTriad
rootProgression I   = nextTriad (IV :: V :: []) (VI :: []) (II  :: III : [])
rootProgression II  = nextTriad (V :: []) (IV :: VI :: []) (I :: III :: [])
rootProgression III = nextTriad (VI :: []) (IV :: []) (I :: II :: V :: [])
rootProgression IV  = nextTriad (V :: []) (I :: II :: []) (III :: VI :: [])
rootProgression V   = nextTriad (I :: []) (IV :: VI :: []) (II :: III :: [])
rootProgression VI  = nextTriad (II :: V :: []) (III :: IV :: []) (I :: [])
rootProgression VII = nextTriad (I  :: III :: []) (VI :: []) (II :: IV  :: V :: [])
```

It is easiest to create a harmonic progression in reverse, to ensure that one ends a musical phrase with either a full (V–I) or half (V) cadence, and work backward from there. So we would like to invert this table to give a list of likely triads to precede a given triad. This would normally be straightforward but here we see a disadvantage of using Agda. As is typical with languages based on dependent type theories, equality is a very subtle concept and even determining if a term is a member of a list is nontrivial.

In Haskell this would be easy—there is an `Eq` typeclass and a method for automatically generating equalities. In the Agda Standard Library [Agda Developers 2020] list membership is a relation with constructive evidence of membership, and requires far more effort we would like given that we currently don't need to do proofs. Alternatives include defining decidable equality for triads (which requires writing out 49 cases) or mapping to natural numbers and using already implemented decidable or boolean equality for those. For now we simply write the inverse function manually, excluding the rare cases.

```
previousTriads : Triad → List Triad
previousTriads I   = V :: IV :: VII :: []
previousTriads II  = VI :: IV :: []
previousTriads III = VI :: VII :: []
previousTriads IV  = I :: V :: II :: III :: []
previousTriads V   = I :: IV :: II :: VI :: []
previousTriads VI  = IV :: I :: II :: V :: VII :: []
previousTriads VII = []
```

4.2 Completing the Harmonization

As Piston does we require the harmonizing triad to include the melody note. We use a function `harmonizations` to generate a list of all candidate harmonizations for a given melody. For space considerations we refer the reader to the supplement `Harmony.agda` for its definition. Here again we encounter difficulties with equality and membership. We use a bit vector to represent the set of scale degrees in a triad, where each scale degree is mapped to an element of `Fin 7`. However we are able to make use of dependent types in the form of vectors to ensure the length of each harmonization matches the melody, and continue to use vectors in the following functions where appropriate.

We next restrict to progressions ending in a half cadence (V) to match Piston's example (`filter halfCadence`). This results in 25 candidate progressions. For each pitch and triad pair we construct a `harmonizingChord` in root position with the root doubled. The function `voiceChord` spaces the notes nicely. All of this work ins done in the function `voicedHarmonizations`.

```
voicedHarmonizations : {n : ℕ} → Vec Pitch n → List (Vec (Vec Pitch 4) n)
voicedHarmonizations {n} ps =
  let ds = map pitchToDegreeCMajor ps
      hs : List (Vec Triad n)
      hs = filter halfCadence (harmonizations ds)
      in map (λ ts → map (λ pt →
             proj₁ pt :: harmonizingChord (proj₁ pt) (proj₂ pt)) (zip ps ts)) hs
```

To determine the "best" of these harmonizations, we now make use of `motionCheck` from the counterpoint work. We compare all six pairs of melodies, count the number of motion errors present, and pick the harmonization which minimizes these flaws. The functon to determine all errors is straightfoward.

```
motionErrors : {n : ℕ} → Vec (Vec Pitch 4) n → List MotionError
motionErrors xs =
  let ss = map (head) xs
      as = map (head ∘ tail) xs
      ts = map (head ∘ tail ∘ tail) xs
      bs = map (head ∘ tail ∘ tail ∘ tail) xs

      sas = map pitchPairToPitchInterval (toList (zip as ss))
      sts = map pitchPairToPitchInterval (toList (zip ts ss))
      sbs = map pitchPairToPitchInterval (toList (zip bs ss))
      ats = map pitchPairToPitchInterval (toList (zip ts as))
      abs = map pitchPairToPitchInterval (toList (zip bs as))
```

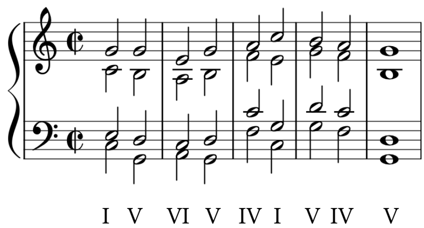

Fig. 6. Harmonization by Music Tools

```
    tbs = map pitchPairToPitchInterval (toList (zip bs ts))
in concatMap checkMotion (sas :: sts :: sbs :: ats :: abs :: tbs :: [])
```

The harmonization produced is shown in Figure 6.

4.3 Comparison with Previous Work

Most of the Haskell-based work [Koops et al. 2013; Magalhães and de Haas 2011; Magalhães and Koops 2014] has focused on harmony. In particular FHARM [Koops et al. 2013] tackles the problem of harmonization of a melody; Figure 5 right shows their harmonization of Piston's example. Their method is quite different from ours. Instead of using Piston's table of root progressions they use a more sophisticated hierarchical grammar from previous work [Magalhães and de Haas 2011]. More specifically, they first require the melody notes to be members of the harmonizing chords, then use their previous work to try to create a harmonic analysis of every possible sequence of chords that arises, choosing the one that most closely fits their grammar.

Although the result produced by FHARM is not bad, it clearly leaves room for improvement. Noted in the paper is that no attention is given to voicing (how each of the four parts flows horizontally and how the lines interact—in other words the counterpoint). They use inversions (in which the third or fifth of the chord is in the bass) to ensure each melody line does not move too much, but as the harmonizing triads are all in close position separated from the melody, the result sounds like a soprano melody with block chord accompaniment, rather than four independent lines.

Furthermore, FHARM always double the melody note, not the root note as Piston does (and as preferred in four-part harmony in general). Also, minimizing the number of parse errors as a means to choose the best harmonization does not necessarily have a musical meaning.

Our harmonization follows Piston's rules more closely and has arguably better spacing of voices; readers can decide for themselves which is more pleasing.

5 CONCLUSION AND FUTURE WORK

We believe that functional programming, and in particular functional programming with dependent types, is a promising way to do software engineering going forward. The languages and tools are perhaps not yet as finely polished and rich as those typically used in production software, but they are fundamentally far more powerful, and thus once the infrastructure (and training of programmers) catches up should become the languages of choice.

Music is an especially valuable domain in which to explore these ideas. We have found it to be a microcosm of issues that arise everywhere in software engineering: modularity, composability, correctness, and data issues such as handling equivalent formulations of the same data as well slightly different formulations. If we can demonstrate that functional programming with dependent types works well in the domain of music, the same techniques can be used in software in general.

Data issues are particularly prevalent at API boundaries between systems, which may internally represent the data in different formats (satisfying modularity and abstraction) but then rely on tedious and potentially error-prone conversions between the formats. In music we see this in wanting to work sometimes with absolute pitch and others with relative pitch (a pair of an octave and a pitch within that octave), or choosing between a pair of pitches and a `PitchInterval`. Also we would like to reuse a function that just works on a pitch to work on a note, which consists of a pitch plus a duration.

For now we have explicitly converted between equivalent representations and explicitly lifted functions, to get a feel for the amount of work required. However in the world of dependent types there are known research techniques for handling these cases, namely transporting across equivalences as in Homotopy Type Theory [Univalent Foundations Program 2013] and ornaments [Dagand 2017]. Cubical Agda [Vezzosi et al. 2019], although still early in development, should be an excellent tool for exploring the extent to which these research ideas can be applied in a practical context.

Aside from these more fundamental ideas there is simply the work of further extending the formalization of music theory to encompass harmonic analysis, more advanced counterpoint, voice leading, and even composition [Schoenberg et al. 1999]. The results so far have been encouraging, but much more needs to be done to find the right abstractions to express the rules in the simplest, clearest and most composable way possible. We expect the world of music theory to be worthy of a long exploration.

REFERENCES

Agda Developers. 2020. Agda Standard Library. https://github.com/agda/agda-stdlib.

E. Aldwell and A. Cadwallader. 2018. *Harmony and Voice Leading*. Cengage Learning.

Youyou Cong and John Leo. 2019. Demo: Counterpoint by Construction. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design* (Berlin, Germany) *(FARM 2019)*. Association for Computing Machinery, New York, NY, USA, 22–24. https://doi.org/10.1145/3331543.3342578

Pierre-Evariste Dagand. 2017. The essence of ornaments. *Journal of Functional Programming* 27 (2017), e9. https://doi.org/10.1017/S0956796816000356

W. Bas De Haas, José Pedro Magalhães, Remco C. Veltkamp, and Frans Wiering. 2011. HarmTrace: Improving Harmonic Similarity Estimation Using Functional Harmony Analysis. In *Proceedings of the 12th International Society for Music Information Retrieval Conference (ISMIR '11)*. 67–72.

W. Bas De Haas, José Pedro Magalhães, Frans Wiering, and Remco C. Veltkamp. 2013. HarmTrace: Automatic Functional Harmonic Analysis. *Computer Music Journal* 37:4 (2013), 37–53. https://doi.org/10.1162/COMJ_a_00209

Richard A Eisenberg and Stephanie Weirich. 2013. Dependently typed programming with singletons. *ACM SIGPLAN Notices* 47, 12 (2013), 117–130.

Johann Joseph Fux. 1965. *The Study of Counterpoint*. W. W. Norton & Company.

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

Hendrik Vincent Koops, José Pedro Magalhães, and W. Bas De Haas. 2013. A Functional Approach to Automatic Melody Harmonisation. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design* (Boston, Massachusetts, USA) *(FARM '13)*. ACM, 47–58. https://doi.org/10.1145/2505341.2505343

José Pedro Magalhães and W. Bas de Haas. 2011. Functional modelling of musical harmony: an experience report. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. ACM, New York, NY, USA, 156–162.

José Pedro Magalhães and Hendrik Vincent Koops. 2014. Functional Generation of Harmony and Melody. In *Proceedings of the Second ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design (FARM '14)*. ACM. https://doi.org/10.1145/2633638.2633645

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers University of Technology.

Walter Piston and Mark DeVoto. 1987. *Harmony*. W. W. Norton & Company.

Curtis Roads, John Strawn, Curtis Abbott, John Gordon, and Philip Greenspun. 1996. *The Computer Music Tutorial*. MIT Press, Cambridge, MA, USA.

A. Schoenberg, G. Strang, and L. Stein. 1999. *Fundamentals of Musical Composition*. Faber & Faber.

Dmitrij Szamozvancev and Michael B Gale. 2017. Well-typed music does not sound wrong (experience report). In *ACM SIGPLAN Notices*, Vol. 52. ACM, 99–104.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.* 3, ICFP, Article Article 87 (July 2019), 29 pages. https://doi.org/10.1145/3341691