Music Theory with Dependent Types (Experience Report)

JOHN LEO, Halfaya Research, USA
YOUYOU CONG, Tokyo Institute of Technology, Japan

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.

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

ACM Reference Format:
John Leo and Youyou Cong. 2020. Music Theory with Dependent Types (Experience Report). *Proc. ACM Program. Lang.* 4, ICFP, Article 42 (August 2020), 8 pages.

1 INTRODUCTION

Edgar Varèse describes music as "organized sound", and throughout history cultures have created 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 sythesis, 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 [???], harmonization of a melody and generation of melodies based on a harmonization [??] and counterpoint [?]. There is also an established Haskell library Euterpea [?] for general music and sound exploration.

To describe rules of basic harmonic structure, ? and its successors use dependent types. (TODO: Give an example.) However Haskell currently has limited support for dependent types, and requires many extentions and tricks such as the use of singleton types [?] to simulate a restricted form of dependent types. (TODO: fix wording)

Authors' addresses: John Leo, Halfaya Research, Bellevue, WA, USA, leo@halfaya.org; Youyou Cong, Tokyo Institute of Technology, Tokyo, Japan, cong@c.titech.ac.jp.

2475-1421/2020/8-ART42 $15.00
https://doi.org/

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 the language Agda [?] since it is fairly mature and aims to be a functional programming language as well as a proof assistant. It also features a Haskell FFI so we can take advantage of existing Haskell libraries, in partiular for MIDI.

Full dependent types allow expression of predicate logic, and it is tempting to take a standard textbook on music theory such as **?** or **?** and attempt to formalize it in type theory. However this is difficult since most "rules" in music theory are not absolute, but rather suggestions of varying degrees (themselves vaguely specified) of importance. Since ultimately the resulting created music is more important than the theory, an alternative is to attempt to indirectly deduce the rules from music samples, using for example machine learning [?]. However.... TODO TODO: Perhaps create rules and weight them, then use ML.

Although formalization of a large amount of music theory is a goal, for now we start with the modest task of expressing a small, relatively strict rule set known as species counterpoint [Fux 1965]. TODO–more here We also present preliminary work on harmonizing a melody using a subset of rules based on **?**, contrasting with existing work by **?**. Notably, since counterpoint and harmony are not separate concepts but in fact deeply intertwined, we are able to reuse counterpoint rules to help develop natural-sounding harmonizations.

We make use of and extend **?**, a library of small tools for analysis and synthesis of music written in Agda, with the goal of eventually being a dependently-typed alternative to Euterpea. Previous work by **?** using Music Tools expressed first-species counterpoint with the Agda type system, so that an error in counterpoint resulted in an Agda type error. Although this takes advantage of Agda's native type checking, it has several downsides. One is that the Agda errors may be difficult to interpret as the corresponding musical error. Another is TODO: flexibility and composability.

This is an experience report, and we highlight both the advantages and disadvantages of using Agda. TODO: MORE

TODO: Talk about conclusion.

TODO: check that bib style is allowed.

2 MUSICAL PRELIMINARIES

Pitches. A *pitch* tells us how high or low a sound is. This notion is represented as natural numbers (of type `Pitch`) in our implementation, with zero representing the lowest possible sound. Throughout the paper, we use the notation `name octave` for pitches, 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`.

Duration. Duration denotes an unspecified unit of time during which a sound or silence lasts. In our implementation, duration is again a natural number (of type `Duration`), with names such as `whole`, `half`, and `quarter`. These values are turned into an absolute length when the music is played at a specific tempo. As an example, `whole` has value `16` and corresponds to 4 seconds if the tempo is 60 beats per minute.

Notes. Combining pitches and duration gives us *notes.* We construct notes (of type `Note`) with the `tone` and `rest` constructors. For example, `tone whole (c 5)` is a whole note whose sound is the middle C, and `rest half` is a half rest.

Intervals. An *interval* represents the difference in pitch between two notes. There are 13 kinds of interval within an octave, and these intervals can be classified from several different perspectives: (i) major or minor; (ii) consonant or dissonant; and (iii) perfect or imperfect. We define intervals as a datatype `Interval`, where each constructor represents one of the 13 intervals.

	per1	min2	maj2	min3	maj3	per4	aug4	per5	min6	maj6	min7	maj7	per8
consonant?	yes	no	no	yes	yes	no	no	yes	yes	yes	no	no	yes

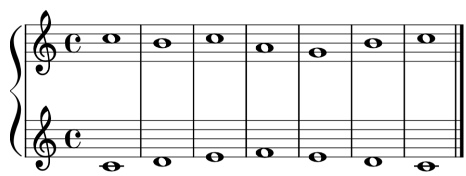

Fig. 1. First Species Counterpoint

3 COUNTERPOINT

TODO: contrast old and new methods

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 counterpoint lines 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 [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. For space reason, we only describe two variants of species counterpoint; other variants can be formalized in an analogous way.

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 and consist only of whole notes. Figure 1 shows an example of first species counterpoint. The lower line is the cantus firmus, which we excerpt from a famous 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 1 is represented as a compound of the following three elements:

```
 PitchInterval : Set
 PitchInterval = Pitch × Interval

first : PitchInterval
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 (more commonly called the *unison*), the 5th, and the 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 an option 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 harmonical consonance and independence of melodic lines. As we saw before, consonant intervals include the unison, 3rd[1], 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 of the latter four intervals. We encode this rule as the `checkIntervals` function, which returns a list of errors correponding 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
... | false | _    = just (dissonant i)
... | _     | true = just (unison p)
... | _     | _    = nothing

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

Motion. The independence of melodic lines is also affected by *motion*, i.e., the way one interval moves to another interval. As can be seen from Figure 2 left, 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), and *oblique* (one line plays the same note). There is a consensus that approaching a perfect interval by parallel or similar motion (as in Figure 2 right) destroys the independence of lines. To rule out this, we define the `checkMotion` function, which lists all occurrences of such invalid motion.

[1]We use “3rd” to mean both the major and minor variants of the 3rd interval, and similarly for the 6th.

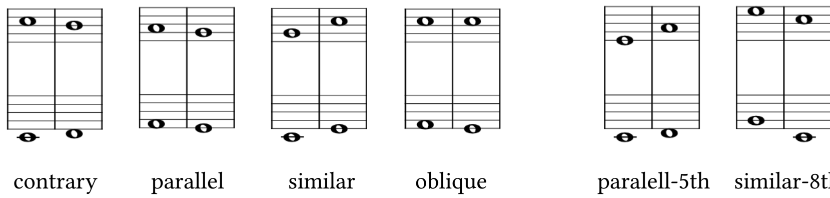

Fig. 2. Four Kinds of Motion (left) and Invalid Patterns (right)

```
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
-- where pairs returns a list of all adjacent pairs in a given list
```

Ending. The ending of the music should express relaxation. Among different intervals, the unison and the 8th are considered most stable, hence the last interval of the music must be either of these intervals. The last interval should also be approached by a *cadence*, a progression that gives rise to a sense of resolution. We encode these rules as the `checkEnding` function. Note that it returns the `tooShort` error when the music does not have enough number of bars to form a valid ending.

```
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)
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)
```

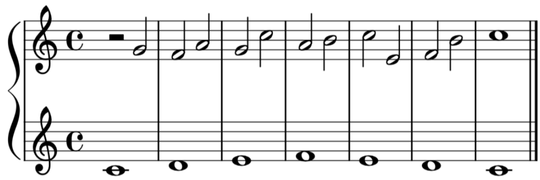

Fig. 3. Second Species Counterpoint

```
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 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, that is:

- The beginning is a perfect consonance
- The middle bars have no dissonant interval or unison
- The middle bars have no perfect interval approached by parallel or similar motion
- The ending is a unison or an octave approached by a cadence

Recall that the functions representing these rules return either an option value or a list of found errors. Therefore, the correctness proofs either take the form `func args ≡ nothing`, or look like `func args ≡ []`.

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

With this record type, we can show that the counterpoint in Figure 1 is correct, since the music is an inhabitant of the `FIrstSpecies` type.

```
fs : FirstSpecies
fs = firstSpecies first1 middle1 last1 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. Figure 3 is an example of such 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:

```
PitchInterval2 : Set
PitchInterval2 = Pitch × Interval × Interval

first2 : PitchInterval
first2 = (c 5 , per5)

middle2 : List PitchInterval2
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. In second species counterpoint, some parts of the music are applied the same rules as first species, while other parts are constrained by new rules.

Beginning. The beginning of the music is *not* allowed to be the unison, as it prevents the listener from recognizing the beginning of the counterpoint line. We restrict the first interval by defining a new error `BeginningError2` and a function `checkBeginning2` that works analogously to `checkBeginning` for first species.

Strong Beats. The rhythmic structure of secound species counterpoint gives rise to the distinction between *strong beats* (the first interval in a bar) and *weak beats* (the second interval). Strong beats in middle bars are constrained by the same rules as in first species. That is, they must all be consonant, non-unison intervals, and in the case of perfect intervals, the motion from the preceding interval (on the weak beat) must be either contrary or oblique. We encode these restrictions as the `checkStrongBeats` function, which, as the `checkIntervals` function for first species, returns a list of found errors.

Weak Beats. Compared to strong beasts, weak beats are constrained in a looser manner. First, they 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 3). Second, they are allowed to be the unison if they are left by step in the opposite direction from their approach (as in bars 5-6 of Figure 3). We implement these rules as the `checkWeakBeats` function, which checks the validity of every three successive beats in the middle bars.

Motion. Parallel and similar motion towards a perfect interval is prohibited across bars. We enforce this rule using the `checkMotion2` function, which uses `expandPitchIntervals2` to convert a list of `PitchInterval2` into a list of `PitchInterval`, and checks every pair of weak beat and strong beat intervals.

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 various properties.

```
record SecondSpecies : Set where
  constructor secondSpecies
  field
```

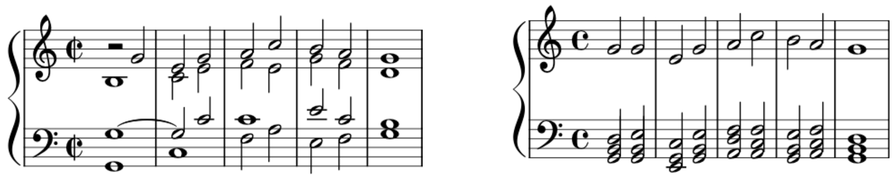

Fig. 4. Harmonization by Piston (left) and FHARM (right)

```
firstBar      : PitchInterval
middleBars    : List PitchInterval2
lastBar       : PitchInterval
beginningOk   : checkBeginning2 firstBar ≡ nothing
strongBeatsOk : checkStrongBeats middleBars ≡ []
weakBeatsOk   : checkWeakBeats middleBars (secondPitch lastBar) ≡ []
motionOk      : checkMotion2 (firstBar ::
                              (expandPitchIntervals2 middleBars) ++
                              (lastBar :: [])) ≡ []
endingOk      : checkEnding2 middleBars lastBar ≡ nothing
```

Using `SecondSpecies`, we can show that the second species counterpoint in Figure 3 is correct.

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

4 HARMONY

TODO: Harmonize melody, use counterpoint for voice leading and to contrain harmony. Mention grid for harmonic analsis, makes use of vectors.

5 CONCLUSION

TODO: easy to express rules, break into parts, reuse in other contexts, extend. Downsides: hard to do some things (equality, sets, etc), Proof emphasis can get in the way, but may want to use in the future. Future work: equivalence, ornaments, ML.

REFERENCES

Johann Joseph Fux. 1965. *The Study of Counterpoint.* 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.