

Harmony and Counterpoint 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, harmony, counterpoint

ACM Reference Format:

John Leo and Youyou Cong. 2020. Harmony and Counterpoint with Dependent Types (Experience Report). *Proc. ACM Program. Lang.* 4, ICFP, Article 42 (August 2020), 7 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, always 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 ([?]).

[more here]

2 MUSICAL PRELIMINARIES

Pitches. In music, a *pitch* is a value denoting how high or low a sound is. Pitches can be represented either as an absolute value or as a relative value paired with the octave the sound belongs to. In our implementation, the two representations of pitches are defined as the `Pitch` and

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

2475-1421/2020/8-ART42 \$15.00


<https://doi.org/>

PitchOctave datatypes, respectively, accompanied by a proof that converting back and forth between them is an identity.

Duration. *Duration* denotes a certain unit of time during which a sound or silence lasts. The unit is unspecified at the composition time; it is instantiated to a concrete value when the music is played at a specific tempo.

Notes. Combining pitches and duration gives us *notes*. In our implementation, we represent notes as a datatype *Note* with two constructors: *tone* for notes with sound and *rest* for those without.

Intervals. *Intervals* are yet another key concept in music. An interval denotes the difference in pitch between two notes. There are 13 kinds of interval within an octave, and these intervals may be classified from several different perspectives: (i) major or minor; (ii) consonant or dissonant; and (iii) perfect or imperfect. As a convention, we refer to min2 and maj2 uniformly as “2nd”, and similarly for other intervals. We also call per1 the *unison* in the rest of this section.



| | 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 |

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 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 [?]. 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.

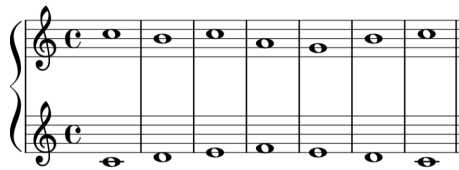


Fig. 1. First Species Counterpoint

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 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, 5th, 6th, and 8th, and among these, the unison is clearly an obstacle to distinguish 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 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
... | false | _ = just (dissonant i)
```

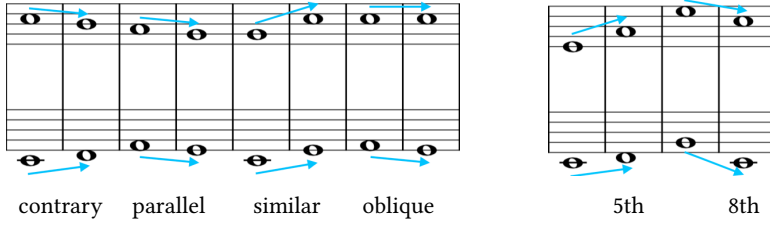


Fig. 2. Left: Four Kinds of Motion, Right: Invalid Motion

```
... | _      | 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). Among these variants, parallel and similar motion makes it harder to distinguish between two voices, especially when the second interval is a perfect one, as in Figure 2 (right). To avoid problematic motion, we define the `checkMotion` function, which lists all invalid motion approaching a perfect interval.

```
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)

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 composed for the Frog's song is correct, since the music inhabits the `FirstSpecies` type.

```

fs : FirstSpecies
fs = firstSpecies first1 middle1 last1 refl refl refl refl

```

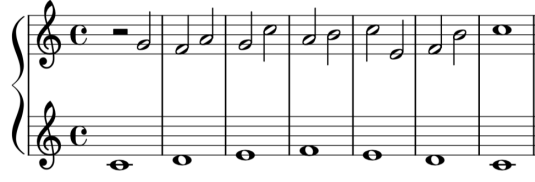


Fig. 3. Second Species Counterpoint

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. As we can see from this example, 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)
```

The rhythmic structure of second species counterpoint gives rise to the distinction between *strong beats* (the first interval in a bar) and *weak beats* (the second interval). Roughly speaking, 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.

Weak beats, on the other hand, 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 a unison if they are left by step in the opposite direction from their approach (as in bars 5-6 of Figure 3).

With these rules in mind, let us look at the definition of `SecondSpecies`, a record type inhabited by correct second species counterpoint:

```
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) ++
                               (lastBar :: [])) ≡ []
endingOk       : checkEnding2 middleBars lastBar ≡ nothing

```

As in `FirstSpecies`, we have three fields holding the musical content, followed by five fields carrying the proofs of various properties. Among the latter group of fields, `beginningOK` is defined analogously to the corresponding field of `FirstSpecies`; the only difference is that the interval must not be a unison, as it prevents the listener from recognizing the beginning of the counterpoint line. The next two fields, `strongBeatsOK` and `weakBeatsOK`, guarantee the correctness of strong and weak beats, respectively. What follows is `motionOK`, which converts a list of `PitchInterval2` into a list of `PitchInterval`, and checks motion accross bars. Lastly, we have `endingOK`, which forces the last two intervals to have a cadence structure.

Using `SecondSpecies`, we can show that our second species counterpoint for the Frog's song is correct.

```

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

```

4 HARMONY

Text here.

5 CONCLUSION

Text here.