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 42 (August 2020), 10 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 [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 its 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].

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 [Norell 2007] 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 Piston and DeVoto [1987] or Aldwell and Cadwallader [2018] 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 [Huang et al. 2017]. However there is the danger that since the rules were never explicit they could end up being violated with poor results; for example parallel fifths and octaves that are extremely rare (and almost always due to special allowed circumstances) in Bach chorales used as training data resulted in a much higher frequency of their use in generated harmonizations [Roberts et al. 2019].

Ultimately it seems some kind of mixture of rules and statistics may be the best way to faithfully capture an effective theory of music. One simple approach would be to develop a logic of rules and their interactions, but attach weights to the rules which could be determined via machine learning techniques.

Our long-term goal is formalization of a large amount of music theory. As a first step 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.

We make use of and extend Leo and Cong [2020], 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 Cong and Leo [2019] 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 that it is difficult to keep the types both simple and flexible.

In this work we present an alternative representation of counterpoint using what could be considered a custom musical type checker written in Agda. This allows us to easily describe fine-grained rules, write custom type errors, and then combine subsets of rules together for larger-scale checking. Music can be easily combined with rules it satisfies as a dependent record type. This record represents a proof certificate that the music follows the asserted rules. Rules can also easily be used in other contexts, satisfying modularity and composability.

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]. 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.

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. However 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 describe future work which may help reduce the tedium.

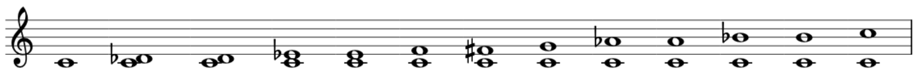
2 MUSICAL PRELIMINARIES

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

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 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. 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 (the first note of a scale; e.g., C in the case of C major) 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 German song called *Froschgesang* (Frog's song). The upper line is the counterpoint composed by the second author.

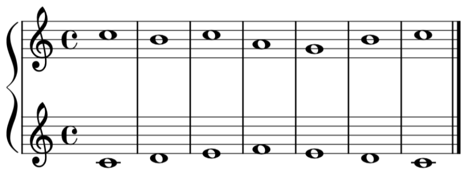


Fig. 1. First Species Counterpoint

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 1st (usually called the *unison*), t5th, 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 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

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

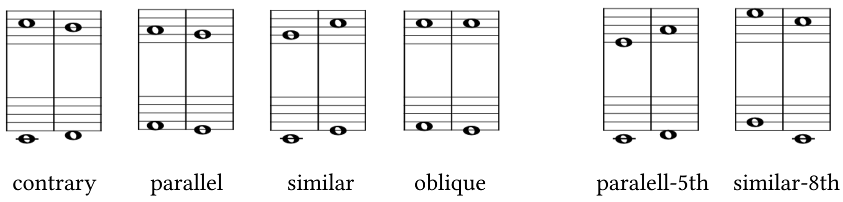


Fig. 2. Four Kinds of Motion (left) and Invalid Patterns (right)

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 Figure 2 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), and *oblique* (one line plays the same note). It is known that approaching a perfect interval by parallel or similar motion (as in Figure 2 right) destroys the independence of lines. To rule out such motion, we define the `checkMotion` function, which checks all pairs of two adjacent intervals 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
```

```
-- where pairs l returns a list of pairs consisting of two adjacent elements of l
```

Ending. The ending of the music should express relaxation. Among different intervals, the unison and 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. 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, tells us which rules are not satisfied.

```
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 (firstBar :: middleBars) ≡ []
```

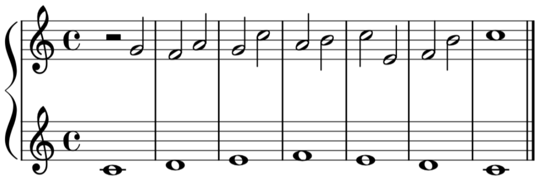


Fig. 3. Second Species Counterpoint

```
  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. This gives rise to the distinction between *strong* beats (the first interval in a bar) and *weak beats* (the second interval). Figure 3 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.

```
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. 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 constrained in a looser manner than strong beats. 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).

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

Ending. The last interval must again 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 disucssed above[2].

```
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
```

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
```

3.3 Comparison with Previous Work

Cong and Leo [2019] implement correct-by-construction counterpoint using an older version of Music Tools. The idea is to represent 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 the existing implementation in two ways. First, while Cong and Leo [2019] rely on the Agda type checker to report errors in the counterpoint, we use our own type checker (implemented as the `checkXXX` functions) and type errors (defined as the `XXXError` datatypes). This allows us to produce error messages that clearly state what kind of mistake the programmer has made. Suppose we have replaced the second interval of Figure 1 by an octave (8th) of D, breaking the rule that perfect intervals cannot be approached by parallel motion. In our implementation, this results in an error in the `motionOk` field, which requires a proof of `checkMotion (firstBar :: middleBars) ≡ []`:

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

[2]As the reader would have imagined, each `checkXXX` function corresponds to one rule that must be satisfied by the counterpoint. The auxiliary function `secondPitch` in `weakBeatsOk` extracts the counterpoint note of the last interval, and `expandPitchIntervals2` turns a list of `PitchInterval2`s into a list of `PitchInterval`s.

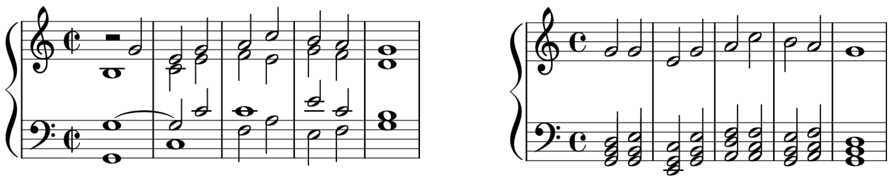


Fig. 4. Harmonization by Piston (left) and FHARM (right)

In Cong and Leo [2019]'s implementation, on the other hand, the error results in the first `PitchInterval` being highlighted in yellow, with the following error message displayed in the Agda buffer:

```
_13 : motionOk (c 5 , c 6) (d 5 , d 6)
```

This indicates the existence of an unsolved metavariable. To a non-expert user, it may be difficult to connect this message to the musical error.

The second improvement from Cong and Leo [2019] is that, instead of incorporating all the rules into the counterpoint datatype, we define each of them as a separate function. This makes it possible to reuse certain rules in a different context, as we will see in the next section.

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 AND FUTURE WORK

We believe that functional programming, and in particular functional programming with dependent types, is the most 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 explicity converted between equivalent representations and explictly 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 composeable way possible. We expect the world of music theory to worthy of a long exploration.

REFERENCES

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.

Cheng-Zhi Anna Huang, Tim Cooijmans, Adam Roberts, Aaron Courville, and Douglas Eck. 2017. Counterpoint by Convolution. In *Proceedings of ISMIR 2017.* https://ismir2017.smcnus.org/wp-content/uploads/2017/10/187_Paper.pdf

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

John Leo and Youyou Cong. 2020. Music Tools. https://github.com/halfaya/MusicTools.

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.

Adam Roberts, Anna Huang, Curtis Hawthorne, Jacob Howcroft, James Wexler, Leon Hong, and Monica Dinculescu. 2019. Approachable music composition with machine learning at scale. In *Proceedings of the 20th International Society for Music Information Retrieval Conference (ISMIR).*

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