Demo: Counterpoint by Construction

Youyou Cong
Tokyo Institute of Technology
Tokyo, Japan
cong@c.titech.ac.jp

John Leo
Halfaya Research
Bellevue, WA, USA
leo@halfaya.org

Abstract

We present *Music Tools*, an Agda library for analyzing and synthesizing music. The library uses dependent types to simplify encoding of music rules, thus improving existing approaches based on simply typed languages. As an application of the library, we demonstrate an implementation of first-species counterpoint, where we use dependent types to constrain the motion of two parallel voices.

CCS Concepts • **Applied computing → Sound and music computing**; • **Software and its engineering → Functional languages**.

1 Introduction

Western music of the common practice period tends to loosely follow sets of rules, which were developed over time to ensure the aesthetic quality of the composition. Among such rules, those for harmony [Piston and DeVoto 1987] and counterpoint (harmonically interdependent melodies) [Fux 1965] are particularly fundamental and continue to be taught to music students, not only as a means to understand the music of that period, but also as a foundation for modern art and popular music.

To analyze and synthesize tonal music, researchers have attempted to encode these rules into programming languages. Functional programming languages seem ideally suited for this task, and in particular, those with a static type system can further guarantee that *well-typed music does not sound wrong*. In the past decade, Haskell has been extensively used to encode the rules of harmony [De Haas et al. 2011, 2013; Koops et al. 2013; Magalhães and de Haas 2011; Magalhães and Koops 2014] as well as counterpoint [Szamozvancev and Gale 2017]. An interesting observation is that, many of the existing encodings rely on some form of *dependent types*, i.e., types that depend on terms. Since Haskell is not a dependently typed language, one has to use language extensions, such as GADTs [Cheney and Hinze 2002] and singleton types [Eisenberg and Weirich 2013], to simulate dependencies. While this allows encoding a wide class of music rules, it can require duplicating code to reflect terms into types, making the implementation less elegant [Monnier and Haguenauer 2010]. This motivates us to explore music programming in a language with intrinsic support for dependent types.

FARM 2019, August, 2019, Berlin, Germany
2019.

We present Music Tools[1], a library of small tools that can be combined functionally to help analyze and synthesize music. To allow simple and natural encoding of rules, we build our library in Agda [Norell 2007], a functional language with full dependent types. As an application of the library, we demonstrate an implementation of species counterpoint, based on the rules given by Fux [1965]. Thanks to Agda's rich type system, we can express the rules in a straightforward manner, and thus ensure by construction that well-typed counterpoint satisfies all the required rules.

2 The Music Tools Library

The goal of Music Tools is to provide a core collection of types and functions that can be used to easily build applications to analyze and synthesize music. It is intended to evolve into a dependently-typed replacement for Euterpea [Hudak and Quick 2018] and borrows many ideas from that library. It is currently restricted to the chromatic scale for simplicity, and like Euterpea is designed to work well with MIDI.

The fundamental type for melody is `Pitch`, which is just a natural number with 0 representing the lowest expressible pitch (it is intended to correspond to MIDI note 0, but the interpretation can change depending upon the application). One can also express pitch as a pair of an octave and relative pitch within the octave, which is more convenient for some applications, and convert between this representation and absolute pitch. One can then prove that converting back and forth is the identity function.

For rhythm the fundamental type is `Duration`, also a natural number, which represents some unspecified unit of time (when the music is played the unit is then specified). A combination of a pitch and a duration gives a `Note`, and notes in turn are made into `Music` via sequential and parallel composition, as in Euterpea.

To play music on synthesizers, one can convert it to MIDI by calling Haskell MIDI libraries via Agda's Haskell FFI. Details can be found in the github repository.

3 Application: First-Species Counterpoint

We now explain how to implement the rule system of first-species counterpoint[2]. In first-species counterpoint, one starts

[1] https://github.com/halfaya/MusicTools
[2] The code is available at https://github.com/halfaya/MusicTools/blob/master/agda/Counterpoint.agda.

with a base melody (the *cantus firmus*), and constructs a counterpoint melody note-by-note in the same rhythm. The two voices are represented as a list of pitch-interval pairs, where intervals must not be dissonant (2nds, 7ths, or 4ths).

```
data IntervalQuality : Set where
  min3  : IntervalQuality
  maj3  : IntervalQuality
  per5  : IntervalQuality
  min6  : IntervalQuality
  maj6  : IntervalQuality
  per8  : IntervalQuality
  min10 : IntervalQuality
  maj10 : IntervalQuality

PitchInterval : Set
PitchInterval = Pitch × IntervalQuality
```

In addition, it is prohibited to move from any interval to a perfect interval (5th or octave) via parallel or similar motion. Therefore, we define a predicate that checks whether a motion is allowed or not.

```
motionOk : (i1 : Interval)
           (i2 : Interval) → Set
motionOk i1 i2 with motion i1 i2
         | isPerfectInterval i2
motionOk i1 i2 | contrary | _     = ⊤
motionOk i1 i2 | oblique  | _     = ⊤
motionOk i1 i2 | parallel | false = ⊤
motionOk i1 i2 | parallel | true  = ⊥
motionOk i1 i2 | similar  | false = ⊤
motionOk i1 i2 | similar  | true  = ⊥
```

The last requirement is that the music must end with a cadence, which is a final motion from the 2nd or 7th degree to the tonic (1st degree). We impose this requirement by declaring two cadence constructors as the base cases of counterpoint (note that the final interval of the cadence is always `(p , per8)` and is thus not explicitly specified). Thus, we arrive at the following datatype for well-typed counterpoint[3].

```
data FirstSpecies : PitchInterval →
                    Set where
  cadence2 : (p : Pitch) →
    FirstSpecies (transpose (+ 2) p , maj6)
  cadence7 : (p : Pitch) →
    FirstSpecies (transpose -[1+ 0 ] p , min10)
  _::_ : (pi : PitchInterval) →
         {pj : PitchInterval} →
         {_ : motionOk pi pj} →
         FirstSpecies pj →
         FirstSpecies pi
```

[3] For readability, we have omitted explicit conversions from `PitchInterval` (which ensures the interval is not dissonant) to the general `Interval`.

Observe that `motionOk` is an implicit argument of the `_::_` constructor. The argument can be resolved automatically by the type checker, hence there is no need to manually supply this proof.

Now we can write valid first-species counterpoint as in the example below.

```
example : FirstSpecies (g 4 , per8)
example =
  (g 4 , per8) :: (c 5 , maj10) ::
  (c 5 , per8) :: (c 5 , maj10) ::
  (e 5 , min10) :: (g 5 , per8) ::
  (cadence2 (c 6))
```

4 Future Work

We intend to expand the Music Tools library to include richer functionality for analysis and synthesis. We are particularly interested in expressing the work done in Haskell on functional harmony in the library, to see just how much dependent types can simplify the representation.

Music is a rich yet circumscribed domain in which issues of equivalence [Tabareau et al. 2018] and ornamentation [Dagand 2017] naturally arise. For example, a musical score can be interpreted either horizontally (counterpoint) or vertically (harmony), and it is important to be able to seamlessly convert between these representations. Similarly, one may wish to treat pitch or rhythm separately as well as combine them, which is a natural application of ornamentation. We plan to examine the extent to which this research can be put to use in a practical domain.

For counterpoint specifically, we plan to represent higher species counterpoint by extending the rules, and to explore automatic generation of species counterpoint. It would be interesting to compare our correct-by-construction counterpoint with that created by machine learning [Huang et al. 2017], which does not have correctness guarantees.

Acknowledgments

The authors would like to thank the participants of the Tokyo Agda Implementors' Meeting, especially Ulf Norell and Jesper Cockx, for many helpful suggestions that improved our Agda code. We also thank the anonymous reviewers for their thoughtful feedback which improved our presentation.

References

James Cheney and Ralf Hinze. 2002. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 90–104.

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 (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 (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

Stefan Monnier and David Haguenauer. 2010. Singleton types here, singleton types there, singleton types everywhere. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification.* ACM, 1–8.

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.

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.

Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proc. ACM Program. Lang.* 2, ICFP, Article 92 (July 2018), 29 pages. https://doi.org/10.1145/3236787