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 parallely sounding 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 independent 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 been attempting to encode various rules into a programming language. It has proven that functional programming languages are 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 other facilities, such as GADTs [Cheney and Hinze 2002] and singleton types [Eisenberg and Weirich 2013], to indirectly simulate dependencies. While this allows to encode a wide class of music rules, it may require duplicating certain pieces of 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 the Agda [Norell 2007] proof assistant, where dependent types are readily available. 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

TODO: describe the key ingredients of the library.

3 Application: First-Species Counterpoint

Now, let us explain how to implement the rule system of first-species counterpoint[2]. In first-species counterpoint, one starts 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
```

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

```
         | 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. 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
```

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

TODO

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.

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.

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.

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.

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