Experience Report: Well-typed music does not sound wrong

Anonymous Author(s)

Abstract

Music description and generation are popular use cases for Haskell, ranging from live coding libraries to automatic harmonisation systems. Some approaches use probabilistic methods, others build on the theory of Western music composition, but there has been little work done on checking the correctness of musical pieces in terms of voice leading, harmony, and structure. Haskell's recent additions to the type-system now enable us to perform such analysis and verification statically.

We present our experience implementing a type-level model of classical music and an accompanying EDSL which enforce the rules of classical music at compile-time, turning composition mistakes into compiler errors. Along the way, we discuss the strengths and limitations of doing this in Haskell and demonstrate that the type system of the language is fully capable of expressing non-trivial and practical logic specific to a particular domain.

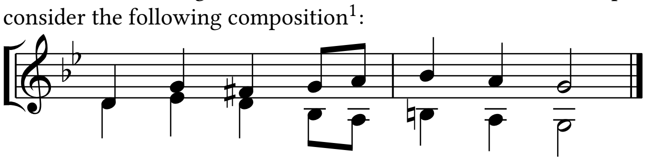
CCS Concepts • **Applied computing → Sound and music computing**; • **Software and its engineering → Functional languages**;

Keywords Type-level computation; Haskell; music theory

ACM Reference format:
Anonymous Author(s). 2017. Experience Report: Well-typed music does not sound wrong. In *Proceedings of Haskell Symposium, Oxford, UK, September 2017 (HASKELL'17),* 6 pages.
DOI: 10.475/123_4

1 Introduction

The connection between music and mathematics has been studied by scholars as early as Pythagoras. These investigations were the beginnings of the field of *Western music theory* – a formal description of what sounds good to the ear and what does not. For example, consider the following composition[1]:



For readers who do not read music: the exact meaning of this depiction is irrelevant, but note that compositions are read from left to right and that, in this example, there are two *voices* – the two series of notes which occur at the same points horizontally.

To ensure that compositions sound good, composers follow strict rules which have been developed over centuries of music tradition. The piece above does not abide by these rules and will sound odd when played. To avoid this, composers have to check by hand, through close inspection of the notes, which rules have been violated. This process is laborious, error-prone and requires a thorough understanding of music theory.

We present *Mezzo*, an embedded domain-specific language for describing music in Haskell which statically enforces the rules of classical music theory. Compositions which break the rules are not valid programs and result in type errors. For example, the composition we gave can be described in the Mezzo EDSL as follows:

```
v1 = d qn :|: g  qn :|: fs qn :|: g en
 :|: a en :|: bf qn :|: a  qn :|: g hn

v2 = d  qn :|: ef qn :|: d  qn :|: bf_ en
 :|: a_ en :|: b_ qn :|: a_ qn :|: g_  hn

comp = defScore (v1 :-: v2)
```

The `:|:` operator is used for sequential composition of notes and `:-:` is used to combine the two voices, `v1` and `v2`, in parallel. The `defScore` function applies a default set of rules. If we attempt to compile this, GHC gives us the following two type errors:

```
error:
  Can't have major sevenths in chords: Bb - B_.
  Parallel octaves are forbidden: A - A_, then G - G_.
```

As expected, the program does not compile since `comp` is musically incorrect. The task of finding the mistakes which would have taken a composer some time to complete was accomplished by Mezzo in a fraction of that time. The type errors also tell us exactly what is wrong and where the errors lie, although we have omitted line and column numbers from the example here.

The first error is caused by a violation of a harmonic rule: major seventh chords, which sound very dissonant, are generally forbidden. The second error is more complex and relates to *counterpoint* – a polyphonic (multi-voice) compositional technique. Its most important consideration is that the melodic lines have to be independent, but give a coherent whole when played together. Composers have to follow strict rules of voice-leading and harmonic motion to ensure this [6]. Whenever two voices sing a perfect interval apart (unison, fifth or octave), they become hard to distinguish and effectively fuse together into one voice. Simply put, series of perfect intervals are not interesting enough to create complex, dynamic music and are therefore forbidden.

To correct the problems, we change the last three notes of the second voice to avoid the major seventh and the parallel octaves:

```
v2 = d  qn :|: ef qn :|: d   qn :|: bf_ en
 :|: a_ en :|: g_ qn :|: fs_ qn :|: g_  hn
```

The corrected code compiles without errors and `comp` is seen as a valid composition that can be used in larger pieces or exported to a MIDI file.

This experience report describes the implementation of Mezzo and also addresses the challenges we faced by using Haskell for type-level computation. Our library provides both a non-trivial and practical use case for advanced type-level features in Haskell and functional programming in general. We also provide evidence that Haskell is more than capable of handling relatively sophisticated

[1] The example is based on http://decipheringmusictheory.com/?page_id=46.

HASKELL'17, Oxford, UK

DOI: 10.475/123_4

type-level computation without being a fully dependently-typed language yet.

2 Music model

In this section, we give a top-down description of the music model implemented in Mezzo and present the majority of the type-level computation techniques used by the library.

Enforcing the musical rules at the type-level buys us many advantages over a more standard implementation at the term-level. For example, we get better integration with existing development tools which can highlight the precise locations in source files at which type errors occur. Users of our library can therefore see where the rules are violated, and we found this very useful in practice. We also benefit from the usual advantages of static typing such as the ability to write functional programs in which only compositions that are guaranteed to sound good may be constructed, or functions which do not need to handle inputs that cannot possibly be musically or structurally valid. However, in our experience, type inference cannot always handle the complex types involved, which makes such programs difficult to write. The leaking of internals in the case of real type errors is also common, but this is a known drawback of EDSL design in general.

2.1 The `Music` data type

Mezzo's music model is responsible for representing musical pieces both at the term- and type-level, as well as expressing and enforcing the composition rules.

The main inspiration comes from *Haskore*, a music description library developed by Hudak et al. [8]. The novelty of Haskore is that it treats music as a recursive structure with two associative operators: sequential (melodic) and parallel (harmonic) composition. In BNF syntax, a piece of music M can be expressed as:

$$M ::= \text{Note} \;|\; \text{Rest} \;|\; M :|: M \;|\; M :\!-\!: M$$

This is translated into Haskell as follows:

```
data Music = Note Pit Dur    | Rest Dur
           | Music :|: Music | Music :-: Music
```

This describes a tree-like structure with the leaves containing notes (with some pitch and duration) or rests (with some duration). Though the `Music` type is fairly simple, it is already capable of expressing a huge variety of musical compositions – however, we have no guarantee that `Music` values will sound good, as there is nothing to constrain their structure.

To enforce rules on compositions, we need to know the detailed structure of them at compile-time. This can be achieved by adding a type argument to the `Music` type, containing some type-level representation of the music (Section 2.2). Ideally, we would like this to depend on the term-level value of `Music m`, which is a typical use-case for *dependently typed programming*. Haskell already supports this through various language extensions [3]. In this case, we can use GADTs [11]: this way, each constructor can determine what `m` should be instantiated with. More complex computation is enabled by the `TypeFamilies` extension, which we use to convert type-level information about pitches and durations into our music representation, as well as to combine these representations. Finally, we encode musical rules as *type class constraints* on the type variables: whenever we construct a new term of type `Music m`, it must follow the composition rules. Our final `Music` type looks like this:

```
data Music m where
    Note  :: NoteConstraints p d
          => Pit p -> Dur d -> Music (FromPitch p d)
    Rest  :: RestConstraints d
          => Dur d -> Music (FromSilence d)
    (:|:) :: MelConstraints m1 m2
          => Music m1 -> Music m2 -> Music (m1 +|+ m2)
    (:-:) :: HarmConstraints m1 m2
          => Music m1 -> Music m2 -> Music (m1 +-+ m2)
```

The separation of structure and constraints makes it easy to extend or even completely change the musical rules implemented, as well as to add new top-level musical constructs, such as chords or chord progressions.

2.2 The pitch matrix

A crucial step in creating a static model of music is finding a suitable representation of musical pieces on the type level. It must have a consistent, but accurate structure that makes rule enforcement as simple as possible. The model must also not discard any relevant musical information: for example, it should always be possible to compose a long melody with a long accompaniment and ensure that all arising harmonic intervals are valid. While intuitive to compose with, the Haskore algebra is too unstructured to formally reason about: for example, it is not clear how one would recursively find two notes which are played at the same time.

We decided on the straightforward approach of keeping the music in a two-dimensional array of notes. The columns of this matrix represent durations and the rows are individual voices. The matrix elements are pairs of pitches and durations, which specify notes. Importantly, all durations in one column are equal: this ensures that notes in the same column are played at the same time.

The implementation of the composition rules relies on the fact that the composed music values have the same "size": sequential pieces must have the same number of voices, and parallel pieces must have the same length. An experienced Haskell programmer would immediately exclaim "Vectors!" – but note that we are at the type level. Thanks to data type promotion [13] and `TypeInType`, this is not an issue: any data type, even GADTs, can be promoted to the type level. All we need is to define the usual `Vector` data type:

```
data Vector :: Type -> Nat -> Type where
    None :: Vector t 0
    (:--) :: t -> Vector t (n - 1) -> Vector t n
```

This vector type is suitable for storing the rows of the matrix (the individual voices), but in those rows we need to store both pitches and durations. Moreover, we want the length of a voice to be the total duration of the notes, so we need to keep the duration on the type level. We do this by defining a new `Elem` type that holds a value (a pitch) and the number of repetitions (the duration), expressed as the proxy `Times` for `Nat`s:

```
data Times (n :: Nat) = T
```

```
data Elem :: Type -> Nat -> Type where
    (:*) :: t -> Times n -> Elem t n
```

This is used to build up an *optimised vector*. Note that in this case, the length of this vector is not the number of elements, but their total duration, so a whole note and 8 eighths have the same length:

```
data OptVector :: Type -> Nat -> Type where
    End  :: OptVector t 0
    (:-) :: Elem t d -> OptVector t (n - d)
```

```
        -> OptVector t n
```

We can now declare a type synonym for matrices:

```
type Matrix t p q = Vector (OptVector t q) p
```

Thanks to GADT promotion, all these types are available at the kind level and we can define type families for common list and matrix operations. In particular, we define horizontal (+|+) and vertical (+-+) concatenation of matrices, as well as means of converting musical values to pitch matrices. For example, FromPitch p d creates a singleton matrix with the pitch p of duration d.

Finally, we need to describe musical values at the type level – this is a straightforward application of data type promotion. All types which describe compositions need term-level values: we accomplish this by creating kind-constrained proxies, such as Pit:

```
data PitchClass = C | D | E | F | G | A | B
data Accidental = Natural | Sharp | Flat
data Octave = Oct_1 | Oct0 | Oct1 | Oct2 | ...

data PitchType = Pitch PitchClass Accidental Octave
               | Silence
data Pit (p :: PitchType) = Pit
```

We also define specialised types for pitch vectors and matrices:

```
type Voice l         = OptVector PitchType l
type PitchMatrix n l = Matrix PitchType n l
```

We can now explicitly specify the type variable for Music m. A minor nuisance here is that kind inference of recursive types can only use monomorphic recursion, just like type inference. If we want polymorphic recursion, which we have in the recursive application of the Music type constructor in :|: and :-:, we need to provide a *complete user-supplied kind signature* (CUSK). Additionally, with -XTypeInType enabled, GHC requires us to quantify all the kind variables in the type definition as shown below. This is explained in more detail in Section 9.11.5 of the GHC 8 User Guide.

```
data Music :: forall n l. PitchMatrix n l -> Type where
    ...
```

2.3 Intervals

The rules implemented in Mezzo mainly constrain the *musical intervals* arising between two composed pieces. To find the interval between two pitches, we declare a type family called MkInterval. It is used in most of the low-level correctness checks. For example, the interval between a C and a G in the same octave and with the same accidental is a perfect fifth, while the interval between a C and a pc2 sharp in the same octave is the interval between the C and a pc2 natural expanded by a semitone:

```
type family MkInterval p1 p2 :: IntervalType where
    MkInterval (Pitch C acc o) (Pitch G acc o) =
        Interval Perf Fifth
   MkInterval (Pitch C Natural o) (Pitch pc2 Sharp o) =
        Expand (MkInterval (Pitch C Natural o)
                           (Pitch pc2 Natural o)) ...
```

2.4 Musical rules

An example of a musical rule is checking harmonic intervals: classically, minor seconds (one semitone) and major sevenths (11 semitones) are to be avoided since they sound very dissonant. To express this limitation, we declare the ValidHarmInterval type class which determines whether an interval is harmonically valid. GHC's custom type error feature (in GHC.TypeLits) lets us specify instances for invalid intervals by making the type error the "precondition", as shown below. Hence whenever GHC tries to determine whether a major seventh is a valid harmonic interval, it encounters a type error. A general, catch-all instance represents valid intervals:

```
class ValidHarmInterval (i :: IntervalType)
instance TypeError (Text "Minor seconds forbidden.")
      => ValidHarmInterval (Interval Min Second)
instance TypeError (Text "Major sevenths forbidden.")
      => ValidHarmInterval (Interval Maj Seventh)
instance {-# OVERLAPPABLE #-} ValidHarmInterval i
```

Note that in the general case we need to permit overlapping instances, which is indicated by a compiler pragma.

We now need to apply this rule to the pitches in our pitch matrix. This is done by a series of simple inference rules, which are easy to express using class constraints on the instance declarations. For example, to check that two pitches (a dyad) are separated by a valid interval, we need to form an interval and establish that it is harmonically valid:

```
class ValidHarmDyad (p1 :: PitchType) (p2 :: PitchType)
instance ValidHarmInterval (MkInterval a b)
        => ValidHarmDyad a b
```

When working with constraints, a useful abstraction is made possible by the ConstraintKinds extension. Constraints (and functions returning constraints) can be passed around as types, which opens the door to many flexible options for validation. For example, we can check if a vector of types satisfies a constraint or a type satisfies all the constraints in a vector. The following definition allows us to apply a binary constraint to two optimised vectors, ensuring that all constraints hold pairwise (the durations can be ignored, as notes in the same column have the same duration):

```
type family AllPairsSatisfy
        (c  :: a -> b -> Constraint)
        (xs :: OptVector a n) (ys :: OptVector b n)
        :: Constraint where
    AllPairsSatisfy c End End = Valid
    AllPairsSatisfy c (x :* _ :- xs) (y :* _ :- ys)
            = ((c x y), AllPairsSatisfy c xs ys)
```

Now we can define validity for harmonic concatenation of two voices. ValidHarmDyad, defined above, is a two-parameter type class of kind PitchType -> PitchType -> Constraint – a suitable first argument to AllPairsSatisfy:

```
class ValidHarmDyadsInVoices
    (v1 :: Voice l) (v2 :: Voice l)
instance AllPairsSatisfy ValidHarmDyad v1 v2
        => ValidHarmDyadsInVoices v1 v2
```

Finally, we use ValidHarmDyadsInVoices to validate the composition of pitch matrices. Given two matrices (v :-- vs) and us (where v is the topmost voice of the first matrix), they can be concatenated if: (1) vs and us can be concatenated, and (2) v can be concatenated with all of the voices in us. The second condition is implemented by mapping ValidHarmDyadsInVectors v (of kind Voice l -> Constraint) over all the voices in us and checking whether all the constraints are satisfied. AllSatisfy applies a unary constraint to all elements of a Vector:

```
class ValidHarmConcat (ps :: PitchMatrix n1 l)
                      (qs :: PitchMatrix n2 l)
```

```
instance ( ValidHarmConcat vs us
         , AllSatisfy (ValidHarmDyadsInVectors v) us
         ) => ValidHarmConcat (v :-- vs) us
```

By translating logical expressions into type class constraints, we can encode most of the low-level musical rules in the type system. We found the pitch matrix representation very well suited for this purpose, as it encapsulates all of the relevant musical information in a structured way that is easy to reason about.

2.5 Rule sets

Mezzo's *rule sets* address the question of flexibility: how can we reconcile formal rule checking with artistic expression? Our solution is to provide users with three levels of rule strictness (including one that does not enforce any musical rules), and let them define their custom rules and correctness checks if they wish. Different parts of a composition can be checked according to different rules.

Rule sets are implemented using constraint kinds and associated type families. The `RuleSet` type class contains associated constraint synonyms for each of the `Music` constructors:

```
class RuleSet t where
    type HarmConstraints t m1 m2 :: Constraint
    type NoteConstraints t p d   :: Constraint ...
```

A rule set is defined as a unit data type and an accompanying instance of `RuleSet`:

```
data Classical = Classical
instance RuleSet Classical where
    type HarmConstraints Classical m1 m2 =
                            ValidHarmConcat m1 m2
    type NoteConstraints Classical p d = Valid ...
```

Finally, we have to parameterise `Music` values by their rule set:

```
data Music :: Type -> PitchMatrix n l -> Type where
  (:-:) :: HarmConstraints rs m1 m2 =>
    Music rs m1 -> Music rs m2 -> Music rs (m1 +-+ m2)
  Note  :: NoteConstraints rs p d =>
    Pit p -> Dur d -> Music rs (FromPitch p d) ...
```

To instantiate `rs`, we create a new type encapsulating `Music` values and rule sets:

```
data Score = forall rs m. MkScore rs (Music rs m)
```

Now we can dynamically change the type checking behaviour by changing the rule set arguments: for example, `MkScore Classical (c qn :-: b qn)` produces a type error, while `MkScore Empty (c qn :-: b qn)` compiles (where `Empty` enforces no rules). As Haskell type classes are open, users are free to define their own rule sets with custom constraints on composition operators, chords, or even notes and rests. For instance, we can implement a rule set for first-species counterpoint by extending the predefined `Strict` rule set with constraints allowing only whole notes and no chords.

3 Music description language

This section showcases some interesting aspects of the Mezzo EDSL which makes use of the type-level model. To increase usability and conciseness, the language provides shorthand methods for note, chord, melody and progression input, covering the most common musical structures composers might use.

3.1 Note and chord input

Mezzo's note and chord input method is based on *continuation-passing style*: it allows musical values to be built via a series of flexible "transformations" with little syntactic interference. For example, a C quarter note can be written as `c qn`, while a D flat major half chord in first inversion is `d flat maj inv hc`. The main advantage of this approach – as opposed to simple constructor functions – is the reuse of syntactic constructs: if the pitch `c` is followed by `qn`, we construct a C quarter note; but if it is followed by `maj qc`, we create a C major quarter chord. The exact details of the implementation are outside the scope of this paper and involve no complex type-level computation. However, we refer the interested reader to Okasaki's paper on *flat combinators* for more information on this style of programming [10].

3.2 Melodies

The input method described above is concise, but still contains a lot of redundancy, especially when writing melodies. For the first voice in Section 1, we had to specify the duration of every note, even though most notes had the same duration, which is commonly the case. It is therefore more convenient to be explicit only when the duration changes, and otherwise assume that each note has the same duration as the previous one. With this in mind, we can use Mezzo's melody construction syntax to describe the melody from Section 1 more concisely:

```
melody :| d :| g :| fs :< e :| a :^ bf :| a :> g
```

Notes are only given as pitches and the duration is either implicit, or explicit in the constructor. For example, `:|` means "the next note has the same duration as the previous note", while `:<` means "the next note is an eighth note". This makes melody input shorter and less error-prone, as most of the constructors will likely be `:|`.

Melodies are implemented as "snoc" lists, i.e. lists whose head is at the end. The `Melody` type keeps additional information in its type variables (like a vector), and has a constructor for every duration:

```
data Melody :: PitchMatrix 1 l -> Nat -> Type where
    Melody :: Melody (End :-- None) Quarter
    (:|) :: (MelConstraints ms (FromPitch p d))
      => Melody ms d -> PitchS p
      -> Melody (ms +|+ FromPitch p d) d
    (:<) :: (MelConstraints ms (FromPitch p Eighth))
      => Melody ms d -> PitchS p
      -> Melody (ms +|+ FromPitch p Eighth) Eighth ...
```

The type keeps track of the "accumulated" music, as well as the duration of the last note. The `Melody` constructor initialises the pitch matrix and sets the default duration to a quarter. The binary constructor `:|` takes the melody composed so far (the tail) and a pitch specifier `PitchS` (the type of the overloaded pitch literals, such as `c`), and returns a new melody with the added pitch and unchanged duration. The other constructors do the same thing, except they ignore the argument `d` of the tail and change the duration of the last note. While the syntax of the constructors might need getting used to, they allow for quick and intuitive melody input.

4 Music rendering

Mezzo can export all well-typed compositions to MIDI files. The principal question is how to reify compositions which exist entirely on the type-level so that we can create the corresponding values on the term-level. Recall that users of Mezzo mainly interact with proxies which contain no term-level information, and types are erased at runtime. To solve this problem, we make use of type classes to reify type-level data.

4.1 The Primitive class

Our aim is to find a primitive representation for all of the musical types that the user is exposed to. That is, to find a function which can convert type-level information into term-level values. Our solution is to define a type class for "primitive" values:

```
class Primitive (a :: k) where
    type Rep a
    prim :: proxy a -> Rep a
```

`Primitive` is poly-kinded, so it can be used with naturals, pitches, etc. Its only method, `prim`, takes the instance type with an arbitrary type constructor, and returns a *representation type* of the value, specified in an *associated type family*. The primitive representation for a pitch would be an integer (e.g. its MIDI number), while for a chord it would be a list of integers (the constituent pitches). As there are no constraints on the representation type, we can be even more flexible: for example, chord types (major, diminished, etc.) are converted into *functions* from integers to integer lists, mapping the MIDI code of the root pitch to the list of codes of the chord pitches.

All we need now is to declare instances of `Primitive` for our types: unfortunately, we have to do this mostly by hand, as Haskell does not have "kind classes" which would let us express that "every type of this kind is a primitive". In our case, we declare separate instances for all of the promoted data constructors of a type:

```
instance Primitive Oct0 where
    type Rep Oct0 = Int ; prim _ = 12 ...
instance Primitive C where
    type Rep C = Int ; prim _ = 0 ...
```

Having done the hard part, reifying pitches (and other compound types) is straightforward:

```
instance (Primitive pc, Primitive acc, Primitive oct)
        => Primitive (Pitch pc acc oct) where
    type Rep (Pitch pc acc oct) = Int
    prim _ = prim (PC @pc) + prim (Acc @acc)
                           + prim (Oct @oct)
```

The `@pc` syntax is possible with the `TypeApplications` extension, which provides a short way of instantiating the polymorphic type variables of a term [5]. The `pc` type variable is bound to the one in the instance declaration, and since we assert that `pc` is an instance of `Primitive`, we can get its primitive representation using `prim`.

4.2 MIDI export

MIDI is a simple, compact standard for music communication, often used for streaming events from electronic instruments. To render compositions as MIDI files, we use a MIDI codec package for Haskell called *HCodecs*[2] by George Giorgidze, which provides lightweight MIDI import and export capabilities. We only needed to add a type for MIDI notes (with their MIDI number, start time and duration) and the functions `playNote` and `playRest` to convert notes and rests into two MIDI events `NoteOn` and a `NoteOff`. Thanks to the algebraic description of `Music` values, converting Mezzo compositions into MIDI tracks is entirely syntax-directed:

```
toMidi (Note pit dur) = playNote (prim pit) (prim dur)
toMidi (Rest dur)     = playRest (prim dur)
toMidi (m1 :|: m2)    = toMidi m1 ++ toMidi m2
toMidi (m1 :-: m2)    = toMidi m1 >< toMidi m2
```

For notes and rests, we use `prim` to get the integer representation of the pitch and duration and convert them into a MIDI track with two events. Sequential composition simply maps to concatenating the two tracks, while parallel composition uses the library's merging operation, denoted here by (><), which interweaves the two lists of messages respecting their timestamps. One of the main benefits of the Haskore system is that the algebraic description maps so elegantly to common list operations, and all the work of converting proxies into primitive values is done by the overloaded `prim` function.

All that is left to do is to attach a header to this track (containing the tempo, instrument name and key signature) and export it as a MIDI file, which is done using HCodecs functions. We also have means of configuring various attributes of the MIDI file, such as tempo, time signature or track name.

5 Related work

Formal descriptions of music are frequently used for algorithmic music composition [9] but have also been applied to analysis and music information retrieval. Martin Rohrmeier developed a formal grammar of functional harmony [12] which was then implemented as a Haskell library, *HarmTrace* [2], for music analysis and composition. This work describes harmonic constructs such as chords and progressions at the type level and has been one of the initial inspirations for Mezzo. In our partial implementation, progressions are indexed by the key, which lets us change the key of the entire progression without altering the schema:

```
inKey c_maj (ph_VI dom_vii0 ton :+ cadence auth_V7)
```

While there is substantial research on generation and analysis of music, little work has been done on checking the correctness of compositions: the system closest to ours is Chew and Chuan's *Palestrina Pal* [7], a Java program for grammar-checking music written in the contrapuntal style of Palestrina. There exist similar commercial programs and composition software plugins such as *Counterpointer*[3] and *Fux*[4], but these are also specialised to counterpoint and do not offer general purpose composition features. We are not aware of related libraries for functional languages or systems that enforce musical rules statically.

Haskell's type-level computation features are seeing increasing adoption and practical use. For example, Augustsson and Ågren describe the implementation of a statically-typed wrapper of a dynamic relational algebra library by describing schemas at the type-level [1]. However, their library does not yet demonstrate the benefits of `TypeInType`.

6 Conclusions

We have described the implementation of Mezzo, a library for composing music which statically enforces that compositions follow the rules of classical music. Users can choose from pre-defined rule sets or add their own and different rule sets can be applied to different parts of a composition.

[2] https://hackage.haskell.org/package/HCodecs

[3] http://www.ars-nova.com/cp/

[4] https://musescore.org/en/project/fux

6.1 Proxies

We chose to use proxies and reification instead of the conventional approach of programming with singletons. This decision is important: instead of trying to merge or mirror the term and type level, we make use of the term-type separation to model the music in two different ways. The term-level algebraic representation is very convenient for composition and recursive traversal, but we need the structured pitch matrix to perform rule-checking effectively. Moreover, abstract musical types (e.g. pitches) are converted directly into concrete values (e.g. MIDI numbers), so having an abstract term-level representations of musical values via singletons (or full dependent types) would bring us no obvious benefits.

6.2 Type-level computation

Haskell has many unique features in its type system, including type classes, functional dependencies, and type families. Mezzo uses most of Haskell's type system features and development has been both really enjoyable and surprisingly easy: data type promotion, GADTs and type families work seamlessly together and there is very little mental overhead needed to think and reason about programs. While we would wish that type families were first-class types so that we could write higher-order type functions, conditionals, data types, and recursion still enabled us to express musical rules effectively.

During development, we have encountered a few limitations and nuisances and some of these are already being addressed. A frequent type error we saw was related to type family applications in type class (or family) instances: this was often triggered when pattern-matching on types whose kind-variables are results of type family applications (e.g. arithmetic) [5]. For example, this is the reason why the `Vector` type's `:--` constructor has an argument of type `Vector (n-1)` instead of the more obvious `Vector (n+1)` in its return type: otherwise, to pattern-match on an argument of type `Vector`, GHC would have to reduce a type family application.

Other causes for unexpected errors were type families, as they may not reduce as far as we might expect. This made debugging difficult and was the reason why we implemented the rule system using type classes instead of type families on constraints: custom compiler errors would not always get triggered if e.g. a custom type error occurred as an argument to a type family.

While type-level programming is already quite pain-free, we thought of a few features that we would have found helpful. The large part of the rule-checking system is built using type classes, but handling overlapping instances made describing recursive rules problematic. In normal usage, closed type classes would not make much sense since the instances rarely overlap, but a separate construct acting as a closed *type predicate* could be useful for verification applications or rule-based systems. Similarly, we often felt that the lack of "kind classes" or type-class promotion forced us to write a lot of repetitive code, e.g. enumerating pitch classes. Kind classes would open the doors to pretty-printing of types, simplified implementation of singletons and ways of adapting other term-level abstractions to the type level.

6.3 Composition using Mezzo

When designing Mezzo's EDSL, our aim was to create a consistent, intuitive syntax for note, chord and melody input, which would be easy to read and write even for non-programmers. The paper could not give much detail on this aspect of the library (the approaches are not specific to type-level computation), but we have received encouraging responses from professional musicians regarding the language. Another topic we omitted for brevity is our partial implementation of the *HarmTrace* model, which lets users compose simple chord accompaniments from progression schemas.

The EDSL, rule sets and various modularisation techniques make Mezzo entirely usable even for large compositions. We have complete, working encodings of Bach's *Prelude in C Major, BWV 846*, Beethoven's *Für Elise* and Chopin's *Prelude, Op. 28, No. 20*. In Bach's piece, we could make good use of the fact that Mezzo is an embedded DSL: to exploit the repetitive rhythmic nature of the piece, we wrote a function that generates an entire bar from the five pitches appearing in it. GHC can infer all the complex types of the functions and rule-checking works as it should.

The performance of our library was not a main goal of our work and we cannot expect a type checker to match the performance of highly optimised machine code execution. Compilation times were slow but not unacceptably so: the average was on the order of 5-10 seconds for shorter compositions, but even a complex piece such as *Für Elise* compiles in under 30 seconds. Albeit this is slower than a fully term-level solution would be, users save "debugging" time by getting a clear description and location of the musical errors, which could not be achieved as conveniently with runtime checks.

Overall, Haskell provided everything we were looking for, if not more: mature and robust type-level computation features, a great medium for implementing embedded domain-specific languages and good library and community support.

[5] This problem is known and tracked under ticket #12564 on GHC Trac.

References

[1] Augustsson, L., and Ågren, M. Experience report: Types for a relational algebra library. In *Proceedings of the 9th International Symposium on Haskell* (2016), ACM, pp. 127–132.

[2] de Haas, W. B., Magalhães, J. P., Wiering, F., and Veltkamp, R. C. Automatic functional harmonic analysis. *Computer Music Journal 37*, 4 (2013), 37–53.

[3] Eisenberg, R. A. *Dependent Types in Haskell: Theory and Practice.* PhD thesis, University of Pennsylvania, 2016.

[4] Eisenberg, R. A., and Weirich, S. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium* (New York, NY, USA, 2012), Haskell '12, ACM, pp. 117–130.

[5] Eisenberg, R. A., Weirich, S., and Ahmed, H. G. Visible type application. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632* (New York, NY, USA, 2016), Springer-Verlag New York, Inc., pp. 229–254.

[6] Fux, J. J. *The study of counterpoint from Johann Joseph Fux's Gradus ad Parnassum.* No. 277. WW Norton & Company, 1965.

[7] Huang, C. Z. A., and Chew, E. Palestrina Pal: a grammar checker for music compositions in the style of Palestrina. In *Proceedings of the 5th Conference on Understanding and Creating Music* (2005), Citeseer.

[8] Hudak, P., Makucevich, T., Gadde, S., and Whong, B. Haskore music notation – An algebra of music. *Journal of Functional Programming 6*, 03 (Nov 2008), 465–484.

[9] Nierhaus, G. *Algorithmic Composition: Paradigms of Automated Music Generation*, vol. 1. Springer Verlag Wien, Jan 2009.

[10] Okasaki, C. Theoretical pearls: Flattening combinators: Surviving without parentheses. *Journal of Functional Programming 13*, 4 (July 2003), 815–822.

[11] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Washburn, G. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2006), ICFP '06, ACM, pp. 50–61.

[12] Rohrmeier, M. Towards a generative syntax of tonal harmony. *Journal of Mathematics and Music 5*, 1 (2011), 35–53.

[13] Yorgey, B. A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., and Magalhães, J. P. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (New York, NY, USA, 2012), TLDI '12, ACM, pp. 53–66.