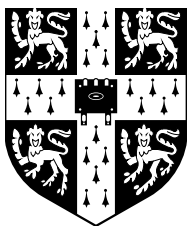
Dmitrij Szamozvancev

Computer Science Tripos – Part II

Well-typed music does not sound wrong

University of Cambridge
Downing College



May 19, 2017

Proforma

Name	**Dmitrij Szamozvancev**
College	**Downing College**
Project Title	**Well-typed music does not sound wrong**
Examination	**Computer Science Tripos – Part II, June 2017**
Word Count	**11,955**[1]
Project Originator	**D. Szamozvancev**
Supervisor	**M. B. Gale**

Original aims of the project

The design and implementation of a Haskell library for music composition which uses a type-level model of music to statically enforce rules of classical music theory. The library should let users describe compositions and check their musical correctness, e.g. presence of dissonant intervals, consecutive fifths, etc., and export compositions as MIDI files.

Work completed

All project requirements have been successfully completed, including a number of planned and unplanned extensions. The library was implemented as a dependently-typed, embedded domain-specific language called *Mezzo*, and includes features such as concise note, chord, melody and progression input, static enforcement of musical constraints, multiple levels of strictness and customisable rendering options. The model was evaluated using unit and integration tests, and several piano compositions have been described to test the library in action.

Special difficulties

None.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Dmitrij Szamozvancev of Downing College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

[Signature]

May 19, 2017

Contents

List of Figures

Acknowledgements

There are many people to thank for devoting their time and effort to this dissertation. In addition to all those who showed encouraging interest in the project idea, I would like to personally thank the following people:

- Simon Peyton Jones, for agreeing to meet me for a discussion about the project and giving valuable advice on future work,
- Richard Eisenberg, for always knowing the way out when I got hopelessly lost in the scary world of `TypeInType`,
- The anonymous review committee of the 2017 Haskell Symposium, for their detailed, constructive feedback on our paper submission,
- Alfie Wright, for proofreading the drafts of this dissertation and sharing useful academic writing advice,
- My parents, for their constant support and stimulating conversations about the nature and philosophy of classical music,
- And of course, Michael Gale, for being a wonderfully encouraging, responsible and helpful supervisor, as well as becoming a great friend in the process.

CHAPTER 1

Introduction

1.1 History and background

Music theory *Music theory*, the formal study of the aesthetic properties of music, aims to provide a scientific basis for music composition, analysis and performance [15]. It might seem surprising that a subjective, creative activity such as music composition can be formalised. But, in fact, a large part of the process is governed by various rules and conventions that have been developed over centuries of music tradition. These rules are dictated by mathematics, human biology and psychology, ethnography and even history, and describe both low-level and global features of compositions [31]. Enforcing them is a simple and largely algorithmic process, but it is also error-prone – indeed, an important part of a composer's education is learning to apply the rules and identify mistakes as quickly as possible.

Computer music Computers drastically changed how music is created and consumed [6]. Not only do they simplify the composing, recording and editing processes, but they also became the foundation of entirely new genres of music. One of the early applications of computers was *algorithmic music composition*, i.e. programming computers to generate music autonomously [33]. Many of the techniques used for automatic composition are based on some encoding of music theory as a probabilistic model [16] or generative grammar [27]. Another popular genre is *live coding*, which involves describing and transforming musical compositions in real time, using a specialised live coding library or domain-specific language [1].

Computer-assisted composition Music creation has become an increasingly digital process, and there are many software tools to assist both professional and novice composers. Many programs offer features such as music notation and playback which are replacing traditional pen-and-paper composition. Other applications aim to provide musical inspiration or even generate parts of a composition automatically, such as the harmony or rhythmic accompaniment. Somewhat surprisingly, there is a lack of tools which automate the task of musical rule enforcement, despite the fact that it is a common, tedious and error-prone process. This project is an attempt to create such a tool.

1.2 Project description

In this project I design and implement an embedded domain-specific language, *Mezzo*, for describing musical pieces, which uses a type-level model of music to statically enforce rules of classical composition. If a composition expressed in the language does not follow the rules of classical music, it is not a valid program and leads to a type error. For example, a consonant fifth interval described by playing a G and a C quarter note at the same time sounds good, so this Mezzo expression compiles:

```
GHCi> g qn :-: c qn
```

However, playing a B and a C at the same time produces a very harsh-sounding, dissonant major seventh interval, so a Mezzo expression describing it does not typecheck:

```
GHCi> b qn :-: c qn

  error: Can't have major sevenths in chords: B and C.
```

The implementation of Mezzo involves extensive use of type-level computation. This field has been gaining attention in the last few years, especially in the area of *dependently-typed programming* [2, 32]. Common use cases for type-level computation are theorem proving and static verification, but this project will serve to show that dependent types can be applied to domains far removed from theoretical computer science or critical systems.

Though there are a number of fully dependently-typed functional languages available, I decided to use *Haskell* [21] as the implementation language. The main appeals of Haskell are its maturity, strong and stable type system, the availability of many external packages (for example, MIDI codecs) and a robust optimising compiler with good support for type-level computation and code generation. Thanks to its clean syntax and functional nature, Haskell is also a great language for implementing embedded DSLs, which is the main deliverable of this project. Additionally, this project provides a practical, nontrivial case study of dependently-typed programming in Haskell and constructive evidence that the language is more than capable of handling sophisticated type-level computation without being a fully dependently-typed language yet.

1.3 Related work

Music composition is a popular use case for Haskell, and several libraries are available for both music description and generation. One of the first related libraries was a composition system called *Haskore* by Hudak et al. [18], which describes music as a recursive algebraic structure with operators for parallel and sequential composition of musical pieces. The Haskore music algebra forms the basis of Mezzo's music model, extended with higher-level structures such as chords and progressions. Haskore was superseded by Euterpea[1] which is used in research on sound synthesis and grammar-based composition [39].

[1] `http://euterpea.com/`

Mezzo borrows ideas from several other Haskell EDSLs, both for music and other media; in particular, the Music Suite[2] composition framework and the vector graphics EDSL Diagrams[3]. However, the note and chord input method is implemented using a technique that I have not seen used for this purpose before.

Research by Magalhães et al. lead to the development of several music analysis and composition tools revolving around type-level modelling of functional harmony [5, 26, 29]. Mezzo uses a similar approach to enforce harmonic structure, but also implements static checks for low-level rules of note and chord composition.

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 Mezzo is Chew and Chuan's *Palestrina Pal* [17], a Java program for rule-checking music written in the contrapuntal style of Palestrina. Similar GUI-based programs and plug-ins are *Counterpointer*[4] and *Fux*[5], but these are also specialised to contrapuntal compositions. I am not aware of similar libraries for functional languages or systems that enforce musical rules statically.

[2] `http://music-suite.github.io/`

[3] `http://projects.haskell.org/diagrams/`

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

[5] `https://musescore.org/en/project/fux`

CHAPTER 2

Preparation

This chapter gives an overview of the topics and approaches explored in preparation for the project, including the basics of music theory and dependently-typed programming in Haskell needed to implement it. An account of the software engineering practices employed during the project implementation is also given.

2.1 Musical preliminaries

This section provides a brief glossary of musical terms that are used in this dissertation. A more detailed overview is given in Appendix A, and a broader treatment of music theory can be found in any standard textbook – I used Walter Piston's *Harmony* [37]. For simplicity, I use analogies based on the keys of a piano. I annotate the examples (Figs. 2.1 to 2.3) with their corresponding encoding in the Mezzo EDLS to give the reader an illustration of how they relate.

2.1.1 Glossary

Semitone The smallest frequency interval used in Western music, found between two adjacent keys on a piano (e.g. a white and a black key). Two semitones make up a *tone*.

Pitch An absolute frequency specified by an *octave* (the range between a frequency and its double), *pitch class* (denoted by a capital letter A-G which represents the position in the octave), and an *accidental* (the shift up or down by a semitone, specified by a ♯ or ♭), corresponding to a key on a piano.

Duration A time interval expressed as a negative power of two: whole, half, quarter, etc. A quarter duration lasts as long as two eighths. A *dotted* duration is a duration extended by its half: a dotted half note lasts as long as three quarters.

Note A musical unit given by a pitch and its duration, e.g. a C natural quarter note or a G sharp dotted eighth note.

Rest A musical unit of silence, given only by its duration.

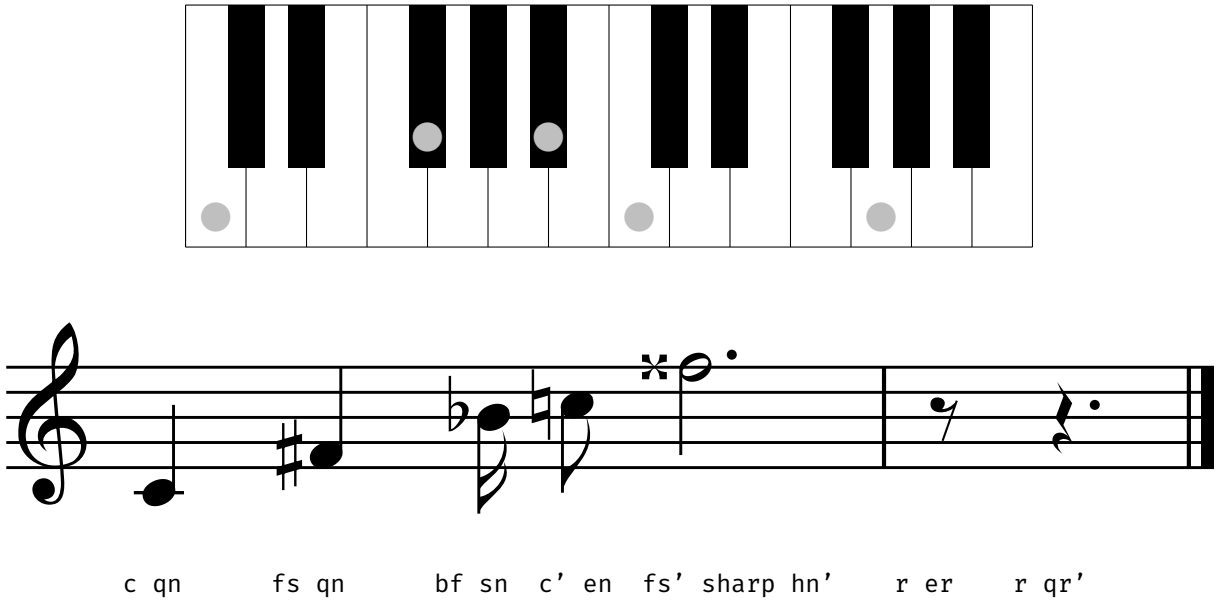


Figure 2.1 – Examples of notes and rests, with the corresponding piano keys. From left to right: C4 quarter, F4 sharp quarter, B4 flat sixteenth, C5 natural sixteenth, F5 double sharp dotted half, eighth rest, dotted quarter rest.

Interval The distance between two pitches (keys on the piano), either played at the same time (harmonic interval), or one after another (melodic interval). An interval is specified by its size (e.g. second, fifth, octave) and class (e.g. perfect, major, augmented).

Perfect interval Unisons, fourths, fifths and octaves are called *perfect intervals* (due to the simple ratio of their frequencies), they can be *diminished* or *augmented* by shrinking or expanding them by a semitone respectively.

Imperfect intervals Seconds, thirds, sixths and sevenths are *imperfect intervals*, they can be *major* or *minor*, and can also be diminished and augmented.

Chord Two or more notes played at the same time, usually separated by major or minor thirds. Chords are specified by their lowest note (*root*), type (e.g. minor third dyad, diminished triad, dominant seventh, doubled augmented triad), and *inversion* (rotation by shifting lower notes up by an octave).

Scale Any subsequence of the 11 pitches in an octave. A *diatonic scale* consists of 7 pitches separated by second intervals whose nature determines the *mode* of the scale (e.g. major, minor). *Tonal music* mostly consists of the pitches of a particular diatonic scale, specified by the *key* (e.g. C major key). The relative position of a pitch in a scale is given by its *scale degree*, e.g. degree III marks the third note of any scale.

Harmonic function The *function* of a chord expresses its role in a key. A *tonic* chord (built on the first degree of a scale) represents stability, while a *dominant* chord (fifth degree) creates tension which has to be *resolved* by returning to the tonic. A *subdominant* (fourth degree) starts creating tension but can be followed by either a dominant or a tonic.

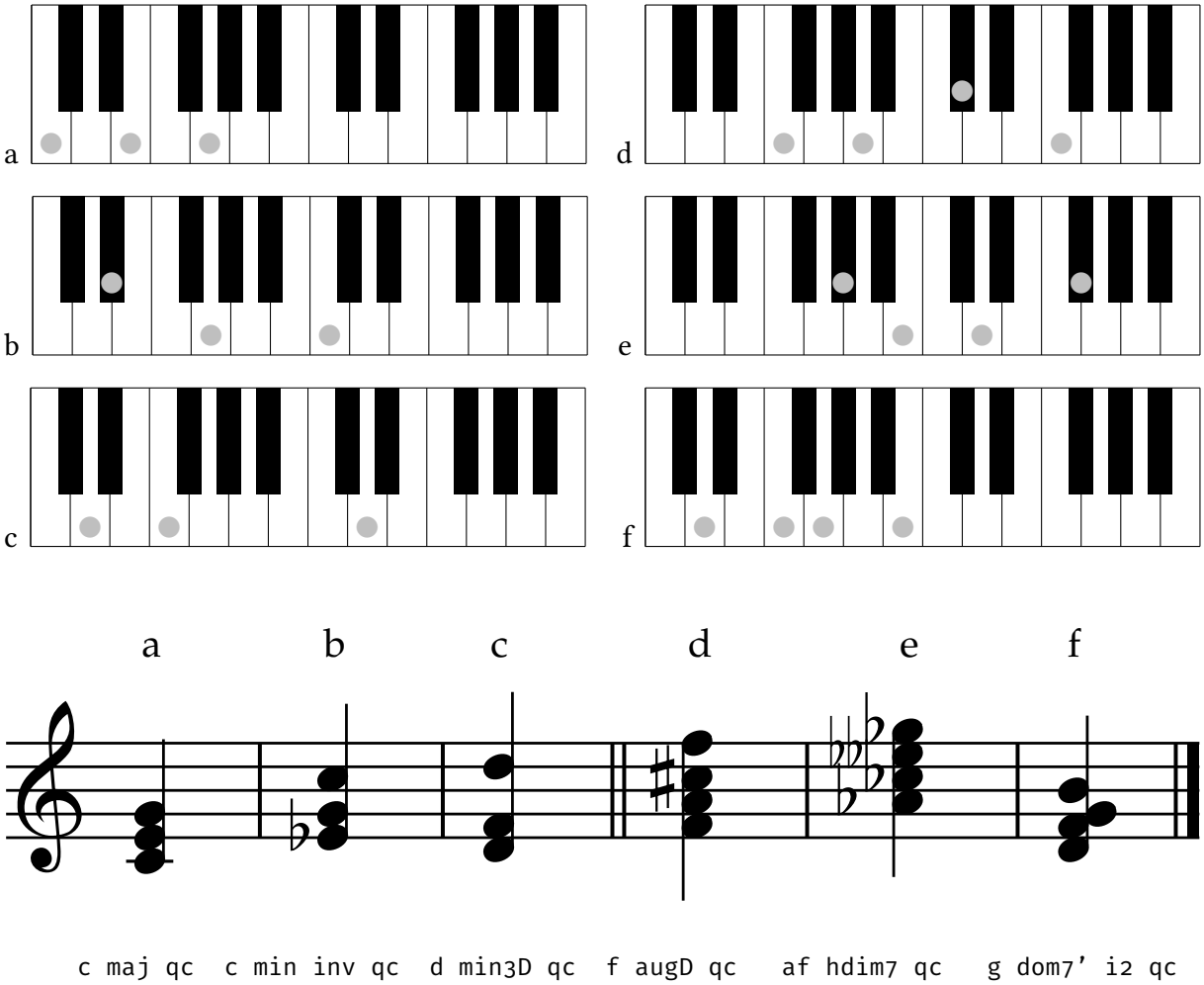


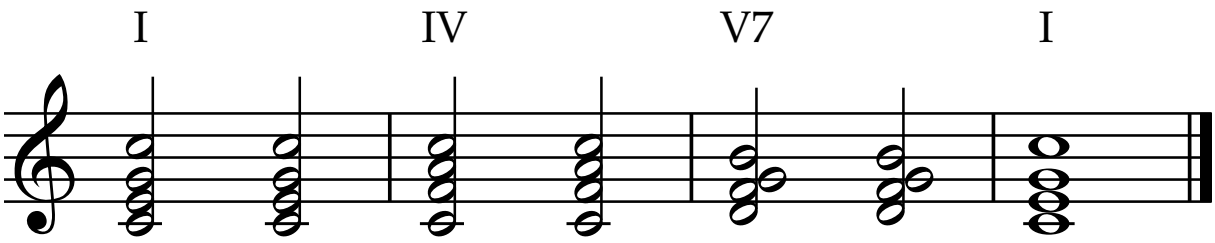
Figure 2.2 – Examples of chords, with the corresponding piano keys: (a) CM triad, (b) Cm triad in first inversion, (c) doubled D minor third dyad, (d) doubled Faug triad, (e) half diminished A♭ seventh chord, (f) dominant G seventh chord in second inversion.

Chord progression A sequence of chords which follows the conventions of functional harmony. For example, a common progression is I–IV–V–I. Progressions often end in a *cadence* to provide a closure to a piece: these create maximum tension (e.g. with a dominant seventh chord) which is then resolved into the tonic of the key.

Voice A single, continuous melodic line of music.

Homophony A compositional technique where a single upper voice is embellished by an accompaniment, e.g. a chord progression.

Counterpoint A polyphonic (multi-voice) compositional technique where each voice sings an independent melody, but they give a coherent whole when played together. *Strict counterpoint* has to follow many harmonic rules to ensure that the piece sounds consonant but the voices stay independent.




```
inKey c_maj (ph_I ton :+ cadence (full subdom_IV auth_V7))
```

Figure 2.3 – Example of a I–IV–V–I chord progression in C major.

2.1.2 Composition rules

To fulfil the minimum project requirements, the Mezzo library must enforce the following musical rules:

Harmonic intervals Minor seconds (one semitone, e.g. C–C♯) and major sevenths (11 semitones, e.g. C–B) sound very dissonant and are forbidden.

Melodic intervals Augmented, diminished and seventh melodic intervals (e.g. C–F♯) are hard to sing and sound disjoint so are forbidden.

Harmonic motion Voices separated by consecutive perfect intervals (fifths, octaves, e.g. C–G then D–A) do not sound sufficiently independent and are forbidden. Similarly, an interval can be followed by a perfect interval only when the voices move in different directions (contrary motion).

Cadences Progressions must end on a first degree chord (full cadence) or sixth degree chord (deceptive cadence) in zeroth or first inversion, to provide the necessary closure to a piece.

2.2 Dependent types in Haskell

This section describes the various extensions and techniques we can use to enable dependently typed programming in Haskell. While the language was not originally designed to have dependent types, they are in the process of being introduced to Haskell's leading compiler, GHC [1]. Mezzo makes use of the latest features to perform complex type-level computation.

2.2.1 Extensions

The Haskell Report 2010 [30] gives the core specification of the language, its syntax, semantics, and standard library. However, the language (and its compiler, GHC) is in very active development, and most new features are added via *language extensions* – some are experimental, others are in widespread production use. This section gives a short account of the extensions

[1] `https://ghc.haskell.org/trac/ghc/wiki/DependentHaskell`

used in Mezzo for type-level computation, developed over the last 17 years – a more in-depth overview can be found in Richard Eisenberg's PhD dissertation [7].

Type classes One of Haskell's unique features is the introduction of ad-hoc polymorphism via *type classes* [13]: means of constraining polymorphic type parameters to get access to more specialised functionality. For example, the class `Show` classifies types which can be pretty-printed: it defines a method `show` which takes a value of the instance type `a` and returns a `String`:

```
class Show a where
    show :: a -> String
```

This specifies a contract: any type that is an *instance* of the `Show` class must implement the method `show`. To pretty-print Booleans, `Bool` can be made an instance of `Show`:

```
instance Show Bool where
    show True  = "True"
    show False = "False"
```

Now, any time we want to constrain a polymorphic type variable to types which can be pretty-printed, we add a *type class constraint*, specifying that a type variable `a` can only be instantiated with types which have an instance of `Show`:

```
exclaim :: Show a => a -> String
exclaim v = show v ++ "!"
```

Generalised algebraic datatypes Algebraic datatypes let us define types as sums and products of other types. For example, polymorphic lists can be recursively defined as:

```
data List a = Empty | Cons a (List a)
```

Thus, a list of integers has the type `List Int`, while a list of lists of Booleans has the type `List (List Bool)`. However, all constructors return a value of type `List a`, and we have no way of constraining the type variable `a` on a per-constructor basis to express invariants. For instance, the above list type cannot enforce the invariant that "the list contains alternating integer and Boolean elements". The `GADTs` [36] extension lets us give explicit type signatures for each of the type constructors, including the type variable in its return type. The above specification can thus be formally expressed as a GADT:

```
data IntBoolList a where
    Empty    :: IntBoolList Int
    ConsInt  :: Int  -> IntBoolList Bool -> IntBoolList Int
    ConsBool :: Bool -> IntBoolList Int  -> IntBoolList Bool
```

Without GADTs, we could not enforce that the types alternate: `Empty` would have the polymorphic type `List a` and could not stand as the second argument for the constructors.

Type families *Type families* [41] open the door to *type-level computation*, enabling us to write functions on types which are evaluated by the type checker at compile time. Suppose that we first define type-level Peano naturals using empty data type declerations:

```
data Z      -- Zero
data S n    -- Successor of a natural number
```

Now we can declare a *closed type family* [8] which calculates the sum of two type-level naturals: this is done by a straightforward recursive definition, except we are pattern-matching and recursing on types:

```
type family Sum a b where
    Sum Z     b = b
    Sum (S a) b = S (Sum a b)
```

Datatype promotion A limitation of basic type-level programming in Haskell is that the kind system (which classifies types just as types classify terms) is very rudimentary, given by the simple grammar:

$$\kappa ::= * \mid \kappa \to \kappa$$

The kind of types is `*` (e.g. `Int :: *`), while the kind of type constructors is an arrow from kinds to kinds (e.g. `List :: * -> *`). This makes it difficult to write "kind-safe" code: for example, in the above definition of `Sum`, `a` and `b` can be any type, not just type-level naturals. This problem was solved by the introduction of *datatype promotion* [46], a way of lifting types to the kind level and data constructors to the type level (see Fig. 2.4). For example, after enabling the extension, the type

```
data Nat = Zero | Succ Nat
```

can also be used as a kind `Nat` (distinct from the kind `*`) and inhabiting types `Zero :: Nat` and `Succ :: Nat -> Nat`. Now we can be more precise in the declaration of `Sum`:

```
type family Sum (a :: Nat) (b :: Nat) :: Nat where ...
```

That is, `Sum` only takes types of kind `Nat` as arguments and returns a type of kind `Nat` as a result. A related addition is *kind polymorphism*, which is a desirable feature once we have a rich kind system. Just as the list length function `length :: [a] -> Int` is polymorphic in the type of the elements, we can define a type family `Length` for type-level (promoted) lists containing elements of the polymorphic kind `k`:

```
type family Length (l :: [k]) :: Nat where
    Length []       = Z
    Length (x : xs) = S (Length xs)
```

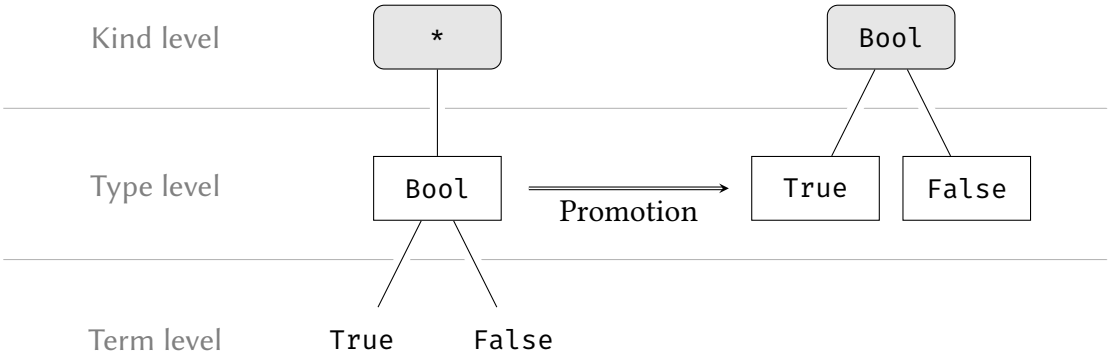


Figure 2.4 – Promotion of datatypes. After promotion, `Bool` becomes a kind, while its constructors `True` and `False` become types. Note that the promoted types `True` and `False` have no term-level inhabitants (but the original, unpromoted `Bool` type is still available).

Constraint kinds By default, Haskell's type classes are a distinct construct from types: they can only be declared using the `class` keyword and can only appear in type class constraints (on the left-hand side of `=>`). The `ConstraintKinds` [4] extension lifts these limitations by unifying types and constraints. More precisely, every type class constraint becomes a simple type with kind `Constraint`, so it can be created and manipulated using type-level functions. Similarly, any type with kind `Constraint` can be used as a type class constraint. Moreover, while type families are not first-class types (i.e. we cannot treat unsaturated type families as types), functions returning constraints can be freely passed around as first-class types:

```
type family ShowOrNum (a :: Bool) :: (* -> Constraint) where
    ShowOrNum True = Show
    ShowOrNum False = Num

f :: ShowOrNum True a => a -> String
f = show    -- Compiles, since 'ShowOrNum True a' evaluates to 'Show a'
```

Merging of types and kinds The above extensions allow for remarkably expressive type-level code: we have a strong kind system, functions on types and constraints, and ways of connecting the type and term level via GADTs. However, for some domains (e.g. generic programming), this is not enough, as there are three distinct levels of organisation (terms, types and kinds) but only two computation environments (run-time and compile-time). From version 8.0.1 onwards, GHC includes the `TypeInType` [45] extension, which solves this discrepancy by unifying types and kinds entirely: it introduces the axiom `* :: *` and renames the "kind" `*` to `Type`. That is, types and kinds can be treated uniformly, giving us kind families, kind-indexed types, GADT promotion, etc. A striking example of this unification is that we can define a heterogenous if-expression at the type level:

```
type family If  (b :: Bool) (t :: k)  (e :: k)  :: k          where ...
type family HIf (b :: Bool) (t :: k1) (e :: k2) :: If b k1 k2 where ...
```

The family `If` is a simple, homogenous type-level conditional expression, which returns `t` if `b` is `True` and `e` otherwise. We use this to statically determine the return kind of `HIf`, based on the truth-value of the argument type `b`: types and kinds can be used in a uniform way.

2.2.2 General approaches

This section introduces some of the techniques for overcoming the limitations of Haskell for dependently-typed programming. The main problems are Haskell's type erasure, and separation of the type and term level, which make type-level information unavailable at runtime.

Proxies Haskell requires all arguments to term-level functions to have types of kind `*`. This means that while promotion lets us have types of any kind, these types are not inhabited by terms and cannot be the types of arguments to functions or data constructors. This limitation makes the following type signature invalid:

```
sum :: (a :: Nat) -> (b :: Nat) -> (Sum a b :: Nat)
```

A simple solution is a *proxy*: a datatype which "holds" an arbitrarily-kinded type variable:

```
data Proxy (a :: k) = Proxy
```

Now, for any type `t`, `Proxy t` is inhabited by a term-level value `Proxy` (see Fig. 2.5), allowing us to pass around types via term-level values:

```
sum :: Proxy (a :: Nat) -> Proxy (b :: Nat) -> Proxy (Sum a b :: Nat)
sum a b = Proxy
```

We can now call the function on two `Proxy` values. The `TypeApplications` [10] extension gives a shorthand way of specifying the type variables in the type of a value:

```
sum (Proxy @5) (Proxy @7) :: Proxy @12
```

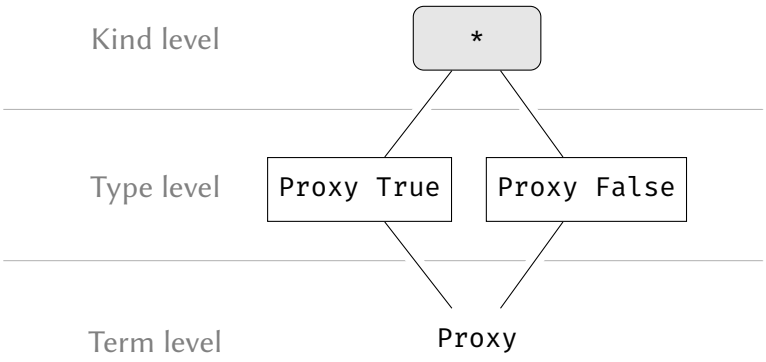


Figure 2.5 – Proxy created for a promoted `Bool` type. The kind `*` makes proxies suitable as function arguments, but they contain no term-level information.

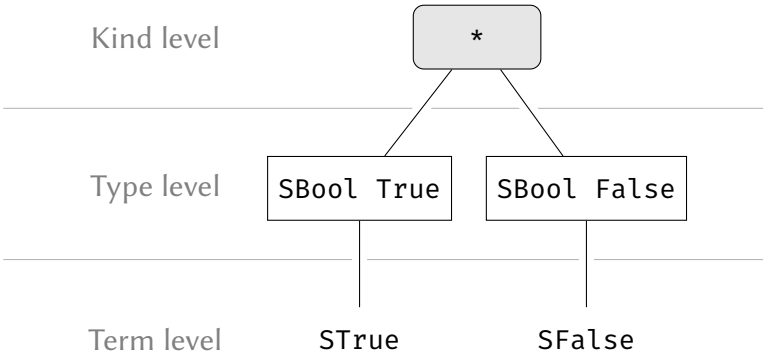


Figure 2.6 – Singleton type created for a promoted `Bool` type. Singletons have kind `*` and their term-level structure is isomorphic to the type-level structure.

Reification Proxies let us pass around types as term-level values, but all the information is still only available in the types; as the types are erased, the type-level values become unavailable at runtime. We need a way to pass down type information to the term level, that is, to *reify* the type. The conventional solution to this problem involves Haskell's type classes: for each reifiable type, we provide a function `reify` which returns the term-level value of the type:

```
class ReifyNat (n :: Nat) where
    reify :: Proxy n -> Int
```

We have to provide instances for each of the types by hand, but for recursive types we can make use of type class preconditions:

```
instance ReifyNat Z where
    reify n = 0
instance ReifyNat n => ReifyNat (S n) where
    reify n = reify (Proxy @n) + 1
```

A `Z` is reified to 0, and an `S n` is reified to the term level value of `n` plus 1.

Singletons The idiomatic way of accessing type-level information at the term level (not used in Mezzo) is entirely mirroring the types as terms (see Fig. 2.6). Types which have a single, unique inhabitant (apart from bottom) are called *singleton types* [9]. A singleton type for natural numbers would be

```
data SNat (n :: Nat) where
    SZ :: SNat Z
    SS :: SNat n -> SNat (S n)
```

With this definition, the only term inhabiting the type `SNat 2` is `SS (SS SZ)` and the type of `SS (SS SZ)` can only be `SNat 2` (thanks to the GADT declaration). Creating singletons for promoted types is a mechanical process and Richard Eisenberg's singletons[2] package can

[2] `https://hackage.haskell.org/package/singletons`

generate them automatically from a type declaration. To avoid having to define singletons by hand or delegate a lot of work to an external package, Mezzo does not use singletons, which ended up saving some work in the definition of the EDSL.

2.3 Software engineering

This section details the software engineering practices followed during the implementation of the project, as well as a formal specification of the requirements.

2.3.1 Starting point and methodologies

At the start of the project, I had a good understanding of music theory and an intermediate grasp of Haskell. However, I had little working experience in type-level programming, which – due to the lack of textbook coverage of the topic – I learned from papers and blog posts. As some of the features I am using were only added to the language in 2016, there was often little information I could find online if I got stuck – in these cases, I contacted Prof. Simon Peyton Jones and Richard Eisenberg for help.

I used the spiral methodology for the implementation, iterating between planning, testing, and adding new features. The type-level model was drafted and refined over the course of several prototypes created during the planning and experimentation phase. Code was regularly committed and pushed to GitHub [3], with large changes implemented in feature branches to allow for quick backtracking. I made good use of GitHub's project management tools to keep track of tasks and new ideas. The project was built and unit tests were run after each push by the continuous integration framework Travis [4]. An early version of the library was uploaded to Haskell's open source package archive, Hackage [5], which had received 73 downloads at the time of writing.

2.3.2 Project requirements

This section expands on the project proposal and specifies the requirements that must be fulfilled for the project to be deemed successful. The requirements for each component are given in order of priority, following the MoSCoW method guidelines.

Type-level model

MM1 The musical piece, or some accurate approximation of it, must reside in the type of a composition. This way, musical composition rules can be enforced by the type checker.

MM2 Violation of a composition rule must produce a compile-time error, that is, a musically incorrect composition must not constitute a valid program.

[3] `https://github.com/DimaSamoz/mezzo`
[4] `https://travis-ci.org/DimaSamoz/mezzo`
[5] `https://hackage.haskell.org/package/mezzo`

MM3 The model MUST enforce rules of harmonic and melodic intervals, harmonic motion and chord progression structure, as described in the proposal and Section 2.1.2.

MS1 The rule checking SHOULD have reasonable performance.

MS2 The model SHOULD support creation of notes, chords and chord progressions, as well as melodic and harmonic composition.

MC1 The model COULD produce custom compiler errors, describing the type and location of the musical error.

MC2 The library COULD support customisation of rules or different levels of strictness.

MW1 The model WON'T be used to generate compositions.

Music description language

LM1 The language MUST be embedded in the Haskell language, i.e. compositions must be compilable by the Glasgow Haskell Compiler without any preprocessing.

LM2 The users MUST be able to interact with the library through term-level values and functions, not types.

LS1 The EDSL SHOULD provide a concise, flexible method of music input, with little mental burden or boilerplate.

LS2 The users SHOULD not need to be concerned with the types of compositions and the compile-time computation happening in the background.

LS3 The users SHOULD be able to input notes, rests and chords, as well as compose pieces melodically or harmonically.

LS4 The users SHOULD be able to use the library with minimal set-up, e.g. module imports.

LC1 The EDSL COULD provide a shorthand notation for chord progression input.

Exporting

EM1 The users MUST be able to export compositions as MIDI files.

ES1 The users SHOULD be able to specify MIDI attributes such as tempo or time signature.

EC1 The library COULD support exporting into music notation formats such as MusicXML, ABC or LilyPond.

CHAPTER 3

Implementation

This chapter describes the implementation of the Mezzo library and EDSL. The system can be split into three main components: the music model, the music description language, and the rendering module (see Fig. 3.1).

Music model The core component of the library is responsible for modelling and checking the correctness of musical compositions. To have static access to musical information, we need a suitable type-level representation of compositions – this is explained in Section 3.1.1. Section 3.1.2 describes the rule system implemented in the library. Section 3.1.3 provides a high-level overview of the central datatype of the model, and how it combines the ideas of *Haskore* with dependent types. Section 3.1.4 gives a detailed look at the functional harmony model, inspired by *HarmTrace*.

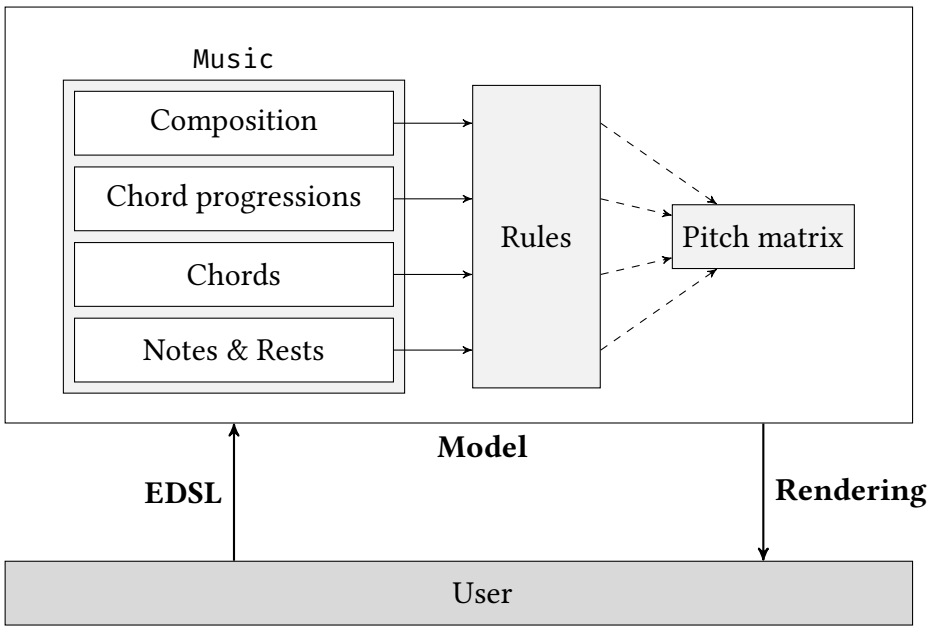


Figure 3.1 – Structural overview of the *Mezzo* library. The user writes compositions using the EDSL which are converted into terms of the `Music` datatype. The correctness of the compositions is checked, and correct pieces get translated into the type-level music representation. Correct compositions can then be rendered into MIDI files.

Music description language The Mezzo MDL provides an interface between the user and the model via an embedded domain-specific language for music composition. The implementation relies heavily on compile-time code generation, introduced in Section 3.2.1. Next, Section 3.2.2 describes a pattern for building fluent EDSLs in a functional setting, and Section 3.2.3 presents shorthand methods for inputting melodies and chord progressions.

Music rendering Mezzo lets users export compositions into MIDI files. Section 3.3.1 addresses the challenge of converting type-level information into primitive values, and Section 3.3.2 shows how these values are then rendered as MIDI files.

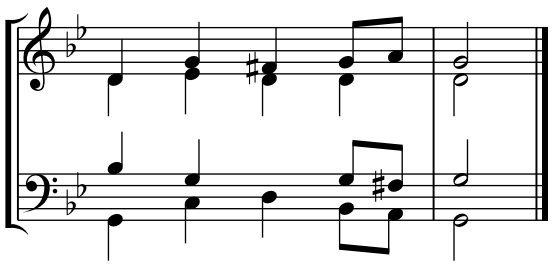
3.1 Music model

This section develops the type-level model of music used by Mezzo and presents the majority of the type-level computation techniques implemented in the library.

3.1.1 The pitch matrix

Our first task is to find a consistent, structured representation of compositions at the type level – without this, we would not be able to analyse music statically. I decided on a straightforward, somewhat brute-force approach: keeping the music in a two-dimensional array of pitches (see Fig. 3.2). The columns of the matrix represent durations and the rows are individual voices. The matrix elements are pairs of pitches and durations (which specify a note). Although it might seem like overkill, we cannot get around the need to keep all relevant information in the types: for example, we should always be able to harmonically compose a long melody with a long accompaniment, and ensure that all the arising intervals sound good.

The implementation of the composition rules requires 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. The usual solution to this is defining a *vector* type: a dependently typed list, where the type contains not just the type of the elements, but the length of the list itself. Using Haskell's built-in type-level naturals, this datatype could be defined as a simple GADT. However, we have to remember that vectors are usually used on the term level, while we are on the type level now. Before GHC 8, it would have been impossible to declare type-level vectors. This is where the `TypeInType` extension comes into play: in GHC 8, any datatype, even GADTs, can be promoted to the type level. All we need is to define the `Vector` datatype:



♩ D	♩ G	♩ F♯	♪ G	♪ A	𝅗𝅥 G
♩ D	♩ E♭	♩ D	♪ D	♪ D	𝅗𝅥 D
♩ B♭	♩ G	𝄽	♪ G	♪ F♯	𝅗𝅥 G
♩ G	♩ C	♩ D	♪ B♭	♪ A	𝅗𝅥 G

Figure 3.2 – The pitch matrix representation of a 4-voice contrapuntal piece.

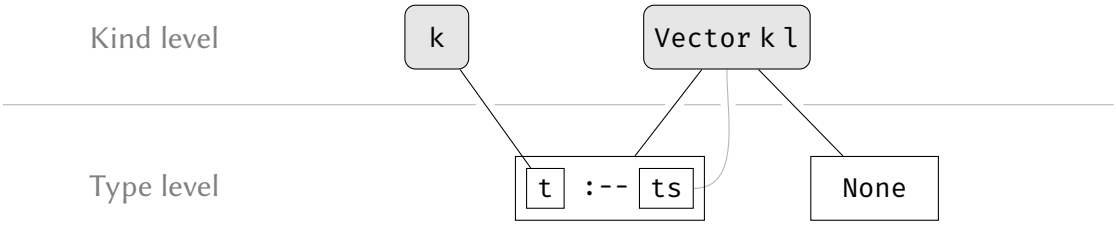


Figure 3.3 – The structure of the promoted `Vector` datatype. A vector of kind `Vector k l` has `l` type elements of kind `k`. Note that the kind variables `k` and `l` are not further classified and have the *kinds* `Type` (or `*`) and `Nat`, respectively.

the empty vector `None` has length 0 while the `(:--)` constructor takes an element of type `t` and a vector of length n, and returns a vector of length $n + 1$ (see Fig. 3.3):

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

This vector type is suitable for storing the rows of our matrix (the individual voices), but in those voices we need to store both pitches and durations. Moreover, the duration of the pitch needs to affect the length of the row, that is, we want the kind to depend on the type of the argument. This dependence on the argument types suggests using a proxy for the number of repetitions, and a GADT for the element type. `(:*)` constructs an `Elem t n` from a value of type `t` and the proxy for the number of occurrences:

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

This dependent element type can now be used to declare the type of vectors optimised for element repetition (see Fig. 3.4): in this case, the `(:-)` constructor takes an element of type `Elem t l` (containing ℓ repetitions of a value of type `t`) and the tail of length n, and returns a vector of length $n + \ell$. This way, the length of an `OptVector` is the sum of the number of repetitions of its elements, i.e. the total duration of the notes:

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

The `Matrix` type is defined as a simple type synonym:

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

Thanks to GADT promotion, all these types are now available at the kind level, and we can declare type families for common list- and matrix operations. In particular, we need horizontal `(+|+)` and vertical `(+-+)` concatenation of matrices:

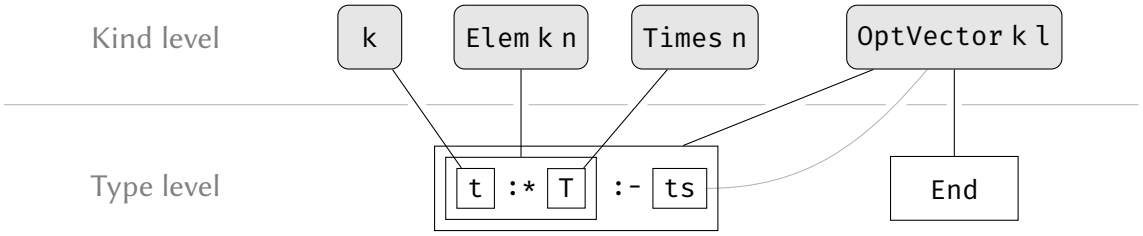


Figure 3.4 – The structure of the `OptVector` datatype. Every element `t` of kind `k` repeated `n` times has the dependent kind `Elem k n`, and the length `l` of the vector is equal to the sum of the length of the individual elements.

```
type family (a :: Matrix t p q) +|+ (b :: Matrix t p r)
            :: Matrix t p (q + r) where
    None          +|+ None          = None
    (r1 :-- rs1) +|+ (r2 :-- rs2) = (r1 ++ r2) :-- (rs1 +|+ rs2)

type family (a :: Matrix t p r) +-+ (b :: Matrix t q r)
            :: Matrix t (p + q) r where
    m1 +-+ m2 = ConcatPair (Align m1 m2)
```

`(+|+)` simply appends the individual rows of the matrices. `(+-+)` is more tricky, since the "grids" of the two matrices might not align: we need to fragment both of them so the number of cells matches up (see Fig. 3.5; the exact details are omitted for brevity).

Finally, we need to describe musical values at the type level – this is a straightforward application of datatype promotion. All types which the user can later interact with need term-level values: this is accomplished by creating kind-constrained proxies. For convenience, I also define specialised types for pitch vectors and matrices.

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

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

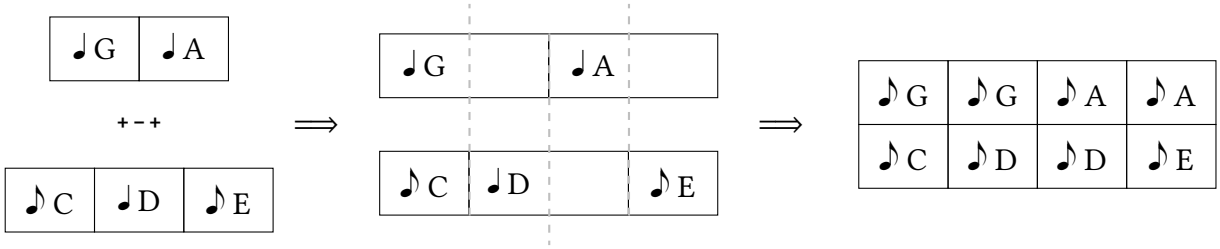


Figure 3.5 – Harmonic concatenation. Note how the quarter notes are fragmented and aligned with the eighth notes in the other voice – this way, the matrix stays well-formed.

A `Partiture n l` is our pitch matrix: it contains `n` voices of length `l` each. The next section describes how the pitch matrix is used to implement the rule-checking component of the model.

3.1.2 Musical constraints

Intervals

The rules implemented in Mezzo mainly constrain the *musical intervals* arising between two composed pieces. I declared a type family called `MakeInterval` to find the interval between two pitches. 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 natural and a `pc2` sharp in the same octave is the interval between the C natural and a `pc2` natural expanded by a semitone:

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

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, the `ValidHarmInterval` type class is declared, classifying intervals which are harmonically valid:

```
class ValidHarmInterval (i :: IntervalType)
```

GHC's *custom type error* feature lets us explicitly state which instances are invalid, and specify an accompanying compiler error. That is, when the parameter `i` is instantiated with an invalid interval, the precondition type class constraint is checked and a type error is encountered. At this point, type-checking fails and the message is displayed as a type error.

```
instance TypeError (Text "Can't have major sevenths in chords.")
      => ValidHarmInterval (Interval Maj Seventh)
instance TypeError (Text "Can't have minor seconds in chords.")
      => ValidHarmInterval (Interval Min Second)
```

Lastly, we want to state that "everything else" is a valid interval. For this, we just state that any interval `i` is valid, and use a compiler pragma to handle overlapping instances correctly: it tells the type checker that it should prioritise more specialised instances declared for a class:

```
instance {-# OVERLAPPABLE #-} ValidHarmInterval i
```

We now need to "apply" this rule to the pitches in our pitch matrix. This is done by a series of inference rules, which are straightforward 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 (MakeInterval a b) => ValidHarmDyad a b
```

When working with constraints, a useful abstraction is enabled 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, checking if a vector of types satisfies a constraint, or a type satisfies all the constraints in a vector. In our case, we apply a binary constraint `c` to two optimised vectors:

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

If the vectors are empty, the constraint trivially holds (expressed as `True ~ True`, a tautology where ~ is the type equality constraint). For the recursive case, we form a tuple, or conjunction, of constraints: apply `c` to the heads as `(c x y)` (we can ignore the durations after the `(:*)`) and recursively apply `c` to the tails of the vectors.

Now we can define validity for the harmonic concatenation of two voices (see Fig. 3.6). `ValidHarmDyad`, as defined above, is a two-parameter type class, so it has 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 (`Partitures`). Given two matrices `(v :-- vs)` and `us` (where `v` is the topmost voice of the

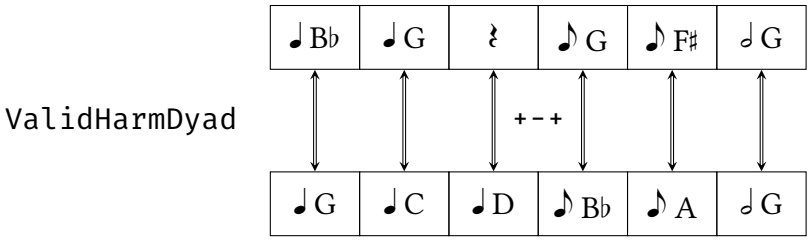


Figure 3.6 – Validating harmonic composition. These pairwise comparisons are made between each pair of voices in the composed pieces.

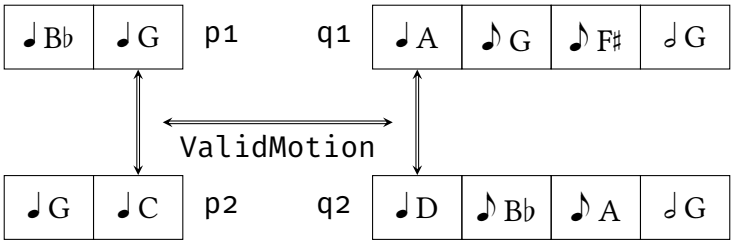


Figure 3.7 – Validating harmonic motion. Note that this example would not typecheck: the C-G interval moves into the D-A interval, causing parallel fifths.

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:

```
class ValidHarmConcat (ps :: Partiture n1 l) (qs :: Partiture n2 l)
instance ( ValidHarmConcat vs us
         , AllSatisfy (ValidHarmDyadsInVectors v) us
         ) => ValidHarmConcat (v :-- vs) us
```

By translating logical expressions into type class constraints, I implement similar techniques for other musical rules. `ConstraintKinds` lets us be very flexible in expressing rules, since we can declare type families that return a `Constraint`. For example, rules of harmonic motion are computed using the `ValidMotion` type family, which takes the four adjacent pitches as arguments and returns a `Constraint` (see Fig. 3.7). `ValidMotion p1 p2 q1 q2` is therefore a valid constraint, so it can stand on the left-hand side of `=>`.

We have now seen how Mezzo stores music at the type level and checks the correctness of musical operations. The source code of the model does not contain a single term-level definition: only types, type classes and type families – yet all of the computation used in the library is performed here, statically.

The next section introduces the musical structure that the users interact with.

3.1.3 The central datatype

This section deals with the datatype that enables interaction between the type-level music model and term-level operations. The main inspiration comes from *Haskore*, an early music description library developed by Paul Hudak's research group [23]. The library treats music as an algebraic structure with two associative operators: sequential (melodic) and parallel (harmonic) composition. In BNF syntax, a musical piece M can be expressed as:

$$M ::= \text{Note} \mid \text{Rest} \mid M :|: M \mid M :–: M$$

That is, a piece of music M is either a note, a rest, two pieces of music composed together sequentially or two pieces of music composed together in parallel (see Fig. 3.8). This model can be used to separate the description and performance of music and establish algebraic

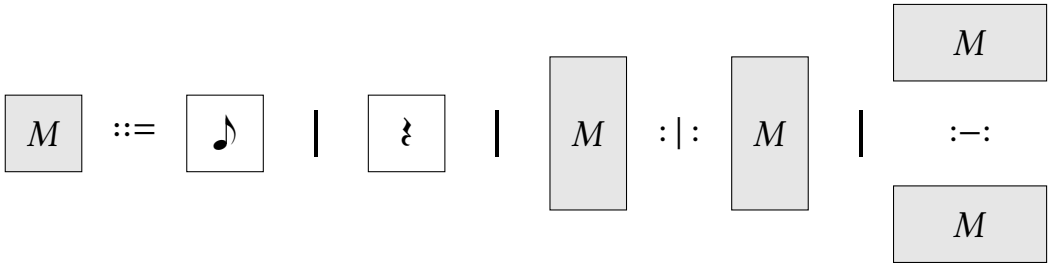


Figure 3.8 – Representation of the *Haskore* music algebra. A piece of music is either a note, a rest, melodic composition of pieces or harmonic composition of pieces.

laws which help formal reasoning about compositions. The approach was extended to other structures as well and generalised as the algebraic theory of polymorphic temporal media [19].

A straightforward translation of the above BNF description into Haskell is as follows:

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

This describes a tree-like structure with the leaves containing a note (with some pitch and duration) or a rest (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 any `Music` value will sound good, as there is nothing to constrain their structure. The rest of this section describes how the Haskore model is augmented with the notion of correctness via dependent types in Mezzo.

To make the library statically "aware" of composition rules, we need to have access to the musical information at compile-time. This can be achieved by adding a type argument `m` to the type above, containing the pitch matrix of the composition. This pitch matrix reflects the term-level contents of a `Music` value, that is, every `Music` value is *dependently typed*. The steps required to transform the Haskore music algebra into a dependently-typed, rule-checked music model are as follows:

1. Add a type variable to the `Music` datatype which represents the pitch matrix of the composition.

2. Turn `Pit` and `Dur` into *proxies*, making the pitch and duration information available at the type level.

3. Define operations which convert pitches and durations into pitch matrices. For example, `FromPitch p d` creates a singleton pitch matrix containing the pitch `p` with duration `d`.

4. Turn `Music` into a *generalised algebraic datatype*, so we have control over the return types of the individual constructors. This is where the functions described in the previous step, as well as the matrix concatenation operations `(+|+)` and `(+-+)` can be applied to the type variables of the constructor arguments.

5. Apply the type class constraints expressing musical rules to the type variables of the constructor arguments. This ensures that a constructor can only be used if the arguments conform to these musical rules: for example, two pieces of music can be combined with `(:-:)` only if their pitch matrices can be harmonically composed.

After completing these steps, we get the central datatype used in Mezzo:

```
data Music :: Partiture n l -> Type where
    (:|:) :: ValidMelComp m1 m2
          => Music m1 -> Music m2 -> Music (m1 +|+ m2)

    (:-:) :: ValidHarmComp m1 m2
          => Music m1 -> Music m2 -> Music (m1 +-+ m2)

    Note  :: ValidNote p d
          => Pit p -> Dur d -> Music (FromPitch p d)

    Rest  :: ValidRest d
          => Dur d -> Music (FromSilence d)
```

The advantage of this approach is that the rules and the structure of music are decoupled: we can easily modify the existing rules, add new ones or completely replace the system with one suitable for, say, jazz compositions. Indeed, Mezzo provides several *rule sets* which enable changing between different levels of strictness. The `Music` datatype can itself be extended with new top-level constructors using the same recipe: define proxies, conversion functions and constraints, then add a data constructor:

```
data Music :: Partiture n l -> Type where
    ...
    Chord :: ValidChord c d
          => Cho c -> Dur d -> Music (FromChord c d)

    Prog  :: ValidProg t p
          => TimeSig t -> Prog p -> Music (FromProg t p)
```

This is how the main Mezzo datatype is defined: the EDSL creates and manipulates `Music` values which can then be rendered into MIDI files. The term-level compositions are structurally different from the type-level pitch matrices: the latter is a simplified representation, abstracting over the tree-like shape of `Music` values, and enabling systematic rule-checking of composition operations.

3.1.4 Harmony model

Mezzo implements a harmonic model heavily inspired by *HarmTrace*, a harmony analysis framework developed by Magalhães et al. [28] and used in the online chord progression

$$
\begin{aligned}
\textit{ProgType} &::= \textit{Cad} \mid \textit{Phrase} := \textit{ProgType} \\
\textit{Phrase} &::= \textit{Ton Dom Ton} \mid \textit{Dom Ton} \mid \textit{Ton} \\
\textit{Cad} &::= \mathrm{V\ I} \mid \mathrm{V7\ I} \mid \mathrm{vii^o\ I} \mid \mathrm{I^6_4\ V7\ I} \mid \mathrm{V7\ vi} \mid \textit{Subdom Cad} \\
\textit{Ton} &::= \mathrm{I} \mid \textit{Ton Ton} \\
\textit{Dom} &::= \mathrm{V} \mid \mathrm{V7} \mid \mathrm{vii^o} \mid \mathrm{II7\ V7} \mid \textit{Subdom Dom} \mid \textit{Dom Dom} \\
\textit{Subdom} &::= \mathrm{IV} \mid \mathrm{ii} \mid \mathrm{iii\ IV} \mid \textit{Subdom Subdom}
\end{aligned}
$$

Figure 3.9 – Simplified syntax of the Mezzo harmonic grammar. In the implementation, all elements are annotated with the key of the piece and length information.

generation tool *Chordify*[1]. The aim of such a model is the abstract representation of functional harmony, e.g. chord progressions. While the HarmTrace model is more robust (handling, for example, borrowed chords and transposition), the implementation in Mezzo includes some ideas that are made possible by the type-level music model, such as abstraction of the key and key-dependent chord quality.

Chord progressions are schemas for harmonising melodies, based on the functional interpretation of various chords in a key. For example, the tonic chord (chord built on the first degree of the scale) represents stability, while the dominant (chord on the fifth degree) creates harmonic tension which has to be resolved. Martin Rohrmeier [40] used these harmonic conventions to develop a formal grammar of harmony, and Mezzo implements a subset of this grammar by converting production rules into datatypes (see Fig. 3.9).

A chord progression consists of a sequence of *harmonic phrases* and ends with a *cadence*, a closing phrase – this ensures that all progressions must have a closure, in accordance with the requirements. This is implemented as a vector-like data structure which ends with a cadence (instead of `Nil`) and keeps the key of the piece and the length of the progressions in its type parameters:

```
data ProgType (k :: KeyType) (l :: Nat) where
    CadPhrase :: Cadence k l -> ProgType k l
    (:=)      :: Phrase  k l -> ProgType k n -> ProgType k (n + l)
```

Phrases consist of a sequence of *tonic* and *dominant regions* and have three forms: tonic-dominant-tonic (I–V–I), dominant-tonic (V–I) or tonic (I). These functional regions are in turn expressed as datatypes resembling CFG rules. For example, a dominant region (`Dominant`) can consist of only a major fifth degree chord (`DomVM`), a secondary dominant (on the second degree) followed by a dominant seventh chord (`DomSecD`), or a subdominant region followed by a dominant region (`DomSD`). `DegreeC` is just a type synonym for a seventh chord, specifying the scale degree, quality, and key of the chord.

[1] `https://chordify.net/`

```
data Dominant (k :: KeyType) (l :: Nat) where
    DomVM   :: DegreeC V MajQ k Inv2 o
            -> Dominant k 1
    DomSecD :: DegreeC II DomQ k Inv0 o
            -> DegreeC V  DomQ k Inv2 (OctPred o)
            -> Dominant k 2
    DomSD   :: Subdominant k ls
            -> Dominant k ld
            -> Dominant k (ls + ld)
```

Haskell's algebraic datatypes make expressing CFG production rules easy, and the use of GADTs ensures that all elements have the same key and keep track of the length of the progressions. Since the `DegreeC` type also holds the chord inversion and the base octave, we can manipulate these so that the notes of the chords are as close together as possible, making the progressions conjunct and fluid – this is not addressed in HarmTrace, as its aim is analysis, not composition. Subdominants, tonics and cadences are described in the same way, and these abstract harmonic types are then converted into concrete chords and pitch matrices.

Next, we move down to the term level and consider Mezzo's music description language, and how it interacts with the type-level model.

3.2 Music description language

Embedded domain-specific languages provide a good compromise between syntactic freedom and built-in functionality: while we are restricted to the syntactic rules of the host language, we get all its syntactic constructs, libraries and compiler with no extra work. Mezzo's *music description language* (MDL) is a DSL embedded into Haskell: it provides a concise and flexible syntax for describing musical pieces and uses Haskell's type inference to enforce the musical rules previously described. This section discusses how the Mezzo MDL is implemented, and how it interacts with the type-level model. I also describe a general pattern for implementing EDSLs inspired by continuation-passing style, which emerged as a result of the work on this library.

3.2.1 Literal values

As explained in the previous section, the type model provides proxies for types that that are exposed to the user (see Section 3.1.1). For example, the proxy for pitch classes defines a value `MkPC` which can have the type `PC (PitchClass C)`, `PC (PitchClass D)`, and so on. The parameter of the `PC` type constructor is a *phantom type*: it is used to tag values of this type but it is not needed to construct them and does not appear in the parameters of a data constructor. This means that creating term-level literals for proxies is trivial (and very boring):

```
_c :: PC (PitchClass C)
_c = MkPC
```

```
_d :: PC (PitchClass D)
_d = MkPC
```

That is, the literal value for the pitch class C is `MkPC` with the type `PC (PitchClass C)`. Defining literals for every musical value would be very repetitive, so most of the declarations are generated at compile-time using *Template Haskell*, Haskell's compile-time metaprogramming framework [43]. For example, the following function generates all the literal declarations for pitch classes (`_c`, `_d`, etc.):

```
pitchClassLits :: DecsQ
pitchClassLits = genLitDecs pcFormatter "PC" ''PitchClass
```

The function `genLitDecs` takes a *formatter* (a function that transforms the name of a type, e.g. `C`, into the name of the constant, e.g. `_c`), the name of the proxy constructor and the type whose promoted data constructors we want to use as the proxy arguments, and returns the list of declarations for the literals. The TH approach really pays off when creating pitch literals for each pitch class, accidental and octave: a 14-line TH function generates 210 literal declarations.

Having the `Music` constructors and literal values for pitch and duration already makes it possible to write music:

```
Note _cn _ei :|: Note _dn _ei :|: Note _en _qu :|: Rest _ha
```

The above syntax is a bit verbose, and most MDLs provide shorthand notation for common composition tasks. The one designed for Mezzo is based around an approach I call *flat builders*: this pattern can be applied to the general task of EDSL design and works well in Mezzo.

3.2.2 Flat builders

Flat builders are inspired by the various attempts to express variable argument and postfix functions in Haskell, such as *continuation-passing style* and the Okasaki's *flat combinators* [34]. Flat builder expressions do not contain parentheses, and they build a value through a series of transformations (cf. the OOP Builder pattern [3, Item 2]). Though the underlying ideas are not new, I attempted to develop them into a general pattern which can be used in the design of natural language-like EDSLs.

Variable argument functions are not natively supported by Haskell, but can be simulated by functions which take their *continuations* as an argument. To encapsulate this idea, I use type synonyms for the three main types of terms in use:

Specifiers Specify an initial value of type `t` to start the "building process". The `forall` keyword binds the type `m` without making it an explicit type parameter of `Spec`.

```
type Spec t = forall m. (t -> m) -> m
```

For example, the function `string` turns its argument into a `String` specifier:

```
string :: String -> Spec String
string inp cont = cont inp
```

Converters Convert a value of type `s` to a value of type `t`

```
type Conv s t = s -> Spec t
```

For example, the function `firstChar` converts a `String` into a `Char` specifier by taking the head element of the input:

```
firstChar :: Conv String Char
firstChar str cont = cont (head str)
```

Terminators Finish building a value of type `t` and return the result of type `r`.

```
type Term t r = t -> r
```

For example, the function `printAscii` finishes building `Char` value and returns it as an ASCII code (using the built-in function `fromEnum`):

```
printAscii :: Term Char Int
printAscii = fromEnum
```

Now, building a value is just a matter of sequencing a specifier, zero or more converters and a terminator, ensuring that the types match up. In the example below, we specify a string, convert it into a character and print it out as an integer:

```
GHCi> string "Hello" firstChar printAscii ⟶ 72
```

Unlike function composition, builders can be read (and written) from left to right with no syntactic interference, which lets us write code that reads like natural language:

```
GHCi> add 5 to 7 and display the result ⟶ "result: 12"
```

In Mezzo, builders are used to construct note values: specifiers for pitches; converters for accidentals; and terminators for durations (see Fig. 3.10). For example, a C quarter note can be written as `c qn`, while a double-sharp F dotted half note is `f sharp sharp hn'`. As specifiers are polymorphic in the type of the final value, we can use the same pitch specifiers for notes and chords: `g qn` is a G quarter note, `g maj inv qc` is a G major chord in first inversion. Though the terminator syntax is different, this only affects the 15 duration literals, and not the 210 pitch literals. As before, builder components are generated by Template Haskell macros.

3.2.3 Melody and harmony

This section provides a brief look at syntactic shorthand notations implemented in Mezzo for melody and harmony input. It also showcases several interesting variations on the classic list and vector data structures.

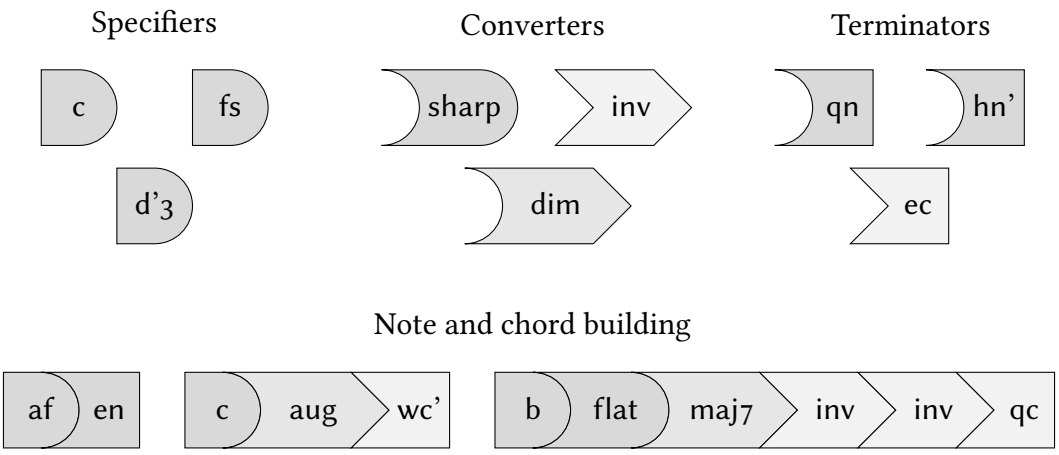


Figure 3.10 – Graphical representation of flat builders.

Melodies

Flat builders are a big step up from literals and constructors, but they are still not the most convenient way to input long sequences of notes that form a *melody*. Since writing melodies is likely to be the most common composition activity, I implemented a melody input method inspired by *Lilypond*[2], a TEX-like music typesetting system.

Using flat builders, a simple melody can be composed as follows:

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

We have to specify the duration of every note, even though it does not change that much. It is 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 above:

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

Notes are only given as pitch specifiers. The duration is either implicit (`(:|)` means "next note has the same duration as previous note") or explicitly given by constructor as an absolute duration (e.g. `(:<)` means "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 difference is that the `Melody` type keeps additional information in its type variables (like a vector), and has a constructor for every duration:

```
data Melody :: Partiture 1 l -> Duration -> Type where
    Melody :: Melody (End :-- None) Quarter

    (:|)   :: (MelConstraints ms (FromPitch p d))
           => Melody ms d -> PitchS p
           -> Melody (ms +|+ FromPitch p d) d
```

[2] `http://lilypond.org/`

```
    (:<)   :: (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 partiture and sets the default duration to a quarter. `(:|)` takes the melody constructed so far (the tail) and a pitch specifier, and returns a new melody with the added pitch and unchanged duration. The other constructors do the same thing, except they change the duration of the last note. While the syntax of the constructors might need some getting used to, they allow for very quick and intuitive melody input.

Chord progressions

Chord progressions are constructed using term-level literals and functions, for example, `dom_V7` for a dominant seventh and `ph_VI` for a dominant-tonic phrase. The subtlety of the model is that it enforces that every harmonic element has the same key, but the keys are implicit, polymorphic type variables (like the `k` in `DegreeC V DomQ k Inv2 o`). We must be able to specify the key of the elements in order to convert them into pitch matrices. We could give the key as an argument to each literal value, but that would quickly become cumbersome. Instead, we make everything a function of the key, including the elements of the phrase list:

```
data PhraseList (p :: ProgType k l) where
    Cdza :: Cad c
         -> Key k -> PhraseList (CadPhrase c)

    (:+) :: InKey k (Phr p)
         -> InKey k (PhraseList ps)
         -> Key k -> PhraseList (p := ps)
```

`PhraseList` is the term-level "singleton" of the promoted type-level `ProgType` described in Section 3.1.4. The `Cdza` constructor takes a `Cadence` proxy as its argument and returns a function from a key to a `PhraseList`. The 'cons' operator `(:+)` takes a function from the key to a phrase (encapsulated by the type synonym `InKey`), and a function from the key to a phrase list, and returns a function from the same key to a new phrase list (see Fig. 3.11). This means

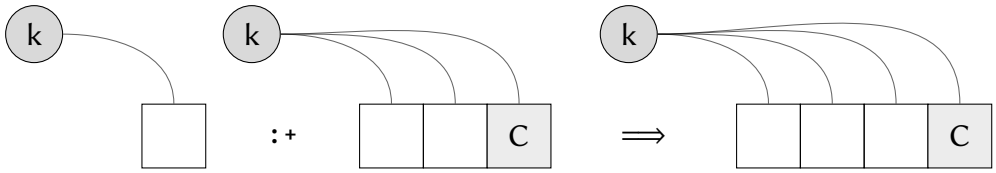


Figure 3.11 – Construction of phrase lists. The phrase and the tail must both depend on the same key k, and the result of the construction is a function from that key to a new phrase list.

that a phrase list is itself a function of the key, and specifying the key of the progression is just a matter of applying the progression to a key:

```
prog = (ph_ivi ton dom_V7 ton :+ cadence auth_V) c_maj
```

An advantage of having both the progressions and the key at the type level is that the quality of a chord (major or minor) can be statically determined from the key. The above progression can be changed from C major to C minor just by altering the key – there is no need to modify the phrase list. If a harmonic element is only valid in a certain mode (e.g. second degree minor subdominants are only available in major mode), correct usage can also be enforced statically via type class constraints. For example, the following progression would produce a custom type error:

```
GHCi> prog = (cadence (full subdom_ii auth_V)) c_min

  error: Can't have a ii subdominant in minor mode.
```

In this section we explored some of the features of the Mezzo EDSL and how it interacts with the type model. The next section describes the MIDI exporting module: while the actual creation of the file is easy using external libraries, there are some interesting challenges involved in converting `Music` terms into exportable values.

3.3 Music rendering

This section presents Mezzo's music exporting module, which renders Mezzo compositions into MIDI files. To accomplish this goal, we need to get information which is entirely at the type-level down to the term level. Users of Mezzo mainly work with proxies, which, as outlined in Section 2.2.2, contain no term-level information. We need to *reify* the types of musical constructs, i.e. make the type-level information available as term-level values. This translation between the two levels can be accomplished using Haskell's type classes.

3.3.1 Reification

Our aim is to find a primitive representation for all musical types that the user can interact with. Specifically, we want a function that can convert a proxy into a suitable term-level value. The solution is a generalisation of the `ReifyInt` type class introduced in Section 2.2.2:

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

The `Primitive` class is poly-kinded, so it can be used with naturals, pitches, chords, and so on. Its method, `prim`, takes a proxy with an arbitrary type constructor (`Dur`, `Pit`, etc.) and returns a *representation type* of the value, specified in an *associated type family*. `Rep` is a function of the instance type, so it reduces to different types for different instances of `a`. 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). 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 MIDI codes of the chord pitches. Similarly, inversions can be represented by the repeated composition of a function which rotates a list of MIDI numbers and adds 12 (the number of semitones in an octave) to the last element.

Now we declare instances of `Primitive` for our types: we have to do this by hand for the constant types (e.g. octaves and pitch classes), but can use recursive calls to `prim` for compound types, provided that we state the preconditions:

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

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

`IntRep` is another example of `ConstraintKinds` in use: it is a constraint synonym expressing that the primitive representation of a type is an integer:

```
type IntRep t = (Primitive t, Rep t ~ Int)
```

The `pc` type variable in the `Pitch` instance above 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`. As the representation is an integer for all three types, we get the MIDI number of the pitch by adding these integers together.

3.3.2 MIDI export

MIDI is a simple, compact standard for music communication, often used for streaming events from electronic instruments. The format describes music as a sequence of *MIDI messages* for various musical events, such as the beginning or the end of a note and tempo change. It is a popular standard in computer music since it abstracts away musical information from instrument sound, so converting written scores into MIDI files is straightforward.

I use a MIDI codec package for Haskell called HCodecs[3] by George Giorgidze, which provides lightweight MIDI import and export capabilities. The basic MIDI representation of a musical note is given as a Haskell datatype, containing the MIDI number of the pitch, its start timestamp and duration (both in MIDI ticks):

```
data MidiNote = MidiNote Int Ticks Ticks
```

[3] `https://hackage.haskell.org/package/HCodecs`

The function `playNote` creates a MIDI track (list of MIDI events) for the note with the given pitch and duration by concatenating a `NoteOn` and a `NoteOff` MIDI event. Thanks to the recursive description of `Music` values, converting Mezzo compositions into MIDI tracks is entirely syntax-directed:

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

For notes and rests, we use `prim` (from the `Primitive` class) to get the integer representation of the pitch and duration and convert them into a MIDI track with two events. Note that Haskell's instance resolution ensures that the correct definition of `prim` is used for each call. Sequential composition simply maps to concatenating the two tracks, while parallel composition uses the library's merging operation, denoted by `(><)`, which interweaves the two lists of messages respecting their timestamps. One of the main benefits of the Haskore model is that the algebraic description maps so elegantly to common list operations, even with a type-heavy implementation. The Mezzo-specific constructions do not require a lot of work either. For example, to render a chord, we first create a list of MIDI notes from its primitive representation, then fold the resulting list of tracks with the merge operator.

```
musicToMidi (Chord c d) = foldr1 (><) notes
    where notes = map (\p -> playNote p (prim d)) (prim c)
```

Finally, we need 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. The produced files can be played in a MIDI editor or converted into other music formats.

This chapter provided an overview of the most interesting components of the Mezzo implementation: it's type-level music model, the EDSL, and the exporting facilities. Next, I examine the achievements of the project, and compare them with the requirements.

CHAPTER 4

Evaluation

This chapter discusses and analyses the implemented project, comparing its results with the requirements set out in Chapter 2.

The aim of the project was to create a dependently-typed music composition library, and this core aim was fulfilled. Two evaluation strategies were used to demonstrate this:

Testing A common way to ensure the correctness of software projects is unit and integration testing, and there is a range of Haskell libraries which simplify these tasks. However, I realised early into the preparation that the usual testing methods would not immediately work with Mezzo: testing usually examines the correct runtime behaviour of a program, but most of the computation in Mezzo happens at compile-time. The ways in which this problem is approached are discussed in Section 4.1.

Examples The proof of the pudding is in the eating, and the proof of a composition library is in writing compositions. The project proposal required that it should be possible to encode a few classical compositions in Mezzo, which are demonstrated in Section 4.2.

I considered other techniques but decided that they would not be valuable measures of the success of this project.

Performance analysis The specification deliberately did not set strict constraints on performance, as it would be naïve to expect the same performance from complex type-level computation as execution of compiled, optimised, low-level code. While checking the correctness of compositions is not fast – on the order of 5-20 seconds, depending on the complexity of the rules and number of voices – I deemed it to be within the "reasonability" requirement. I did some performance analysis and optimisation (see Section 4.3), but there is more work that can be done.

User study While one of the deliverables of the project is an EDSL, I decided that testing its usability is outside the scope of the project: comparison to other EDSLs would be uninformative, and testing the correctness of the rule enforcement can be done without external participants. While the language was developed with usability in mind, this is ultimately not the main concern of the project.

4.1 Testing

Most of the important computation in Mezzo happens at compile-time, so testing these computations must also happen at compile-time. One approach could be putting each unit test into a separate file and checking whether they compile if they should, or fail to compile if they shouldn't. However, this is a laborious way of testing (though, due to an unforeseen bug in GHC, this is what I had to do to test the rule enforcement – see Section 4.1.4), one that can be simplified and automated using *testing frameworks*. Unfortunately – and understandably – most of the testing frameworks for Haskell operate at runtime. Therefore, our task is to write runtime unit tests for compile time computation – while this might sound counter-intuitive, there are Haskell features and libraries which allow us to do just that.

4.1.1 Deferred type errors

A naïve approach to testing compile-time code is writing down a term that should compile, and if it compiles, then the computation must be correct. There are two problems with this approach. First, it is not guaranteed that code which compiles is correct: for example, stuck type families do not produce compile-time errors in Haskell. Second, the method does not work well with testing frameworks and continuous integration: these assume that the code being tested compiles, and the only errors it produces happen at runtime. The latter problem can be solved by a recent addition to Haskell's type checker: *deferred type errors* [44]. This compiler option enables a file to compile even if it contains type errors, and turns those errors into runtime exceptions if the incorrect piece of code is evaluated. This way, all of our unit tests compile, and type errors can be caught at runtime.

4.1.2 Type equality

Unit testing generally consists of building *assertions*: for example, we assert that a function returns the expected value for some input, and the testing framework notifies us if that is ever not the case. The simplest form of assertions is equality of term-level values. For Mezzo, our task is to define a notion of *type equality* and make it "available" as a term-level assertion: for instance, we might wish to assert that `MakeInterval (Pitch C Natural Oct4) (Pitch E Flat Oct4)` equals the type `Interval Min Third`. There are multiple techniques for turning type equality into term-level tests, such as explicit type signatures or equality constraints. A neat approach uses the fact that `id`, the identity function, must have the same argument and return types. If the function type `Proxy t1 -> Proxy t2` is inhabited by `id`, we know that `t1` and `t2` must be equal. The technique implemented in Mezzo is based on the similar idea of *heterogenous propositional type equality* [25], defined as a GADT:

```
data (a :: k1) :~: (b :: k2) where
    Refl :: a :~: a
```

This defines a datatype `(:~:)` of kind `k1 -> k2 -> Type` with a single constant inhabitant `Refl`. The type of this inhabitant is `a :~: a`, indicating that the two sides of `(:~:)` must be equal. If we have a value `Refl` of type `t1 :~: t2`, we know that `t1` is equal to `t2` – that is,

we have a *term-level proof* of the equality of the two types. While this formulation has more complex uses (for example, in generic programming), I only use it as a "unit test" for type equality:

```
mkIntval :: MakeInterval (Pitch C Natural Oct4) (Pitch E Flat Oct4)
         :~: Interval Min Third
mkIntval = Refl
```

This definition type-checks only if `Refl` is a valid inhabitant of the type, and `Refl` is a valid inhabitant of the type only if `MakeInterval (Pitch C Natural Oct4) (Pitch E Flat Oct4)` evaluates to `Interval Min Third`. Moreover, by deferring type errors to runtime, type equality can be tested with normal unit test assertions, as described next.

4.1.3 Typeability assertion

Type equalities give us term-level unit tests for compile-time computation, and deferred type errors turn type errors into runtime errors. All we need now is to turn proofs of type equality into *assertions*, which are then handled by the Haskell test framework, HSpec[1]. For this, I use the should-not-typecheck[2] package by Callum Rogers. It defines an assertion, `shouldNotTypecheck`, that passes only if its argument does not compile. I made a simple variant of this assertion which does the opposite: passes only if its argument type-checks. The main idea behind these assertions is that they force the evaluation of the argument in an environment that handles exceptions; if a `TypeError` exception (via deferred type errors) is caught, a pass (or failure) is asserted:

```
shouldTypecheck :: NFData a => a -> Assertion
shouldTypecheck a = do
  result <- try (evaluate (force a))  -- Using Haskell's do-notation
  case result of
    Right _ -> return ()  -- Test passes
    Left (TypeError msg) -> assertFailure ("Term didn't compile.")
```

This lets us write our unit tests in an idiomatic way:

```
main = hspec $
       describe "MakeInterval" $
         it "should correctly create intervals from pitches"
           (shouldTypecheck mkIntval)
```

If `mkIntval` type-checks, the test is passed – otherwise, we get a test failure instead of a compiler error or runtime exception. Similar assertions test most of the type-level computation happening in Mezzo, and this helped locate several bugs during the development process.

[1] http://hackage.haskell.org/package/hspec
[2] http://hackage.haskell.org/package/should-not-typecheck

4.1.4 Testing of musical rules

The methods discussed so far are used to unit test various type-level functions used in Mezzo, but the enforcement of musical rules is more complex and requires integration testing. Originally, I planned to use `shouldNotTypecheck` to test the musical rules: for example, `shouldNotTypecheck (c qn :-: b qn)` would pass, indicating that major sevenths are correctly forbidden by the language. Unfortunately, it turned out that custom type errors are not converted into runtime errors properly. This is related to a compiler bug (a so called *GHC panic*, an exception that should never occur) I discovered which is encountered when evaluating terms with deferred custom type errors. As I did not find any reports of this particular problem on GHC's issue tracking page, I submitted a bug report myself[3], which was subsequently addressed and the fix is scheduled to be included in the next GHC release. Later it turned out that this fix solved other long-standing reported problems caused by the same compiler bug.

While this bug prevents us from writing automated tests for custom type errors at the time of writing, we can still test the rules manually. I created two sets of tests: one for musical values that should compile (i.e. which do not violate musical rules, such as `c qn :-: g qn`, a perfect fifth), the other for values that should not (i.e. which are musically incorrect, such as a major seventh harmonic interval, `c qn :-: b qn`). Testing is just a matter of compiling the tests and ensuring that this either succeeds for correct values, or produces the suitable type error for musical mistakes. Below are examples of the "failing" tests, together with the compiler errors they produce:

```
-- Flattening the lowest note would bring its MIDI code outside
-- the range of allowed values, so this should be disallowed.
test "too low" $ cf_5 qn
-- error: Note can't be lower than C natural of octave -1: Cb_5.

-- In the strictest rule set, major seventh chords are forbidden.
test "major seventh" $ d maj7 qc
-- error: Can't have major seventh chords: D Maj7.

-- Minor second degree subdominants are only allowed in major key.
test "ii subdominant" (prog $ cadence (full subdom_iii_IV auth_V))
-- error: Can't have a ii subdominant in minor mode.

-- Augmented melodic intervals are difficult to sing and are
-- therefore disallowed in choral music.
test "augmented melodic C-Fs" $ c qn :|: fs qn
-- error: Augmented melodic intervals are forbidden: C and F#.

-- Minor second harmonic intervals sound very harsh and dissonant.
test "minor second harmonic G-Gs" $ g en :-: gs en
-- error: Minor second harmonic intervals are forbidden: G and G#.
```

[3] https://ghc.haskell.org/trac/ghc/ticket/13487

```
-- Voices singing in parallel fifths do not sound sufficiently
-- independent so are disallowed in contrapuntal music.
test "parallel fifths" $ (a qn :|: g wn) :-: (d qn :|: c wn)
-- error: Parallel fifths are forbidden: A and D, then G and C.
```

4.2 Examples

Unit tests are useful for formal evaluation, but the best way to find bugs and deficiencies in a music library is to try it out on real compositions. The project proposal required that the library should be able to encode the following piano compositions: Bach's *Prelude in C Major, BWV 846*, Beethoven's *Für Elise* and Chopin's *Prelude, Op. 28, No. 20*. These have all been successfully implemented, and their full source code can be found in Appendix B. This section discusses Chopin's and Bach's pieces, as well as simple examples of contrapuntal and homophonic composition. I chose these examples to showcase the different compositional techniques that Mezzo supports (see Fig. 4.1).

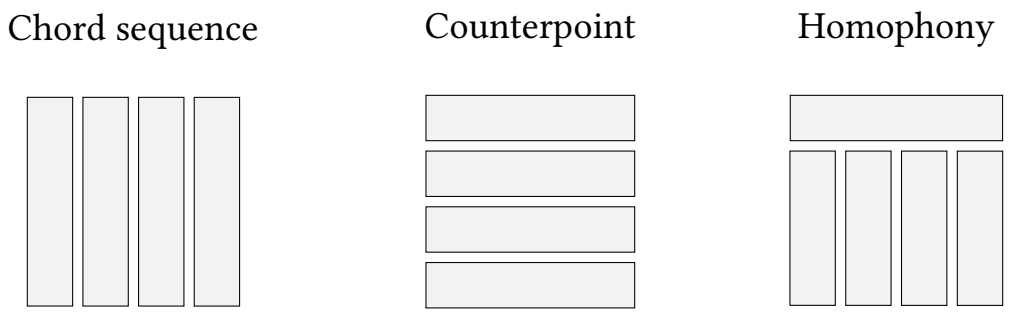
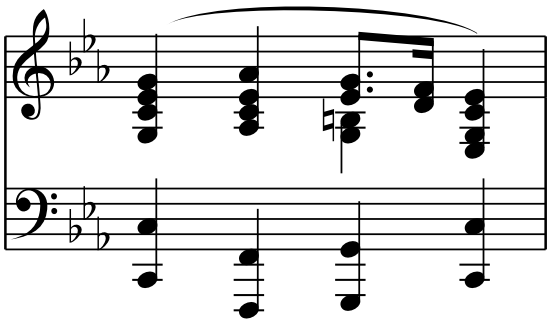


Figure 4.1 – Common musical structures that Mezzo supports. The blocks represent coherent musical elements, e.g. chords (vertical) or voices (horizontal).

4.2.1 Chopin's *Prelude*

Chopin's *Prelude* was chosen to test Mezzo's chord composition capabilities. As discussed in Section 3.2.2, Mezzo uses flat builders to describe chords: e.g. `c sharp majD inv qc` is a doubled C sharp major quarter chord in first inversion. The whole piece consists of a sequence of chords: doubled triads and seventh chords in the right hand, and octaves in the left. The piece could be transcribed in almost its entirety – however, I occasionally had to leave out a few notes as they would create forbidden intervals which Mezzo pointed out. This demonstrates that the library works, as well as the fact that it is perfectly common for composers to break the rules for artistic effect. Nevertheless, this does not invalidate the purpose of the library, since it aims to catch unintentional mistakes, not deliberate ones. If the rule-checking ever gets in the way, it can be made less restrictive or turned off completely (see Section 4.3). The first measure of the composition can be seen on Fig. 4.2.



```
rh1 = c_ minD' i2 qc
  :|: af_ majD qc
  :|: (ef maj3 ec' :|: d min3 sc
      :-: g_ maj3 qc)
  :|: c_ minD inv qc
lh1 = c__ oct qc :|: f_3 oct qc
  :|: g_3 oct qc :|: c__ oct qc
```

Figure 4.2 – First measure of Chopin's *Prelude*.

4.2.2 Bach's *Prelude*

Bach's *Prelude* is a great example of the advantages of making Mezzo an embedded DSL: we can make use of standard Haskell functionality in our compositions. For example, we can abstract out certain properties of musical pieces using functions. The *Prelude* consists of 32 rhythmically identical bars followed by a short conclusion. The repeated bars only differ in the 5 pitches they contain, while their order and rhythm is always the same. Instead of typing all measures out one-by-one, we can abstract out the pitches from the structure of the bar – that is, create a function from the five pitches to the entire measure:

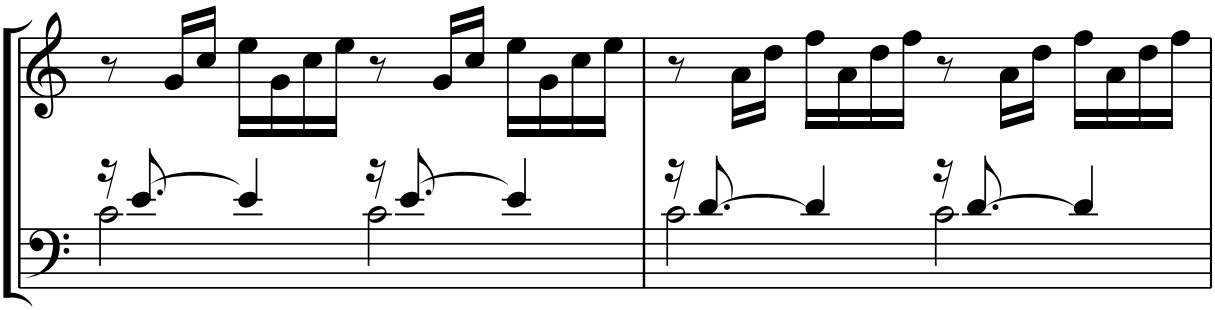
```
-- One bar in the top voice
v1b p1 p2 p3 = r er :|: notes :|: notes :|: r er :|: notes :|: notes
    where notes = p1 sn :|: p2 sn :|: p3 sn

-- One bar in the middle voice
v2b p = r sr :|: p qn' :|: r er :|: p qn' :|: r sr

-- One bar in the bottom voice
v3b p = p hn :|: p hn

-- Construct single bar given five pitches
bar p1 p2 p3 p4 p5 = v1b p3 p4 p5 :-: v2b p2 :-: v3b p1
```

Amazingly, this works without any type signatures: Haskell's type inference is able to deduce the most general type for these functions, despite the fact that the types and constraints involved are very complex.

With this "bar-generating" function, the entire piece can be described in very little code. The first two bars are demonstrated in Fig. 4.3.

4.2.3 Contrapuntal composition

Counterpoint is a form of polyphonic writing, characterised by strict rules of voice leading and harmonic motion [11]. Pieces are usually written for four singing voices which have to be kept independent throughout the composition. On its strictest setting, Mezzo is able to register all of



```
bar c e g c' e' :|: bar c d a d' f'
```

Figure 4.3 – First two measures of Bach's *Prelude*.

the common mistakes, such as concealed fifths or parallel octaves. As an example, I transcribed a piece from an online chorale composition guide[4] into Mezzo (see score on Fig. 4.4). In its first part, the guide deliberately composes an incorrect piece, and then points out the voice leading In the second part, the guide corrects the mistakes and presents a new version – however, Mezzo is still able to detect an error which the author of the tutorial missed.

```
v1 = melody :| d :| g :| fs :< g :| a :^ bf :| a
            :| a :| a :| d' :| c' :| bf :| a :> g

v2 = melody :| d :| ef :| d :| d :| d :| d
            :| cs :| d :| d :| ef :| d :| d :> bf_

v3 = melody :| bf_ :| g_ :| a_ :< g_ :| fs_ :^ g_ :| a_
            :| a_ :| fs_ :| g_ :| g_ :| g_ :| fs_ :> g_

v4 = melody :| g__ :| c_ :| c_ :< bf__ :| a__ :^ g__ :| f__
            :| a__ :| d__ :| bf__ :| c_ :| d_ :| d_  :> g__
--                    ^ The guide used a 'd_' which would have caused
--                      a concealed octave with the second voice.

comp = play v1 :-: play v2 :-: play v3 :-: play v4
```

4.2.4 Homophonic composition

Homophony is a compositional texture where a principle musical line is sung in the top voice while the lower voices provide an accompaniment – the lower voices sing in *chords*, rather than *lines*. This texture is very common in non-choral music: in popular music, a singer with musical accompaniment is essentially a form of homophony. I used the *Happy Birthday* song to test out Mezzo's melody and chord progression input (see Fig. 4.5) – while the chord progression

[4] http://decipheringmusictheory.com/?page_id=46

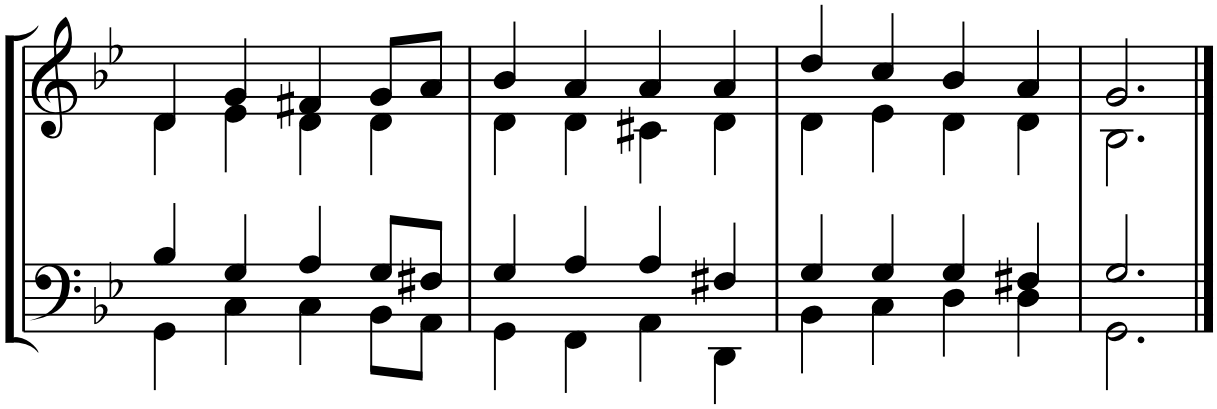


Figure 4.4 – Example contrapuntal composition.

generation is still fairly rudimentary, the EDSL provides an easy way to input progression schemas which are statically guaranteed to sound good.

```
mel = melody :~| r :~| r :<. g :<< g :^ a :| g :| c' :> b
             :<. g :<< g :^ a :| g :| d' :> c'
             :<. g :<< g :^ g' :| e' :| c' :| b :| a
             :<. f' :<< f' :^ e' :| c' :| d' :>. c'

chords =
      ph_IVI (ton_T_T ton ton) (dom_D_D dom_V dom_V) (ton_T_T ton ton)
   :+ cadence (full subdom_ii auth_V)

comp = play mel :-: prog triple (inKey c_maj chords)
```

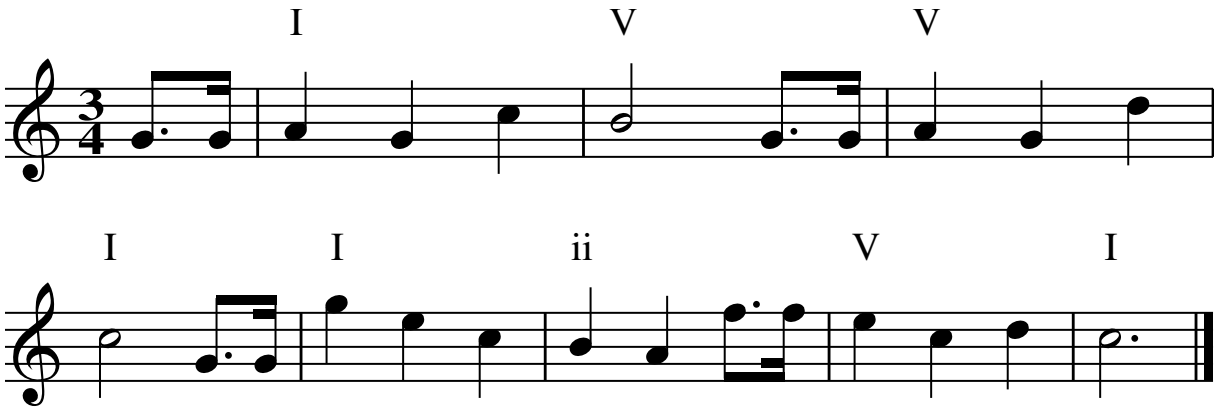


Figure 4.5 – The *Happy Birthday* song with a chord progression accompaniment.

4.3 Further developments

By using Mezzo to encode real compositions as shown above, it became clear that small additions to the library which improve performance or usability could be made. I decided to include them for completeness. We submitted a paper based on this project to Haskell'17 (Appendix C) before I made these additions, which now also address some of the reviewers' comments.

Rule sets The music checking rules were initially hardcoded into the language, making the library quite inflexible. Moreover, I realised that not all types of music have to follow all of the rules described (for example, rules of harmonic motion do not apply to homophonic music). For this reason, Mezzo implements several *rule sets* for different levels of rule strictness, including one which does not enforce any rules, allowing for complete flexibility.

Scores It is quite common for compositions to have global attributes such as tempo, key or time signature. Different sections of a piece might have a different set of these attributes, e.g. in the case of a key or tempo change, but this cannot be handled purely with composition of `Music` values. Mezzo therefore includes a `Score` type which encapsulates a piece of music and its attributes, and lets users compose different scores using *sections* (see below). The attributes affect both the `Music` pieces, e.g. the key and time signature of progressions, and the metadata of the MIDI files produced, e.g. the tempo. We can also set non-musical attributes such as the title of the piece and the rule set used to check the correctness of the piece. The syntax for creating scores uses flat builders:

```
sco = score setTitle "Composition"
            setTimeSig c_maj
            setKeySig quadruple
            setTempo 120
            setRuleSet free
            withMusic (c qn :-: b qn)
```

Although the above composition includes a dissonant major seventh, the file still compiles, as we turn off rule-checking by switching to the `free` rule set. Note that we changed the type checking behaviour through a value-level definition.

Sections Type-checking can be a time-consuming process, especially if the musical piece is long. Mezzo lets us break up large compositions into *sections* and "pre-render" these sections into simple `MidiTracks` with no type information to keep track of. This process is quite natural, as most compositions have a clear modular structure: for example, *Für Elise* (see Appendix B) can be broken down into a repeating theme and several independent episodes. Low-level harmonic and melodic rules do not apply across these sections, so it is unnecessary to keep the whole piece in the types at all times. Instead, we convert all sections into lists of MIDI events, which are then arranged into the correct order and rendered into the MIDI file. This significantly simplifies composition of large pieces, as we only keep as much information in the types as we need for a particular section.

Performance An issue mentioned in the beginning of this section is the performance of the library: type-checking compositions takes a relatively long time. The bottleneck is the `MakeInterval` function, which is called for almost every pair of pitches. As it is a recursive function, it was not difficult to increase the number of base cases, so the recursion can terminate early. This simple modification resulted in significant increase in compilation speed, especially for compositions with several voices or chords (see Fig. 4.6).

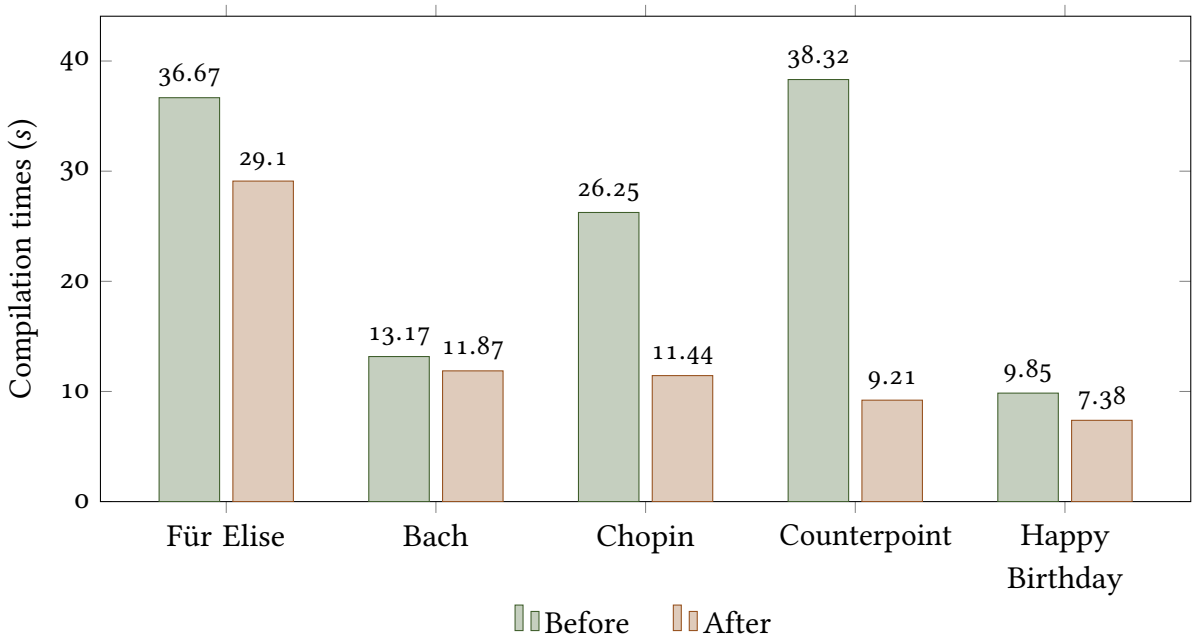


Figure 4.6 – Comparison of compilation times before and after the optimisation of `MakeInterval`. The time averages were computed from five sets of measurements taken in identical conditions.

This chapter presented several evaluation strategies used to test the correctness and expressiveness of the Mezzo composition library. The last chapter presents a summary of the results and concludes the dissertation.

CHAPTER 5

Conclusion

The project was a success. Its aim was the creation of a music composition library and EDSL which can enforce various musical rules statically. The three main components – the type-level model, the music description language and the exporting module – have been built, and the library was evaluated via unit tests and example compositions. The project also served as a case study in dependently typed programming in Haskell, examining various approaches of type-level computation, EDSL design, reification and testing. This chapter concludes the dissertation and proposes ideas for further extensions to the library.

5.1 Results and accomplishments

All of the minimum success criteria given in the project proposal have been met. Moreover, nearly all of the extensions have been implemented, including the non-MUST criteria of the project specification. It was understood that automatic generation of music would be a difficult task, as we would need to generate random types, and I am uncertain whether that is possible. LilyPond output – while desirable – was deemed to be of low importance for the purposes of this project.

5.1.1 Type-level model

- Every musical piece contains its static representation, a *pitch matrix*, in the type (MM1).
- The `Music` type provides constructors for notes, rests, chords, progressions, and sequential and parallel composition (MS2).
- The correctness of every construct is enforced statically (MM2), either via GADTs or type class constraints, producing custom type errors (MC1).
- Mezzo provides three levels of strictness: no enforcement, classical rules, and strict rules (MC2).
- Compilation times are entirely acceptable (MS1).

5.1.2 Music description language

- The Mezzo EDSL is fully embedded into Haskell (LM1), requiring few dependencies, and a single module to import into the source file (LS4).
- Users do not need to interact with the types (LM2), and ideally the type-level music model is hidden from view (LS2).
- The library provides easy means of note, rest, chord and composition input (LS1, LS3), as well as shorthands for melodies and chord progressions (LC1).

5.1.3 Exporting

- Users can export Mezzo compositions into MIDI files with custom attributes such as tempo, title or key signature (EM1, ES1).

5.2 Future work

While the core of the project is complete, there are further ideas or improvements which can be implemented in the future. Listed below are a few of the limitations and extensions which could be addressed.

- Mezzo progressions are functions of the key, while melodies are not – it should not be difficult to add a similar abstraction of composing melodies via *scale degrees* instead of absolute pitches.
- There is currently no way to change the *dynamics* of pieces: every note has the same volume. This could be solved by adding dynamics attributes to scores.
- Progressions are rhythmically very rigid: there is no way to modify the number of notes played or create arpeggios.
- In many cases, the complex type-level model is a disadvantage, especially in real error messages: these are usually very cryptic and give little idea of what could be going wrong. Leaking of internals through error messages is a common problem in EDSL design which has been addressed to some extent by the Helium compiler [14] and recent research by Serrano and Hage [42] – however, no Haskell implementation of their approach exists yet.
- There are other interesting models of music composition, such as the T-calculus [24] and tiled temporal media [22], chord spaces [38], and music calculi [35]. These could be integrated into Mezzo.

Overall, the project achieved its goals, and I think it became an interesting case study in both music formalisation and dependently-typed programming in Haskell.

Bibliography

[1] Samuel Aaron and Alan F. Blackwell. From Sonic Pi to Overtone: Creative musical experiences with domain-specific and functional languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, pages 35–46. ACM, 2013.

[2] David Aspinall and Martin Hofmann. Dependent types. *Advanced Topics in Types and Programming Languages*, pages 46–86, 2005.

[3] Joshua Bloch. *Effective Java.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[4] Max Bolingbroke. Constraint Kinds for GHC. `http://blog.omega-prime.co.uk/?p=127`, 2011. [Online; accessed 31/03/2017].

[5] W. Bas de Haas, José Pedro Magalhães, Frans Wiering, and Remco C. Veltkamp. Automatic functional harmonic analysis. *Computer Music Journal*, 37(4):37–53, 2013.

[6] Roger T. Dean. *The Oxford Handbook of Computer Music.* OUP USA, 2009.

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

[8] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 671–683, New York, NY, USA, 2014. ACM.

[9] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130, New York, NY, USA, 2012. ACM.

[10] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. Visible type application. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, pages 229–254, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[11] Johann Joseph Fux and Alfred Mann. *The study of counterpoint from Johann Joseph Fux's Gradus ad Parnassum.* WW Norton & Company, 1965.

[12] George Giorgidze. HCodecs. `https://hackage.haskell.org/package/HCodecs`, 2014. [Online; accessed 15/10/2016].

[13] Cordelia V. Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[14] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 62–71, New York, NY, USA, 2003. ACM.

[15] Michael Hewitt. *Music Theory for Computer Musicians*. Nelson Education, 2008.

[16] Lejaren A. Hiller and Leonard M. Isaacson. *Experimental Music: Composition with an electronic computer*. Greenwood Publishing Group Inc., 1959.

[17] Cheng Zhi Anna Huang and Elaine Chew. Palestrina Pal: a grammar checker for music compositions in the style of Palestrina. In *Proceedings of the 5th Conference on Understanding and Creating Music*. Citeseer, 2005.

[18] Paul Hudak. Haskore music tutorial. In *Second International School on Advanced Functional Programming*, pages 38–68. Springer Verlag, LNCS 1129, August 1996.

[19] Paul Hudak. An algebraic theory of polymorphic temporal media. In *International Symposium on Practical Aspects of Declarative Languages*, pages 1–15. Springer, 2004.

[20] Paul Hudak. Haskore. `https://hackage.haskell.org/package/haskore`, 2016. [Online; accessed 15/10/2016].

[21] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[22] Paul Hudak and David Janin. Tiled polymorphic temporal media. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, pages 49–60. ACM, 2014.

[23] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – An algebra of music. *Journal of Functional Programming*, 6(03):465–484, Nov 2008.

[24] David Janin, Florent Berthaut, Myriam Desainte-Catherine, Yann Orlarey, and Sylvain Salvati. The T-calculus: Towards a structured programming of (musical) time and space. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, pages 23–34, New York, NY, USA, 2013. ACM.

[25] Simon Peyton Jones, Stephanie Weirich, Richard A Eisenberg, and Dimitrios Vytiniotis. A reflection on types. In *A List of Successes That Can Change the World*, pages 292–317. Springer, 2016.

[26] Hendrik Vincent Koops, José Pedro Magalhães, and W. Bas de Haas. A functional approach to automatic melody harmonisation. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, pages 47–58, New York, NY, USA, 2013. ACM.

[27] Fred Lerdahl and Ray Jackendoff. *A generative theory of tonal music.* The MIT Press, Cambridge. MA, 1983.

[28] José Pedro Magalhães and W. Bas de Haas. Functional modelling of musical harmony: An experience report. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 156–162, New York, NY, USA, 2011. ACM.

[29] José Pedro Magalhães and Hendrik Vincent Koops. Functional generation of harmony and melody. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, FARM '14, pages 11–21, New York, NY, USA, 2014. ACM.

[30] Simon Marlow et al. Haskell 2010 Language Report.

[31] Josh H. McDermott, Alan F. Schultz, Eduardo A. Undurraga, and Ricardo A. Godoy. Indifference to dissonance in native Amazonians reveals cultural variation in music perception. *Nature*, 535(7613):547–550, 2016.

[32] James McKinna. Why dependent types matter. *ACM SIGPLAN Notices*, 41(1):1–1, 2006.

[33] Gerhard Nierhaus. *Algorithmic Composition: Paradigms of Automated Music Generation.* Springer Verlag Wien, Jan 2009.

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

[35] Yann Orlarey, Dominique Fober, Stéphane Letz, and Mark Bilton. Lambda calculus and music calculi. In *Proceedings of the International Computer Music Conference*, pages 243–243. International Computer Music Accociacion, 1994.

[36] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM.

[37] Walter Piston. *Harmony. Revised and expanded by Mark DeVoto.* New York: WW Norton, 1978.

[38] Donya Quick and Paul Hudak. Computing with chord spaces. In *International Computer Music Conference*, September 2012.

[39] Donya Quick and Paul Hudak. Grammar-based automated music composition in Haskell. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, pages 59–70, New York, NY, USA, 2013. ACM.

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

[41] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 51–62, New York, NY, USA, 2008. ACM.

[42] Alejandro Serrano and Jurriaan Hage. Type error diagnosis for embedded DSLs by two-stage specialized type rules. In *European Symposium on Programming Languages and Systems*, pages 672–698. Springer, 2016.

[43] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM.

[44] Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 341–352, New York, NY, USA, 2012. ACM.

[45] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. *SIGPLAN Notices*, 48(9):275–286, September 2013.

[46] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.

APPENDIX A

Music theory

This Appendix provides a more in-depth overview of musical concepts discussed in this dissertation.

Musical primitives

The basic primitives of music are *notes* and *rests*: they are combined into larger melodic and harmonic units to create compositions.

Notes

Notes are specified by a *pitch* and a *duration*: the pitch describes how high or low a note is (e.g. the key pressed on the piano), while the duration is the length of time the note is played for (e.g. how long the piano key is pressed).

- A pitch is given by its *pitch class*, *accidental* and *octave number*. The musical range is divided into *octaves* numbered from -1 to 8, and the pitch class – one of C, D, E, F, G, A or B – is the relative position of the note in the octave (white keys on a keyboard). The accidentals alter the pitch: a *sharp* raises it by a semitone (key to the right), a *flat* lowers it by a semitone (key to the left), a *natural* does not change it (and is the default accidental). Some notes can be expressed in two ways, these are called *enharmonic equivalents*: for example, a B flat is the same as an A sharp.
- Durations are expressed as negative powers of two: whole, half, quarter, eighth, sixteenth, thirty-second, etc. That is, a whole note lasts as long as two successive half notes, and a half note is as long as four eighths. A duration of a note can be extended by its half with a dot: a dotted quarter note is as long as three eighths.

Rests

Rests are periods of silence and are only specified by a duration, expressed in the same way as note durations.

Intervals

A very important concept in music is that of an *interval*, which characterises the number of semitones between two pitches. We can talk about *harmonic intervals* between notes played at the same time, and *melodic intervals* between notes played one after the other. An interval is specified by its *class* (major, minor, perfect, augmented, diminished) and *size* (unison, second, third, …, seventh, octave). Unisons, fourths, fifths and octaves are the *perfect* intervals and can be *diminished* or *augmented* by shrinking or expanding them by a semitone respectively. All other intervals can be *major* (sounding bright) and *minor* (sounding serious) and can also be diminished and augmented.

Chords

A *chord* is two or more notes played at the same time. A *triad* consists of three notes, usually separated by two thirds. Depending on the class of the thirds (major or minor), we have major, minor, augmented and diminished triads, each with its own character: for example, a major triad sounds stable and positive, while a diminished triad sounds tense and suspicious. A triad is often *doubled*: the lowest note is repeated an octave higher, which gives a 4 note chord with the same type as the underlying triad. *Seventh chords* consist of 4 notes separated by thirds; the most important type is the *dominant* (or major/minor) seventh chord, which creates a large amount of harmonic tension. Chords can be *inverted* by raising the lower pitches by an octave.

Scales

An octave spans 12 pitches separated by semitones, and a *scale* is any subsequence of those pitches. All 12 pitches make up the *chromatic scale*, while *diatonic scale* is a 7-pitch subset of that, with one, possibly altered, pitch for each pitch class. Scales are given by their *keynote*, i.e. first pitch, and *mode*, which is usually major or minor. *Tonal music* pieces are mostly built from the pitches of a specific scale, with occasional accidentals: for example, a piece in the key of C major would mostly consist of pitches from the C major scale (and would be played on the white keys of the piano keyboard, as they represent the C major scale). The index of a pitch in a scale is the *scale degree*, given in Roman numerals. For example, the degree I of a C major scale is a C, while the degree VI of a D minor scale is a B flat.

Progressions

Chord progressions are sequences of chords following a pattern. Most patterns are based on the notion of *functional harmony*, where the chords built on the degrees of a scale have a specific function (role). The chord built on the first degree (the *tonic*) represents stability and completion. The chord of the fifth degree (the *dominant*) creates tension and is usually resolved by a tonic. The fourth degree chord (the *subdominant*) builds moderate tension and can be resolved into the tonic or followed by a dominant. A chord progression would therefore follow the functional roles of the chords: a common example is I–IV–V–I. The last 2-3 chords of a progression are often called a *cadence* and provide the closure for a piece.

Counterpoint

Counterpoint is a way of composing polyphonic (multi-part) pieces, often for several singing voices. The most important consideration is that the melodic lines have to be independent, but give a coherent whole when played together. There are several different types (*species*) of counterpoint of increasing complexity, each having to adhere to strict rules of voice-leading and harmonic motion.

APPENDIX B

Mezzo compositions

To fulfil the project requirements, the Mezzo library has to be robust enough to describe real piano compositions. Due to their differences in complexity, structure and musical style, I chose the following pieces:

1. Johann Sebastian Bach: *Prelude in C Major, BWV 846*
2. Frédéric Chopin: *Prelude, Op. 28, No. 20*
3. Ludwig van Beethoven: *Für Elise*

Below is the full Mezzo source code for all three compositions. Note that since these pieces are not strictly composed contrapuntal chorales, there is bound to be some use of dissonance and disjunct voice-leading for musical effect – in these cases, either a note was removed or the type-checking was bypassed to allow the piece to compile. All compositions type-check an can be rendered to MIDI files.

Johann Sebastian Bach: *Prelude in C Major*

This prelude has a simple, repetitive structure and little rhythmic variation. In computer science, repetition is usually handled by abstraction and parameterisation – and that is exactly what is used in the description of this piece. Making use of the fact that Mezzo is embedded into Haskell, we can define a function which creates one bar of the prelude, given the five pitches that occur in it. This way, the bulk of the composition can be described by a series of function calls. Note that – fortunately – we do not need to supply any type signatures: Haskell's type inference finds the most general type for us.

```
import Mezzo
-------------------------------------------------------------------------
-- Bar repetitions
-------------------------------------------------------------------------

-- One bar in the top voice
v1b p1 p2 p3 = r er :|: notes :|: notes :|: r er :|: notes :|: notes
    where notes = p1 sn :|: p2 sn :|: p3 sn
```

```
-- One bar in the middle voice
v2b p = r sr :|: p qn' :|: r er :|: p qn' :|: r sr

-- One bar in the bottom voice
v3b p = p hn :|: p hn

-- Render out a single bar given five pitches
bar p1 p2 p3 p4 p5 = section "bar" $
                        score setTempo 100
                            setKeySig c_maj
                            setRuleSet free
                            withMusic music
    where music = v1b p3 p4 p5 `hom` v2b p2 `hom` v3b p1

bars =
 [ bar c   e   g  c' e',  bar c    d   a  d' f'
 , bar b_  d   g  d' f',  bar c    e   g  c' e', bar c    e   a   e'  a'
 , bar c   d   fs a  d',  bar b_   d   g  d' g', bar b_   c   e   g   c'
 , bar a_  c   e  g  c',  bar d_   a_  d  fs c', bar b_   d   g   d'  f'
 , bar g_  bf_ e  g  cs', bar f_   a_  d  a  d', bar f_   af_ d   f   b
 , bar e_  a_  c  g  c',  bar e_   f_  a_ c  f,  bar d_   f_  a_  c   f
 , bar g__ d_  g_ b_ f,   bar c_   e_  g_ c  e,  bar c_   g_  bf_ c   e
 , bar f__ f_  a_ c  e,   bar fs__ c_  a_ c  ef, bar af__ f_  b_  c   d
 , bar g__ f_  g_ b_ d,   bar g__  e_  g_ c  e,  bar g__  d_  g_  c   f
 , bar g__ d_  g_ b_ f,   bar g__  ef_ a_ c  fs, bar g__  e_  g_  c   g
 , bar g__ d_  g_ c  f,   bar g__  d_  g_ b_ f,  bar c__  c_  g_  bf_ e
 ]

-------------------------------------------------------------------------
-- Conclusion and end chord
-------------------------------------------------------------------------

concV1 = play $ melody :~< r
            :<< f_ :| a_ :| c :| f :| c :| a_ :| c
            :| a_ :| f_ :| a_ :| f_ :| d_ :| f_ :| d_ :~< r
            :<< g :| b :| d' :| f' :| d' :| b :| d'
            :| b :| g :| b :| d :| f :| e :| d

concV2 = r sr :|: c_ wn :|: b_ hn' :|: r er'

concV3 = c__ wn :|: c__ wn
```

```
conc = section "Conclusion" $
            score setTempo 100
                  setKeySig c_maj
                  withMusic (concV1 `hom` concV2 `hom` concV3)

endChord = section "End chord" $
            score setTempo 100
                  setKeySig c_maj
                  withMusic (c maj inv wc `hom` c__ oct wc)

main = renderSections "rendered/Bach.mid"
        "Johann Sebastian Bach - Prelude in C Major" $
        bars ++ [conc, endChord]
```

Frédéric Chopin: *Prelude, Op. 28, No. 20*

While Bach's Prelude almost entirely consists of a repeated melodic phrase, Chopin's Prelude is a sequence of complex chords and little melodic movement. Mezzo's symbolic chord input can handle all but the most non-standard chords that appear in the composition, making the structure of the piece very clear.

```
import Mezzo
-------------------------------------------------------------------------
-- First part
-------------------------------------------------------------------------

part1Rh1 = c_ minD inv inv qc :|: af_ majD qc
  :|: ((ef maj3 ec' :|: d min3 sc) :-: g_ maj3 qc) :|: c_ minD inv qc
part1Lh1 = c__ oct qc :|: f_3 oct qc :|: g_3 oct qc :|: c__ oct qc

part1Rh2 = af__ majD' i2 qc :|: cs_ majD inv qc
  :|:((c min3 ec' :|: bf_ min3 sc):-: ef_ maj3 qc):|: af__ majD inv qc
part1Lh2 = af_3 oct qc :|: df_3 oct qc :|: ef_3 oct qc :|: af_3 oct qc

part1Rh3 = b__ dimD inv qc :|: c_ majD inv qc
  :|: pad2 ((g en' :|: f sn) :-: c qn) :|: pad (c_ min inv inv qc)
part1Lh3 = g_3 oct qc :|: c_3 oct qc :|: f_3 oct qc :|: c__ oct qc

part1Rh4 = d_ dom7 inv qc :|: g_ majD qc
  :|: pad (((b en' :|: a sn) :-: fs qn :-: d qn) :|: g_ maj inv qc)
part1Lh4 = d__ oct qc :|: g_3 oct qc :|: d_3 oct qc :|: g_3 oct qc

part1Rh = part1Rh1 :|: part1Rh2 :|: part1Rh3 :|: part1Rh4
part1Lh = part1Lh1 :|: part1Lh2 :|: part1Lh3 :|: part1Lh4
```

```
part1 = section "First part" $
            score setKeySig c_min
                  setTempo 30
                  withMusic (hom part1Rh part1Lh)

-------------------------------------------------------------------------
-- Second part
-------------------------------------------------------------------------

part2Rh1 = ef maj3D qc :|: ef fourthD qc
  :|: (d' qn :-: (af en' :|: gf sn) :-: d qn) :|: d fourthD qc
part2Lh1 = c__ oct qc :|: c_ oct qc :|: b__ oct qc :|: bf__ oct qc

part2Rh2 = c fifthD qc :|: d maj3D qc
  :|: g_ maj' i2 ec' :|: pad (a_ min3 inv sc) :|: g_ maj inv qc
part2Lh2 = a__ oct qc :|: af__ oct qc :|: g__ oct qc :|: f__ oct qc

part2Rh3 = c fifthD qc :|: af_ maj3D qc
  :|: ((g en' :|: f sn) :-: g_ fifth qc) :|: c_ min' i2 qc
part2Lh3 = ef__ oct qc :|: f__ oct qc :|: b_3 oct qc :|: c__ oct qc

part2Rh4 = af__ majD' i2 qc :|: df_ majD inv qc
  :|:((ef en' :|: d sn):-: g_ maj3 qc :-: f_ qn):|: pad (c_ min inv qc)
part2Lh4 = af_3 oct qc :|: df_3 oct qc :|: g_3 oct qc :|: c_3 oct qc

part2Rh = pad (part2Rh1 :|: part2Rh2 :|: part2Rh3) :|: part2Rh4
part2Lh = part2Lh1 :|: part2Lh2 :|: part2Lh3 :|: part2Lh4

part2 = section "Second part" $
            score setKeySig c_min
                  setTempo 30
                  withMusic (hom part2Rh part2Lh)

-------------------------------------------------------------------------
-- End chord
-------------------------------------------------------------------------

end = section "End chord" $
        score setKeySig c_min
              setTempo 30
              withMusic (hom (c minD wc) (c_ fifth wc))
```

```
main = renderSections "rendered/Chopin.mid"
        "Frederic Chopin - Prelude in C Minor"
        [part1, part2, part2, end]
```

Ludwig van Beethoven: *Für Elise*

A structurally diverse piano piece with thematic episodes and variations, complex melody, rhythm and harmony, showcasing most of the composition features implemented in Mezzo. It is also a good example of *modular composition*: the independent episodes are compiled and pre-rendered into simple MIDI tracks, which are then "glued together" to form a large-scale composition. This way, there is no need to keep the entire piece in a single type, as the notes appearing in one episode have no effect on a repetition of the theme. This simple optimisation makes a big difference: without it, the compiler runs out of stack space after composing only the first two sections; with it, the entire composition compiles in less than 40 seconds.

```
import Mezzo
-------------------------------------------------------------------------
-- Fur Elise theme
-------------------------------------------------------------------------

-- Refrain

refrCoreRh = play $ melody :<< e' :| ds' :| e' :| ds' :| e'
                           :| b :| d' :| c' :^ a :~< r :< b :~<< r

refrCoreLh = play $ melody :~> r
            :<< a__ :| e_ :| a_ :| c :| e :| a
            :<< e__ :| e_ :| gs_ :~<. r

refrRh = refrCoreRh :|: e sn :|: gs sn :|: c' en :|: r sr :|: e sr
     :|: refrCoreRh :|: e sn :|: c' sn :|: b sn

refrLh = refrCoreLh :|: a__ sn :|: e_ sn :|: a_ sn :|: r sr
     :|: refrCoreLh

refrBh = hom refrRh refrLh
     :|: hom (a qn) (a__ sn :|: e_ sn :|: a_ sn :|: r sr)
     :|: hom refrRh refrLh

refrain = section "refrain" $
            score setTempo 90
                setKeySig a_min
                withMusic refrBh
```

```
-- Variation
varRh = play $ melody
    :< a :~<< r
    :<< b :| c' :| d' :<. e'
    :<< g :| f' :| e' :<. d'
    :<< f :| e' :| d' :<. c'
    :<< e :| d' :| c' :< b
    :<< e :| e :| e' :| e :| e' :| e' :| e''
    :<< ds' :| e' :| ds' :| e' :| ds' :| e' :| ds'

varLh = play $ melody
    :<< a__ :| e_ :| a_ :~<. r
    :<< c_ :| g_ :| c :~<. r
    :<< g__ :| g_ :| b_ :~<. r
    :<< a__ :| e_ :| a_ :~<. r
    :<< e__ :| e_ :~>. r :~< r

variation = section "variation" $
                score setTempo 90
                    setKeySig a_min
                    setRuleSet free
                    withMusic (hom varRh varLh)

theme = refrain ++ variation ++ refrain


-------------------------------------------------------------------------
-- First episode
-------------------------------------------------------------------------

ep1Rh1 = pad2 (a en :|: r sr)
     :|: pad (c maj3 inv sc :|: c fourth inv sc) :|: c maj inv sc

ep1Rh2 = play $ melody :| c' :<<. f' :<<< e' :< e' :| d'
                       :<<. bf' :<<< a' :<< a' :| g'

ep1Rh3 = play $ melody :<< f' :| e' :| d' :| c'

ep1Rh4 = play $ melody :< bf :| a :<<< a :| g :| a :| bf :^ c'
                       :<< d' :| ds' :<. e' :<< e' :| f' :| a :^ c'
                       :<<. d' :<<< b
```

```
ep1Lh1 = pad2 (a__ sn :|: e_ sn :|: a_ sn)
     :|: pad (c sn :-: bf_ sn :|: c sn :-: a_ sn)
     :|: (c sn :-: bf_ sn :-: g_ sn)

ep1Lh2 = play $ melody
            :<< f_ :| a_ :| c :| a_ :| c :| a_
            :| f_ :| bf_ :| d :| bf_ :| d :| bf_ :| f_ :| e

ep1Lh3 = bf_ sn :-: g_ sn :-: f_ sn :|: pad2 (e sn)
     :|: bf_ sn :-: g_ sn :-: f_ sn :|: pad2 (e sn)

ep1Lh4 = pad (play $ melody :<< f_ :| a_ :| c :| a_ :| c :| a_
                            :<< f_ :| a_ :| c :| a_ :| c :| a_
                            :<< e_ :| a_ :| c :| a_)
     :|: d_ oct sc
     :|: pad (play $ melody :<< f_ :| g_ :| e :| g_ :| e :| g_ :| f)

ep1part = score setTempo 70
                setKeySig c_maj
                withMusic

ep1p1 = section "1st episode, part 1" $
            ep1part (hom ep1Rh1 ep1Lh1)

ep1p2 = section "1st episode, part 2" $
            ep1part (hom ep1Rh2 ep1Lh2)

ep1p3 = section "1st episode, part 3" $
            ep1part (hom ep1Rh3 ep1Lh3)

ep1p4 = section "1st episode, part 4" $
            ep1part (hom ep1Rh4 ep1Lh4)

episode1 = ep1p1 ++ ep1p2 ++ ep1p3 ++ ep1p4

-------------------------------------------------------------------------
-- Second episode
-------------------------------------------------------------------------
ep2Rh1 = play $ melody :<<< c' :| g' :| g :| g' :| a :| g'
                       :| b :| g' :| c' :| g' :| d' :| g'
                       :| e' :| g' :| c'' :| b' :| a' :| g'
                       :| f' :| e' :| d' :| g' :| f' :| d'
```

```
ep2Lh1 = pad (c maj3 ec :|: pad (r sr)
     :|:      g sn :-: f sn
     :|:      e min3 sc)
     :|:      g sn :-: f sn :-: d sn
     :|:      c maj ec
     :|: pad (f_ maj3 ec
     :|:      g_ maj3 ec)

ep2Rh2 = play $ melody :<<< e' :| f' :| e :| ds' :| e' :| b
                       :| e' :| ds' :| e' :| b :| e' :| ds'
                       :<. e' :<< b :| e' :| ds' :<. e' :<< b
                       :| e' :| ds' :| e' :| ds' :| e' :| ds'

ep2Lh2 = gs_ min3 ec :|: pad (r qr :|: r wr)

ep2part = score setTempo 70
                setKeySig c_maj
                withMusic

ep2p1 = section "2nd episode, part 1" $
            ep2part (hom ep2Rh1 ep2Lh1)

ep2p2 = section "2nd episode, part 2" $
            ep2part (hom ep2Rh2 ep2Lh2)

episode2 = ep2p1 ++ ep2p1 ++ ep2p2


-------------------------------------------------------------------------
-- Third episode
-------------------------------------------------------------------------

-- Repeat a composition 6 times.
repeat6 n = n :|: n :|: n :|: n :|: n :|: n

ep3Rh1 = pad3 (a en :|: r qr)
     :|:       cs dim7 inv qc' :|: pad (d min inv qc)
     :|: pad2 (cs' min3 sc :|: d' min3 sc)
     :|: pad  (d dim' i2 qc :|: d dim' i2 ec :|: a min qc')

ep3Lh1 = rep :|: rep :|: rep :|: rep :|: rep
    where rep = repeat6 (a__ sn)
```

```
ep3Rh2 = pad2 (d min3 inv qc
     :|:       c maj3 inv sc  :|: b_ min3 inv sc)
     :|: pad  (fs_ dim' i2 qc)
     :|: pad2 (a_ min3 inv ec :|: a_ min3 inv ec
     :|:       c maj3 inv ec :|: b_ min3 inv ec
     :|:       a_ min3 inv qc')

ep3Lh2 = repeat6 (d__ fifth sc)
     :|: repeat6 (ds__ sn :-: a__ sn)
     :|: e__ fourth sc :|: e__ fourth sc :|: e__ fourth sc
     :|: e__ fourth sc :|: e__ maj3 sc :|: e__ maj3 sc
     :|: a_3 oct sc :|: a__ sn :|: a__ sn
     :|: a__ sn :|: a__ sn :|: a__ sn)

ep3Rh3 =       cs dim7 inv qc' :|: pad  (d min inv qc)
     :|: pad2 (cs' min3 sc :|: d' min3 sc
     :|:       d' min3 qc :|: d' min3 ec :|: d' min3 qc'
     :|:       ef maj3 inv qc
     :|:       d min3 inv sc :|: c min3 inv sc)
     :|: pad  (bf_ maj inv qc
     :|:       d min ec :|: d dim qc
     :|:       d dim ec :|: a_ min inv qc)
     :|: pad3 (r er)
     :|: pad  (b_ fourth inv ec :-: gs_ en)
     :|: pad3 (r qr)

ep3Lh3 = aNotes :|: aNotes :|: aNotes
     :|: bfNotes :|: bfNotes :|: bfNotes :|: bNotes
     :|: c_ qn :|: r er :|: e_ en :|: r qr
    where aNotes = repeat6 (a__ sn)
          bfNotes = repeat6 (bf__ sn)
          bNotes = repeat6 (b__ sn)

ep3part = score setTempo 90
                setKeySig d_min
                withMusic

ep3p1 = section "3rd episode, part 1" $
           ep3part (hom ep3Rh1 ep3Lh1)

ep3p2 = section "3rd episode, part 2" $
           ep3part (hom ep3Rh2 ep3Lh2)
```

```
ep3p3 = section "3rd episode, part 3" $
            ep3part (hom ep3Rh3 ep3Lh3)

episode3 = ep3p1 ++ ep3p2 ++ ep3p3


-------------------------------------------------------------------------
-- Fourth episode
-------------------------------------------------------------------------

ep4Rh1 = tripletE a_ c e :|: tripletE a c' e'
     :|: tripletE d' c' b :|: tripletE a c' e'
     :|: tripletE a' c'' e'' :|: tripletE d'' c'' b'
     :|: tripletE a' c'' e'' :|: tripletE a'' c'3 e'3
     :|: tripletE d'3 c'3 b'' :|: tripletE bf'' a'' gs''

ep4Lh1 = pad2 (a_3 en)
    :|: pad2 (r er) :|: a_ min ec :|: a_ min ec
    :|: pad2 (r er) :|: a_ min ec :|: a_ min ec
    :|: pad2 (r er) :|: a_ min ec :|: a_ min ec

ep4Rh2 = tripletE g'' fs'' f'' :|: tripletE e'' ds'' d''
     :|: tripletE cs'' c'' b' :|: tripletE bf' a' gs'
     :|: tripletE g' fs' f'

ep4p1 = section "4th episode, part 1" $
            score setTempo 90
                  setKeySig a_min
                  setRuleSet free
                  withMusic (hom ep4Rh1 ep4Lh1)

ep4p2 = section "4th episode, part 2" $
            score setTempo 90
                  setKeySig a_min
                  withMusic ep4Rh2

episode4 = ep4p1 ++ ep4p2
```

```
--------------------------------------------------------------------------
-- End chord
--------------------------------------------------------------------------

endChord = section "End chord" $
            score setTempo 90
                  setKeySig a_min
                  withMusic (hom (a qn) (a_3 oct qc))

main = renderSections "rendered/FurElise.mid"
        "Ludwig van Beethoven - Fur Elise"
        [ theme
        , episode1, episode2
        , theme
        , episode3, episode4
        , theme
        , endChord
        ]
```

APPENDIX C

Haskell Symposium paper

An Experience Report based on this project has been submitted to the early track of the ACM SIGPLAN Haskell Symposium 2017, part of the 22nd ACM SIGPLAN International Conference on Functional Programming. A board of anonymous reviewers provided feedback on the paper, some of which has been incorporated into the library (see Section 4.3). The reviews and the paper are included in their entirety – note that the latter was written based on an unfinished implementation of Mezzo and does not cover some later additions included in this dissertation.

Review 1

This experience report describes a novel application of dependently-typed Haskell programming, the representation of musical scores that are statically checked for correctness using the type system. It showcases recent developments in GHC Haskell, in particular the TypeInType language extension introduced in GHC 8.0. A key theme is the use of dependent types and ConstraintKinds in EDSL design.

The paper is generally well-written. It introduces concepts clearly and without requiring substantial background in either music theory or dependent types. The subject matter is interesting, and the authors have managed to pack a lot of detail into a six-page experience report.

My most significant criticism of the paper is that it does not sufficiently justify why it takes the approach of representing music and encoding musical rules at the type level. The use of dependent types leads to significant complexity and, as the authors observe in section 7.2, extremely long compile times. While the existence of case studies such as the present paper is valuable (and may help motivate improvements in GHC's support for dependently-typed programming), it seem to me that the particular problem at hand might be better solved by a "normal" term-level program.

It would surely be easier to take a simple algebraic representation of music (as presented in Section 3.1) and write a functional program to check its correctness. Lifting such a program to the type level is then an independent problem, potentially even solvable automatically (e.g. by the "singletons" library). Thus, I would like to see more of an argument in the paper about the benefit dependent types bring here.

On the one hand, checking correctness in the type system makes it possible to write functions that are guaranteed to construct correct outputs for all inputs. Are there such

examples, and how easy are they to write using Mezzo?

On the other hand, this style of dependently-typed programming means that functions consuming music do not have to deal with incorrect input. The paper gives an example of a function consuming music, `musicToMidi`, but this does not seem to rely on the music having any structure beyond that of the algebraic data type. Are there examples of functions that can make use of the additional type structure?

Overall, I would like to see this paper accepted, but perhaps re-submission to the regular track would give the authors the opportunity to further draw out the key ideas and argue more clearly the case for dependent types in this domain. Correspondingly, I have ranked the paper as "weak accept" at this stage.

More specific comments: The paper could be a little more self-contained. There are various types referenced without explicit definition (e.g. `Elem`, which presumably has constructor `:*`), in some cases without even being mentioned in the text (e.g. `Quarter`, `PitchS`). I appreciate that space constraints make this difficult, of course.

p2, c2, l27. What is the `q` in `NoteConstraints p q`? Should it be `d`? Relatedly, I can understand `MelConstraints` and `HarmConstraints` as imposing the composition requirements, but what is the purpose of `NoteConstraints` and `RestConstraints`?

p3, c1, l34. Having only a limited background in music theory, I wasn't previously familiar with the word partiture. A brief definition in the text might help.

p3, c1, l36. Can you unpack this paragraph a bit more? Polymorphic recursion in data types, and the corresponding need for CUSKs, may be a bit obscure.

p3, c1, l56. The `MakeInterval` type family seems to be neither defined nor referenced anywhere else in the paper, so this sentence is unnecessary.

p3, c2, l35. The definition of `AllPairsSatisfy` ignores the durations of the notes. This seems surprising, as it appears that `ValidHarmDyadsInVectors` will apply `ValidHarmDyad` to notes that occur at different times (but at the same index in the vectors). Or is there some constraint elsewhere that the melodies being harmonically composed have the same rhythm?

p4, section 4.2. The `flat builders` idea is nicely presented, and the choice of terminology and types definitely help make it comprehensible. However, this section feels a bit out of step with the rest of the paper, as it is suddenly about CPS rather than dependent types. Nice as the idea is, perhaps the space might be better used with more of a focus on the key message of the paper.

p5, section 5.2. It is nice that the extra type information does not get in the way of producing MIDI output, but not terribly surprising. This section could perhaps be condensed into an example of the use of `prim`.

p5, c2, l34. What is `Chord`? It doesn't seem to be defined anywhere.

p5, section 6. Modulo the usual problems of space, it would be nice to see a bit more discussion of the relationship between Mezzo and previous work such as HarmTrace and Haskore.

p6, c1, l5. Is there really hardly any other work on checking the correctness of compositions programmatically? I'm not familiar with the literature, but this seems surprising. Or is the claim limited to systems that enforce rules using a type system or similar static analysis?

p6, section 7.1. The authors' perspective on the current state of dependently-typed programming in GHC Haskell is valuable, and perhaps this section could be expanded. What would Mezzo look like in a dependently-typed language (such as Idris or Agda)? Which things would be easier or harder? Do the benefits of dependent types in Haskell justify the additional complexity introduced?

p6, c1, l38. I'd appreciate a better explanation of the problem identified by reference to GHC Trac 12564.

Review 2

The paper presents Mezzo, a Haskell EDSL for describing "correct" polyphonic compositions in Haskell. Compositions are encoded in Haskell's type system, and as a result the EDSL uses all the latest type-system extensions implemented in GHC.

I find quite a few of the design decisions presented in this paper to be highly questionable.

The motivation of the paper seems reasonable – certainly it would be nice to have a tool that composers may use to spot potential problems with their piece. The choice to encode these checks in the type system of the language, however, leads to a number of undesirable consequences:

1. "incorrect" music cannot be expressed at all, no matter what the composer intended;
2. "correctness" is rigid: it is difficult to modify or extend the rules;
3. "correctness" is limited: only rather simple properties of the composition can be validated.

I think that (1) is undesirable, as the notion of "correct" in the setting of music is in many ways up to the composer. So it seems much more desirable to have a system that points out potential mistakes, rather than rejecting them outright. Similarly, in the context of education, it is desirable to be able to express "incorrect" programs, so that students can experiment, and see for themselves what is "wrong" with the composition, rather than rejecting the composition with an error message.

About the rigidity (2): it seems quite difficult to add new rules to the system, or change the existing rules. This is in part because the rules are encoded using quite a lot of sophisticated type-level machinery. The other problem, however, is that they are baked into the language. In contrast, if the validation function was just a simple Haskell function, that given a composition returned some potential problems, it would be easy to extend the system to support new "correctness" checks, or even check multiple "correctness" properties on the same composition. In contrast, with the Mezzo approach, one would have to change the whole language.

About (3): the checks that Mezzo performs are fairly simple; it is not at all obvious that more complex checks can be implemented at all, certainly not easily.

All of these issues would be completely circumvented if the implementation took a more conventional approach, and emphasized programming in Haskell, rather than programming in Haskell's type system.

In terms of the structure of the paper – the explanations are fairly clear. However, most of the paper is spent on the details of the encodings used Mezzo's implementation, and few

alternative design consideration are mentioned. The experience of designing the language is mentioned only briefly in Section 7. There is no description of the experience using Mezzo at all, and some of the statistics reported at the end of the paper suggest that it might be quite unusable, as even short pieces take a very long time to compile. The assertion that the type-level work somehow speeds up the generation of MIDI seems misleading – MIDI is a simple format, and generating it would be very quick on pretty much any modern computer.

Review 3

This paper shows how to implement functions that check that a musical composition follows the rules of classical music composition. The cool thing is that it is done at the type level, and the result is thus an EDSL for correct musical compositions. Compositions that violate the rules cause type errors, which are expressed in readable domain specific terminology, thanks to recently added functionality in GHC for producing custom type error messages.

So I guess the paper thus serves as a showcase for what can be achieved with GHC's fancy type system extensions. But since type level computations are rather inefficient in GHC, this approach does not seem to give you a practically useful system. Even one of the small examples in the paper (a composition containing only 16 notes) takes close to 3 seconds to type check (on my laptop), and the authors mention that a larger example takes 35 seconds to type check.

So I am wondering how this system would compare (in code size, efficiency and maintainability, etc) to a solution using the first simple algebraic type from section 3.1 for music and a composition rule checking function implemented as a normal, value-level function, perhaps encapsulated as an abstract data type to enforce the composition rules as an invariant? In the typical use case, would it be a big loss to get run-time errors instead of compile-time type errors when the composition rules are violated?

I am naturally also wondering what it would be like to do this in a language with proper support dependent types, such as Agda or Idris. Would you still have the same efficiency problems? (Probably not, I am guessing…)

Also, this is not the first Haskell Symposium Experience Report that makes use of GHC's type system extensions to create a type safe EDSL: The 2016 paper "Experience Report: Types for a Relational Algebra Library" by Lennart Augustsson and Mårten Ågren seems somewhat similar, one difference being that the library presented in that paper was practically useful…

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 recently introduced type-system features can be used to perform such low-level 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 is 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**; *Domain specific 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

Millennia ago, Pythagoras famously investigated the mathematical connection between the lengths of strings in a string instrument and the pitches of the corresponding notes. These experiments were the beginnings of the field of *Western music theory*, a formal description of what sounds good to the ear and what does not. The principles of music theory have acted as the foundation of music composition for centuries and, alongside a variety of other methods, have often been applied to the problem of *algorithmic music generation* [9].

We present *Mezzo*[1], an embedded domain-specific language in Haskell for describing music. We take inspiration from formal systems of music theory, which can be used to analyse and compose music based on a description of classical music rules. However, unlike previous work in the area, we do not use such a model to generate new music, but instead check the *correctness* of user-made compositions with respect to the rules – an essential task which composers have been doing by hand for centuries.

[1] https://hackage.haskell.org/package/mezzo

HASKELL'17, Oxford, UK

DOI: 10.475/123_4

Mezzo makes extensive use of recent additions to Haskell's type system and, in particular, the new `TypeInType` extension [15] which removes the distinction between types and kinds. At the core of Mezzo is a type-level model of music, allowing compositions to be represented entirely on the type-level. This allows for the rules of music theory to be expressed as type families and type classes which are then later enforced as type constraints. Compositions which do not follow these rules do not satisfy the corresponding constraints and lead to errors at compile-time.

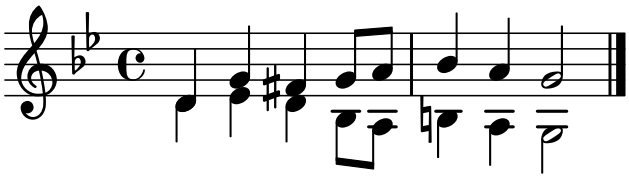
Users can interact with the library through an embedded domain-specific language, which provides constructs for composing music using notes, melodies and chords. These basic constructs are in effect the singleton values of the corresponding notes in the type-level model, which are then combined into larger compositions using a range of combinators. Where possible, composition errors lead to custom type errors which explain the musical rule that was violated. Finally, if a composition type-checks, it can be exported to a MIDI file.

In this experience report, we discuss the challenges we faced in implementing Mezzo in Haskell, related to type-level computation, dependently typed EDSL design, and reification. We believe that 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 type-level computation without being a fully dependently-typed language yet.

2 Example

This section provides an overview of some musical concepts and the rules that the library enforces through a simple example. A complete introduction to music theory is beyond the scope of this paper, but may be found in any standard music textbook – we suggest Walter Piston's *Harmony* [12].

As an introductory example, we will attempt to compose a simple contrapuntal melody for two voices[2].



Counterpoint is a polyphonic (multi-part) compositional technique. Its most important consideration is that the melodic lines have to be independent, but give a coherent whole when played together. To ensure this, composers have to follow strict rules of voice-leading and harmonic motion which were developed centuries ago and followed since then [3].

The above composition can be described in the Mezzo EDSL as shown below by specifying the voices as sequences of notes, such as `d qn` which represents a D quarter note, with `:|:` acting as the sequential composition operator. The different voices are then composed in parallel using `:-:`.

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

```
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 = v1 :-: v2
```

However, this composition has a few mistakes, and indeed, Mezzo catches them and indicates this with type errors at compile-time:

```
error:
    • Can't have major sevenths in chords.
    • In the expression: v1 :-: v2
error:
    • Direct motion into a perfect octave is forbidden.
    • In the expression: v1 :-: v2
```

Bar 2 starts with a major seventh chord, which sounds very dissonant and is generally forbidden – Mezzo therefore lets us know that a harmonic rule was broken. The second issue is more complex and stems from the fact that counterpoint requires the voices to be *independent*, in the sense that all voices should be discernible at all times. 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 (or at least to be avoided). Mezzo sees the parallel octaves in the last two notes and notifies us in the form of a compiler error.

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 Mezzo code compiles without errors – from now on, `comp` is seen as a valid composition and can be used in larger pieces or exported into a MIDI file.

3 Music model

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

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. [6]. 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` could be expressed as:

```
M ::= Note  |  Rest  |  M :|: M  |  M :-: M
```

This lets us separate the *description* and *performance* of music and establish algebraic laws which help formal reasoning about compositions.

3.1 The `Music` datatype

A straightforward translation of the above BNF description into Haskell is as follows:

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

This describes a tree-like structure with the leaves containing a note (with some pitch and duration) or a rest (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 any `Music` value will sound good, as there is nothing to constrain their structure.

To make our library statically "aware" of composition rules, we need to have access to the musical information at compile-time – this can be achieved by adding a type argument to our type, containing some type-level representation of the music. 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 alrady supports many of the desired features through various language extensions [1]. In this case, we can use the `GADTs` extension to enable the definition of *generalised algebraic datatypes* [11]: this way, we can be specific about what the type variable should contain for each possible constructor. More complex computation is enabled by the `TypeFamilies` extension, which we use to convert type-level information about pitches and durations (passed as type arguments) into our music representation, as well as to combine these representations. Finally, we encode musical rules as *type class constraints* on the type variables: this way, whenever we construct a new term of type `Music m`, it must follow the implemented composition rules. Our final `Music` type looks like this:

```
data Music m where
    Note  :: NoteConstraints p q
          => 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.

We've seen how the main Mezzo data type is defined: the EDSL manipulates `Music` values which can then be rendered into MIDI files. But how is music actually represented at the type level? We explore this in the next section.

3.2 The pitch matrix

To make rule description and enforcement as simple as possible, we want to represent music at the type-level in a consistent, structured way. We decided on a straightforward, somewhat brute-force approach: keeping the music in a two-dimensional array of pitches. The columns of the matrix represent durations and the rows are individual voices. The matrix elements are pairs of pitches and durations (which specify a note).

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 on the type level now. However, thanks to data-type promotion [16] and `TypeInType`, this is not an issue: in GHC 8, we can promote any data-type, even GADTs, without doing any extra work. All we

need is to define the `Vector` data-type (using `GHC.TypeLits` to get efficient naturals and arithmetic at the type level):

```
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 our matrix (the individual voices), but in those rows we need to store both pitches and durations. We do this by defining a new `Elem` type that holds the value and the duration, and use it to build up an *optimised vector*. Note that the length of this vector is not the number of elements, but their total duration, so a whole note and 8 eights have the same length. We can also declare a type synonym for matrices.

```
data OptVector :: Type -> Nat -> Type where
    End  :: OptVector t 0
    (:-) :: Elem t l -> OptVector t (n - l)
         -> OptVector t n
type Matrix t p q = Vector (OptVector t q) p
```

Thanks to GADT promotion, all these types are now available at the kind level, and we can declare 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.

Finally, we need to describe musical values at the type level – this is a straightforward application of datatype promotion. All types which the user can later interact with need term-level values: we accomplish this by creating kind-constrained proxies. For convenience, we also define specialised types for pitch vectors and matrices.

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

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

We can now explicitly specify the type variable for `Music m`. An important thing to note here is that we have polymorphic recursion in the application of `Music`, so GHC requires us to provide a *complete user-supplied kind signature* (CUSK) by quantifying all the kind variables in the type definition.

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

Lastly, we show how musical rules can be expressed as type class constraints.

3.3 Musical constraints

In Mezzo, a piece of music is either correct or incorrect. That is, correctness is just a *predicate* on `Music` values, and this can be implemented using Haskell's *type classes*, acting as compile-time type predicates. This view of type classes is made even more powerful by the `ConstraintKinds` extension, which lets us treat constraints as "first-class" types of kind `Constraint`.

The rules implemented in the library mainly apply at the composition of `Music` values and constrain the *musical intervals*, which are described by their size (unison, second, third, etc.) and type (perfect, minor, major, augmented, diminished). We use the `MakeInterval` type family to find the interval between two pitches: its definition relies heavily on type-level pattern matching and recursion, but is otherwise unchallenging.

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: its instances are only the intervals which are allowed in harmony. However, we can do better, using GHC's custom type error feature (in `GHC.TypeLits`): we make the type error the "precondition" to an instance of an invalid interval, so whenever GHC tries to determine whether a major seventh is a valid harmonic interval, it encounters the type error:

```
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 ValidHarmInterval i -- General case
```

Note that we need to account for overlapping instances – for purposes of space we omit the `OVERLAPPING` and `OVERLAPPABLE` pragmas in this paper, but they have to be added to make instance resolution deterministic. In general, we omit some uninteresting corner cases too for the sake of brevity.

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, we know that two voices can be harmonically composed if all of the pitch pairs form valid harmonic intervals.

When working with constraints, a useful abstraction is made possible by the `ConstraintKinds` extension. Constraints (even unsaturated ones) can be passed around as types, which opens the door to many flexible options for validation: for example, checking if a vector of types satisfies a constraint, or a type satisfies all the constraints in a vector. In our case, we apply a binary constraint to two optimised vectors:

```
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`, which checks if two pitches form a valid interval, is a two-parameter type class, so it has kind `a -> b -> Constraint` – exactly what `AllPairsSatisfy` needs.

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

We use similar techniques to implement the other musical rules. The details do get a bit complicated, but ultimately it is just a matter of translating logical rules into type class constraints. The extension lets us be very flexible in expressing rules, since we can even declare type families that return a `Constraint`. For example, constrains for harmonic motion are computed based on the adjacent pitch pairs.

4 Music description language

Among the most important considerations for DSL design are expressiveness and conciseness. Haskell's minimalistic syntax makes

it a great choice for implementing EDSLs and provides a lot of flexibility with its built-in constructs and abstractions. In this section we discuss how Mezzo's music description language is implemented and how the difficulties arising from a heavily typed model can be overcome. We also describe a general pattern for implementing fluent EDSLs inspired by continuation-passing style.

4.1 Literal values

As explained in the previous section, the type model provides proxies that the user can interact with – these are just constant values with a phantom type variable. This means that creating literal term-level values for proxies is trivial (and very boring):

```
_c :: PC (PitchClass C) ; _c = PC
```

In fact, this is so boring that code for most of the DSL literals are generated at compile-time using Template Haskell [14]. The approach really pays off when creating literal values for pitches for each pitch class, accidental and octave: the 14-line function generates 210 literal declarations.

Having literal values and smart constructors already makes it possible to write simple music:

```
noteP _cn ei :|: noteP _dn ei :|: noteP _en qu
```

The above syntax is still a bit verbose, and most MDLs provide shorthand notation for common composition tasks. Since Mezzo handles both note-level and chord-level input, we decided on an approach that makes notation concise and flexible. This pattern, which we call *flat builders*, can be applied to the general task of EDSL design and works well in Mezzo.

4.2 Flat builders

Flat builders are inspired by the various attempts to express variable argument and postfix functions in Haskell, such as *continuation-passing style* and the Okasaki's *flat combinators* [10]. 'Flat', as the expressions do not contain parentheses, and 'builders', as they build a value through a series of constructions and transformations. The ideas here are not new, but we feel that having suitable types and a consistent terminology makes CPS programming simpler.

Variable argument functions are not explicitly supported by Haskell, but can be simulated by functions which take their *continuations* as an argument. Instead of using the `Cont` monad, we use simple type synonyms to encapsulate the "building blocks":

- *Specifiers*: specify an initial value of type `t` to start the "building process".

  ```
  type Spec t = forall m. (t -> m) -> m
  ```

- *Converters*: convert a value of type `s` to a value of type `t`.

  ```
  type Conv s t = s -> Spec t
  ```

- *Terminators*: finish building a value of type `t` and return the result of type `r`.

  ```
  type Term t r = t -> r
  ```

As a simple example, consider an expression which takes a string, converts it into a character by taking the first element and prints out the ASCII code for the element:

```
string :: String -> Spec String
firstChar :: Conv String Char
printAscii :: Term Char Int
string "Hello" firstChar printAscii --> 72
```

Unlike simple function composition, builders can be read (and written) from left to right with no syntactic interference, which makes them a good choice for EDSL development:

```
add 5 to 7 and display the result --> "result: 12"
```

In Mezzo, builders are used to construct note values: specifiers for pitches, converters for accidentals, terminators for durations. For example, a C natural quarter note can be simply written as `c qn`, while a double-sharp F with a dotted half note is `f sharp sharp hn'`. The main advantage of using builders is that the specifiers don't have to know the final return type of the builder, which makes them very reusable. We can use the same pitch specifiers for notes and chords: `c sharp qn` is a C sharp quarter note, `c sharp maj inv qc` is a C sharp major chord in first inversion. Though the terminator syntax is different, this only affects the 6 duration literals, and not the 210 pitch literals. As before, builder components are generated by Template Haskell macros.

4.3 Melodies

Flat builders are a big step up from literals and constructors, but still not the most convenient way to input long sequences of notes. Since writing melodies is likely to be the most common composition activity, we implemented a melody input method inspired by LilyPond, a TeX-like music typesetting system. The top voice of the example in Section 2 was written as:

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

We have to specify the duration of every note, even though changes of duration in melodies are not very common. 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 constructor to describe the melody above:

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

Notes are only given as pitches, the duration is either implicit (`(:|)` means "next note has the same duration as the previous note") or explicit in the constructor (e.g., `(:<)` means "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, that is, lists whose head is at the end. The difference is that the `Melody` type keeps additional information in its type variables (like a vector), and instead of 2 constructors it has 25:

```
data Melody :: Partiture 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 partiture and sets the default duration to a quarter. `(:|)` takes the melody constructed so far (the tail) and a pitch specifier, and returns a new melody with the added pitch and unchanged duration. The other constructors do the same thing, except they change the duration of the last note. While memorising the constructors might

take some time, they allow for very quick and intuitive melody input.

5 Music rendering

Mezzo can export all well-typed compositions to MIDI files. Adding other audio formats in the future should not be difficult and would follow the same principles outlined in this section. The principal question we address in this section is the one of how to reify compositions which exist entirely on the type-level so that we can use them generate corresponding values on the term-level. As you might recall, users of Mezzo mainly interact with proxies, which contain no term-level information. This is deliberate since we do not wish to duplicate information which might be prone to errors. To solve this, we once again make use of type classes.

5.1 The `Primitive` class

Our aim is to find a *primitive representation* for all of the musical types that the user can interact with. 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 types, naturals, pitches, etc. Its main 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.

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, it's not too bad: we just 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, pitches (and other compound types) are 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 [2]. 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`.

5.2 MIDI export

MIDI is a simple, compact standard for music communication, often used for streaming events from electronic instruments. The format describes music as a sequence of *MIDI messages* for various musical events, for example, the beginning or the end of a note, tempo change, etc. It is a popular standard in computer music since it abstracts away musical information from instrument sound, so converting written scores into MIDI files is straightforward.

We use a MIDI codec package for Haskell called *HCodecs*[3] by George Giorgidze, which provides lightweight MIDI export and import capabilities. We only needed to add a type for MIDI notes (with their number, start time and duration) and functions to convert notes 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:

```
musicToMidi (Note pitch dur) =
            playNote (prim pitch) (prim dur * 60)
musicToMidi (Rest dur) =
            playRest (prim dur * 60)
musicToMidi (m1 :|: m2) =
            musicToMidi m1 ++ musicToMidi m2
musicToMidi (m1 :-: m2) =
            musicToMidi m1 >< musicToMidi 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 `merge` operation, which interweaves the two lists of messages respecting their timestamps. We think that one of the main benefits of the Haskore system is that the algebraic description maps so elegantly to common list operations, even with a type-heavy implementation. The Mezzo-specific constructions do not require a lot of work either:

```
musicToMidi (Chord c d) = foldr1 (><) notes
  where notes = map (`playNote` prim d * 60) (prim c)
```

We first create a list of MIDI notes from the primitive representation of the chord, then fold the resulting list of tracks with the merge operator.

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. In the future, we plan to add ways to configure the attributes and metadata of this file, but the existing implementation already produces valid, playable MIDI files.

6 Related work

One of the first experiments in algorithmic composition, the *Illiac Suite* in 1957 [4], consisted of four pieces of increasing musical complexity: the first two used the rules of counterpoint to generate one- and four-part melodies, while the second two used stochastic processes and Markov chains to compose more experimental-sounding music. Joint research into linguistics, music theory and cognition resulted in Lerdahl and Jackendoff's influential *generative theory of tonal music* [7], which relates the structure of compositions with musical understanding through well-formedness and preference rules – a model that inspired several implementations and derived theories. Martin Rohrmeier developed a formal grammar of functional harmony [13] which was then implemented as a Haskell

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

library, *HarmTrace* [8], for music analysis and composition. This work describes harmonic constructs like chords and progressions at the type level and has been one of the initial inspirations for Mezzo.

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* [5], a Java program for grammar-checking music written in the contrapuntal style of Palestrina. We are not aware of similar libraries for functional languages or systems that enforce musical rules statically.

7 Conclusions

We have described the implementation of Mezzo, a library for composing music which statically enforces that compositions follow the rules of classical music. This means that composers no longer have to check that they follow the rules by hand. An EDSL provides a straightforward interface to the library to compose music and well-typed compositions can be exported to MIDI files.

7.1 Type-level computation

Haskell has long been a playground for type system experimentation, starting with type classes and functional dependencies through GADTs and type families to merging types and kinds entirely. Mezzo uses most of these features, and development has been really enjoyable and surprisingly easy: promotion, GADTs and type families work seamlessly together and there is very little mental overhead needed to think and reason about programs. Of course, we don't get the full power of functional programming at the type level (type-families are not "first-class types"), but conditionals, datatypes and recursion go a really long way.

During development, we have encountered a few limitations and nuisances – 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 by performing natural arithmetic or pattern-matching, and the solution wasn't always obvious[4]. Another cause for unexpected errors was non-covering type families: type families are accepted even if they are stuck. 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 wouldn't always get triggered, as they are accepted arguments to type families.

While type-level programming is already quite pain-free, we thought of a few feature ideas 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. While closed type classes wouldn't make much sense (in their normal use, instances rarely overlap), a separate construct acting as a closed *type predicate* could be useful. 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., for enumerating pitch classes. They would open the doors to pretty-printing of types, simplified implementation of singletons and ways of adapting Haskell design patterns to the type level.

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

7.2 Performance

One thing we haven't touched upon yet is *performance*, mainly since it was not our main concern. It is safe to say that we cannot expect a type checker to match the performance of highly optimised machine code execution, and accordingly, compile times were quite slow. Checking harmonic motion rules is a complicated operation: a short 4-voice composition takes more than 35 seconds to type-check. This means that Mezzo is most likely too cumbersome to use for actual music education at the moment, but we feel that this does not reduce its value as an example of what is possible to do in Haskell's type-system. Of course, all this upfront work means that runtime performance is great: the composition that takes 37 seconds to compile can be converted into a MIDI file almost instantaneously.

Overall, we feel that 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. We firmly believe that in a few years Haskell will be on par with the dependently typed languages used in research and is already an excellent choice for anyone wanting to look into type-level programming.

References

[1] Richard A Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. Dissertation. University of Pennsylvania.

[2] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632.* Springer-Verlag New York, Inc., New York, NY, USA, 229–254. DOI: http://dx.doi.org/10.1007/978-3-662-49498-1_10

[3] Johann Joseph Fux. 1965. *The study of counterpoint from Johann Joseph Fux's Gradus ad Parnassum.* Number 277. WW Norton & Company.

[4] Lejaren A Hiller and Leonard M Isaacson. 1959. *Experimental Music; Composition with an electronic computer.* Greenwood Publishing Group Inc.

[5] Cheng Zhi Anna Huang and Elaine Chew. 2005. Palestrina Pal: a grammar checker for music compositions in the style of Palestrina. In *Proc. of the 5th Conf. on Understanding and Creating Music.* Citeseer.

[6] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. 2008. Haskore music notation – An algebra of music. *Journal of Functional Programming* 6, 03 (Nov 2008), 465–484.

[7] Fred Lerdahl and Ray Jackendoff. 1983. *A generative theory of tonal music.* The MIT Press, Cambridge. MA.

[8] 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. DOI: http://dx.doi.org/10.1145/2034773.2034797

[9] Gerhard Nierhaus. 2009. *Algorithmic Composition: Paradigms of Automated Music Generation.* Vol. 1. Springer Verlag Wien.

[10] Chris Okasaki. 2003. THEORETICAL PEARLS: Flattening Combinators: Surviving Without Parentheses. *J. Funct. Program.* 13, 4 (July 2003), 815–822. DOI: http://dx.doi.org/10.1017/S0956796802004483

[11] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-based Type Inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06).* ACM, New York, NY, USA, 50–61. DOI: http://dx.doi.org/10.1145/1159803.1159811

[12] Walter Piston. 1978. Harmony. (Revised and expanded by Mark DeVoto). *Londres: Victor Gollancz LTD* (1978).

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

[14] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02).* ACM, New York, NY, USA, 1–16. DOI: http://dx.doi.org/10.1145/581690.581691

[15] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. *SIGPLAN Not.* 48, 9 (Sept. 2013), 275–286. DOI: http://dx.doi.org/10.1145/2544174.2500599

[16] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12).* ACM, New York, NY, USA, 53–66. DOI: http://dx.doi.org/10.1145/2103786.2103795

APPENDIX D

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Typesafe music composition

D. Szamozvancev, Downing College
21 October 2016

Project Originator: D. Szamozvancev
Project Supervisor: M. B. Gale
Director of Studies: Dr R. Harle
Project Overseers: Prof J. Crowcroft & Dr T. Sauerwald

Introduction

Music composition was among the first applications of digital computers, with experiments going back to the late 1950s [16]. Since then, a lot of new approaches have been developed and computer-assisted music composition has almost started its own musical genre. This project aims to exploit the formal nature of the music-theoretical rules governing the classical music of the common practice period, and express them as types and type-level operations such that these rules would be enforced statically by the type checker. The end product would be a Haskell combinator library for music composition, one that could assist composers in writing music and music students in learning and applying the rules of music composition.

The language choice has fallen on Haskell due to its strong, static type system and a variety of type-level extensions. Features like GADTs, type families and (nearly) dependent types enable programmers to express complex assumptions about the problem domain and let the type checker enforce those assumptions. I intend to make use of these advanced type-level features to encode some basic rules of chord construction, voice-leading and rhythm in a type-based model. Haskell's purely functional nature lends itself well to the implementation of a combinator-based, intuitive and expressive composition library, where the typed harmonic model can be utilised.

Starting point

The field of computer-assisted music composition is very rich with research into various composition and analysis techniques [33]. I have spent some time exploring the rule-based methodologies, especially the work of Magalhães et al. [29] which the project aims to build upon and extend. I have also looked into the research of the Yale Haskell Group on music generation [39] and functional composition libraries [23]. The relevant Tripos courses are *Foundations of Programming*, *Semantics of Programming Languages* and *Types*.

I have some piano and music theory background and am fairly proficient with Haskell, albeit I will go into further depth regarding its type-level extensions. Up to this point only small, experimental code snippets have been written to test out the MIDI library for Haskell.

Resources required

For the project I will be using my own computer, a 2013 MacBook Pro with a 2 GHz Intel Core i7 CPU and 8 GB of RAM. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. If the computer should fail, I will continue work on an MCS machine. The project files will be backed up to a USB drive and cloud storage twice daily, and I will frequently push code changes to an online GitHub repository.

The main piece of software, the GHC compiler, is freely available online for download. Other than that, I will be using a text editor and a music player for MIDI playback, and possibly software to convert MIDI into audio files.

Work to be done

The main components of the project are the following:

1. A precise, extensible model of tonal harmony and melody expressed as a collection of Haskell datatypes. The model should statically enforce various rules of music theory, i.e. ensure that a term violating these rules would not typecheck.

2. A flexible combinator library for music composition which lets users create simple musical pieces that follow the common principles of composition.

3. A way of exporting the created pieces as MIDI or audio files – this would use existing codec libraries available for Haskell.

4. Evaluation of the above components by examining the correctness of the model and testing it with unit tests.

Success criteria

The project will be deemed successful if the type model can enforce a set of common rules of classical composition, and the composition library is flexible enough to create reasonably

complex piano music. As these criteria are fairly subjective, I will list the concrete requirements that need to be achieved:

1. The type system must rule out the following violations:
   (a) Major seventh and minor second intervals in chords
   (b) Consecutive fifths and octaves
   (c) Ending a piece on a scale degree chord other than I or V
   (d) Ending a piece on a chord in second inversion
   (e) Leaps by large or augmented intervals
   (f) Rhythm that is independent of the underlying metre
2. The user must be able to implement the following pieces using the library:
   (a) Johann Sebastian Bach: *Prelude in C Major, BWV 846*
   (b) Ludwig van Beethoven: *Für Elise*
   (c) Frédéric Chopin: *Prelude, Op. 28, No. 20*

Possible extensions

By its nature such an art-oriented project is very extensible, e.g. by adding new rules, composition techniques, etc. A few ideas are listed below.

- A highly desired extension would be using the harmony model to generate music automatically. This would involve building a system for stochastic music composition and could be further extended by adding new genres, scales, instruments, interaction and customisation.
- Type errors are the bane of every programmer, and they get even worse when complicated type-level computation is involved. It would be great to customise the error messages so the fault is described in music-theoretical terms.
- Counterpoint is a method of composing complex, multi-part music with its own set of rules. These could be added to the harmony model.
- The library can also be extended with many extra combinators for various composition techniques and combined with the automatic generation system to, for example, harmonise a user-supplied melody, create a cadence, arpeggiate a chord, etc.

Timetable

The schedule I plan to stick to is as follows:

1. **Michaelmas weeks 2–4** Research on music theory, algorithmic composition and type-level computation in Haskell.

2. **Michaelmas weeks 5–6** Make draft of the type model. I expect the first attempt to "evolve" as features are being added, so I decided to schedule in a period of experimentation. The final type model will be started from a clean slate, with the benefit of hind sight.

3. **Michaelmas weeks 7–8** Start implementing the type model.

4. **Michaelmas vacation** Continue work on the type model. Along the way, unit tests will be written to ensure that the model rejects terms that violate musical rules. By the end of the vacation the implementation should fulfil the first set of success criteria.

5. **Lent weeks 0–2** Write progress report and create the exporting module. The abstract Haskell terms have to be converted into a valid MIDI representation. This step will either use HCodecs [12], a codec library, or build on Haskore [20], a Haskell music creation system.

6. **Lent weeks 3–4** Begin work on the composition library. This will most likely involve a lot of back-and-forth tweaks between the library, type model and exporting module.

7. **Lent weeks 5–6** Finish work on the composition library.

8. **Lent weeks 5–6** Test the project against the success criteria. Depending on the type model, a formal proof of its correctness could be attempted as well.

9. **Lent weeks 7–8** Finish up the main project and start drafting the dissertation.

10. **Easter vacation:** Write dissertation.

11. **Easter weeks 0–2:** Proofreading, final testing and submission.