Well-typed Music Does Not Sound Wrong
(Experience Report)

Dmitrij Szamozvancev
University of Cambridge
ds709@cam.ac.uk

Michael B. Gale
University of Warwick
m.gale@warwick.ac.uk

Haskell Symposium 2017
Oxford, United Kingdom
8 September 2017
Does this piece sound good?
import Mezzo

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

main = playLive (v1 :-: v2)
import Mezzo

\[ v1 = d \ qn :|: g \ qn :|: fs \ qn :|: g \ en :|: a \ en :|: bf \ qn :|: a \ qn :|: g \ hn \]

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

main = playLive (v1 :-: v2)
import Mezzo

- Major sevenths are not permitted in harmony: Bb and B_
- Direct motion into a perfect octave is forbidden: Bb and B_, then A and A_
- Parallel octaves are forbidden: A and A_, then G and G_
Mezzo example

import Mezzo

- Major sevenths are not permitted in harmony: Bb and B_
- Direct motion into a perfect octave is forbidden: Bb and B_, then A and A_
- Parallel octaves are forbidden: A and A_, then G and G_
import Mezzo

- Major sevenths are not permitted in harmony: Bb and B_
- Direct motion into a perfect octave is forbidden: Bb and B_, then A and A_
- Parallel octaves are forbidden: A and A_, then G and G_
import Mezzo

   :|: a qn :|: a qn :|: g hn :|: g hn

main = playLive' $ score setRuleSet strict withMusic comp
import Mezzo

v1 = d qn :|: g qn :|: fs qn :|: g en


main = playLive (v1 :-: v2)
Mezzo example

```
import Mezzo

v1 = d qn :|: g qn :|: fs qn :|: g en


main = playLive (v1 :-: v2)
```
Western tonal music is governed by rules:

- What notes sound good together, or in sequence
- How voices should interact
- How a piece should be structured

Learning and following rules requires care, attention, and time
A Haskell EDSL for music composition

Maintains a static model of music
A dependently typed music algebra

Converts composition mistakes into type errors
Compiler errors describe the nature and location of mistakes
Behind the scenes
The Mezzo recipe

1. Take the Haskore music algebra
The Mezzo recipe

2. Add some dependent types
The Mezzo recipe

3. Hide everything under an EDSL
The Mezzo recipe

4. Add MIDI export functionality
The Haskore music algebra

An algebraic description of music
Primitives and two composition operators

\[ M ::= \text{NOTE} \mid \text{REST} \mid M \\ M :\cM \mid M :\cM \]

- **Primitive values**
- **Harmonic composition**
- **Melodic composition**
The Haskore music algebra

An algebraic description of music
Primitives and two composition operators

\[ M ::= \text{NOTE} \mid \text{REST} \mid M :-: M \mid M ::|: M \]

Primitive values
Harmonic composition
Melodic composition

data Music = Note Pit Dur
\mid Rest Dur
\mid Music :-: Music
\mid Music ::|: Music
The Haskore music algebra

\[
((D5 : | : C5) :-: F4) :|: (B4 :-: G4)
\]
The Haskore music algebra

\[ ((D5 :|: C5) ::: F4) :|: (B4 ::: G4) \]
The Haskore music algebra

\[
\begin{array}{c}
\text{Type level} \\
\hline
\text{Term level} \\
\end{array}
\]

\[
\begin{array}{c}
\text{D5} \\
\text{C5} \\
\text{F4} \\
\text{B4} \\
\text{G4} \\
\end{array}
\]
The Haskore music algebra
The Haskore music algebra

Intuitive for composition
Unsuitable for verification
Pitch matrix

\[
\begin{array}{ccc}
\dagger D5 & \dagger C5 & \dagger B4 \\
\dagger F4 & \dagger F4 & \dagger G4
\end{array}
\]
The Haskore music algebra

<table>
<thead>
<tr>
<th>D5</th>
<th>C5</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4</td>
<td>F4</td>
<td>G4</td>
</tr>
</tbody>
</table>

Type level

Term level

D5  C5  F4  B4  G4
Pitch matrix

Alternative music format, suitable for verification
Has a clear, rigid, non-hierarchical structure

Reflects the visual layout of the score
Obvious relationship between parallel and successive notes

<table>
<thead>
<tr>
<th>♮ D5</th>
<th>♮ C5</th>
<th>♮ B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>♮ F4</td>
<td>♮ F4</td>
<td>♮ G4</td>
</tr>
</tbody>
</table>
Our aim is to store the pitch matrix on the type level
Enables static verification of the rules

Need to enforce invariance of dimensions
A simple type-level list of lists would not suffice

<table>
<thead>
<tr>
<th>♮D5</th>
<th>♮C5</th>
<th>♮B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>♮F4</td>
<td>♮F4</td>
<td>♮G4</td>
</tr>
</tbody>
</table>
Where GHC 8 comes into play

**Problem:** keeping a size-indexed matrix on the type level requires GADT promotion
Where GHC 8 comes into play

*Problem*: keeping a size-indexed matrix on the type level requires GADT promotion

---

In 67465497/ghc:

- Add kind equalities to GHC.

This implements the ideas originally put forward in "System FC with Explicit Kind Equality" (ICFP'13).

There are several noteworthy changes with this patch:

- We now have casts in types. These change the kind of a type. See new constructor `CastTy`.

- All types and all constructors can be promoted. This includes GADT constructors. GADT pattern matches take place in type family equations. In Core, types can now be applied to coercions via the `CoercionTy` constructor.
Where GHC 8 comes into play

*Problem*: keeping a size-indexed matrix on the type level requires GADT promotion

*Solution*: upgrade to GHC 8!

With GHC 8, GADTs can be promoted just like any other type

Enabled by the TypeInType extension
Where GHC 8 comes into play

*Problem*: keeping a size-indexed matrix on the type level requires GADT promotion

*Solution*: upgrade to GHC 8!

With GHC 8, GADTs can be promoted just like any other type

Enabled by the TypeInType extension

data Vector :: Type → Nat → Type where
  None :: Vector t 0
  (∶--) :: t → Vector t (n - 1) → Vector t n
Dependent Haskore algebra

data Music where
  Note :: Pit → Dur → Music
  Rest :: Dur → Music
  (:::-) :: Music → Music → Music
  (::|::) :: Music → Music → Music
Dependent Haskore algebra

data Music :: ∀ n l. PitchMatrix n l → Type where
  Note :: Pit p → Dur d → Music (FromPitch p d)
  Rest :: Dur d → Music (FromSilence d)
  (:-:): :: Music m1 → Music m2 → Music (m1 ++ m2)
  (::|::): :: Music m1 → Music m2 → Music (m1 |+| m2)
Dependent Haskore algebra

data Music :: ∀ n l. PitchMatrix n l → Type where
  Note :: Pit p → Dur d → Music (FromPitch p d)
  Rest :: Dur d → Music (FromSilence d)
  (:-:) :: Music m1 → Music m2 → Music (m1 ++ m2)
  (|:|) :: Music m1 → Music m2 → Music (m1 +|+ m2)
Dependent Haskore algebra

data Music :: ∀ n l. PitchMatrix n l → Type where
  Note  :: Pit p → Dur d → Music (FromPitch p d)
  Rest  :: Dur d → Music (FromSilence d)
  (:-::) :: Music m1 → Music m2 → Music (m1 +-+ m2)
  (::|::) :: Music m1 → Music m2 → Music (m1 +|+ m2)

A vector of vectors of pitches
A promoted GADT
Dependent Haskore algebra

```haskell
data Music :: ∀ n l. PitchMatrix n l → Type where
  Note :: Pit p → Dur d → Music (FromPitch p d)
  Rest :: Dur d → Music (FromSilence d)
  (:::-::) :: Music m1 → Music m2 → Music (m1 +-+ m2)
  (::__:) :: Music m1 → Music m2 → Music (m1 +|+ m2)
```

Promoted musical values:

```haskell
data PitchClass = C | D | E | F |...
data PitchType = Pitch PitchClass

Kind-constrained proxies:

```haskell
data Pit (p :: PitchType) = Pit
```
Dependent Haskore algebra

data Music :: ∀ n l. PitchMatrix n l → Type where
  Note :: Pit p → Dur d → Music (FromPitch p d)
  Rest :: Dur d → Music (FromSilence d)
  (:-:) :: Music m1 → Music m2 → Music (m1 +-+ m2)
  (::|::) :: Music m1 → Music m2 → Music (m1 +|+ m2)

Type families for constructing
and combining pitch matrices

Concatenation respects the matrix dimensions
Made possible by the length-indexing in the kinds

type family (a :: PitchMatrix n k) +|+
  (b :: PitchMatrix n l)
  :: PitchMatrix n (k + l) where ...
Dependent Haskore algebra

\[ \text{FromPitch D5} \rightarrow \boxed{\text{D5}} \]
Dependent Haskore algebra

FromSilence ↦ [ ]
Dependent Haskore algebra

\[
\begin{array}{cc}
\dd D5 & \dd C5 \\
\dd F4 & \dd F4
\end{array}
: | :
\begin{array}{c}
\dd B4 \\
\dd G4
\end{array}
\rightarrow
\begin{array}{ccc}
\dd D5 & \dd C5 & \dd B4 \\
\dd F4 & \dd F4 & \dd G4
\end{array}
\]
Dependent Haskore algebra

\[ \begin{array}{ccc}
\hat{D}_5 & \hat{C}_5 & \hat{B}_4 \\
\hat{F}_4 & \hat{F}_4 & \hat{G}_4
\end{array} \]
Dependent Haskore algebra

```haskell
data Music :: ∀ n l. PitchMatrix n l → Type where
  Note  :: Pit p → Dur d → Music (FromPitch p d)
  Rest :: Dur d → Music (FromSilence d)
  (-::-) :: Music m1 → Music m2 → Music (m1 ++- m2)
  (|:|:) :: Music m1 → Music m2 → Music (m1 +||+ m2)
```

Type families for constructing and combining pitch matrices

Concatenation respects the matrix dimensions
Made possible by the length-indexing in the kinds
Musical constraints

data Music :: \forall n l. PitchMatrix n l \to Type where
Note :: Pit p \to Dur d \to Music (FromPitch p d)
Rest :: Dur d \to Music (FromSilence d)
(::--) :: Music m1 \to Music m2 \to Music (m1 ++ m2)
(:|::) :: Music m1 \to Music m2 \to Music (m1 |+ m2)

We have static access to musical values through type variables

We impose type class constraints to limit the usage of the Music constructors
Musical constraints

data Music :: ∀ n l. PitchMatrix n l → Type where
  Note :: ValidNote p d
         ⇒ Pit p → Dur d → Music (FromPitch p d)
  Rest :: ValidRest d
         ⇒ Dur d → Music (FromSilence d)
  (:-:) :: ValidHarmComp m1 m2
         ⇒ Music m1 → Music m2 → Music (m1 ++ m2)
  (:-|:) :: ValidMelComp m1 m2
         ⇒ Music m1 → Music m2 → Music (m1 +|+ m2)

We have static access to musical values through type variables

We impose type class constraints to limit the usage of the Music constructors
Musical constraints

A series of inference rules as class hierarchies

“Axioms” specify valid and invalid intervals

Domain-specific error messages with GHC’s custom compiler errors feature

class ValidMelInterval (i :: IntervalType)
instance TypeError
  (Text "Major sevenths forbidden.")
  ⇒ ValidMelInterval (Interval Maj Seventh)
instance {-# OVERLAPPABLE #-} ValidMelInterval i
Musical constraints


handle overlapping instances. In normal usage, closed type classes would not make much sense as the instances rarely overlap, but a separate construct acting as a closed type predicate could be useful for type-level programming and verification. Similarly, we often

```haskell
class ValidMelInterval (i :: IntervalType) where
instance TypeError
  (Text "Major sevenths forbidden.") => ValidMelInterval (Interval Maj Seventh)
instance {-# OVERLAPPABLE #-} ValidMelInterval i
```

Some text from the document.
Musical constraints

A series of inference rules as class hierarchies. 

"Axioms" specify valid and invalid intervals. 

Domain-specific error messages with 

GHC’s custom compiler errors feature. 

handle overlapping instances. In normal usage, closed type classes would not make much sense as the instances rarely overlap, but a separate construct acting as a closed type predicate could be useful for type-level programming and verification. Similarly, we often

Haskellers leave no feature unabused.
Musical constraints

A series of inference rules as class hierarchies

“Axioms” specify valid and invalid intervals

Domain-specific error messages with GHC’s custom compiler errors feature

```haskell
class ValidMelInterval (i :: IntervalType)
instance TypeError
    (Text "Major sevenths forbidden.")
    ⇒ ValidMelInterval (Interval Maj Seventh)
instance {-# OVERLAPPABLE #-} ValidMelInterval i
```
Musical constraints

A series of inference rules as class hierarchies

“Axioms” specify valid and invalid intervals

Domain-specific error messages with GHC’s custom compiler errors feature

Rules propagate the interval axioms to the pitch matrix verification

class ValidMelLeap (p :: PitchType)
    (q :: PitchType)
instance ValidMelInterval (MakeInterval p q)
    ⇒ ValidMelLeap p q
Musical constraints

Constraints connect the pitch matrix with the Haskore algebra.

The rules are enforced any time a `Music` constructor is used.

`ConstraintKinds` allows us to treat and manipulate constraints as types.

Flexible means of validation, such as computed or partially applied constraints.

The rule system is extensible and customisable.

Constraints are further parameterised by `rule sets`. 

52
Rule sets

Allow for customisation of rule-checking
Not all genres of music follow the same rules

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

data Classical = Classical
instance RuleSet Classical where ...

data Empty = Empty
instance RuleSet Empty where ...
```
Rule sets

Allow for customisation of rule-checking
Not all genres of music follow the same rules

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

Music values are parameterised by rule sets
Rule-checking behaviour can be modified dynamically

data Score = ∀ rs m. MkScore rs (Music rs m)
MkScore Classical (c qn :-: b qn) ✗
MkScore Empty (c qn :-: b qn) ✓
Also in the paper

Details of the pitch matrix implementation
Treatment of duration and fragmentation

Construction of intervals

Some features of the EDSL
Note, chord and melody input

Reification and MIDI rendering
Summary and conclusions

Mezzo is a music composition library and EDSL with static rule-checking of musical scores

Exploits the term-type separation to manipulate two different models of music
No singletons required!

Built on the Haskore algebra, augmented with dependent types
Makes use of GADT promotion, type families and constraint kinds
Well-typed Music
Does Not Sound Wrong
(Experience Report)

github.com/DimaSamoz/mezzo
hackage.haskell.org/package/mezzo

ds709@cam.ac.uk  |  m.gale@warwick.ac.uk
Advantages of static typing

Static, compile-time verification

Source location of mistakes

Two, distinct views of music
Haskore algebra for composition,
pitch matrix for verification

Simple term-level programming
Disadvantages of static typing

Complex type-level programming
But not much harder than doing the same thing on the term level

Slower compilation
But time is saved on finding the mistakes

Term-type separation
Can be handled with standard Haskell techniques
Musical constraints

Musical constraints are implemented as a series of “inference rules” via type classes.

GHC’s custom type error feature lets us specify which instances are invalid, and provide an explicit, domain-specific error message.

class ValidMelInterval (i :: IntervalType)
instance TypeError (Text "Major sevenths forbidden.")
  => ValidMelInterval (Interval Maj Seventh)
instance {-# OVERLAPPABLE #-} ValidMelInterval i

Class with no methods – an open type predicate.
A type is either an instance (a valid melodic interval) or not (an invalid melodic interval).
Musical constraints

Musical constraints are implemented as a series of “inference rules” via type classes.

GHC’s custom type error feature lets us specify which instances are invalid, and provide an explicit, domain-specific error message.

class ValidMelInterval (i :: IntervalType)
instance TypeError (Text "Major sevenths forbidden.") => ValidMelInterval (Interval Maj Seventh)
instance {-# OVERLAPPABLE #-} ValidMelInterval i

If \( i \) is unified with a major seventh interval, a type error is encountered (uses GHC.TypeLits).
Musical constraints

Musical constraints are implemented as a series of “inference rules” via type classes.

GHC’s custom type error feature lets us specify which instances are invalid, and provide an explicit, domain-specific error message.

class ValidMelInterval (i :: IntervalType)
instance TypeError (Text "Major sevenths forbidden.")  
          ⇒ ValidMelInterval (Interval Maj Seventh)
instance {-# OVERLAPPABLE #-} ValidMelInterval i

Otherwise, the interval is valid. We need to handle overlapping instances, as Haskell type classes are open and not checked in order.
Musical constraints

class ValidMelInterval (i :: IntervalType)
instance TypeError (Text "Major sevenths forbidden.")
  ⇒ ValidMelInterval (Interval Maj Seventh)
instance {-# OVERLAPPABLE #-} ValidMelInterval i

class ValidMelLeap (p :: PitchType) (q :: PitchType)
instance ValidMelInterval (MakeInterval p q)
  ⇒ ValidMelLeap p q

class ValidMelAppend (v :: Voice l1) (w :: Voice l2)
instance ValidMelLeap (Last v) (Head w)
  ⇒ ValidMelAppend v w

class ValidMel (p :: PitchMatrix n k) (q :: PitchMatrix n l)
instance (ValidMelAppend v w, ValidMelConcat vs ws)
  ⇒ ValidMelConcat (v :-- vs) (w :-- ws)