Introduction

Music is a Haskell library for music representation and manipulation. It is partially a development of the ideas outlined by Paul Hudak [1], which has previously been implemented by libraries such as *Haskore*, *Euterpia* and *temporal-media*. *Music* is entirely separate from these libraries, however.

The main goal of *Music* is to generalise standard music notation, allowing the user to express any kind of music, including Western, non-Western, classical, popular, traditional, instrumental, vocal and electronic music, using a common, semantically well-behaved representation. To achieve this goal, the library avoids depending on specific pitch names, tuning systems, rhythms and instructions; instead provides multiple common representations of these concepts which can be combinedinto a coherent musical language. More importantly, it provides a set of polymorphic types and functions which can be used in conjunction with the musical representation of choice. This way *Music* is able to express a proper superset of the music expressible in standard notation.

This flexibility is attained by a somewhat involved use of the Haskell type system, the details of which need not be known precisely by users of the library. Tentative users are probably better of reading this reference documentation and adapting its examples as a starting point, before trying to fully digest the type signatures of the reference documentation, however.

Design principles

Relative values

TODO Emphasis on *relative* values rather than *absolute*

TODO Unit values and binary operations are heavily used in this style.

Continous values

TODO Emphasis on *continous values*, postponing sampling to the final rendering phase

---

[1] Hudak, Paul (2003) *An Algebraic Theory of Polymorphic Temporal Media*

TODO This is similar to the approach taken in vector graphics, but dissimilar to traditional computer music systems.

Time as the central musical parameter

TODO The library is organized around the *Temporal* type class and its subclasses *Timed* and *Delayed*

TODO Other properties are treated *Temporal* transformers

Use of standard type classes

- Use of standard type classes
  - *Monoids* for composition
  - *Functors* for transforming structures without respect to time
  - *Applicatives* for transforming structures with respect to time
  - *Monads* for transforming and flattening nested structures

Terminology

Music use standard music terminology where appropriate. The meaning of terms such as *pitch*, *dynamics*, *duration*, *intonation*, *phrase* should be straightforward.

It is worth noting that standard music theory concepts may often be understood in a both a general and a more specific sense. For example, *pitch* may refer to property of having a discernible frequency, or to a set of values such as *A5* and *Eb4*. In *Music* the more general sense is used unless otherwise noted.

Time

In *Music*, the property of *temporality* (being able to be composed in time) is separated from that of *duration* (having a known extent in time) and *position* (having a known position in time). This separation allows for representation of a wider range of musical structures. Temporal values can be though of as moments in time in which some *event* occurs. The purpose of the temporal abstraction is to describe just the temporal properties, leaving other properties, such as pitch, timbre etc abstract.

Basic composition

Temporal

Temporal values are captured by the `Temporal` type class. Each implementation of temporal is a type constructor parameterized on its content, i.e. if `t` is a temporal type constructor and `a` is a concrete type, `t a` is a temporal structure of `a` values which can be composed in time.

```
class Temporal d where
    instant :: d a
    (|||)   :: d a -> d a -> d a
    (>>>)   :: d a -> d a -> d a
    (<<<)   :: d a -> d a -> d a
```

As can be seen, `Temporal` defines three binary operations for parallel, sequential and reverse sequential composition. Intuitively, reverse sequential composition is a synonym for ordinary sequential composition with its arguments reversed, so `a >>> b` (read as *a* followed by *b*) is equivalent to `b <<< a` (read as *b* preceded by *a*). Both parallel and sequential composition is associative, mening that `(a >>> b) >>> c` is equivalent to `a >>> (b >>> c)`. Parallel composition is also commutative, meaning that `a ||| b` (read as *a* with *b*) is equivalent to `b ||| a`.

`Temporal` also defines `instant`, which is the unit value for both parallel and sequential composition. It can be thought of as an infinitely brief moment in time. Both sequential and parallel composition form a monoid with instant.

Loop and reverse

Music also provides two subclasses of `Temporal`, providing operations supported by many, but not all of the temporal implementations. These are `loop`, which repeats a temporal value and reverse, which retrogrades it.

```
class Temporal d => Loop d where
    loop :: d a -> d a

class Temporal d => Reverse d where
    reverse :: d a -> d a
```

Timed values

TODO restrictions on time values:

```
class (Enum t, Ord t, Real t, Fractional t) => Time t Source
```

The main characteristic of durational values is their ability to be prolonged, shortened or scaled. In standard notation, duration is represented by a combination of divisivion into bars and beats, represented by vertical barlines and beaming, and note values, represented by note head shape and number of flags or beams.

```
class Time t => Timed t d where
    duration :: d a -> t
    stretch  :: t -> d a -> d a
```

A characteristic of positional values is their ability to be moved forward and backwards in time. Position is implicit in standard notation, but is often encountered in audio editing software. Some scorewriting software such as Sibelius of Finale allow the user to view to position of a note by selecting it.

```
class Time t => Delayed t d where
    rest   :: t -> d a
    delay  :: t -> d a -> d a
```

The meaning of Time

In the definition of `Timed` and `Delayed`, we left the representation of time itself in the abstract, save for the restrictions grouped together under the `Time` type class.

Implementations

Score

A *Score* is a a container of discrete events. It implements all the temporal type classes, and provides the constructor `note`, which lifts a single value into a temporal value. Several derived combinators for constructing scores are provided, but these can all be defined in terms of `note`.

The `note` constructor creates temporal values of duration one. To get another duration, the `stretch` function should be used.

```
auld = g
   >> stretch 3 $ c >> c >> stretch 2 (c >> e)
   >> stretch 3 $ d >> c >> stretch 2 (d >> e)
   >> stretch 3 $ c >> c >> stretch 2 (e >> f)
   >> stretch 6 $ a
```

There is also a module providing instances of the standard numeric type classes for `Score` (implemented in terms of `stretch`), which allows for a more concise syntax:

```
auld' = g
   >> 3 * c >> c >> 2 * (c >> e)
   >> 3 * d >> c >> 2 * (d >> e)
   >> 3 * c >> c >> 2 * (e >> f)
   >> 6 * a
```

Event list

For each score there is a corresponding `EventList`, which can be created by the `render` function. As `render` is overloaded, it is necessary to provide a full type signature:

```
p :: Score Double StdNote
el = render p :: EventList Double StdNote
```

Segment

Pitch

Like time, pitch is represented using relative values.

TODO linear (for hertz) and logarithmic (for octaves, equal temperament, cents etc) newtype wrapper TODO scales

Dynamics

Phrasing

Space

Rendering

Sound

MIDI

OpenSoundControl

Graphics

Standard notation

MusicXML

Abc Notation