Report on the Haskore Computer Music System

Bogdan Dumitriu José Pedro Magalhães

June 16, 2005

1 Summary

Haskore was designed to allow musicians to describe and process their music using an embedded language built on top of Haskell. *Haskore* is available as a number of modules which allow for different functionality in terms of composing music, interpreting music and going back and/or forth from the internal music format to external formats such as MIDI or C-Sound.

A major part of *Haskore* is dedicated to music representation and composition. All major musical concepts are available for use in the form of constructors for the `Music` type.

```
data Music =
    Note Pitch Dur [NoteAttribute] -- a note   \ atomic
  | Rest Dur                       -- a rest   / objects
  | Music :+: Music                -- sequential composition
  | Music :=: Music                -- parallel composition
  | Tempo (Ratio Int) Music        -- scale the tempo
  | Trans Int Music                -- transposition
  | Instr IName Music              -- instrument label
  | Player PName Music             -- player label
  | Phrase [PhraseAttribute] Music -- phrase attributes
  deriving (Show, Eq)
```

A `Note` is used to represent a musical note, with a pitch, a duration and possibly some attributes (such as the volume at which it is supposed to be played, for example). A `Rest` simply indicates silence for a certain amount of time. Sequential (`:+:`) and parallel (`:=:`) composition concatenate pieces of music so that they are played either one after the other or one at the same time with the other. `Tempo` is used to indicate faster or slower playing of a piece of music, while `Trans` is used to add an offset to all the pitches of notes in a piece of music. The last three constructors, `Instr`, `Player` and `Phrase` are used for markup of music, so that a player has additional indications as to how to play it.

A `Music` can either be written by hand (which is rather tedious), generated by some mathematical algorithms (fractal music, for example) or it can be read from a MIDI file. Either way, such a piece of music is actually an abstract, high-level representation of the musical parts that are fixed. There are parameters, though, that can be changed when transforming this piece to a concrete, absolute representation. There are mainly two ways of doing this: by using the `Player` concept and by using the `Performance` concept.

A `Player` represents a nice abstraction of a real player, which of course will always give a personal interpretation to a piece of music, for example by making the rests longer or shorter than specified. *Haskore* uses markup to indicate that a piece of music should be played by a certain player. However, this is merely a `String` which has to be associated with an implementation of a `Player` in order to actually interpret the music. A `Player` is specified by a number of functions, among which one that says how the notes are played by this player.

A `Performance`, on the other hand, is the low-level representation of music in *Haskore*, which can be obtained by interpreting a piece of `Music` in a certain `Context`. The `Context` is simply a fixation of some reference parameters (such as the base volume, the base pitch or the base duration) together with a specification of a number of default things to be used, when some specialized ones are not available (such as a default player or a default instrument). Apart from the `Context`, the other thing that is needed in order to transform a piece of `Music` into a `Performance` is a mapping of player names to actual `Player`s. This high-to-low level transformation is done with this function:

```
perform :: PMap -> Context -> Music -> Performance
```

Last, but not least, it is clear the this system would be completely useless unless it was able to interact with some external formats. Although the original plans were ambitious, the reality is only about half way through in terms of format support. Thus, reading and writing from/to MIDI format is supported and writing to C-Sound format is supported. When writing, functions take a `Performance` to a MIDI or C-Sound file, while when reading a `Music` instance is generated from the MIDI file. From the user's point of view, this interaction is trivial, it only consists of calling one function, either for reading or for writing. Behind the scenes, of course, a lot of work has to be done in order to perform the transformations.

2 Advantages of this embedding

As usual, higher-order functions and laziness play a fundamental role in making the embedding powerful: one can easily map transformations over a `Music`, construct infinite musical objects, and even use Haskell's purity to reason mathematically over musical concepts, as explained in Hudak's paper.

The primitives chosen and their combinators are clear for non-Haskell programmers, since they represent the fundaments of music theory. The syntax also seems quite intuitive and easy to use.

Being built on top of another language also makes life easier for *Haskore* when parsing MIDI files or outputting to one of the two supported formats, using `Writer`s and `Reader`s in the monadic style.

However, there is more in Haskell that could have been used by this embedding. Almost no type classes are used, and one would expect that using those would make the approach more elegant and easier to extend (for instance, it would be good to be able to `fmap` over a musical `Phrase`).

3 Personal experience

To present this report, we spent sometime dealing with *Haskore*, thus getting acquainted with its use and underlying structure. The immediate impression one gets is that the fact that *Haskore* is not actively developed for more than 5 years makes its use harder: it only works with Hugs, and with recent versions of the interpreter some changes are necessary in the source code.

Furthermore, some of the implementations are not as powerful as one would expect. For instance, the `retrograde` and `inverse` only work on sequences of music, but one can (somewhat) easily construct versions that support parallel composition as well: for the `retrograde`, this just implies careful handling of parallel notes with different durations, adding a rest to the shorter one. The `inverse` requires clever handling of the first note (on each horizontal line) to make the inversion always refer to the same starting point, but also raises some questions on what to do when a line "forks" into two lines — possibly the author kept his implementation simple to avoid running into these problems.

The process of reading from MIDI files produces `Music` objects which do not conform to the musical structure one would expect: instead of musical lines paralleled with each other, one finds small blocks of parallel components sequentially connected, which disrupts any attempt to do any clever musical processing on a read file. This, together with the absence of a GUI, makes it impossible to use anything other than small fragments of `Music`.

On error reporting, apparently no care has been taken. In fact, some functions will even end in a pattern match failure, which is for sure something that someone not familiar with Haskell will not understand.

Concluding, we can say that the *Haskore* system was a good idea but, to allow for nowadays usage, further work on the project would be necessary, alongside with active development.