

The Monad.Reader/Issue2/Haskore

From HaskellWiki

< The Monad.Reader | Issue2

This article needs reformatting! Please help tidy it up.--WouterSwierstra 14:14, 9 May 2008 (UTC)

Contents

- 1 Haskore
 - 1.1 Preparation
 - 1.2 Building blocks of Music
 - 1.3 Output
 - 1.4 Functional Music
 - 1.4.1 Scale Theory
 - 1.4.2 Making it Audible

1 Haskore

This Article will be about Haskore, which is a Haskell library for describing music. It follows an approach of describing a domain specific language and thus reduces complications of arbitrary language decisions. Imagine, for example, a structure describing Music in any imperative language and compare this to the simplicity of a Haskore representation.

A core of the Haskore system is Score data, which is stored as a Type called `Music`. Score data is usually represented like this:

attachment:haskore1-ex1.gif

This sequence of Symbols, while looking relatively simple to the musician's eye, gives us a lot of information about the music we associate with it. Some of the information encoded here would be the number of notes struck, their lengths, how they overlap or don't.

Implicitly, we assume that we're dividing an octave into 12 "halftones" (look at the numeric values of $(12\text{th root of } 2)^n$ for $n=0\dots 12$ and compare to "simple" fractions to gain understanding of the significance of this number: $n=7$ is the "fifth" (c-g), $n=4$ the "major third", $n=3$ the "minor third" etc.), that a certain note called "a" represents 440 oscillations per second and a few other, even more arcane things.

We shall see that we need to give our computer all this information to reproduce the music that we associate with these notes.

1.1 Preparation

Get [attachment:tmr-Haskore.tar.gz Haskore]. This version of Haskore is ancient but stable. I corrected it slightly to account for features that changed in the meanwhile.

Unpack the file. It provides some documentation and the Haskore sources.

To use Haskore interactively, change to `Haskore/Src` and start `hugs` (you could also use `ghci`, but be sure to put the file `Haskore/ghc_add/IOExtensions.lhs` into the `Haskore/Src` Directory before.) Type `:l HaskoreLoader` and `:m Basics` to initialize Haskore for immediate Experimentation. `:l example` will load Haskore, some declarations in the examples in this text, and import `TestHaskore` which will save us some time and brains by defining reasonable defaults for some features.

Follow-ups of this article will need `Csound`. Most distributions of operating Software will allow for a relatively easy installation.

1.2 Building blocks of Music



Haskore offers a data type called `Music` that represents - as you might have guessed - music. The "atoms" of music, notes, can be generated by giving their "pitch class" (This is where the implicit assumption that our octave is divided into 12 pitches shows up) , octave, duration, and a List of Attributes , like in:

```
#!syntax haskell Basics> :t (c 1 (1%4) []) c 1 (1 % 4) [] :: Music
```

This snippet would represent a "c4" note, played for a fourth measure. The infix operator `%` is used to create a rational number. This way we can easily specify triplets, for example, which are harder in inherently quantized environments.


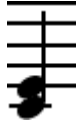
The names Haskore gives to the "pitch classes" are, as one would expect, the names used in the Anglo-Saxon languages, that are, `a b c d e f g`. Sharp and Flat pitches are available via `as` and `af`, respectively. Note that this encoding is an absolute one and does not differentiate in any way among "enharmonics", like `es` and `f`

Now how do we make this single note a music? We will have to combine it with other notes. There are two obvious way to do this.

||<^> attachment:haskore1-ex2.gif ||<#eeeeee> :+ : ||<^>  || = ||<^> 

The sequential composition, expressed by the operator `(:+:) :: Music -> Music -> Music`

results in a value that represents both values in temporal composition (I am tempted to write "played one after the other", but there is no playing going on for now, so this would be a bad idea)

`||<^> attachment:haskore1-ex2.gif ||<#eeeeee> ::= ||<^>  || = ||<^>  ||`

The parallel composition `:::` has the same type, but composes both values to one that represents them simultaneously ("played at the same time").

Using `:t`, we can see that both Operators take two `Music` values and return a `Music` value. Using these Features and the rests (which are named `qnr`, `hnr` etc., for *quarter note* rest, half note rest), we can already construct a lot of music.

Other useful operators (Actually, all the "operators" mentioned are just infix type constructors for `Music` values - see `Basics.lhs` line 34...43. The semantics of the constructed Score is to be added later) are `Trans :: Int -> Music -> Music` and `Tempo :: Ratio Int -> Music -> Music`. Use them to Transpose tunes, or to change their speed.

The list at the end of each note does not seem to make much sense until now. It is intended to hold notewise attributes. For example, the Volume of a Note can be kept here, since it might be different for each single Note. `c 4 (1%4) [Volume 50]`, for example, would represent a quarter "c 4", played at "Volume 50". While we have a clear definition for "c 4" and "1%4", we don't have one for "Volume 50". This will become important now, when we want to make our music audible.

1.3 Output

What is missing now to play that music? Since there is no inherent support for `Music` in "Computers" (Turing-Equivalent Machines), we need to output something that a given synthesis equipment understands. A canonical choice for score data would be `midi`. The only information still missing in our `Music` Data is `midi` channel numbers.

A Haskore abstraction for converting score Data to something closer to acoustical reality is a "Performance". There is a function `perform :: PMap -> Context -> Music -> Performance` that can convert `Music` to a `Performance`, given a `PMap` (a Mapping from player Names to Players) and a `Context` (which is not interesting right now, but can control how various performances will be coordinated).

For example, we can turn an arbitrary `Music` Value to a `Performance` like this:

```
#!syntax haskell Main> perform (\_->defPlayer) defCon example1
[Event{eTime=0.0,eInst="piano",ePitch=48,eDur=0.5,eVol=113.5,pFields=[]}]
```

Using some default Values we nicked from `TestHaskore.lhs`. We see that the `Volume 100` note attribute was converted to an event volume of `113.5`. Considering that result, it's questionable if the default values were chosen all that wisely.

Using

```
#!syntax haskell Main> perform (\_->defPlayer) defCon example2
[Event{eTime=0.0,eInst="piano",ePitch=48,eDur=0.5,eVol=113.5,pFields=[]},
```

```
Event{eTime=0.5,eInst="piano",ePitch=48,eDur=0.5,eVol=113.5,pFields=[]},
```

```
...
```

We see that a performance is a flat list of `Events` as opposed to a `Score` value, which is rather tree-like in structure.

Now we are ready to write these events out to some musical format, for example `midi`. We needed some additional information to write out the `midi` file, namely a "patch map" to map the instrument name "piano" to the `midi` "Acoustic Grand Piano" (Instrument 1) on Channel 1. For other instruments, you could just extend the list. (For a list of instrument names, see `Haskore/Src/GeneralMidi.lhs`)

```
#!syntax haskell Main> outputMidiFile "example2.mid" (performToMidi (perform (\_->defPlayer) defCon
example2) [("piano","Acoustic Grand Piano",1)])
```

This call gives no visible output. After that, you should, however, find `example2.mid` in your current directory. Open it with your favourite (I recommend "Rosegarden" [1] (<http://www.rosegardenmusic.com/>) on Unix-derivate systems) `midi` Sequencer/Editor tool, or play it back. For ease of use i put all these bits together to a function in `example.lhs`

```
#!syntax haskell Main> midiout "example2.mid" example2
```

1.4 Functional Music

How could functional programming help us specify music? Haskell variables can of course take `Music` values, and build other values from them, so we can for example `Transpose` a given piece of music.

We could, for example, write a function that converts a list of intervals (integers) and a `Music` value to a chord.

```
#!syntax haskell mychord intervals base = map (\n->Trans n base) intervals
minor = [0,3,7] major = [0,4,7]
```

0 is the prime, 3 the small third, 4 the large third and 7 the fifth. Now we can specify a simple chord progression:

```
#!syntax haskell example3 = (c 4 (1%4) [Volume 100]) :+:
```

```
(g 4 (1%4) [Volume 100]) :+:
(f 4 (1%4) [Volume 100]) :+:
(c 4 (1%4) [Volume 100])
```

```
example4 = mychord major example3
```

As we see, `mychord` works with any music value. What it can't do is building different chords on top of a sequence of notes. So:

```
#!syntax haskell example5 = (mychord major (c 4 (1%4) [Volume 100])) :+:
```

```
(mychord minor (d 4 (1%4) [Volume 100])) :+:
(mychord major (g 4 (1%4) [Volume 100])) :+:
(mychord major (c 4 (1%4) [Volume 100]))
```

gives us a sequence with different kinds of chords.

1.4.1 Scale Theory

Now as one might know, different "Modes" of (European, traditional) Music use the same sequence of intervals, just starting from a different point in the sequence (`Mode`) and note (`Key`, `Tonic`). Using the Major scale as the original one:

```
#!syntax haskell > maj_skips = [2,2,1,2,2,2,1]
```

we declare a helper function `runsum`, which just sums up numbers in a list continuously.

```
#!syntax haskell runsum = scanl (+) 0
```

Now we can declare all the scales based on the major scale in one function, and for example, have a look at the intervals of the minor scale.

```
#!syntax haskell scale kind = runsum (drop kind (cycle maj_skips))
```

```
#!syntax haskell Main> take 8 (scale 5) [0,2,3,5,7,8,10,12]
```

We need `cycle` because scales repeat all 8 "steps" (every octave). The intervals of the major scale, taken from the sixth (since we start counting with 0: 5), give the (natural or aeolian) minor scale.

We declare a simple melody "step" wise, as in "steps" of a scale (the 8th step being the octave, 12 halftones, and the other steps depending on the exact scale used)

```
#!syntax haskell simplemelody = [(0,1%4),(5,1%2),(4,1%8),(3,1%8),(2,1%4),(5,1%2),(0,1%4)]
```

```

* Specify a value of type (Ratio Int->Music) and call it
base (as it will become the base tone (Tonic) of our melody,
if we give it an arbitrary length)
* Find out how many halftones are between the base of a scale and
the "step" wanted: trans n = (fromInteger (scl !! n))
* Transpose the base note, given a length to complete it, about that
amount, to get the ultimate result.

```

```
#!syntax haskell
```

```

realize :: Int -> (Ratio Int->Music) -> (Int,Ratio Int) -> Music
realize kind base (n,len) = Trans (trans n) (base len)
  where
    trans n = (fromInteger (scl !! n))
    scl = (scale kind)

```

We'll write another helper, that realizes a few notes and puts them in a sequence:

```

#!syntax haskell testrealize kind base melody = allseq $ map (realize kind base) melody

```

1.4.2 Making it Audible

Now we can realize our melody in an arbitrary scale, on an arbitrary base pitch, like for example:

```

#!syntax haskell Main> midiout "major.mid" (testrealize 0 (\l->(c 4 l [Volume 100])) simplemelody)
Main> midiout "minor.mid" (testrealize 5 (\l->(d 4 l [Volume 100])) simplemelody)

```

in c major, and then in d minor. This task (transpose and change mode) makes a nice (and often-cursed) exercise for music students. Thanks to Haskell we were able to solve it in some 20 lines of code.

Now of course we also want to describe music that's not only single-voiced. For example, we could want to describe the a'th three and four note chord in our scale:

```

#!syntax haskell tri a = [a,a+2,a+4] tet a = [a,a+2,a+4,a+6]

```

and put the chords numbered 1, 5, 4 and 1 after each other (if you ever thought you couldn't tell a *I-IV-V-I progression* even if you saw one, now you did), putting in a four note chord here and there, and adding an octave to the last I:

```

#!syntax haskell test2d:: (Int,Ratio Int) test2d = [alllength (1%2) (tri 0),

```

```

    alllength (1%2) (tet 3),
    alllength (1%2) (tet 4),
    alllength (1%2) (8:(tri 0))]
  where alllength l= map (\a->(a,l))

```

Now we only need to map realize twice to that, and then fold twice (first in parallel, then serially) to make this a Music value.

```

#!syntax haskell rea2d kind base melody = allseq $ map allpar $ map (map (realize kind base))
melody

```

Try:

```

#!syntax haskell midiout "iivviprog.mid" (rea2d 5 (\l->(f 5 l [Volume 100])) test2d)

```

And listen to it.

This would be all for this issue of TMR. If you should feel bored, try Haskore yourself. For example, you could:

```
* Try to write an own melody, either using realize to later change scale, or without.  
* Put fitting chords along simplemelody, or put a melody along test2d  
* Read in some existing midi files using readMidi and try to analyze the resulting Music values. (for example, a  
* If you're really bored: get some sheet music and realize it in Haskore.
```

Anyway, stay tuned for the next Issue of TMR. If you have any questions, join us on freenode (just point your IRC client to irc.freenode.net), channel #haskell, and don't hesitate to ask me.

Bastiaan Zapf ([freenode basti_](http://irc.freenode.net))

CategoryArticle

Retrieved from "http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue2/Haskore"

- This page was last modified 19:12, 11 August 2008.
- Recent content is available under a simple permissive license.