AlgoRhythm: A Library for Algorithmic Music Composition

AFP: Final Report

Joris ten Tusscher, Cas van der Rest, Orestis Melkonian

1 Domain

We set out to implement a Haskell library for algorithmic music composition.

1.1 Algorithmic music composition

The process of algorithmic music composition is mostly helpful as an additional aid to a composer's toolbox, allowing for generation of vast and highly diverse musical material. The composer's role then is to refine this material by identifying interesting segments and combine them in a tasteful, coherent whole.

1.2 Music representation

An immediate requirement to facilitate algorithmic music composition is the ability to concisely represent music. One could use the universal standard for musicians of western music, namely sheet scores. This representation, although incredibly helpful for real-time interpretation by human performers, is not fitted to computational manipulation. An appropriate representation would utilize the added expressiveness computation bestows upon us.

We refrain from taking music analysis into consideration, since it is outside the scopes of our library and could be independently performed at a separate stage of a composer's workflow. Hence, we do not provide any functionality to pre-existing pieces of music.

1.3 Music generation

Apart from representing music, there is a need to randomly generate a variety of interesting musical pieces. Furthermore, one would want to have some control over the generation procedure, such as requiring the output of melodies respecting a specific structural rule.

Naturally, our library should output generated music in some common format. We chose to support common sheet notation in PDF and digital formatting in MIDI.

1.4 Motivation

This project is an homage to the late Paul Hudak, one of the creators of Haskell and professional jazz pianist. His work combined functional programming and computer music, which essentially gave birth to this particular line of research (for current work, see ICFP's FARM workshop). We were greatly influenced by his general datatype-oriented view of polymorphic temporal media [**?**], his music Haskell library Euterpea and his linguistic approach to music generation [**?**].

2 Main Concepts & Techniques

2.1 Representation DSL

For representing music, we defined a highly-abstract `Music` datatype, polymorphic to the specific elements of music manipulated through time. Hence, this datatype only specifies the operations invariant across all such specific representation, such as sequential/parallel composition of music and basic construction of music events or silence (of a certain duration), as shown below:

```
data Music a = Music a :+: Music a
             | Music a :=: Music a
             | Note Duration a
             | Rest Duration
```

Since our datatype is polymorphic, we can freely change abstraction levels of music representation (as can be witnessed by our applications). To allow for playback/rendering/exporting, we define a core representation `MusicCore = Music (Pitch, Attributes)`, which every other abstraction should convert to.

Futhermore, we define common constants (e.g. scales, chords), useful types of music (e.g. `Melody = Music Pitch`, `Rhythm = Music ()`, `Harmony = Music Chord`), as well as operations often used in algorithmic music composition, such as transposition, inversion, mirroring, scaling and repetition. These are implemented as typeclasses, since multiple music types could be manipulated in the same way (e.g. we could transpose both a `Pitch` and a `Scale`).

2.2 Export functionality

Export functionality is provided for both `.midi` files and `.ly` files, which can be converted to PDF using Lilypond.

2.2.1 Musical Score

Any value of type `Music` can be converted to a Lilypond file representing a musical score using `writeToLilypondFile "music.ly" m` functions, provided that `m` is a member of the `ToMusicCore` type class, meaning that it can be converted to the `MusicCore` type. Assuming that the user has Lilypond installed on their system, the generated file can be converted into a PDF using `lilypond music.ly`. An example of a generated score can be found in figure ?.

2.2.2 MIDI files

The music representation DSL is very different from how music is represented in MIDI: whereas the DSL has Notes, Rests and methods for music composition, MIDI is just one big chain of events. For example, a Note in the DSL with duration d would be represented in MIDI by a NoteOn event with a certain timestamp t and a NoteOff event with timestamp $t + d$. There are *many* differences however, and therefore the decision was made to not write all the conversion code manually. Instead, Euterpea has been used to convert music from the DSL to a MIDI file, or to a MIDI stream that can be played in real time. Since Euterpea's DSL is very similar to the DSL that was created for this project, it made MIDI export a lot easier.

2.3 Generation

monadic interface, selectors, etc... A simple, constraint based, monadic interface is included in order to allow a user to program their own music generators. This interface is defined as a combined `State` and `IO` monad: `type GenericMusicGenerator st s a = StateT (st s) IO a`.

We provide an instance of this interface to generate for the types used in the representation DSL, defined as `type MusicGenerator s a = GenericMusicGenerator GenState s a`.

2.3.1 Default generation state

The `GenState` datatype consists of several values of the type `Entry s a`, where `a` is the type of the value that is contained within the `Entry`, and `s` is the state that is kept when selecting entries from the `GenState`. Each `Entry` consists of three elements: `values :: [(Weight, a)]`, `constraints :: [a -> Bool]` and `selector :: Selector s`. The `values` can be used to generate certain values with a higher probability, `constraints` allow the exclusion of certain values, and `selector` describes how to select a value from this entry, where the `Selector` type is defined as follows: `type Selector s a = s -> [(Weight, a)] -> IO (a, s)`. Selection based upon previously generated values is thus possible, since `Selector` is essentially a state monad.

2.3.2 Composing generators

An auxiliary data type `Accessor` is defined, that describes how to get and set values in the generator state:

```
data Accessor st s a = Accessor
  { getValue :: st s -> Entry s a
  , setValue :: Entry s a -> st s -> st s
  }
```

The library functions for adding constraints and selectors are parameterized with an accessor, so that the right `Entry` is manipulated.

2.4 Grammars

To accommodate a linguistic approach to music generation, we provide a DSL for defining generative grammars. These are similar to context-free grammars, with the addition of being probabilistic (i.e. allow assigning weights to individual rules), temporal (i.e. rules are parametric to duration) and node-sharing (i.e. allow repetition of generated symbols using *let*-expressions).

For brevity's sake, we eschew a formal definition of our grammars, and immediately give the definitions of the corresponding Haskell datatypes. We also refer the reader to our grammar implementations for examples.

```
data Grammar meta a = Grammar { initial :: a, rules :: [Rule meta a] }
data Rule meta a = (a, Weight, Duration -> Bool) :-> (Duration -> Term meta a)
data Term meta a = Prim (a, Duration)                                  -- primitive
                 | Term meta a :-: Term meta a                          -- sequence
                 | Aux Bool meta (Term meta a)          -- auxiliary modifiers
                 | Let (Term meta a) (forall b. Term () b -> Term () b) -- let
```

Given an initial symbol and some initial value of type `input`, the grammar expansion will begin rewriting by a randomly-picked activated rule, up to fixpoint. A rule's body can be a primitive terminal symbol of a certain duration, a sequence of terms, a term wrapped with auxiliary metadata which will be used in a post-processing step, or a let-expression of a term, which will share a rewritten term in the let's body (when fixpoint is reached).

The symbol type `a` along with the metadata type `meta` and the input type `input` must be implement a way to strip off metadata (possibly converting to some more concrete representation. This is captured by the *Expand* typeclass:

```
class Expand input a meta b | input a meta -> b where
  expand :: input -> Term meta a -> IO (Term () b)
```

When the aforementioned fixpoint is reached, we are left with metadata-enriched terms and *let*-expressions. The expansion proceeds as follows: the auxililary metadata are stripped-off (by `expand`) and the *let*-expressions are unfolded. This is then trivially converted to our `Music` datatype, since we only have to deal with primitive values (i.e. `Notes`) and sequential composition (i.e. `:+:`).

2.5 Dynamic performance

One aspect of music in which humans distinguish themselves from computers is dynamic performance, e.g. playing notes with a different volume based on one's interpretation of the music. To mimic this behaviour, a heuristics based approach was used. Firstly, for every note, the absolute start time is calculated from when the music started, as well as the pitch value of the note represented as an integer (where a higher integer means a higher note). One might notice that this is very similar to a rough transcription of the music to scores, since sheet music has the same two-dimensional representation. Then, the music is clustered to find notes that are close to each other. The k-means clustering algorithm was used for this, since it is a very famous algorithm for clustering data in a multi-dimensional Euclidean spaces. The k was set to be the total time of the music divided by the most popular time of a standard measure: four beats. Next, per cluster, the time and pitch value, both in range [0,1], of the note within the cluster are calculated, so the highest note in every cluster has a y-coordinate of 1, the lowest a value of 0. The first note in every cluster an x-value of 0, the last a value of 1. Next, a mapping function that takes the relative x and y coordinate of a note as input and again returns a double in the range [0,1] is used to calculate the volume of all notes in the music. 0 means very soft, 1 very loud. Although MIDI reserves 7 bits for the dynamics of a note, Euterpea, which is used for conversion to MIDI, only supports about 10 different dynamics, so a lot of information stored in the volume Double is lost when it is converted to its Euterpea representation.

3 Results

3.1 Music DSL

To showcase the expressive power of our DSL for representing and manipulating music, we present a very short piece of code, which however employs a variery of operators (transposition, retrograde, time-scaling, delay) and generates a rather complex musical effect (audio):

```
hypnotic :: Melody
hypnotic = 2%5 *~ cascades :+: (cascades ><)
  where cascades = rep id      (%> sn) 2 cascade
        cascade  = rep (~> M3) (%> en) 5 run
        run      = rep (~> P4) (%> tn) 5 (D#3 <| tn)
        rep :: (Melody -> Melody) -> (Melody -> Melody) -> Int -> Melody -> Melody
        rep _ _ 0 _ = (0~~)
        rep f g n m = m :=: g (rep f g (n - 1) (f m))
```

3.2 Generation DSL

The monadic interface allows for relatively easy manipulation of the generation process. There are a few ways in which a user may influence the way in which the music is generated. They in-

clude adding constraints, choosing selectors and setting the frequency/probability for individual elements.

```
-- Add a new constraint (only quarter notes or faster)
addConstraint duration (<=(1%4))

-- Provide a custom selector
putSelector pitchClass mySelector

-- Provide a custom distribution for elements (favor higher octaves)
putOptions dynamics
  [ (0.01, PPPPP)
  , (0.05, PPPP)
  ...
  , (0.3, FFFF)
  ]
```

An aggregate example where the interface is used to generate a melody in the C major scale over a G7 chord with a length of 4 whole notes is shown below:

```
melodyInC :: MusicGenerator () MusicCore
melodyInC = do
  addConstraint pitchClass (inScale C major)
  options <- getOptions pitchClass
  rhythm  <- boundedRhythm (4 * wn) Medium
  -- Chord tones are 4 times as likely to be picked
  let options' = map
        (\(w, v) ->
          if v `elem` (instantiate G d7 :: [PitchClass])
            then (4 * w, v) else (w, v)) options
  -- set options and generate pitches
  putOptions pitchClass options'
  pitches <- replicateM (length rhythm) (pitchClass??)
  -- put everything together into a piece of music
  let fullPitches = (flip (<:) $ []) <$> (zipWith (#) pitches (repeat 4))
  let gmaj7 = (toMusicCore . chord .
        map (Note (4 * wn) . (flip (#)) 3)) (G =| d7)
  return $ gmaj7 :=: line (zipWith (<|) fullPitches rhythm)
```

The monadic interface allows for easy implementation of simple heuristics, as demonstrated above. A more involved example of this can be found in the `Generate.Applications.Diatonic` module, which generates melodies based on the idea that smaller intervals (steps and skips) between consecutive notes are more common than larger intervals (leaps).

3.3 Grammars

To showcase the generative power of our grammars, we implement grammars all basic elements of music, namely harmony, melody and rhythm. To save space, we only explain the intuition behind each grammar and refer the reader to the actual implementations.

3.3.1 Tonal harmony [code]

Harmonic structure is argued to exhibit recursion and hierarchical organization, hence fit nicely to our grammar framework. We directly implement the generative grammar proposed by Rohrmeier[**?**], which captures harmonies on diatonic cadences (i.e. movements between chords of a different degree in some scale). The grammar is based on a Schenkarian view of harmony[**?**], namely the abstraction of an elaborate chord progression as a single scale degree. For instance, the cadence V-I can be seen as a harmonic element based on the tonic (I).

The grammar rewrites basic scale degrees into more elaborate ones, making sure the produced chord progressions have appropriate durations. One important addition is that of *key modulation*, where an auxiliary key wraps a grammar term and makes the generated subprogression be interpreted in a transposed key from the original. This is handled particularly well by our `Aux` terms and `Expand` typeclass.

3.3.2 Jazz melodic improvisation [code]

For melodic improvisation, we implemented the context-free grammar behind the open-source educational tool Impro-Visor[**?**], which aims to train novice musicians in jazz improvisation. The grammar consists of two-levels, one to produce an interesting rhythmic structure of a given length and one to fill these values with notes having a certain function against some harmonic context (e.g. chord tones, approach tones). However, these could be inhabited by many possible concrete pitches, therefore a stochastic post-processing step heuristically assigns theses values (based on pitch distance and preferred octaves). This grammar does not use the metadata-enrichment features of our grammars.

3.3.3 Tabla rhythmic improvisation [code]

Last but not least, we implement a grammar for improvising rhythmic sequences for Tabla, an Indian percussion instrument. We base our implementation on a proposed grammar used in the expert system *Bol Processor BL1*[**?**]. The rules operate on a fixed number of syllables, since this is how Tabla music is written (i.e. each syllable corresponds to a particular stroke of the drum set).

Technical note Since there is no standard MIDI encoding of tabla sounds, we had to provide a custom MIDI mapping from notes to percussion sounds. Interestingly, this was elegantly modelled by our `ToMusicCore` typeclass, which provided the custom mapping.

3.4 Dynamic performance

Although the clustering of notes that is used to add dynamics, users can still define their own mapping function that converts the x and y coordinate of a note to its dynamics, but some standard functions have been provided as well. One very simple example is a 1 oscillation long sine wave based on Note's x-value. Such a function might not always give good results, because notes will sometimes be soft or loud purely based on their relative time within their cluster, and not based on their pitch height in the cluster. Another example is the quantile function of the exponential distribution, that takes the Note's y-value (relative pitch height within the cluster) as input.

3.5 Produced songs [code]

We accumulate our results in several musical pieces automatically generated by our library (up to instrument assignment). You can see the corresponding music scores here and the audio files

in AlgoRhythm's dedicated Soundcloud account.

4 Reflection

more type trickery

Although the monadic interface works reasonably well for implementing simple heuristics, actually generating more complex pieces of music turned out to be quite hard. We think that this flaw is fundamental to our approach, as the `GenericMusic` type is very *flat*, in the sense that it is only aware of the most fundamental aspects of music, such as notes and durations. Since musical compositions often exhibit structure on many levels, generated music that only concerns about the smallest building blocks will not be very convincing.

As it turned out, the selected monadic approach does not scale well in order to incorporate these larger structures into it's output, which makes it a bit of a dead end for more complex purposes.

Regarding grammars, one problem we were not able to overcome, was implementing a grammar for jazz chord progressions, as proposed by Steedman[**?**], since it required context-sensitive features and that would make our grammar implementation overly complicated and necessarily slower.