Final Report

Advanced Functional Programming

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 [1], his music Haskell library Euterpea and his linguistic approach to music generation [2].

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

lilypond, MIDI, etc...

2.3 Generation

monadic interface, selectors, etc...

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

k-means clustering, etc...

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))
```

genDSL+chaos example

3.2 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.

3.2.1 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[3], 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[4], 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.

3.2.2 Jazz melodic improvisation [code]

For melodic improvisation, we implemented the context-free grammar behind the open-source educational tool Impro-Visor[5], 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.

3.2.3 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*[6]. 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.

3.3 Dynamic performance

combine with previous section?

3.4 Magnum Opus [code]

We accumulate our results in a musical piece automatically generated by our library (up to instrument assignment). We attach the corresponding musical score in the appendix and provide an audio link in our Github repository.

4 Reflection

more type trickery
GenDSL abstraction

Regarding grammars, one problem we were not able to overcome, was implementing a grammar for jazz chord progressions, as proposed by Steedman[7], since it required context-sensitive features and that would make our grammar implementation overly complicated and necessarily slower.

APPENDIX: Suite for Koto, Sitar, & Tablas

generate score/audio

References

[1] P. Hudak, "An algebraic theory of polymorphic temporal media," in *International Symposium on Practical Aspects of Declarative Languages*, pp. 1–15, Springer, 2004.

[2] D. Quick and P. Hudak, "Grammar-based automated music composition in haskell," in *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, pp. 59–70, ACM, 2013.

[3] M. Rohrmeier, "Towards a generative syntax of tonal harmony," *Journal of Mathematics and Music*, vol. 5, no. 1, pp. 35–53, 2011.

[4] A. Forte and S. E. Gilbert, *Introduction to Schenkerian analysis*. Norton, 1982.

[5] R. M. Keller and D. R. Morrison, "A grammatical approach to automatic improvisation," in *Proceedings, Fourth Sound and Music Conference, Lefkada, Greece, July.*, 2007.

[6] B. Bel, "Modelling improvisatory and compositional processes," *Languages of Design, Formalisms for Word, Image and Sound*, vol. 1, no. 1, pp. 11–26, 1992.

[7] M. J. Steedman, "A generative grammar for jazz chord sequences," *Music Perception: An Interdisciplinary Journal*, vol. 2, no. 1, pp. 52–77, 1984.