Status Report

Advanced Functional Programming

Joris ten Tusscher, Cas van der Rest, Orestis Melkonian

1 Techniques

1.1 Main Datatypes

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
```

For instance, in Western classical music it would appropriate to instantiate this type with an element representing musical notes, represented as pairs of the pitch class (one of the 12 available ones) and the octave (how low/high is this note played). In South India's vocal tradition *Konnakol*, on the other hand, one would only care about the rhythmic values (i.e. the durations), thus `Music ()` would be more intuitive.

This freedom essentially enable us to choose the abstraction level we wish our music programs to be written in. Most importantly, we define a core representation `MusicCore`, which is what every other abstraction should convert to, in order to allow for rendering/exporting in different formats. The typeclass is given below:

```
class ToMusicCore a where
  toMusicCore :: Music a -> MusicCore
```

We, naturally, focus on western music and define further helpful types/synonyms to represent common concepts encountered in music theory and practice (+ their conversions to `MusicCore`). Furthermore, we provide operations often used in algorithmic music composition, such as transposition, inversion, mirroring, scaling and repetition. These are implemented as typeclasses, in order to allow user-defined `Music` instances to extend them.

Last but not least, we provide commonly-used *constants* of these musical concepts, such as popular scales/chords/durations.

For the generator DSL, we define a monadic interface that combines the `State` and `IO` monad: `type MusicGenerator a = StateT GenState IO a`. In the future, this type might become polymorphic in the type of the state, but as of yet this is not the case. The `GenState` type is defined as follows:

```
data GenState = GenState { selector :: Selector
                         , pc       :: Entry PitchClass
                         , oct      :: Entry Octave
                         -- ... etc ...
                         }
```

Where `type Entry a = ([a], [Constraint a])`, representing all possible values for a certain type, and a list of constraints that are applied to that type. The type `Selector` represents a function that selects a value from a list of possible values, and is defined as `type Selector = forall a . [a] -> IO a`

1.2 Libraries

In order to avoid re-inventing the wheel, we utilized pre-existing Haskell libraries for exporting to MIDI/score formats. Specifically, we use **lilypond**[1] to export music to Lilypond's digital score format, which can then be easily converted to PDF, etc... To enable MIDI export and music playback, we use **Euterpea**[2]. Since each of these libraries defines their own music datatype, integrating them entails converting from our `MusicCore` type, which could be easily and naturally covered by the target types. For testing, we use HUnit[3] and QuickCheck[4].

2 Achievements

Thus far, our main achievements are an extensive DSL for representing/transforming music and the ability to export in MIDI and score formats. Design and implementation of the Generation DSL has started, but is still incomplete. Our code is available on Github[5].

2.1 Music DSL

We have implemented a rather expressive, but simple, EDSL for representing music and transformation. To give you a taste of the available operators, let us consider the example of expressing the harmonic progression of a famous song structure, known as the *12-bar blues*. Given a particular tonic (i.e. note), one can derive 12 bars of chords, moving only between three basic (dominant) chords (I, IV and V) in a certain order. Moving from the tonic to the other ones is achieved simply by transposing a `Chord` by a certain `Interval`. The program expressing this structure is given below:

```
-- Useful type synonyms
type Chord = [Pitch] ; type Melody = Music Pitch ; type Harmony = Music Chord
-- Abstract harmonic structure
bluesProgression :: Pitch -> Harmony
bluesProgression p =
  let tonic = p=|d7 <| wn
  in  line $ map ($ tonic) [id, id, id, id, (~> P4), (~> P4), id, id, (~> P5), (~> P5), id, id]
-- Concrete harmonic structure (Repeat C blues progression four times)
cBlues :: Harmony
cBlues = 4 ## bluesProgression (C#3)
```

Given a way to improvise over such $D7$ chords, i.e. a function of type `Chord -> Melody`, one could derive an improvisation line on top of this harmony, as demonstrated below:

```
-- Parallel composition of harmonic progression and improvisation line (played an octave higher)
cBluesImprov :: Melody
cBluesImprov = chords cBlues :=: flatten (improviseOverD7 <$> cBlues) ~> P8
-- Create a line from a 4-note chord (we use postfix operators for silence)
improviseOverD7 :: Chord -> Melody
improviseOverD7 [a, b, c, d] =
  a <| qn :+: (b <~ Mi2) <| en :+: b <| en :+: (c <| qn :=: d <| qn) :+: (en~~) :+: d <| en
```

We attached the generated score (i.e. `writeToLilypond cBluesImprov`) in the Appendix.

[1] https://hackage.haskell.org/package/lilypond
[2] https://hackage.haskell.org/package/Euterpea
[3] https://hackage.haskell.org/package/HUnit
[4] https://hackage.haskell.org/package/QuickCheck
[5] https://github.com/omelkonian/afp-project

2.2 Generation DSL

Goal of the generation DSL is to provide an interface for a programmer to quickly generate random pieces of music. We aim to enable the user to customize the generation process by either constraining which values can be generated, or by supplying functions that generate values for the appropriate types in place in place of just sampling them from their domain with uniform probability. We refer to the latter as `Selector`, as they select values from a certain domain.

2.3 Monadic Interface

The generation DSL is provided as a monadic interface that a programmer can use to steer the generation process. The `GenState` type contains the internal state of a generator, consisting of the types to generate for, and a `Selector`, and can be manipulated through several library functions, on top of the functions that the generic `StateT` interface provides.

A function `runGenerator :: MusicGenerator a -> IO a` is provided that can be used to actually run a generator, alongside several functions that generate values for the data types used in our representation DSL. Below is a small example of a generator that generates a short melody.

```
-- Generator execution
main :: IO ()
main =
  melody <- runGenerator randomMelody
  -- from here, you could export to a MIDO or score file

-- Generator definition
randomMelody :: MusicGenerator Melody
randomMelody = do
  notes <- replicateM 10 genNote     -- Generate random notes
  let melody = line notes            -- Combine into a melody
  return $ melody :+: invert melody -- Sequence melody with it's inversion
```

As can be seen from the example above, the generator DSL integrates very nicely with the existing representation DSL. Furthermore, since the generator interface is a monad, generators can be manipulated in the same way as any other monad, making it very easy to compose larger, more complex generators from smaller ones.

2.4 Future Extensions

Although the basic structure for the generation DSL is in place, there is still quite a lot of work to do. Up until now, work on the generation DSL has mainly focused on finding a suitable approach. Now that we have one, we can start defining library functions that operate on the data types of the representation DSL. A few of the more notable parts of the DSL that we will be working on in the upcoming weeks are:

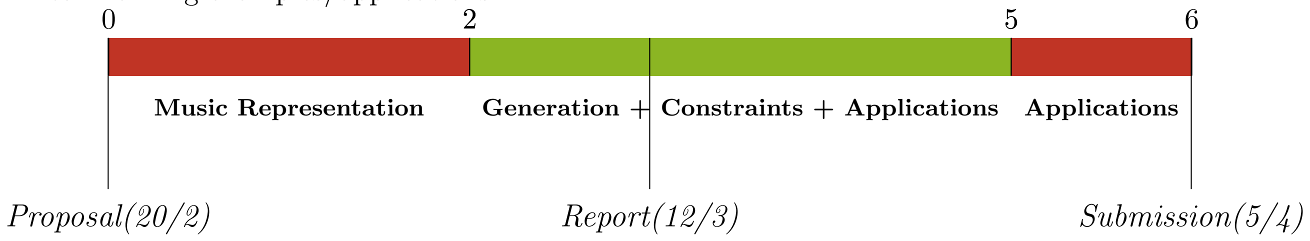
- Suitable library functions for manipulating constraints, i.e. an intuitive way of adding (and removing) constraints from the generation process. We intend to add a `local` function that will allow for partial generators to run under a different set of constraints without the need to remove and re-add all existing ones.

- The possibility to apply a custom `Selector` to a generator to allow for non-uniform selection of values. The aim for this approach is to allow for many different possible ways of generating music besides complete randomness, such as chaos functions or L-systems. An added benefit is extensibility, as a programmer may come with their own procedure for determining appropriate musical values, which can then be applied to any `MusicGenerator`.

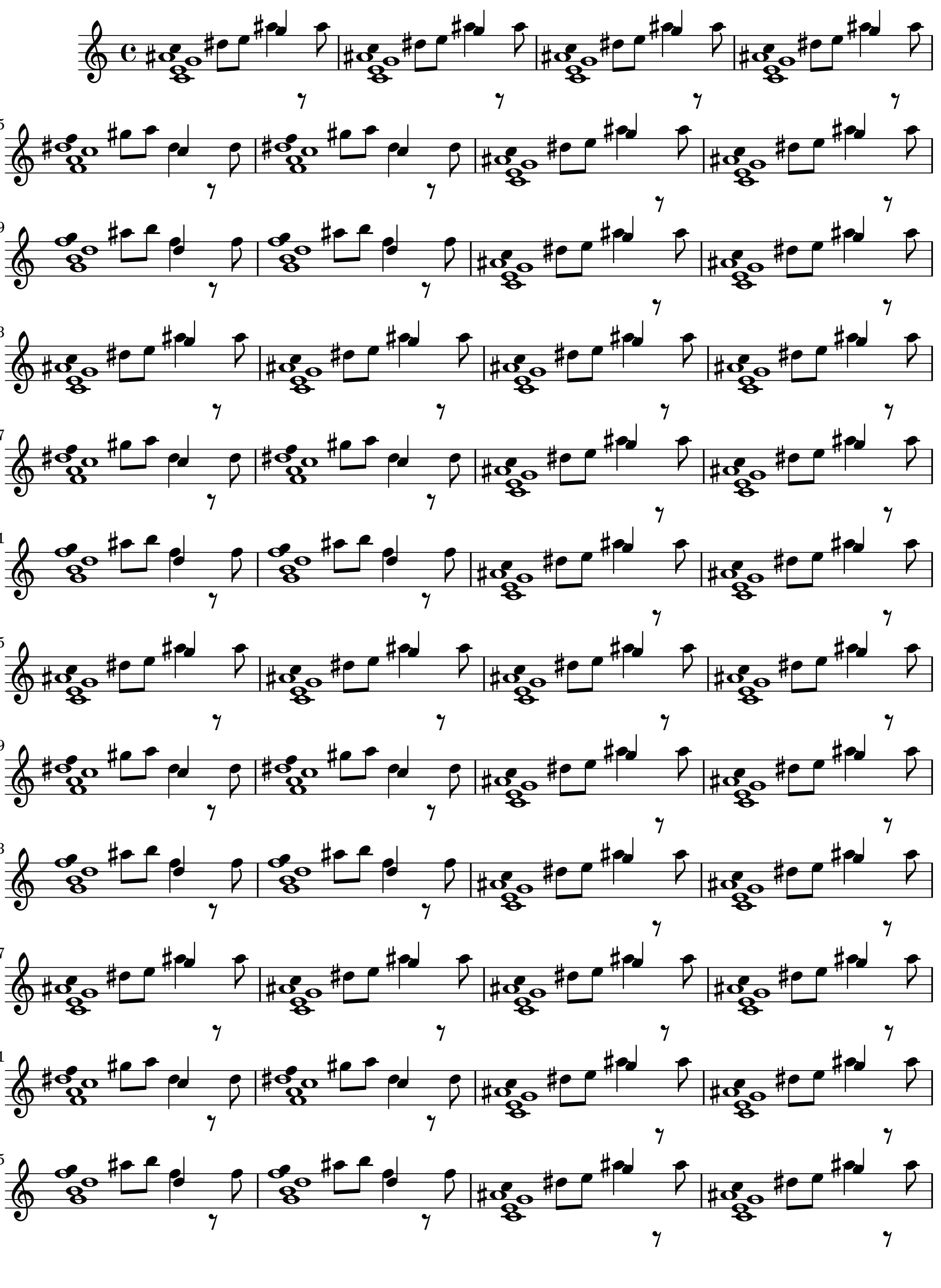
Beside the extensions mentioned above, we also intend to include some small examples of applications for generators. Possibilities for this include automatically generating melodies over a certain chord progression or generating random exercises for certain scales or chords.

3 Planning

We are more or less on track with our initial planning, but we realised we cannot tackle Generation and Constraints separately. Thus, we decided to aim for a more integrated and iterative solution, where we gradually design/implement the GenDSL with constraints in mind. Since this task is not easily distributed amongst us, we will simultaneously work on GenDSL applications (i.e. chaotic systems, probabilistic rewriting grammars) from week 4 onwards, retrospectively adapting to new requirements we may encounter. Finally, the last week will be solely devoted to finalizing examples/applications.



APPENDIX: Generated Score for Blues Progression



5
9
13
17
21
25
29
33
37
41
45