Scala Music Generation

Valérian Pittet Nada Amin Viktor Kuncak
EPFL
{first.last}@epfl.ch

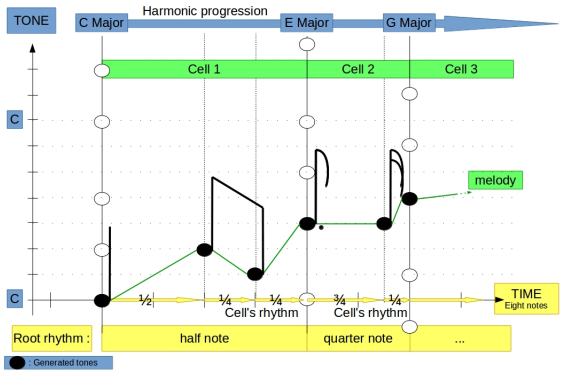

Figure 1. Concepts

Abstract

We present a case study in Scala of automated music generation.

Categories and Subject Descriptors H.5.5 [*Information Interfaces and Presentation*]: Sound and Music Computing

Keywords music generation, automatic composition

1. Music Generation

1.1 Concepts

The generated piece is divided into cells. Each cell has a duration, a harmonic chord, a rhythmic pattern and a melody. These four properties constrain each other. In particular, the melody is constrained so that its first note is part of the harmonic chord, and so that there is one note for each element of the rhythmic pattern. Figure 1 summarizes these concepts.

We represent these four properties by mutually-constraining grammars.

```
val chords: Grammar[Chord]
val root: Grammar[RootRythm]
val cells: Grammar[RythmCell]
val tones: Grammar[Tone]
```


In the remaining subsections, we show how to set these grammars to generate increasingly better-sounding patterns. We focus mainly on improving the generation of the melody, so that it is not too jumpy and has a nice shape.

1.2 Purely Random Melody

For now, we fix the chords to repeat a harmonic sequence well-known from classical music: `I - IV - V - I`. The sequence is repeated three times, which defines the duration of the piece.

```
val chords0: Grammar[Chord] =
  Triad(I) ** Triad(IV) ** Triad(V) ** Triad(I)

lazy val chords: Grammar[Chord] =
  repeat(3)(chords0)
```

We also fix the rhythm using infinite deterministic grammars. The duration of a cell is a half-note (`H`), while a cell has three notes each time with the pattern quarter-note eight-note eight-note (`Q +: E +: E`).

```
lazy val root: Grammar[RootRythm] =
  H ** root

lazy val cells: Grammar[RythmCell] =
  (Q +: E +: E) ** cells
```

Finally, the melody is defined by a grammar of tones. For now, the grammar simply allows any tone for each note. Figure 2 depicts this unconstraining grammar.

```
lazy val tones: Grammar[Tone] =
  (I || II || III || IV || V || VI || VII) ** tones
```

Here is a sample result for the first harmonic sequence:

```
Triad(I) I I III
Triad(IV) IV II III
Triad(V) VII V III
Triad(I) I VI II
```

We see that each first note in a cell is constrained to be part of the chord triad. Otherwise, the melody is unconstrained.

1.3 Flowing Melody

Our first improvement is to create a more flowing melody by changing the grammar of tones, as illustrated in Figure 3. Starting with the tonic `I`, we recursively constrain the next tone to stay within a range of the previous tone. We give the most weight to a single step increase or decrease.

```
lazy val tones: Grammar[Tone] = nextTone(I)
def nextTone(t: Tone): Grammar[Tone] =
  t ** (
    (nextTone(t decreaseBy 2), 1.0) ||
    (nextTone(t decreaseBy 1), 2.0) ||
    (nextTone(t), 0.5)              ||
    (nextTone(t increaseBy 1), 2.0) ||
    (nextTone(t increaseBy 2), 1.0)
  )
```

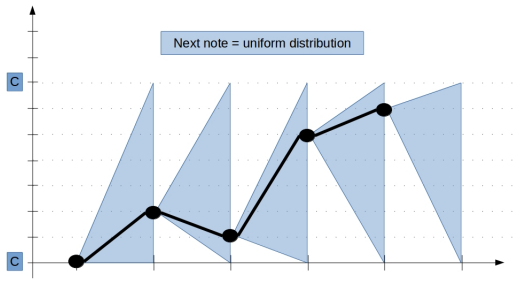

Figure 2. Uniform Random Tones

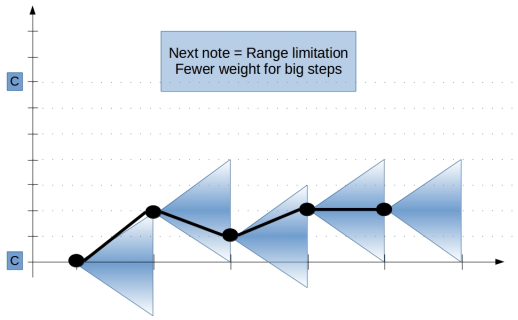

Figure 3. Weights and Bounded Tones

Here is a sample result for the first harmonic sequence:

```
Triad(I) I II II
Triad(IV) I VII I
Triad(V) II III IV
Triad(I) III III IV
```

Like before, we see that each first note in a cell is constrained to be part of the chord triad. In addition, notes take mostly single steps.

1.4 Intertial Melody

Instead of fixing the weights associated with increasing or decreasing the tone, we can also vary the weight according to the past shape of the melody, for example, whether it is already increasing or decreasing. Figure 4 illustrates this scheme.

Here is a sample result for the first harmonic sequence:

```
Triad(I) I II IV
Triad(IV) IV VI VII
Triad(V) II III V
Triad(I) III I II
```

1.5 Variations

So far, we have fixed the harmonic chords and the rhythm. Using alternations in these grammars, we can also introduce variations in those aspects.

```
val chords0: Grammar[Chord] =
```

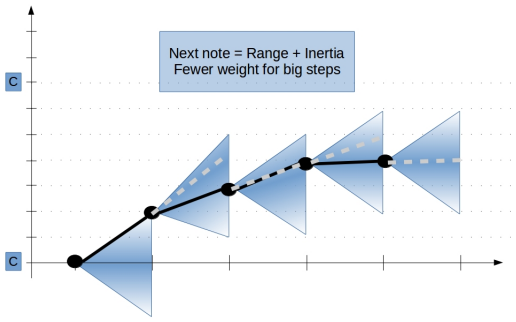

Figure 4. Inertial Tone

```
  Triad(I) **
    ( Triad(V) ||
      Triad(IV) ** Triad(V) ||
      Triad(IV) ** Triad(V) ** Seventh(V)
    ) ** Triad(I)

override
lazy val chords: Grammar[Chord] =
  repeat(3)(chords0)

override
lazy val root: Grammar[RootRythm] =
  ((Q ** Q) || H) ** root

override
lazy val cells: Grammar[RythmCell] =
  ( (Q +: E +: E) || ((Q-) +: E) ) ** cells
```

1.6 End Control

We now focus on controlling the end of the generation. Previously, the chords grammar defined the length of the piece. Now, we tweak the generation so that the grammars setting the rhythm must also end for the piece to end. This entails changing these infinite grammars with variations that can possibly end, while also giving us the freedom to have a different rhythm at the end.

```
lazy val root: Grammar[RootRythm] =
  (rootBody ** root) || rootEnd

val rootBody: Grammar[RootRythm] =
  (Q ** Q) || H
val rootEnd: Grammar[RootRythm] = H

lazy val cells: Grammar[RythmCell] =
  (cellsBody ** cells) || cellsEnd

val cellsBody: Grammar[RythmCell] =
  (Q +: E +: E) || ((Q-) +: E)
val cellsEnd: Grammar[RythmCell] =
  RythmCell(H::Nil)

lazy val tones = nextTone(I) ** tonesEnd

val tonesEnd: Grammar[Tone] = I
```

Here is a sample output:

```
Triad(I) I VI IV
Triad(IV) VI I
Triad(V) II III I
Triad(I) III V IV
Triad(I) III III
Triad(IV) IV IV
```

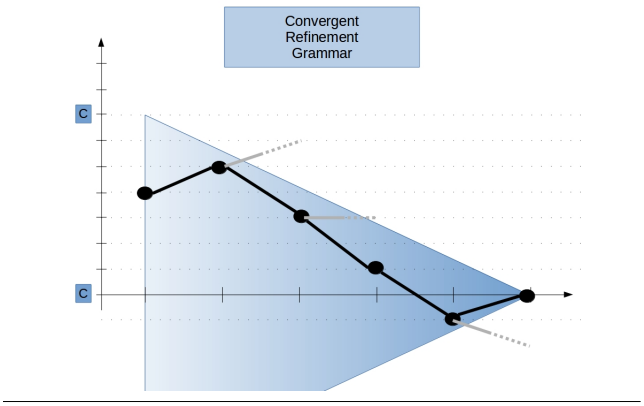

Figure 5. Convergent Refinement

```
Triad(V) V III
Triad(I) V III II
Triad(I) III V
Triad(IV) VI IV
Triad(V) V III
Triad(I) I
```

1.7 Convergence

As we get closer to the end, we would like the tones to converge to the tonic. For this, we leave the tones grammar as is, but from the chords grammar, we send a message to constrain further the tones generated. Figure 5 illustrates the constraint on convergence: at step n, the constraint enforces that only n notes above and below the tonic can be generated.

```
lazy val chords: Grammar[Chord] =
  ( repeat(3)(chords0) **
    MelodyRefine[Chord](converge(10)) **
    repeat(2)(chords0))

// creates an infinite grammar that converges in n steps
def converge(n: Int): Grammar[Tone] =
  // still allows some oscillation after converged
  if (n < 2) converge(2)
  else {
    Production(
      (for (i <- -n to n)
       yield (Word(I increaseBy i), 1.0)).toList) **
    converge(n-1)
  }
```

Here is a sample result:

```
Triad(I) I I VII
Triad(IV) I I
Triad(V) II I
Triad(I) III III
Triad(I) V V VII
Triad(IV) I I
Triad(V) II IV
Triad(I) V III
Triad(I) V V
Triad(IV) VI IV
Triad(V) II III
Triad(I) I VII
Triad(I) I I II
Triad(IV) I VI IV
Triad(V) II III
Triad(I) I II
Triad(I) III I VII
Triad(IV) I I I
Triad(V) II VII
Triad(I) I
```

1.8 Outlook

Because the melody is already constrained by a harmonic progression, we can nicely play the chords at the beginning of each cell. On the other hand, we don't attempt to do any voice leading for the harmonic chords.

2. Related Work

[1, 2]

Acknowledgments

References

[1] J. P. Magalhães and H. V. Koops. Functional generation of harmony and melody. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling and Design*, FARM '14, 2014.

[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 and Design*, FARM '13, 2013.