Scala music generation

Valérian Pittet

June 14, 2014

Abstract

The present report relates to the Scala music generation project. The main goal was to implement a domain specific language adapted to music description and providing useful operators to manipulate melodies.

It presents first how the language was implemented with its different features. Then it discuss the case study that was driven using that defined language. Different approaches are considered and showed with their respective trade-offs.

Finally a small section describes the enhancements that could be brought to the project.

Contents

1 Introduction

Before speaking about highly abstract concepts on music generation, one needs to have a dedicated support to build the abstractions. This is achieved by implementing a domain specific language for music description and generation.

The goal of this project was to design such a language providing brief representation and expressive manipulation methods for abstractions over musical language complexities to conceive more advanced features.

The final DSL is the result of an incremental process, considering at each step which functionalities to provide and how to represent them for the best expressiveness with concise and readable code. The traditional music representation and structures are the guidelines for the DSL specification. This comes to a language representing all the element entangled in traditional music (as tonality, rhythm, ...) independently and combining them by simple operators.

"By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race." (Alfred North Whitehead, 1861-1947)

2 Data representation

When referring to music description, the first problem to solve is the inherent multidimensional representation of traditional music. In a single note, its shape, its position, its color and additional marking describe height, duration and even relationship to other notes of the partition. Especially, simultaneous notes can be written at the same position.

On the other hand, code is written in a linear fashion, disallowing easy simultaneous note description. Taking the midi file format as an example, it solves this problem using note events (note on and note off) with occurring time relative to the previous event. Parallel notes are easily represented with successive zero-delay notes, but at the cost of big trade-offs. Each note generates two distinct and independent events. The concept of chord is not well represented as the first described note delay has to be different from the other ones. Also, the note duration is not explicit and has to be recomputed.

Here, we want more intuitive concepts to describe parallel and sequential patterns. Notes are described as a tuple of height and duration with two composition modes: parallel or sequential. It is enough to consider a melody as a set of multiple tracks played together (parallel composition of sequential notes) with the structure :

```
class Parallel(tracks: List[Sequential])
class Sequential(notes: List[Note])
```

This is also equivalent to having a sequence of chords played one after each other (sequential composition of parallel notes) with the structure :

```
class Sequential(melody: List[Parallel])
class Parallel(notes: List[Note])
```

However both representations are not convenient when it comes to a single voice that splits into two parts for a short time. In any case, one has to consider this small second line from the beginning until the end, almost filled with musical rests. The solution is to generalize those representations to a single one where parallel and sequential segments can be both composed of notes and any kind of segment. Thus the melodies representation is a tree structure where nodes are parallel or sequential compositions and leaves are notes.

2.1 Tree representation

As we want a common interface for sequential and parallel segments, their implementation is a simple syntax tree. This is achieved using the following definitions.

```
trait MusicalSegment {
  def melody: List[MusicalSegment]
}
abstract class SequentialSegment(
    melody: List[MusicalSegment])
  implements MusicalSegment
abstract class ParallelSegment(
    melody: List[MusicalSegment])
  implements MusicalSegment
```

And last but not least, the note representation fits in parallel and sequential segments as leaves of the tree :

```
case class Note(tone: Tone, duration: Duration)
  implements MusicalSegment {
  def melody = this :: Nil
}
```

2.2 Tone and pitch

While note duration is pretty straightforward to represent, heights have different ways of being implemented. The most natural is to use the standard notes name : C, D, E, F etc. As the scale uses twelve half tones, this system can be extended to C, C♯, D, D♯ , E, F, ... This first solution is however not convincing as many operators like `stepUp` are context dependent. `C.stepUp` will not output the same value in a C-major or F-major scale. Musically, it makes more sense to represent heights independently from the context.

That's why here, tones have been chosen. They represent the seven different steps of the scale and are denoted with roman numbers (I, II, III, IV, V, VI, VII). The octave can be specified and tones may be altered or not (♭, ♯, ♮). The choice of tones over pitches has many advantages.

For example, the same melody could be played in major mode or minor mode only changing the scale in which it is interpreted. Minor harmonic is not even harder to manage. Additionally, this hides the irregularity of steps separated by only one halftone. Using tone representation makes it trivial to implements functions like `I + 1 //returns II` .

Looking into more details, this also allows to easily distinguish similar pitches such as C♯ and D♭. Regarding only playing melody this does not make any difference, but when computing intervals or even outputting partitions, it will get more weight.

3 Data description

The next point in this DSL definition is operators that manage the data expression. They are the basic components used to build data instances.

3.1 Base operators

As discussed before, there are two kinds of composition, sequential `+(that: MS): SS` [1] and parallel `|(that: MS): PS` . Each of them will add the `that` argument in the callee's melody. Note that those operators behave according to callee and

[1]For the sake of conciseness, `MusicalSegment` is shorten `MS` , `SequentialSegment` : `SS` and `ParallelSegment` : `PS`

argument. Using `|` on sequential segment will not include the argument in the callee's melody. This would break the contract of `|` that provides parallel composition. In this case, a new parallel segment is created with the callee and the argument as melody.

```
val mySeq: SS = ...
// includes newNote in mySeq
mySeq + newNote
// creates new parallel segment
val depth2 = mySeq | newNote
```

Two more operators `++(that: MS): SS)` and `||(that: MS): PS` complete the basic composition functionality. Each creating a new segment consisting of the callee and the argument. They provide full control on the tree structure, forcing new node insertion. This will prove helpful to implement good recursive operators.

These are the basis for the melody description and can basically describe any melody. However they are not efficient for writing many notes. This leads to a first improvement of defining multiplicative concepts over the additive ones.

3.2 Repetition operators

First the multiplication by an `Int` is defined as repeating the same melody n times. Usually, each sequential operator should have its parallel counterpart. But in this case, repeating the same melody in parallel is nearly idempotent (as the same thing gets played at the same time). Thus this operator returns the melody repeated n times sequentially.

The next multiplicative operator is a generalization of the first one. Rather than repeating n times the same melody, it applies each time a specified transformation. Because the melody changes, sequential and parallel operators are distinguished. They are defined as

```
def fillSeq(trans: (MS=>MS)*): SS

def fillPar(trans: (MS=>MS)*): PS
```

The method implementation infers the identity at the beginning of each transformation sequence as the scope of this operation is to repeat a segment and not to map it to another one.

As application example, `fillPar` can be used to express a canon from a single melody:

```
val canon: MS = ...
canon fillPar (canonShift + _)
```

where `canonShift + _` prepends an empty note of appropriate duration to shift the second voice.

4 Data manipulation

As we are now able to create simple melodies, the goal is to develop them in more complex and interesting structures. This concept requires to run through the tree exploiting its recursive structure and selecting the element(s) to modify.

4.1 Note mapping

A good first attempt to achieve this functionality is a note mapping method.

```
def mapNotes(trans: (Note => MS)*): MS
```

`mapNotes` traverses the whole tree and rebuilds it applying to each note the provided transformation.

For more expressiveness, it is possible to provide multiple transformations. The fist function maps the first note, the second function the second note and so on. When all functions are used, the mapping restart to the first function.

This for example enables simply creating two interleaving patterns that apply one after each other to the notes.

However, several limitations of this mapping call for a more general form. The first remark is that this kind of function may target not only notes but also complete melody parts, responding to a given predicate. This predicate may consider composition type (parallel, sequential), tree depth or any other boolean expression on a subtree.

Then music is never completely regular. One may wish to specify a more complex pattern instead of a simple cycle when applying transformations. Thus each transformation must have additional parameters to specify application conditions. Those concepts are combined as follows.

4.2 General mapping

The generalization of `mapNotes` is defined as
`mapIf(tList: TransformList): MS`

The class `TransformList` provides a list of transformations and selective informations to choose when to apply which transformation. Selection informations are divided in two main categories.

The first one, called selector, determines which parts of the melody are valid candidates for the transformation application. Of course, there may be multiple valid candidates in a single branch of the tree. The mapping simply considers valid candidates as nodes of the tree, hence the subcandidates will not be considered. The selector can be seen as a boolean predicate that can make checks on the type of the nodes (e.g. distinguish parallel and sequential) and cast them to apply any other user-defined boolean function.

This selector is the reason why the `SequentialSegment` and `ParallelSegment` classes are abstract. The user can extend them to create its own custom types and use them in the selector.

The other category enables breaking the simple cyclic application of transformations from `mapNotes` . Each transformation is associated with parameters specifying its range of application and its period. The selected transformation for the n^{th} candidate is the first one in the list such that n is in the range and

$$(n - \text{range\_begin}) \mod \text{period} == 0$$

If there is a candidate that conforms to the selector but there is no transformation that meets the application constraints, identity is applied and this candidate is still considered as a leaf for the mapping.

For example, repeating all flat chords of a melody (parallel set of notes) is coded with :

```
val melody: MS = ...
melody mapIf (
  isPar given (_.height==1)
  thenDo (_ *2))
```

And if we need every third chord to be not repeated :

```
melody mapIf (
  isPar given (_.height==1)
  thenDo (
    identity,
    period = 3,
    from = 2))
  orDo (_ *2))
```

where `isPar` , `given` , `thenDo` and `orDo` are built-in features to hide the `TransformList` instantiation.

5 Case study

To get a good insight of the expressivity of the DSL, I took the piece : Recuerdos de la Alhambra, *Francisco Tarrega* as a case study. In this guitar piece, tremolo[2] composes the entire soprano melody. The rhythm is thus quite stable and the melody holds many repetitions.

In total, I used three different approaches to implement it with different results.

5.1 Redundancy abstraction

The aim of this approach is to avoid writing each repeated note using functions that generate those repetitions.

First, I separated the piece into four voices. This makes redundancy and similarities more visible in each voice. This case study is especially adapted to this approach as it presents the property of having lots of repeated notes (tremolo in soprano and repetitive bass). Repetitions building also specifies the rythmic pattern.

On a single voice, repetitions are encoded as `mapNotes` functions and when needed, different `mapNotes` are organized in a `mapIf` call to modify the repetitive pattern being used.

This approach avoids writing repeated notes and requires a single 'core' melody. As it also leaves the rythm specification to an annex function, writing the initial melody is very straightforward. This implementation is very flexible and can reproduce the original melody without any concerns.

However the base melody still has to be fully hand-written and this representation does not allow deeper analysis of the melody.

[2]the same note played many times (three in this case) very quickly

5.2 Pattern extraction

The pattern extraction approach is a response to the problem of the redundancy abstraction approach.

Build on top of the first implementation, it extracts the melodic patterns present in the core melodies of the previous approach. With a single reference note, the pattern generates a whole phrase. Hence each phrase is represented by a reference note and the voices are simple sequences of those referent.

Even though the final written melody is much smaller than in the previous case, this solution as many limits. Sticking to original melody produce over-fitted and very specific patterns. The final implementation uses some of them only once and do not produce any benefit. Others patterns have parameters specifying a small variation useful for specific cases and produce bad results in other context or with different values.

Furthermore, the harmonic dependencies between different lines becomes hidden by all the abstractions. The final melody appears to sound good because it is written so and modifying a parameter has uncontrolled effects.

5.3 Harmonic analysis

A deep analysis of the piece reveals that each phrase turns out to be written in a given tonality, not necessarily the main one of the piece. Occurrences of altered notes outside the original scale generate those transpositions.

Using this information, context is enough to generate many behaviors seen in other approaches as exceptions. This approach thus spits the piece into phrases and not into voices anymore. It extract the scale of the phrase and provides it as parameter for new phrase patterns.

To avoid the over-fitting problem from the pattern extraction approach, I ignored some exceptions to the rules, especially transitions between phrases. As the tonality inside a phrase is fixed and respected, the result of a standalone phrase is good. Unfortunately, the transition between the sequential phrases is not controlled anymore.

This makes the final approximation of the original melody pretty bad. But in exchange, randomly composing the different patterns with simple constraints on tonalities and chaining is easy and produces acceptable results.

6 Conclusion

According to this case study, the implemented features proved to be useful and filled their intended purpose. But it appears that support is missing for better representation capabilities.

Pattern abstraction can be realized using many different methods. It would need exploration and specification of standard procedures and would eventually provide a coding context as support to guide the abstraction in the right way. This would require performing more and simpler case studies, analysing the abstraction process and translating it as a language feature.

The pattern abstraction domain especially needs a clearer specification for rhythm description. For now, melody and rhythm as always been represented together. Separating them will enhance modularity and give better understanding of the melodies. In order to achieve this, the rhythm domain needs more investigations. Thoughful primitives dedicated to rhythm would be a great addition to the language.

Finally, tree structure is still hard to configure with custom types for the best use of the `mapIf` function. It is first necessary to achieve a good understanding of the tree structure behavior under recursive methods. Then it will be possible to add interfaces for those custom types to be taken into account by existing methods.