F♭ a Language for Music Composition

Joachim Tilsted Kristensen
tilsted@diku.dk

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science, University of Copenhagen
Universitetsparken 5, DK-2100 Copenhagen Ø, Denmark

Abstract

In this paper we introduce the declarative language F♭ for music composition. F♭ supports music composition at a higher level of abstraction than traditional sheet music. Composers are able to express themselves at a high level of abstraction throughout the entire process of composing a piece, and we sketch some of the available transformation of the various component of a composition. F♭ is currently still at a prototype stage (version 0.8), but we hope to evolve the language to be useful in many areas of computer aided music composition and analysis.

1 Motivation

Modern western music notation is organized in a way that a practiced musician is able to read and execute its contents simultaneously. As such, music sheets may be viewed as a machine language that happens to be interpreted at runtime, on an organic machine.

In this perspective, the music composer is a programmer who writes programs directly as low-level imperative instructions like *"play the high E quietly for one bar"* or *"wait 2 bars, then play an A-major chord"*. However, while the sheet music is at a rather low-level of abstraction the composer is more likely to reason about the composition at a higher level of abstraction. Like: *"It is a twelve-bar blues"*, or *"The theme is played twice, but in different keys"*.

Music theoretical concepts are often taught and understood separately, and even though some work has been done in the area of providing programming tools for music composition, we are not aware of any that take the approach of separating a *composition* into *harmony*, *rhythm* and *voices* treating them as different but connected concerns.

Regarding harmony, the "obvious" choice of primitives for formalizing music theory for a mathematician might be to describe a *pitch* by its absolute value in some sense (a frequency in Hz, or a note on the piano). However, *absolute* pitch is rather uncommon among practicing musicians and composers. In fact, most modern ear-training systems such as Solfège, prefer reasoning about pitches *relative* to the harmonic material in which they are evaluated (typically a scale or a chord).

In more harmonically involved genres, such as jazz, this harmonic material will change with each of the chords used to compose the music. Musicians communicate their choices when *playing over changes* by breaking down the way they picked this harmonic material for each chord[1].

[1]See `http://www.youtube.com/watch?v=xB1VnbKhgtg&t=4m0s`

So, for a music composition language to be useful in the whole process for a composing musician, we believe that scales, chords and other harmonic theories should be the primitives of harmony, and the way harmonic values are constructed should be expressed in terms of the materiel they are picked from.

In the early 80s, MIDI was invented as a means of communication between personal computers and external devices. A MIDI-file is structured much like a music sheet, it is a sequence of events/signals. MIDI is a standard found everywhere in music recording, production, sequencing and composition software. Thus, many music programs can import a MIDI-file and render it to anything from synthesized audio to engraved music sheets. The current version of F♭ [2] can already render compositions in to MIDI file. And we see a number of use-cases in which the composer writes an F♭ program that describes the way the music is composed from music theoretical constructs, and then render it into a MIDI-file that may be imported into various music programs.

2 F♭

The main goals for the current version of F♭ is that it should use notation and terms that are familiar to music composers, and we want to support and encourage the music composers to handle harmony, rhythm and voices as separate notions, that can be combined in different ways.

When it comes to harmony, we represent values in terms of abstract concepts such as the step of a scale. The idea is then to compose these abstract values in different contexts where their meanings become concrete. An example of such an abstract value looks like:

```
phrase abstract_phrase = {1 1 5 5 6 6 5}
```

where the keyword `phrase` states that the value `abstract_phrase` is an abstract musical phrase, and the brackets `{` and `}` denotes that *these numbers represent steps in a music scale.*

When talking about rhythms, we instead use the brackets `| · |` to denote that *these numbers represent reciprocal subdivisions of a measure.* Thus, a rhythm must always be declared by a meter. An example of such a rhythm in $\frac{4}{4}$, may be denoted as:

```
rhythm little_star  (4 / 4) 1 |: 4 4 4 4 | 4 4 2   :|
```

Now, to make the music concrete, the phrase must be part of a sequence, played by an instrument in some key. We

call such a sequence a `voice`, and an example could look like:

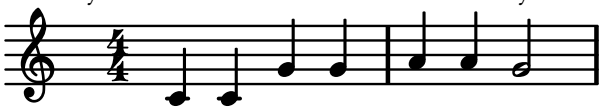
```
midi  piano = 0, 127
voice twinkle_twinkle piano {C} ionian {abstract_phrase}
```

Where the keyword `midi` denotes that *"this value is a pair of bytes, that define an instrument and its initial volume in the general MIDI standard"*, The keyword `voice` is used to declare a voice. In this case, we declare the voice `twinkle_twinkle` , played by piano, in the key of C-ionian (major). playing the abstract musical phrase that we declared earlier. Since it is so short, we could have just typed in the melody directly, but in larger examples it makes sense to structure voices in phrases and/or chords, there are some lager examples available in the github page[2]. The keyword `C` refers to the middle C in standard western music theory. There are 7 such keywords named `A, B, C, D, E, F` and `G`. Other music notes, may be reached by applying operators to them. For instance, the postfix operator "+" brings a note up one octave, "−" moves a note down one octave, and "#" and "b" move notes up and down by semitones, All operators apply to scalesteps as well as concrete notes.

To put together a voice and a rhythm we use the with-operator ("<-"), as an example, we might want to play the above declared voice `twinkle_twinkle`, with the rhythm `little_star`

```
composition example0 = twinkle_twinkle <- little_star
```

Which then evaluates to some MIDI-music which we could similarly have been declared in a music-sheet by:

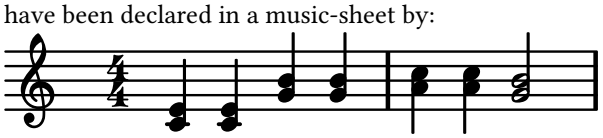


At first, this way of composing music seems abstract and complicated in comparison with libraries such as Jython-Music, JMusic or Euterpea, while also being less concise than markup-language counterparts such as LilyPond or Alda. However, it has the advantage, that we may later manipulate the meaning of the music in ways that were not possible in either. For instance, we can change the key from a major-key to a minor key without having to worry about the distance inbetween notes. That is, in C-major the third step is an E, and in $D_\flat$-minor the third step is also E, but the first step in the two scales are different. So, the transformation of changing keys is not as trivial as that of transposing a piece into a different tonality, but in this language both transformations can be done effortlessly.

As an example, we might want to add another voice to our example melody, playing the same thing, but one third above the original melody. This corresponds to playing the changing the key to E-phrygian (minor):

```
composition example1 =
   ({E} phrygian twinkle_twinkle <- little_star ) +
                 twinkle_twinkle <- little_star
```

Which evaluates to some MIDI-music which could similarly have been declared in a music-sheet by:



One last thing to note, is that since the values that the above piece are constructed from all have clear semantics, the errors can become quite helpful. As an example, if we had declared the rhythm above as

```
rhythm little_star2  (4 / 4) 1 |: 4 4 4 4 | 4 2 :|
```

The current implementation returns an error:

```
The rhythm 'little_star2' does not satisfy the time
signature '4/4', as it adds up to '3/4' in measure 2.
```

In the future, we would like to aid the user with error messages, perhaps such as the ones suggested by [3].

3 Near Future and Related Work

There are other DSLs for music composition, notably Haskore and Euterpea[1]. However, the primitives of Euterpea is at the same level of abstraction as sheet music, and if you want to work with higher level notions you have to build those yourself in Haskell (which has both pros and cons). While it may be possible to make $F_\flat$ an embedded DSL (EDSL) build on top of Euterpea. We have currently elected not to do so, because the music composers we have talked to, have been reluctant to pay the barrier of entry to install and learn Haskell to work with Euterpea.

In modern music theory, one of the voices is often a melody, and the other voices depend not only on the melody, but on a sequence of chords, and a relationship between those chords. We plan to try to support this dependency, either by declaring voices in terms of chords and other voices, or by introducing new constructs that relates notes to the other notes sounding at the same time.

Another related language Mezzo[3] uses Haskell's type system to check for rules of classical western music voicing. We would also like to support user provided rules. Likewise, in the current version of $F_\flat$ some music theoretical constructs, such as the diatonic scales, are built in keywords. Such fundamental constructs should be primitives that the user can define themselves. We don't see any reason why pitch should be limited only to the semitones of western music. However we plan to provide western musical theory the current version of $F_\flat$ supports as a library.

References

[1] P. Hudak and D. Quick. *The Haskell School of Music: From Signals to Symphonies*. Cambridge University Press, Sept. 2018.

[2] J. T. Kristensen. Fb v.0.8. `https://github.com/FurryRogue/f-flat`.

[3] D. Szamozvancev and M. B. Gale. Well-typed music does not sound wrong (experience report). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, pages 99–104, New York, NY, USA, 2017. ACM.