Demo: Composable Compositions with Tonart

Jared Gentner
jagen315@gmail.com

Abstract

This demo introduces Tonart, a language and metalanguage for practical music composition. The object language of Tonart is abstract syntax modeling a traditional musical score. It is extensible- composers choose or invent syntaxes which will most effectively express the music they intend to write. Composition proceeds by embedding terms of the chosen syntaxes into a coordinate system that corresponds to the structure of a physical score. Tonart can easily be written by hand, as existing scores are a concrete syntax for Tonart. The metalanguage of Tonart provides a means of compiling Tonart scores via sequences of rewrites. Tonart's rewrites leverage context-sensitivity and locality, modeling how notations interact on traditional scores. Using metaprogramming, a composer can compile a Tonart score with unfamiliar syntax into any number of performable scores.

In this demo, we will make a small composition using Tonart. We will construct this composition by manipulating notations representing abstract music objects. These will eventually be compiled into a digital score representation, as well as a computer performance. We will add in an especially abstract object at the end, and use our creativity to compile it into something performable.

ACM Reference Format:
Jared Gentner. 2024. Demo: Composable Compositions with Tonart. In *Proceedings of FARM '24.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Demo

1.1 Introduction

This demo is intended to show how Tonart can be used for practical composition with a score containing abstract music objects. Notation softwares such as MuseScore[1] offer a score interface and many surface level transformations over notes and attributes of notes. However, they fail to provide operations over objects like chords, scale degrees, and voice leadings. Existing computer music libraries such as Euterpea [1] provide these operations or the means to build them, but do not provide them within a score-like interface. A score-based DSL is fundamental to leveraging existing knowledge and skills with score notation, a centuries old medium.

[1]https://musescore.org/en

FARM '24, September 2, 2024, Milan, Italy
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.2 Tonart Syntax

Below is the base Tonart syntax. The undefined nonterminals are extension points which will be elaborated as we progress through the demo.

```
⟨form⟩     ::= ⟨art-id⟩
            |  ⟨object⟩
            |  ⟨rewriter⟩
            |  ⟨context⟩
            |  ( @ ( ⟨coord⟩* ) ) ⟨form⟩* )
⟨program⟩  ::= ( define-art ⟨art-id⟩ ⟨form⟩* )
            |  ( realize ⟨realizer⟩ ⟨form⟩* )
```

A *context* is a coordinate structure. Tonart's primary coordinate structure is called `music`.

```
⟨context⟩ ::= ( music ⟨<form>⟩* )
```

Music has two coordinates,

```
⟨coordinate⟩ ::= ( interval ( ⟨number⟩ ⟨number⟩ ) )
              |  ( voice ⟨id⟩* )
```

which are orthogonal and represent the horizontal (time) and vertical (voice) dimensions of a physical score.

The @ form is used to embed objects into a context at given coordinates.

Tonart is compiled into Racket[2] by *realizers.*

Note that we will not actually write out the `music` form in this demo, as the realizers we are using treat their toplevel forms as music implicitly.

1.3 Composing In Tonart

We will begin composing with only one object and one realizer.

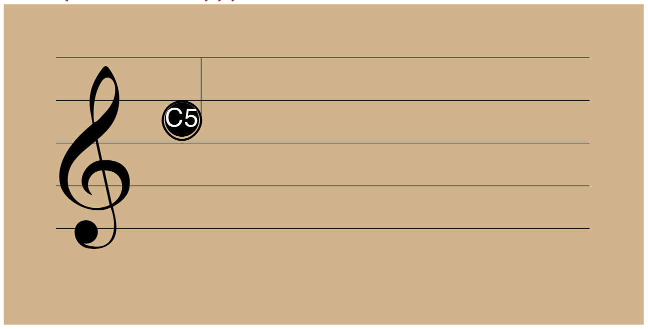
```
⟨object⟩   ::= ( note ⟨pitch⟩ ⟨accidental⟩ ⟨octave⟩ )
```

[2]Tonart is written as an embedded DSL in Racket. It is a syntactic abstraction, running entirely at compile time, utilizing Racket's module system to implement its own extension mechanisms and libraries. [2]

```
⟨realizer⟩ ::= ( staff-realizer
                 ( ⟨number⟩ ⟨number⟩ ) )
```

——

```
(realize (staff-realizer [300 150])
  (@ [(interval [0 4]) (voice soprano)]
    (note c 0 5)))
```



This is a note called C5, or, C in the fifth octave. It is sung by the soprano voice, for four beats.

To play this note from the computer, we will convert it into a frequency. A frequency will be represented by the tone object. To turn notes into tones, we will use a straightforward rewriter called note->tone. Tonart rewriters are not composed into the context like objects; instead, they transform the context by adding, deleting, and modifying existing objects.

```
⟨object⟩   ::= ...
             |  ( tone ⟨frequency⟩ )
⟨rewriter⟩ ::= ( note->tone )
⟨realizer⟩ ::= ...
             |  ( sound-realizer )
```

——

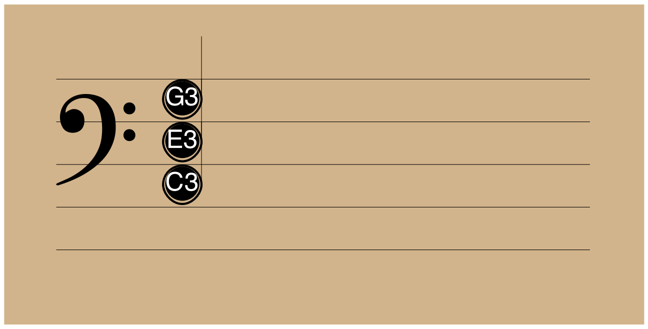
```
(realize (sound-realizer)
  (@ [(interval [0 4]) (voice soprano)]
    (note c 0 6))
  (note->tone))
```

Now we will add a harmony to this note. We will express the harmony as a chord.

```
⟨object⟩   ::= ...
             |  ( chord ⟨pitch⟩ ⟨accidental⟩
                  ( ⟨quality⟩ ) )
⟨rewriter⟩ ::= ...
             |  ( chord->notes ⟨octave⟩ )
```

——

```
(realize (staff-realizer [300 150])
  (@ [(interval [0 4]) (voice accomp)]
    (chord c 0 [M])
    (chord->notes 3)))
```



We have not yet discussed putting objects one after another in time. We could of course use consecutive intervals. However, this gets unwieldy. We are instead going to establish a concept of a *sequence* of notes.

```
⟨context⟩    ::= ...
              |  ( seq ⟨form⟩* )
⟨coordinate⟩ ::= ...
              |  ( index ⟨number⟩* )
```

We define a new context. This context is called seq and has one coordinate, index, representing the position of an object in the context. seq contexts can be embedded in music contexts, allowing us to express an ordered sequence directly in a score, without giving specific lengths to the notes or other objects it contains.
Next, we define syntax for rhythms, which are, for our purposes, a series of consecutive durations.

```
⟨object⟩   ::= ...
             |  ( rhythm ⟨number⟩* )
⟨rewriter⟩ ::= ...
             |  ( apply-rhythm )
```

Now we can do something more complex with the soprano.

Note: Instead of writing,

```
(seq
  (@ [(index 0)] (note a 0 3))
  (@ [(index 1)] (note b 0 3))
  ...)
```

I will write (seq (notes [a 0 3] [b 0 3] ...)).

Below, a melody, in the soprano, is bound to melody. The definition does not rewrite with apply-rhythm immediately. That is fine, as we will apply the rhythm at the end. It is often best in Tonart to express music declaratively, by writing down all objects first and saving the rewriting for the end.

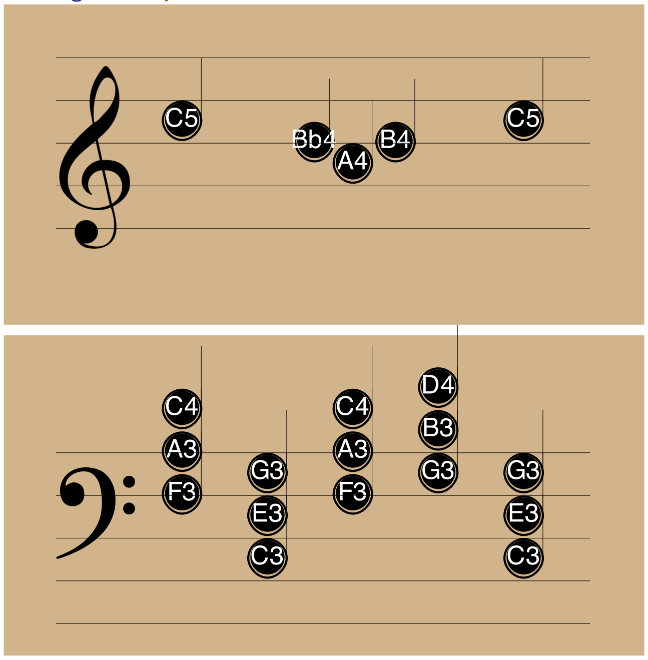
```
(define-art melody
  (@ [(voice soprano)]
    (seq (notes [c 0 5] [b -1 4] [a 0 4]
                [b 0 4] [c 0 5]))
    (rhythm 3/2 1/2 1/2 3/2 2)))
```

Now, we supply a harmony and harmonic rhythm, which the accompaniment will outline.

```
(define-art harmony
    (seq (chords [f 0 M] [c 0 M] [f 0 M]
                 [g 0 M] [c 0 M]))
    (rhythm 1 1 1 1 2))
(define-art accomp
  (@ [(voice accomp)]
    harmony))
```

A very nice cadence. To see it visualized:

```
(define-art song-notes
  melody accomp
  (apply-rhythm) (chord->notes 3))
(realize
  (staff-realizer [300 300])
  song-notes)
```



To hear it:

```
(realize (sound-realizer)
  song-notes (note->tone))
```

1.4 Finale

To finish off, we will try adding a more obscure object to our composition.

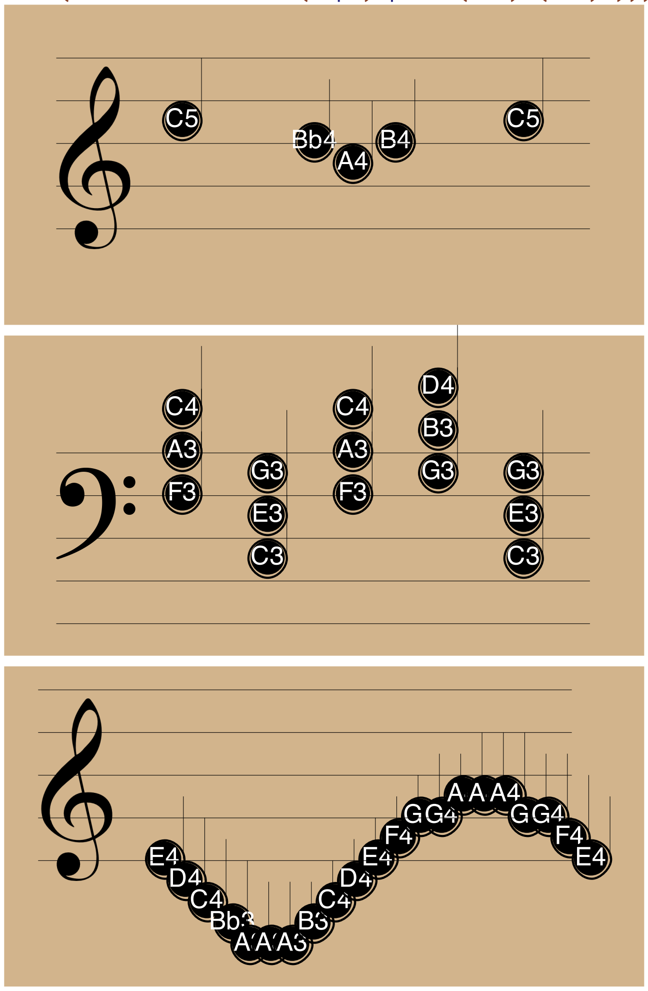
```
⟨object⟩   ::= ...
            |  ( function ( ⟨id⟩ ) ⟨expr⟩ )
⟨rewriter⟩ ::= ...
            |  ( function->notes
                 ( ⟨number⟩ ⟨number⟩ )
                 ( ⟨note⟩ ⟨note⟩ ) )
```

`function` is a mathematical function.

`function->notes` applies to functions, and it creates a melody that fits within the surrounding harmony and matches the contour of the function.

Here is the finished work. I will use `(uniform-rhythm 1/4)` as a shorthand for `(rhythm 1/4 1/4 1/4 ...)`. The function is $sin(x)$ over $(-\pi, \pi)$.

```
(realize (staff-realizer [300 450])
  song-notes
  (@ [(voice countermelody)]
    harmony (apply-rhythm)
    (@ [(interval [0 5])]
      (function (x) (sin x))
      (uniform-rhythm 1/4))
    (@ [(interval [5 6])]
      (note e 0 4))
    (function->notes [(- pi) pi] [(a 3) (a 4)])))
```



References

[1] P. Hudak. Euterpea. 2014. http://euterpea.com

[2] Flatt, Matthew. Composable and Compilable Macros: You Want it When? In *Proc. ACM Intl. Conf. Functional Programming*, pp. 72–83, 2002.