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 build a Tonart by adding standard music objects and rewriters one by one. We will compile all objects into a digital score representation, as well as a computer performance. We will add in a surprise 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

We will begin this demo with a quick look at Tonart's syntax. After that we will immediately start composing. Tonart is extensible at several points. To reflect this, we'll build our demo Tonart syntax a few forms at a time, writing small examples as we go, in each language.

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

We begin with a 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.

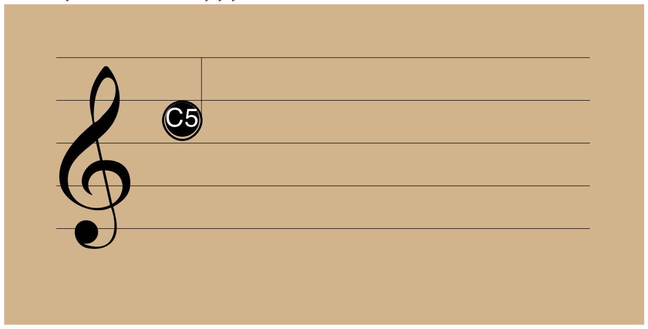
1.3 Composing In Tonart

Tonart is compiled into Racket by *realizers*. We will begin composing with only one object and one realizer.

```
⟨object⟩   ::= ( note ⟨pitch⟩ ⟨accidental⟩ ⟨octave⟩ )
⟨realizer⟩ ::= ( staff-realizer ⟨clef⟩ )
```

——

```
(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 put 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 )
```

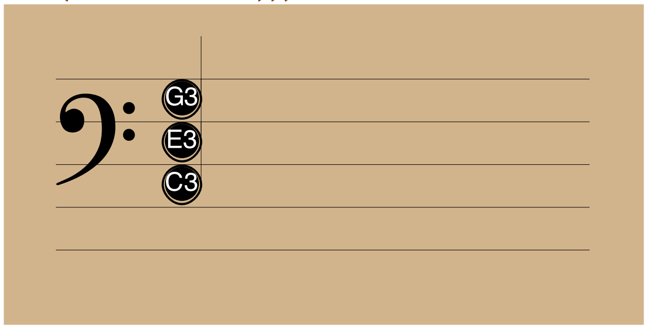
```
(realize (sound-realizer)
  (@ [(interval [0 4]) (voice soprano)]
    (note c 0 6))
  (note->tone))
```

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 ⟨number⟩* )
⟨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 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 use apply-rhythm immediately. That is fine, as we will apply the rhythm at the end. It is often best in Tonart to express the music by writing down all objects first and saving the rewriting for the end.

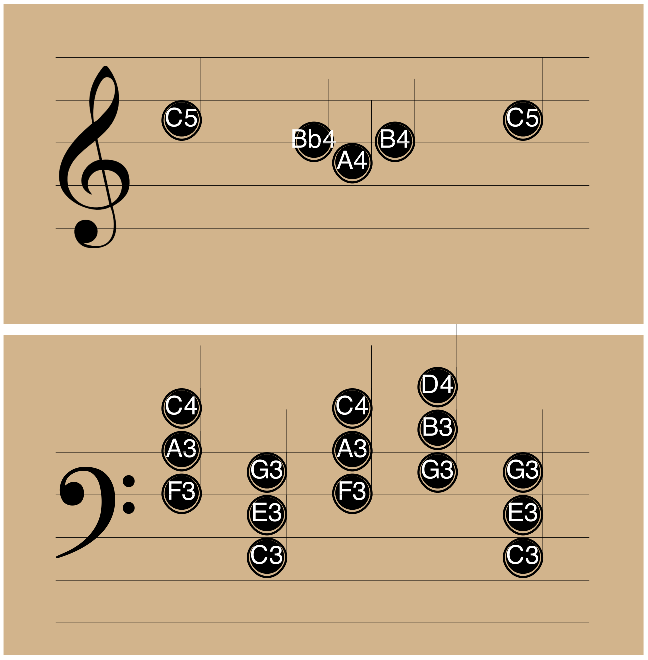
```
(define-art melody
  (@ [(voice soprano)]
    (seq (notes [c 0 5] [b -1 4] [a 0 4]
                [b 0 4] [c 0 5]))
    (rhythm 3/2 1/2 1/2 3/2 2)))
```

Now, we supply a harmony.

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

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

Now we will try compiling a piece with a more obscure object.

```
⟨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 an example. I will use `(uniform-rhythm 1/4)` as a shorthand for `(rhythm 1/4 1/4 1/4 ...)`. The function is $sin(x)$ over $(-\pi, \pi)$.

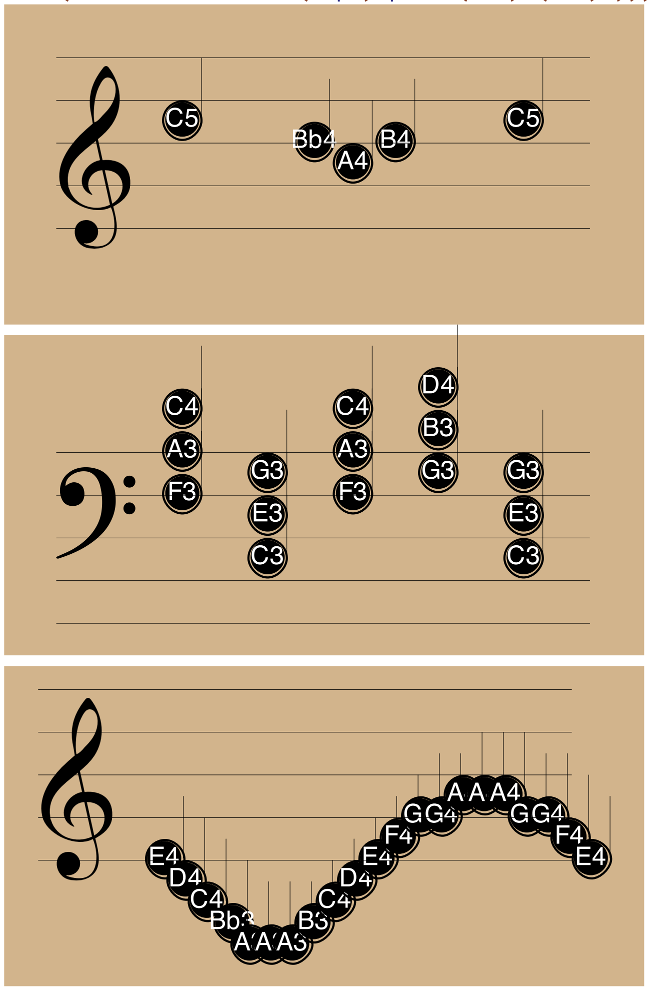
```
(realize (staff-realizer [300 450])
  song-notes

  (@ [(voice countermelody)]

    harmony (rhythm 1 1 1 1 2)
    (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)])))
```


C5
Bb4
A4
B4
C5
C4
A3
F3
G3
E3
C3
C4
A3
F3
D4
B3
G3
G3
E3
C3
E4
D4
C4
Bb3
A A A3
B3
C4
D4
E4
F4
G G4
A A A4
G G4
F4
E4