Demo: A Geometric Approach to Generate Musical Rhythmic Patterns in Haskell

Xavier Góngora
Universidad Nacional Autónoma de México
Mexico City, Mexico
xavier.gongora@comunidad.unam.mx

Abstract

We present work-in-progress on RTG, a domain specific language embedded in Haskell designed to explore the affordances of geometry as a means to generate and manipulate rhythmic patterns in live coded music. Examples of how simple geometry is capable of producing interesting rhythms are shown to support our use of binary lists as a pattern representation. We introduce Erlangen's Program notion of geometry as encoded in *groups*, using such structure as the focus of a combinator interface based on an archetypal `RhythmicPattern` type implemented using a type class. Examples of the interface usage are provided. Future work targets the definition of `Group` instances for the *rhythmic pattern types* such that the *group* laws are fulfilled and its operations lift to the interface in a musically coherent and engaging way.

CCS Concepts: • **Applied computing** → **Sound and music computing**; • **Software and its engineering** → *Domain specific languages.*

Keywords: geometry, rhythm, music, pattern, group theory, functional programming, Erlangen program, live coding

ACM Reference Format:
Xavier Góngora. 2024. Demo: A Geometric Approach to Generate Musical Rhythmic Patterns in Haskell. In *Proceedings of the 12th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design (FARM '24), September 2, 2024, Milan, Italy.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3677996.3678295

1 Introduction

In previous work [5] we showcased design criteria for RTG:[1] a live coding library for the generation and manipulation of rhythm, influenced by the Tidal Cycles language.[2] Its central idea is the definition of *group structures* for rhythmic pattern types (*i.e.* rhythm families defined by computational and geometric considerations, represented using algebraic data types), so that rhythms can *transform* one another.

Our canonical rhythmic pattern type is that of Euclidean rhythms, shown briefly in section 2. Rhythmic patterns are considered language primitives as values of such types. The goal is to develop a concise and terse domain specific language such that the combination and transformation of rhythms is done within musically significant geometric constraints that afford effective exploration strategies during a live coding performance. To the best of our knowledge this is a novel approach to rhythm manipulation which is yet to prove itself before an audience.

Studies of rhythm's geometric properties are usually intended for music theoretic analysis or music information retrieval. In contrast, our approach is led by the intuition that rhythm itself is a geometric aspect of time and the search for the creative potential of such an aspect. This made us speculate about an inner *geometry* of rhythm, hidden in its *group of transformations*. We discuss *group* structure and the Erlangen Program as the origin of our concept of geometry in section 5.

The current prototype is build using lists and abstracts musical rhythm to its bare bones. We motivate this simplification in section 2 by showing that interesting structure persists in this setting. Next, in sections 3 and 4, we use the Haskell's type system to implement a shared interface for distinct *group operators* that includes some basic combinators.[3] Afterwards, in section 6, we provide examples of the interface use.

2 What about Geometry in Rhythm?

Rhythm is a multidimensional phenomenon, too complex to account for in any single framework. In all its generality, it could be described as a natural law underlying all dimensions of being. In the case of music, aspects like volume, timbre and cultural context underlie our perception (or induction) of rhythm [2]. What is certain, and a useful approach towards

[1] The library's source code repository: https://github.com/ninioArtillero/RTG

FARM '24, September 2, 2024, Milan, Italy

ACM ISBN 979-8-4007-1099-5/24/09
https://doi.org/10.1145/3677996.3678295

[2] Tidal Cycles oficial website: https://tidalcycles.org/

[3] We use the terms *operator* and *operation* in an interchangeable fashion, as in many contexts they are equivalent. The former is more common in programming, while the later in arithmetic.

its computational abstraction, is that rhythm is fundamentally bound to the arrangement of sound events in time. This arrangement alone produces salient perceptual structures, most notably meter and rhythmic grouping [2, 13], which seem to correlate with geometric properties.

Toussaint's [2020] working hypothesis is that geometric properties of rhythms translate to musical qualities that consistently make them being perceived as *good*, evidenced by their repeated appearance among diverse cultural contexts and musical styles. His paradigmatic example is the *Son Clave* found in music genres related to the african diaspora (its name comes from cuban music). It is a 5 onset pattern in a 16 isochronous (*i.e.* of the same time duration) pulse measure. We can represent this pattern as a list, where "1" stands for a sound *onset* and "0" for a *rest*.[4]

```
son = [1,0,0,1,0,0,1,0,0,0,1,0,1,0,0,0]
```

An important family of pattern examples are Euclidean rhythms. These are constructed using Björklund's algorithm, which is equivalent to Euclid's algorithm for computing the maximum common divisor of a pair of integers. Geometrically, the `(k,n)` Euclidean rhythm can be described as the configuration that maximizes the evenness of `k` ones among `n-k` zeros in a binary pattern [12]. For example, the Euclidean rhythm denoted by `(7,12)` is a common West African bell pattern [12].

```
e (7,12) = [1,0,1,1,0,1,0,1,1,0,1,0]
```

Another interesting example discussed in Toussaint [2020] is that of the rhythmic pattern used by Steve Reich in his piece "Clapping Music". Out of the 495 possible 8 onset in 12 pulses measure patterns, this pattern can be selected by means of regularity and diversity constraints, like allowing only one-pulse rests and excluding cyclic permutations. A key constraint, which is relevant to the structure of "Clapping Music", is that the pattern doesn't align with itself while rotating it step-wise until it reaches the starting position.[5] This pattern happens to be a *rotation* of another one obtained as the result of superposing the three clapping patterns performed in the Beer Dance of the Lala people [13].

```
clap1     = [1,0,0,0,1,0,0,0,1,0,0,0]
clap2     = [1,0,1,0,1,0,0,1,0,1,0,0]
clap3     = [1,0,1,0,1,1,0,1,0,1,0,1]

beerDance = [1,0,1,0,1,1,0,1,1,1,0,1]

reichP    = [1,1,1,0,1,1,0,1,0,1,1,0]
```

[4]The terms *onset* and *offset* are standard in the theory of music, and refer to the start and end (respectively) of a sound event associated with a specific source. As part of the simplicity of our approach, we do not consider offsets.
[5]Steve Reich's "Clapping Music" starts with two performers clapping this pattern in unison. One of them will carry on this way the whole piece. After every twelfth repetition, the other performer shifts its pattern by one step (rotation). The piece ends after both performers play in unison again.

Incidentally, "Clapping Music" is a good fit for naming the domain of rhythm the present work focuses on. Clapping is commonly used when communicating a rhythmic pattern. This might be related to its short and fast decaying envelope resembling a time point in perception. These examples show that engaging clapping music can be generated out of a simple geometric procedure such as the Euclidean algorithm or by the operation of superposition that underlies polyrhythms.[6]

3 Representing Rhythm

Our model of rhythm is a sequence of isochronous sound onsets and rests called a `RhythmicPattern`. We implement this in Haskell by first defining a `Binary` data type to represent onsets and rests.

```
data Binary = Onset | Rest
  deriving (Eq, Ord)

instance Show Binary where
  show Rest  = show 0
  show Onset = show 1

instance Semigroup Binary where
  Rest  <> Onset = Onset
  Onset <> Rest  = Onset
  _     <> _     = Rest

instance Monoid Binary where
  mempty = Rest

instance Group Binary where
  invert = id
```

Listing 1. A binary data type representing sound onsets and rests

We will discuss groups with some detail in section 5, but for the time being we note that the `Group` instance provided for the `Binary` type is isomorphic to the integers modulo 2 (which is the only group structure possible for a two value type).[7] We'll use this type in the definition of our archetypal rhythmic pattern type to have a standard way of combining onsets and rests. Now we define `Rhythm` as a `newtype` wrapper around `Pattern`, which is a type synonym for lists.

```
type Pattern a = [a]

newtype Rhythm a =
```

[6]It might be argued that these properties are arithmetic rather than geometric. Nevertheless, both fields of mathematics are intricately related to each other [11], so for us this is a matter of appreciation and not an assertion on mathematical classification.
[7]The `Group` type class is exported by the `Data.Group` module from the `groups` package. https://hackage.haskell.org/package/groups

```
  Rhythm { getRhythm :: Pattern a }

instance Functor Rhythm where
  fmap f (Rhythm xs) = Rhythm (fmap f xs)

instance Applicative Rhythm where
  pure xs = Rhythm $ pure xs
  Rhythm fs <*> Rhythm xs =
    Rhythm (zipWith ($) fs xs)

instance Semigroup a =>
  Semigroup (Rhythm a) where
  Rhythm pttrn1 <> Rhythm pttrn2 =
    Rhythm $ pttrn1 `EuclideanZip` pttrn2

instance (Semigroup a, Monoid a)
  => Monoid (Rhythm a) where
  mempty = Rhythm $ repeat mempty

instance (Semigroup a, Monoid a, Group a)
  => Group (Rhythm a) where
  invert = fmap invert
```

Listing 2. Rhythm defined as a newtype wrapper around the pattern type

The previous declarations are pretty straight forward, but the `Semigroup` instance has some crucial consequences for us, so we look into it. The constraint there is caused by the `EuclideanZip` function. It works by zipping the `k` elements of the smaller list as evenly as possible among the `n` elements of the bigger list, combining then with the `Semigroup` operator, according to the Euclidean pattern `(k,n)` (see section 2). In the case when both lists have the same length, it simply does **zipWith** (<>) pttrn1 pttrn2.

This is not the only operation possible. Indeed, from Haskell's type system point of view, we could implement it with any operation imaginable as long as it fulfills the corresponding function types (signatures). Nevertheless, one design strategy is looking for simplicity and symmetry [5]. That such an operation is a good fit is to be decided by its fulfillment of group laws and its (musical) performance.

4 The Interface

Now we are in a position to display the full interface to manipulate rhythmic patterns.

```
type RhythmicPattern = Rhythm Binary

class Semigroup a => Rhythmic a where

  -- Minimal complete definition.
  toRhythm :: a -> RhythmicPattern

  -- Group operator lifting.
  (&) :: Rhythmic b =>
    a -> b -> RhythmicPattern
  x & y = toRhythm x <> toRhythm y

  (!&) :: a -> a -> RhythmicPattern
  x !& y = toRhythm (x <> y)

  -- Basic combinators.
  inv  :: a -> RhythmicPattern
  co   :: a -> RhythmicPattern
  rev  :: a -> RhythmicPattern
  (|>)  :: Rhythmic b =>
    a -> b -> RhythmicPattern
  (<+>) :: Rhythmic b =>
    a -> b -> RhythmicPattern
```

Listing 3. The main rhythmic pattern interface

The default implementation of the basic combinators is not shown. They are group inversion, complement, reverse, sequencing and superposition, respectively. We have also provided specifications for this functions as formal properties verifiable by `QuickCheck`, but are omitted here. Formal verification, by proof or property based testing, is important to assert a group structure.

The group lifting functions combine patterns using *group* operations. As shown by their default implementations, values of any two rhythmic types can be combined with the `RhythmicPattern` group operation after lifting the values using the `&` operator. The `!&` operator combines the elements of the same type using their group operation before lifting the result. They don´t necessarily match, as the corresponding group operations can be different. When group operators are curried, each element of a group can be seen as a *transformation* for elements of the same group. Also, transformations of objects and spaces form groups, and this simple fact take us near the realm of geometry.

5 Groups and Geometry

Our focus on the search, definition and use of group operations for rhythmic pattern is founded on speculative possibilities informed by the Erlangen Program. In this section we introduce it along with the laws defining the (algebraic) group structure.

The Erlangen Program is a research program proposed by Felix Klein at the end of the nineteen century, which ultimately asserts the study of geometric structure and properties is equivalent to that of the invariants under a *group of transformations* [14].

As an example, a polygon's enclosing area, perimeter and number or vertices are invariants of the group of isometric

movements of the plane (any combination of translations, reflections and rotation). On the other hand, a regular polygon on a specific configuration, for instance a triangle centered in the coordinate origin on the plane, is invariant only under a *subgroup* of the group of isometries of the whole plane.

In this context, a *group* is defined as a set with an associative binary operation (here represented polymorphically by `<>`), having a unique identity element and a unique inverse for each element. In terms of Haskell's standard type classes, a group is a type with a `Subgroup` and `Monoid` instances, and also a `Group` instance implementing the invert function, fulfilling the following laws (*i.e.* all expressions evaluate to **True** for any elements x, y and z of the group):

```
(x <> y) <> z == x <> (y <> z)
mempty <> x == x
x <> mempty == x
invert x <> x == mempty
x <> invert x == mempty
```

Listing 4. Groups laws as Haskell boolean expressions

As simple as they are, the group laws encode the mathematical notion of symmetry [14]. They tell us that property preserving transformations compose and they can always be undone by a corresponding (backwards) transformation. At the time of Erlangen's Program publication [7], geometry was blooming into a diverse subject with the emergence of non-Euclidean geometries, differential geometry and projective geometry. Perhaps this contrasted sharply with the previous monolithic classical geometry (that of Euclid's) and a unifying framework was called forth. The Erlangen Program today is perhaps an artifact in the history of mathematics, but it underlies many ground-breaking developments in the theoretical physics of the twentieth centry, such as Kaluza-Klein theory, Yang-Mills theory, string theory and non-commutative geometry [6].

6 Usage and Examples

Euclidean rhythms (see section 2), onset binary patterns (the archetypal RhythmicPattern type, as defined in listing 3) and *time patterns* are our starting cases of rhythmic pattern types. Time patterns are a collection of prescribed patterns, based on equal-tempered scales and popular rhythmic *timelines* [13].[8]

```
diatonic :: Pattern Time
diatonic = [0/12, 2/12, 4/12, 5/12,
            7/12, 9/12, 11/12]

rumba :: Pattern Time
rumba = [0/16, 3/16, 7/16, 10/16, 12/16]
```

[8]Timelines are short *ostinatos* that contextualize the other rhythmic and melodic elements in a piece within a given style of music.

The elements of these patterns are rationals numbers normalized to range from 0 to 1 non-inclusive, as 1 matches the beginning of the next *cycle*. They are eventually converted to the binary representation for playback (as all rhythmic pattern types).

Rhythmic patterns are played in a cyclic fashion, repeated indefinitely during a performance, and their playback *tempo* is derived from a variable that defines how many times it is played per second.

The group operations `<>` of the three types are implemented as follows: As time patterns are just type synonyms, we use the standard list append as operation. In the binary representation this operation lifts to the superposition of the patterns. Euclidean rhythms use modular arithmetic on their tuple components. With its current operation the Euclidean type has the following Cartesian product as denotation: $\mathbb{Z}_n \times \{n\} \times \mathbb{Z}_n$ for all $n \in \mathbb{N}$.

Elements of both types can be lifted to a RhythmicPattern using the toRhythm function, and are combined using the euclideanZip function described at the end of section 3. Space constraints prevent us from showing the implementations or detailed examples of the individual operations.[9]

What we have seen so far allows us to express the combination of scales and rhythms.

```
playR $ diatonic & rumba & diminished
-- [1,0,0,1,1,0,1,1,1,0,0,1,1,0,0,0]
```

The playR function implements rhythm playback. It is currently a quick and dirty solution using the SuperDirt sound engine and a clap sample.[10] Euclidean rhythms can be thrown into the mix.

```
playR $ e(3,8) & e(13,33) & son
-- [1,0,0,1,0,1,1,0,1,0,1,0,0,1,0,
--   1,0,0,1,0,0,0,0,1,0,1,0,0,1]

playR $ e(3,8) !& e(13,33) & son
-- too long to display
```

When two patterns are combined with the & operator, the RhythmicPattern operation is used, while !& uses the specific rhythmic pattern type operation (so both patterns must be of the same type). In other words, we have means to choose the operation to use: each type's own group operation or the RhythmicPattern one. Is also of note that these operators substitute the direct use of `<>`. This is an elegant way of experimenting with different combinators, thinking of rhythmic patterns both as language primitives and group elements leveraging ad-hoc polymorphism.

[9]The current implementation of the operations described can be found at this point in the source code history: https://github.com/ninioArtillero/RTG/tree/a3f90bb70105628c6eca17cf35c09b9ad7f28951

[10]SuperDirt source code repository: https://github.com/musikinformatik/SuperDirt

7 Discussion

Future work is aimed at testing and comparing alternative representations of rhythmic patterns as the group operations proposed so far have failed to fulfill group laws. We have focused on a variety of zip-like operations differing in the way list elements are matched when list lengths are not equal. Zip-like operations are natural candidates considering rhythm is "temporal media". As pointed out by Apfelmus [2019], the other natural applicative structure for lists (associated with the standard monad instance) is analogous to a product which might not be suitable for temporal media. This could be signaling that either a relaxation of the group axioms is called forth or that a different representation is needed. Both cases would reveal an actual fact about the structure of rhythmic patterns.

Also, rhythmic structure has the peculiarity of being a second order pattern in the sense that the insertion of a single onset changes the whole structure [10].

From this considerations, the implementation of the rhythmic pattern representation as a binary list might need to be enriched to carry more structural information. Maybe a state monad to keep track of context information such as rhythmic grouping (or clustering, to avoid confusion) and meter, to bind it through exotic pattern combinators. Another approach that has proven musically robust is Tidal Cycles representation of patterns [8, 9] based on the Functional Reactive Programming idea of a "behavior" as a function of continuous time [4]. This approach has had a long life cycle and has been subject to many refinements. Leveraging some of this work might be of great value to our task.

8 Conclusions

We exposed the rhythmic pattern representation that sits at the core of RTG's prototype and showed some examples of its intended use. The current aim is to provide a new way of combining rhythmic patterns for live performance using group structures.

During development the abstraction power of standard type classes, like `Functor` and `Applicative`, play the role of a gravitational field guiding design decisions and implementation strategies. Along our way, we keep finding clues and intuitions supporting our search for the geometric structures of rhythmic patterns. Existing limitations and inconsistencies might be clues of where the inner geometry of time and rhythm lies. Perhaps we need "some deeper structure, going far beyond the parts, points, local neighborhoods... and fragmented classical geometrical views in general" [3]. For certain, geometry is a field of mathematics that stills holds promise for the domain of music creation.

Acknowledgments

We thank Roberto Velasco, Juan S. Lach, and the anonymous reviewers for their valuable feedback. This ongoing research is supported by the *Consejo Nacional de Humanidades, Ciencias y Tecnologías* (Conahcyt) as part of my PhD studies under the supervision of Hugo Solís at the *Universidad Nacional Autónoma de México* (UNAM).

References

[1] Heinrich Apfelmus. 2019. Demo: Functors and Music. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design*. ACM, Berlin Germany, 52–55. https://doi.org/10.1145/3331543.3342582

[2] Eric F. Clarke. 1999. Rhythm and Timing in Music. In *The Psychology of Music (Second Edition)*, Diana Deutsch (Ed.). Academic Press, San Diego, 473–500. https://doi.org/10.1016/B978-012213564-4/50014-7

[3] Micho Durdevich. 2017. Music of Quantum Circles. In *The Musical-Mathematical Mind: Patterns and Transformations*, Gabriel Pareyon, Silvia Pina-Romero, Octavio A. Agustín-Aquino, and Emilio Lluis-Puebla (Eds.). Springer International Publishing, Cham, 99–110. https://doi.org/10.1007/978-3-319-47337-6

[4] Conal Elliott. 2009. Push-Pull Functional Reactive Programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1596638.1596643

[5] Xavier Góngora. 2023. Rhythm, Time and Geometry. In *Algorithmic Pattern Salon*. Then Try This. https://doi.org/10.21428/108765d1.e65cd604

[6] Lizhen Ji and Athanase Papadopoulos (Eds.). 2015. *Sophus Lie and Felix Klein: The Erlangen Program and Its Impact in Mathematics and Physics*. Number 23 in IRMA Lectures in Mathematics and Theoretical Physics. European Mathematical Society, Zürich.

[7] Felix Klein. 1983. A Comparative Review of Recent Researches in Geometry. *Bulletin of the New York Mathematical Society* 2, 10 (July 1983), 215–249.

[8] Alex McLean. 2014. Making Programming Languages to Dance to: Live Coding with Tidal. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design (FARM '14)*. Association for Computing Machinery, New York, NY, USA, 63–70. https://doi.org/10.1145/2633638.2633647

[9] Alex McLean. 2020. Algorithmic Pattern. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. Birmingham City University, Birmingham, UK, 265–270. https://doi.org/10.5281/zenodo.4813352

[10] Orestis Melkonian, Iris Yuping Ren, Wouter Swierstra, and Anja Volk. 2019. What Constitutes a Musical Pattern?. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design*. ACM, Berlin Germany, 95–105. https://doi.org/10.1145/3331543.3342587

[11] John Stillwell. 1998. *Numbers and Geometry*. Springer, New York, NY. https://doi.org/10.1007/978-1-4612-0687-3

[12] Godfried Toussaint. 2005. The Euclidean Algorithm Generates Traditional Musical Rhythms. In *Renaissance Banff: Mathematics, Music, Art, Culture : Conference Proceedings 2005*, Reza Sarhangi and Robert V. Moody (Eds.). Bridges Conference, Southwestern College, Winfield, Kansas, 47–56.

[13] Godfried T Toussaint. 2020. *The Geometry of Musical Rhythm: What Makes a "Good" Rhythm Good?* (second edition ed.). CRC Press, Boca Raton London New York.

[14] I. M. Yaglom. 1988. *Felix Klein and Sophus Lie: Evolution of the Idea of Symmetry in the Nineteenth Century*. Birkhäuser, Boston.

Received 02-JUN-2024; accepted 2024-07-02