AlgoRhythm

A Library for Algorithmic Music Composition

Joris ten Tusscher, Cas van der Rest, Orestis Melkonian

April 5, 2018

Universiteit Utrecht

Some definitions

- **Melody**: Notes played in *sequence*
- **Chords/harmony**: Notes played *simultaneously*
- **Scale**: a sequence of ascending notes, beginning and starting on the same note.

i.e: C major = C, D, E, F, G, A, B, C

Or in intervals: 2, 2, 1, 2, 2, 2, 1

Some definitions

A piece of music is said to be in a **key** if it (primarily) uses notes from a certain scale

Diatonic music is music that uses scales that have the same pattern as we saw before (2,2,1,2,2,2,1).

Music DSL: Representation

Basically, you want to know when to make noise and when to remain silent.

Two pieces of music can be composed in parallel or sequentially.

```
type Duration = Rational

data Music a = Music a :+: Music a
             | Music a :=: Music a
             | Note Duration a
             | Rest Duration
```

Music DSL: Representation

In order to provide export functionalities, we use a `MusicCore` type and a typeclass `ToMusicCore`.

```
type PitchClass = C | Cs | D ... As | B
type Octave = Oct0 | Oct1 ... Oct5 | Oct6
type PitchAttribute = Dynamic Dynamic
                    | Articulation Articulation

type MusicCore =
  Music ((PitchClass, Octave), [PitchAttribute])
```

This ensures that all the necessary information is there when exporting a piece of music

Music DSL: Representation

(Abstract) scales and chords are represented as intervals between notes, i.e:

```
major = [P1,M2,M3,P4,P5,M6,M7] -- Major scale
d7b5 = [P1, M3, A4, Mi7] -- Half diminished chord
```

There are many constants for various scales and chords (, as well as common durations:

```
qn = 1%4
```

Music DSL: Manipulation

Music can be constructed and manipulated using various operators

```
-- quarter note C in the 4th octave, played softly
let n = (C#4 <: [PPP]) <| qn

-- A half note rest
let r = (hn~~)

-- Instantiate an abstract chord
let cMaj7 = ((C =| maj7) <#) 3 <|| wn
```

Music DSL: Manipulation

A melody in our DSL:

```
line [ (C#6 <: [Dynamic PP]) <| wn
     , (D#6 <: [Dynamic MP]) <| wn
     , (hn~~)
     , (C#6 <: [Dynamic F_]) <| qn
     , (D#6 <: [Dynamic F_]) <| qn
     , (C#6 <: [Dynamic F_]) <| qn
     , (B#5 <: [Dynamic F_]) <| qn
     , (D#6 <: [Dynamic MF]) <| qn
     , (C#6 <: [Dynamic MP]) <| wn
     , (F#5 <: [Dynamic P ]) <| wn
     ]
```

Music DSL: Manipulation

There's also some operators for common operations:

```
-- Transposition
C ~> M3 == E

-- Retrograded (mirroring)
let music' = (music><)

-- Time scaling
let music' = music *~ (1%5)
```

Also, `Music` is a functor!

```
^^Ilet rhythm = const () <$> music
```

Focus on Generation, Ignore Analysis

Generation: monadic interface

We provide a simple monadic interface to allow users to steer the generation process

```
type MusicGenerator s a = GenericMusicGenerator GenState

type GenericMusicGenerator st s a = StateT (st s) IO a
```

Generation: monadic interface

The `GenState` type can be used to generate for the `MusicCore` data type.

```
data Entry s a = Entry { values      :: [(Weight, a)]
                       , constraints :: [Constraint a]
                       , selector    :: Selector s a
                       }

data GenState s = GenState { state      :: s
                          , pc     :: Entry s PitchClass
                          , oct    :: Entry s Octave
                          , dur    :: Entry s Duration
                          , itv    :: Entry s Interval
                          , dyn    :: Entry s Dynamic
                          , art    :: Entry s Articulati
                          }
```

Generation: monadic interface

`Selector` determines how values are selected from the possible options and `Accessor` tells you how to get to a value in the generation state.

```
type Selector s a = s -> [(Weight, a)] -> IO (a, s)

data Accessor st s a = Accessor
  { getValue :: st s -> Entry s a
  , setValue :: Entry s a -> st s -> st s
  }
```

Default accessors are included for `MusicCore`: `octave`, `duration`, etc ...

Generation: monadic interface

Generate a few pitches within the E phrygian dominant scale:

```
gen :: MusicGenerator () [PitchClass]
gen = do
  pitchClass >! (inScale E (harmonicMinor ~> P5)
  20 .#. (pitchClass??)
```

Running the generator (in the IO monad):

```
notes <- runGenerator () gen
```

Generation: diatonic improvisation

So, how do we go about generating some actual music?

Generation: diatonic improvisation

Generating something playable (attempt 1):

```
playable = do
  pitches   <- 20 .#. (pitchClass??)
  durations <- 20 .#. (duration??)
  octaves   <- 20 .#. (octave??)
  return (line $ zipWith (<|)
    ((flip (<:) $ []) <$> zipWith (#) pitches octaves)
    durations)
```

Depending on your taste, this will probably sound rubbish

Generation: diatonic improvisation

Generating something playable (attempt 2):

```
playable = do
  pitchClass >! (inScale C major)
  options <- (pitchClass?+)
  rhythm  <- boundedRhythm (1 * wn) High
  -- set options and generate pitches
  pitchClass >+ map
        (\(w, v) ->
          if v `elem` (G =| d7 :: [PitchClass])
            then (4 * w, v) else (w, v)) options
  pitches <- (length rhythm) .#. (pitchClass??)
  -- put everything together into a piece of music
  let fullPitches = (flip (<:) $ []) <$>
        (zipWith (#) pitches (repeat 4))
  let gmaj7 = (toMusicCore . chord .
        map (Note (1 * wn) . (flip (#)) 3)) (G =| d7)
  return $ gmaj7 :=: line (zipWith (<|) fullPitches rhyt
```

Generation: diatonic improvisation

Using a simple heuristic and some tweaking, we can improve upon this result:

```
diatonicPhrase dur density key scale chord octD = do
    durations <- boundedRhythm dur density
    octaves <- genAspect octave 4
      (length durations) 2.0
        (map (Arrow.first fromIntegral) octD)
    pitches <- genAspect pitchClass key
      (length durations) 1.3
      (mergeWeights
        (intervalWeights key  scale)
        (semiChordWeights key chord))
    let fullPitches = ((flip (<:) $ [])
          <$> (zipWith (#) pitches octaves))
    return $ line
      (zipWith (<|) fullPitches durations)
```

Generation: diatonic improvisation

However, as it turns out, this approach doesn't scale too well

Music is structured in many different ways; both on the micro and macro level.

The `MusicGenerator` type simply doesn't allow you to generate macro-level structures for your music in a nice way. It is too focused on the fundamentals.

Chaos in music

- Chaos system: n start values, n update functions. f_x calculates x_{i+1} given x_i.
- Chaos: small difference in init values gives very different results.

Chaos in music: example

Table 1: $f_x = max\ (-1)\ (min\ 1\ (1 - rx^2))$

r	1.9521	1.9621	0.25
x	1.2	1.18	1.18
x_0	-1.0	-1.0	0.8937
x_1	-0.9521	-0.9621	0.8002
x_2	-0.7695	-0.8161	0.8398
x_3	-0.1561	-0.3070	0.8236
x_4	0.9524	0.8149	0.8304
x_5	-0.7708	-0.3031	0.8276
x_6	-0.1598	0.8196	0.8287

Chaos in music: hard to get right

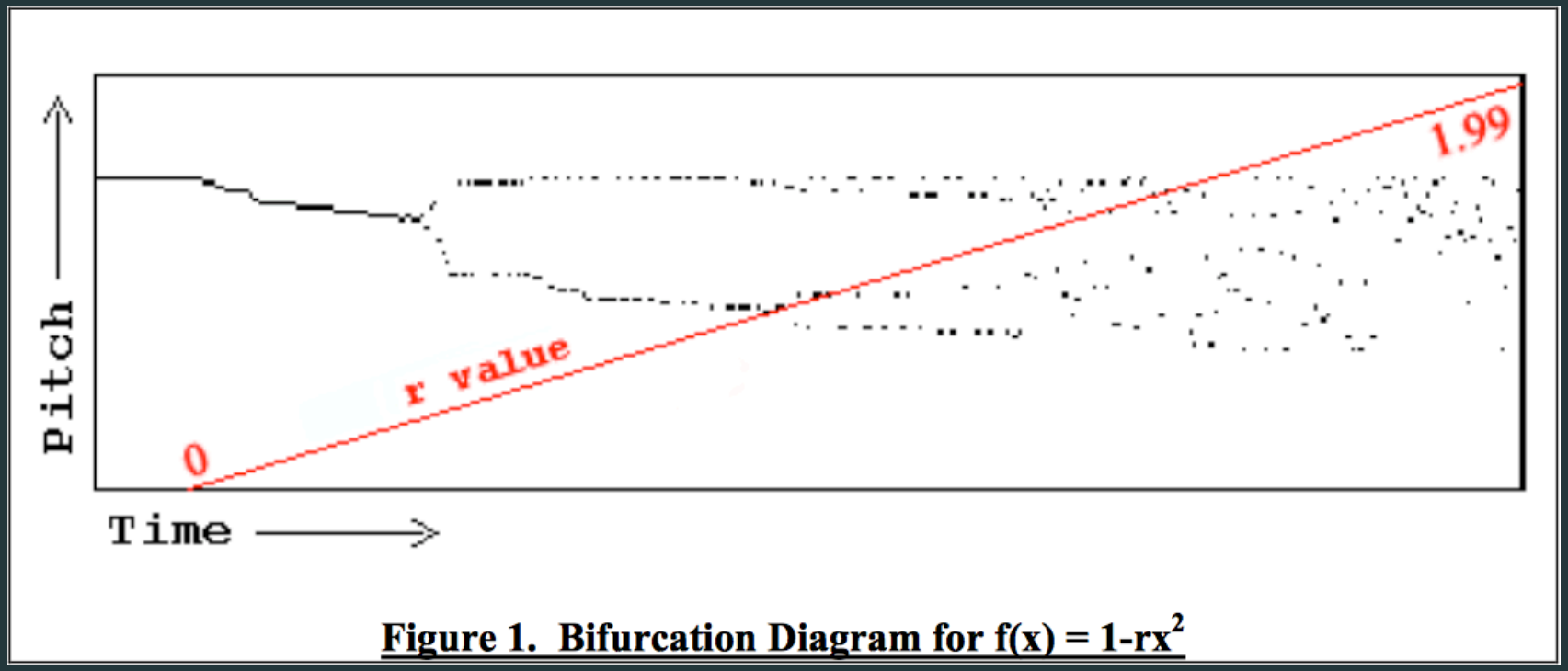

Figure 1. Bifurcation Diagram for $f(x) = 1-rx^2$

Walker, Elaine. "Chaos melody theory." Music Technology New York University, Master's thesis (2001).

Dynamic Performance: 1 (Cluster notes)

K-means

- x: absolute start time of note
- y: pitch, represented as integer
- k: total music time / beats per standard bar

Dynamic Performance: 2 (Map to dynamics)

1. Convert x (abs. time) and y (int pitch) to relative values per cluster in range [0,1].
2. Call mapping function on every (x,y) pair
3. Convert mapping function result to dynamics
4. Add dynamics to note that (x,y) belongs to.

Grammars: Properties

(Generative) *context-free grammars*, with a few extra features:

- **Temporal**: Rules are parametric to duration
- **Probabilistic**: Rules can be assigned weights
- **Graph**: Allow node sharing (using *let*-expressions)

Grammars: Definition

```
data Grammar meta a =
    a |: [Rule meta a]
data Rule meta a =
    (a, Weight, Dur -> Bool) :-> (Dur -> Term meta a)
data Term meta a =
    a :%: Dur
    | Term meta a :-: Term meta a
    | Aux Bool meta (Term meta a)
    | Let (Term meta a) (Term meta a -> Term meta a)

(a, w) -| f = (a, w, f) :-> (a :%:)
a |->  b = a :-> const b
a |--> b = (a, 1, always) |-> b
($:)  = Aux False
(|$:) = Aux True
```

Grammars: Generation

```
gen :: (Eq a, Eq meta, Expand input meta a b)
    => Grammar meta a -> input -> Dur -> Music b
gen gr i t = rewrite gr t >>> unlet >>> expand i >>> toMusic
```

1. Given an initial duration, rewrite until fixpoint
   ```
   rewrite :: (Eq a, Eq meta)
           => Grammar meta a -> Dur -> Term meta a
   ```
2. Unfold *let*-expressions
   ```
   unlet (Let x f) = f x
   unlet x         = x
   ```
3. Expand auxiliary wrappers
   ```
   class Expand input meta a b | input meta a -> b where
       expand :: input -> Term meta a -> Term () b
   ```
4. Convert to music
   ```
   (:%:) ~> (<|)
   (:-:) ~> (:+:)
   ```

Grammars: Tabla Rhythm

```
tabla :: Grammar () Syllable
tabla = S |:
  [ S  |--> TE1 :-: XI
  , XI |--> TA7 :-: XD
  , XD |--> TA8
  , XG |--> TB2 :-: XA
    ...
  , TE4 |--> Ti :-: Rest :-: Dha :-: Ti
  , TC2 |--> Tira :-: Kita
  , TB3 |--> Dha :-: Tira :-: Kita
  , TD1 |--> Rest
    ...
  ]
instance ToMusicCore Syllable where
    ...
```

Grammars: Tonal Harmony

```
harmony :: Grammar Modulation Degree
harmony = I |:
  [ -- Turn-arounds
    (I,  8, (> wn)) :-> \t ->
      Let (I:%:t/2) (\x -> x :-: x)
  , (I,  6, (> hn) /\ (<= wn)) :-> \t ->
      II:%:t/4 :-: V:%:t/4 :-: I:%:t/2
  , (I,  2, (> hn) /\ (<= wn)) :-> \t ->
      V:%:t/2 :-: I:%:t/2
  , (I,  2) -| (<= wn)
    ...
    -- Modulations
  , (V,  5, (> hn)) :-> \t -> Modulation P5 $: I:%:t
  , (V,  3) -| always
  , (II, 2, (> hn)) :-> \t -> Modulation M2 |$: I:%:t
  , (II, 8) -| always
    ...
  ]

instance Expand Config Degree Modulation SemiChord where
    ...

voiceLead :: Music SemiChord -> IO (Music Chord)
```

Grammars: Jazz Improvisation

```
melody :: Grammar () NT
melody = MQ |:
  [ -- Abstract Rhythm { MQ ~> Q }
    (MQ,  1, (== qn)) |-> Q:%:qn
  , (MQ, 25, (> (hn^.))) :-> \t -> Q:%:hn :-: MQ:%:(t - hn)
    ...
    -- Concrete Rhythm { Q ~> MN }
  , (Q, 47, (== wn)) |-> MN:%:qn :-: Q:%:hn :-: MN:%:qn
  , (Q,  6, (== hn)) |->
      MN:%:(qn^^^) :-: MN:%:(qn^^^) :-: MN:%:(qn^^^)
    ...
    -- Abstract Melody { MN ~> N }
  , (MN, 1, (== wn)) |-> N:%:qn :-: N:%:qn :-: MN:%:hn
  , (MN, 1, (== qn)) |->
      N:%:(en^^^) :-: N:%:(en^^^) :-: N:%:(en^^^)
    ...
    -- Concrete Melody { N ~> NT }
  , (N, 50, (== qn)) |-> ChordTone:%:qn
  , (N, 45, (== qn)) |-> Rest:%:qn
  , (N,  1, (== en)) |-> ApproachTone:%:en
    ...
  ]

mkSolo :: Music SemiChord -> Music NT -> IO Melody
```

Demo: Code

```
orientalAlgebras = do
  let ?config = MusicConfig
    { basePc       = A
    , baseOct      = Oct3
    , baseScale    = arabian
    , chords       = equally allChords
    , scales       = equally allScales
    , octaves      = [(20, Oct4), (15, Oct5), (5, Oct6)]
    , restWeight   = 0, ...
    , tempo        = 6%5
    , instruments = [Piano, Sitar, Tabla]
    , beat = sn
    }
  let t = 12 * wn
  har <- voiceLead  <$> runGrammar harmony t
  mel <- mkSolo har <$> runGrammar melody  t
  rhy <- runGrammar tabla t
  writeToMidiFile "out.mid" (dyn (har :=: mel :=: rhy))
```

Demo: Music score