Passager, for string orchestra

Hans Höglund 2012

March 11, 2012

Introduction

This document describes a piece for string orchestra. It is a literate program, which may be read as well as executed.

To convert into a readable Pdf file, install Pandoc and use

```
markdown2pdf -o MusicaVitae.pdf MusicaVitae.lhs
```

To run the program, install the Haskell platform and the `Music.Time` module. For correct playback of intonation, use a sampler that support Midi Tuning Standard, such as Timidity.

```
{-# LANGUAGE
    TypeSynonymInstances,
    FlexibleInstances,
    MultiParamTypeClasses,
    DeriveFunctor #-}

module Music.Projects.MusicaVitae
(
-- * Preliminaries
-- ** Instruments and tuning
    Part(..),
    Section(..),
    Tuning(..),
```

```
    partSection,
    sectionTuning,
    partTuning,
    ensemble,
    sectionParts,
    isViolin, isViola, isCello,
    highParts, lowParts,
    highViolinParts, highViolaParts, highCelloParts,
    lowViolinParts, lowViolaParts, lowCelloParts,
    doubleBass,
    Doubling(..),
    PartFunctor(..),

-- ** Time and pitch
    Dur(..),
    Pitch(..),
    Str(..),
    Scale(..),
    step,
    scaleFromSteps,
    PitchFunctor(..),

-- ** Dynamics
    Level(..),
    Dynamics(..),
    levelAt,
    ppp, pp, p, mf, f, ff, fff,
    cresc,
    dim,
    LevelFunctor(..),

-- ** Articulation
    Articulation(..),
    Phrasing(..),
    staccato,
```

```
    tenuto,
    legato,
    portato,

-- ** Playing techniques
    RightHand(..),
    LeftHand(..),
    Stopping(..),
    Stopped,
    Technique,

-- ** Intonation
    Intonation(..),
    intonation,
    cueIntonation,
    raisedIntonation,

-- ** Cues
    Cue(..),

-- * Midi rendering
    renderCue,

-- * High-level constructors
-- ** Open strings
    openString,
    openStringPizz,
    openStringJete,
    openStrings,

-- ** Natural harmonics
    naturalHarmonic,
    naturalHarmonicPizz,
    naturalHarmonicJete,
    naturalHarmonics,
```

```
-- ** Quarter stopped strings
    quarterStoppedString,
    quarterStoppedStrings,

-- ** Stopped strings
    stoppedString,
    stoppedStringPizz,
    stoppedStringJete,
    stoppedStrings,

-- ** Tremolo
    stoppedStringTrem,
    naturalHarmonicTrem,
)
where

import Prelude hiding ( reverse )

import Control.Applicative
import Data.Convert ( convert )
import qualified Data.List as List

import Music
import Music.Time.Overlay
import Music.Time.Tremolo
import Music.Time.Functors
import Music.Render
import Music.Render.Midi
import Music.Inspect
import Music.Util.List
import Music.Util.Either
```

Preliminaries

In this chapter we will describe the musical preliminaries needed to compose the piece.

Instruments and parts

The instrumentation is as follows:

- Violin I-IV
- Viola I-II
- Cello I-II
- Double Bass

A basic idea of the piece is to combine (slightly) different tunings of the instruments using open-string techniques and harmonics. For this purpose, we will split the ensemble into three sections, each using a different tuning:

- Odd-numbered Vl, Vla and Vc parts tunes A4 to 443 Hz (A3 to 221.5 Hz)
- Even-numbered Vl, Vla and Vc parts tunes A4 to 437 Hz (A3 to 218.5 Hz)
- Double bass tunes A1 to 55 Hz

The other strings should be tuned in relation to the A-string as usual.

```
data Part
    = Violin Int
    | Viola  Int
    | Cello  Int
    | DoubleBass
    deriving ( Eq, Show )

data Section
    = High | Low | Middle
    deriving ( Eq, Show, Enum, Bounded )

type Tuning = Frequency
```

We now define the relation between these types as follows:

```
partSection   :: Part -> Section
sectionTuning :: Section -> Tuning
partTuning    :: Part -> Tuning

partSection ( Violin 1 ) = High
partSection ( Violin 2 ) = Low
partSection ( Violin 3 ) = High
partSection ( Violin 4 ) = Low
partSection ( Viola 1 )  = High
partSection ( Viola 2 )  = Low
partSection ( Cello 1 )  = High
partSection ( Cello 2 )  = Low
partSection DoubleBass = Middle

sectionTuning Low    = 440 - 2
sectionTuning Middle = 440
sectionTuning High   = 440 + 2

partTuning = sectionTuning . partSection
```

Then add some utility definitions to quickly access the various parts:

```
ensemble     :: [Part]
sectionParts :: Section -> [Part]

isViolin, isViola, isCello, isDoubleBass :: Part -> Bool

highParts, lowParts
    :: [Part]
highViolinParts, highViolaParts, highCelloParts
    :: [Part]
lowViolinParts, lowViolaParts, lowCelloParts
    :: [Part]
doubleBass :: Part

ensemble
```

```
    = [ Violin 1, Violin 2, Violin 3, Violin 4
      , Viola 1, Viola 2, Cello 1, Cello 2, DoubleBass ]

sectionParts s  =  filter (\x -> partSection x == s) ensemble

highParts  =  sectionParts High
lowParts   =  sectionParts High

isViolin     ( Violin _ )    =  True
isViolin     _               =  False
isViola      ( Viola _ )     =  True
isViola      _               =  False
isCello      ( Cello _ )     =  True
isCello      _               =  False
isDoubleBass ( DoubleBass )  =  True
isDoubleBass _               =  True

highViolinParts  =  filter isViolin (sectionParts High)
highViolaParts   =  filter isViola  (sectionParts High)
highCelloParts   =  filter isCello  (sectionParts High)
lowViolinParts   =  filter isViolin (sectionParts Low)
lowViolaParts    =  filter isViola  (sectionParts Low)
lowCelloParts    =  filter isCello  (sectionParts Low)
doubleBass       =  DoubleBass
```

All parts may be doubled. If several parts are doubled but not all, the musicians should strive for a balance between the two main tuning sections (i.e. avoid doubling just the upper parts or vice versa).

Certain cues are required to be played by a single musician even if the parts are doubled, which will be marked *solo*. These passages should be distributed evenly among the musicians, instead of being played by designated soloists.

```
data Doubling = Solo | Tutti
    deriving ( Eq, Show )
```

The `PartFunctor` class defined useful operations for mapping over part and doubling.

```
class PartFunctor f where
    setPart     :: Part -> f -> f
    mapPart     :: (Part -> Part) -> f -> f

    setDoubling :: Doubling -> f -> f
    mapDoubling :: (Doubling -> Doubling) -> f -> f

    setPart     x = mapPart (const x)
    setDoubling x = mapDoubling (const x)
```

Time and pitch

We will use the temporal operations form `Music.Time` for composition on both event level and structural level. We use floating point values to represent durations.

```
type Dur = Double
```

For simplicity, we will use Midi numbers for written pitch. Sounding pitch will of course be rendered depending on tuning and playing technique of the given part.

String number will be represented separately using a different type (named `Str` so as not to collide with `String`).

```
type Pitch = Int

data Str = I | II | III | IV
    deriving ( Eq, Show, Ord, Enum, Bounded )
```

A scale is conceptually a function from steps to pitches. This relation is captured by the `step` function, which maps steps to pitches. For example, `major 'step' 3` means the third step in the major scale.

The simplest way to generate a scale is to list its relative steps, i.e. `2,2,2,1,2` for the first five pitches in the major scale. This is captured by the function `scaleFromSteps`.

```
newtype Scale a = Scale { getScale :: [a] }
    deriving ( Eq, Show, Functor )

step :: Scale Pitch -> Pitch -> Pitch
```

```
step (Scale xs) p = xs !! (p `mod` length xs)

scaleFromSteps :: [Pitch] -> Scale Pitch
scaleFromSteps = Scale . accum
    where
        accum = snd . List.mapAccumL add 0
        add a x = (a + x, a + x)

retrograde :: Scale Pitch -> Scale Pitch
retrograde = Scale . List.reverse . getScale
```

The `PitchFunctor` class defines a useful operation for mapping over pitch. This generalizes to scales and scores containing pitched elements.

```
class PitchFunctor f where
    setPitch :: Pitch -> f -> f
    mapPitch :: (Pitch -> Pitch) -> f -> f
    setPitch x = mapPitch (const x)

instance PitchFunctor (Pitch) where
    mapPitch f x = f x

instance PitchFunctor (Scale Pitch) where
    mapPitch f = fmap f

invert :: PitchFunctor f => f -> f
invert = mapPitch negate
```

Dynamics

We use a linear representation for dynamic levels. Level 0 corresponds to some medium level dynamic, level 1 to extremely loud and level –1 to extremely soft. A dynamic is a function from time to level, generalizing crescendo, diminuendo and so on.

```
type Level = Double

newtype Dynamics = Dynamics { getDynamics :: Dur -> Level }
```

```
    deriving ( Eq )

instance Show Dynamics where
    show x = ""

levelAt :: Dynamics -> Dur -> Level
levelAt (Dynamics n) t = n t

ppp, pp, p, mf, f, ff, fff :: Dynamics
ppp = Dynamics $ const (-0.8)
pp  = Dynamics $ const (-0.6)
p   = Dynamics $ const (-0.3)
mf  = Dynamics $ const 0
f   = Dynamics $ const 0.25
ff  = Dynamics $ const 0.5
fff = Dynamics $ const 0.7

cresc, dim :: Dynamics
cresc = Dynamics id
dim   = Dynamics (succ . negate)

instance Num Dynamics where
    (Dynamics x) + (Dynamics y)  =  Dynamics (\t -> x t + y t)
    (Dynamics x) * (Dynamics y)  =  Dynamics (\t -> x t * y t)
    signum (Dynamics x)          =  Dynamics (signum . x)
    abs (Dynamics x)             =  Dynamics (abs . x)
    fromInteger n                =  Dynamics (const $ fromInteger n)

class LevelFunctor f where
    setLevel :: Level -> f -> f
    mapLevel :: (Level -> Level) -> f -> f
    setLevel x = mapLevel (const x)

    setDynamics :: Dynamics -> f -> f
    mapDynamics :: (Dur -> Level -> Level) -> f -> f
```

```
    setDynamics n = mapDynamics (\t _ -> n `levelAt` t)

instance LevelFunctor Dynamics where
    mapLevel f (Dynamics n) = Dynamics (f . n)
    mapDynamics f (Dynamics n) = Dynamics (\t -> f t $ n t)
```

Articulation

```
data Articulation
    = Straight
    | Accent   Double Articulation
    | Duration Double Articulation
    deriving ( Eq, Show )

data Phrasing
    = Phrasing
    | Binding Double Phrasing
    | Begin Articulation Phrasing
    | End Articulation Phrasing
    deriving ( Eq, Show )

staccato :: Articulation -> Articulation
staccato = Duration 0.8

tenuto :: Articulation -> Articulation
tenuto = Duration 1.2

legato :: Phrasing -> Phrasing
legato = Binding 1.2

portato :: Phrasing -> Phrasing
portato = Binding 0.8
```

Playing techniques

The piece makes use of different playing techniques in both hands.

The `RightHand` type is parameterized over time, articulation, phrasing and content. The `LeftHand` type is parameterized over pitch and string.

```
data RightHand t c r a
    = Pizz   c a
    | Single c a
    | Phrase r [(t, a)]
    | Jete   r [a]
    deriving ( Eq, Show )

data LeftHand p s
    = OpenString            s
    | NaturalHarmonic       p s
    | NaturalHarmonicTrem   p p s
    | NaturalHarmonicGliss  p p s
    | QuarterStoppedString  s
    | StoppedString         p s
    | StoppedStringTrem     p p s
    | StoppedStringGliss    p p s
    deriving ( Eq, Show )

leftHand :: RightHand t c r a -> [a]
leftHand (Pizz   c x)   =  [x]
leftHand (Single c x)   =  [x]
leftHand (Phrase c xs)  =  map snd xs
leftHand (Jete   c xs)  =  xs
```

As the intonation will be different between open and stopped strings, we define a function mapping each left-hand technique to a stopping. This stopping also distributes over right-hand techniques (for example, an the intonation of a natural harmonic is open, whether played *arco* or *pizz*).

```
data Stopping = Open | QuarterStopped | Stopped
    deriving ( Eq, Show )

class Stopped a where
    stopping :: a -> Stopping
```

```
instance Stopped (LeftHand p s) where
    stopping ( OpenString           s     ) = Open
    stopping ( NaturalHarmonic      x s   ) = Open
    stopping ( NaturalHarmonicTrem  x y s ) = Open
    stopping ( NaturalHarmonicGliss x y s ) = Open
    stopping ( QuarterStoppedString s     ) = QuarterStopped
    stopping ( StoppedString        x s   ) = Stopped
    stopping ( StoppedStringTrem    x y s ) = Stopped
    stopping ( StoppedStringGliss   x y s ) = Stopped

instance Stopped a => Stopped (RightHand t r p a) where
    stopping ( Pizz   c x )       =  stopping x
    stopping ( Single c x )       =  stopping x
    stopping ( Phrase r (x:xs) )  =  stopping (snd x)
    stopping ( Jete   r (x:xs) )  =  stopping x

instance PitchFunctor (LeftHand Pitch s) where
    mapPitch f ( StoppedString        x s   ) = StoppedString      (f x) s
    mapPitch f ( StoppedStringTrem    x y s ) = StoppedStringTrem  (f x) (f y) s
    mapPitch f ( StoppedStringGliss   x y s ) = StoppedStringGliss (f x) (f y) s
    mapPitch f x                              = x

instance PitchFunctor a => PitchFunctor (RightHand t c r a) where
    mapPitch f ( Pizz   c x )   =  Pizz   c (mapPitch f x)
    mapPitch f ( Single c x )   =  Single c (mapPitch f x)
    mapPitch f ( Phrase r xs )  =  Phrase r (fmap (\(d,p) -> (d, mapPitch f p)) xs)
    mapPitch f ( Jete   r xs )  =  Jete   r (fmap (mapPitch f) xs)
```

Intonation

Many playing techiniques in the score calls for open strings. In this case intonation is determined solely by the tuning.

In some cases, open-string techniques are used with an above first-position stop. This should make the open string pitch rise about a quarter-tone step (or at least less than a half-tone step).

Where stopped strings are used, intonation is determined by context:

- In solo passages, intonation is individual. No attempt should be made to synchronize intontation (on long notes et al) for overlapping solo cues.
- In unison passages, common intonation should be used.

```
data Intonation
    = Tuning
    | Raised
    | Common
    | Individual
    deriving ( Eq, Show )

intonation :: Doubling -> Technique -> Intonation

intonation Tutti t = case stopping t of
    Open           -> Tuning
    QuarterStopped -> Raised
    Stopped        -> Common

intonation Solo t = case stopping t of
    Open           -> Tuning
    QuarterStopped -> Raised
    Stopped        -> Individual

cueIntonation :: Cue -> Intonation
cueIntonation (Cue p d n t) = intonation d t

raisedIntonation :: Cent
raisedIntonation = 23 Cent
```

Cues

A *cue* is an action taken by a performer on time.

```
type Technique =
    RightHand
```

```
        Dur
        Articulation
        Phrasing
        (LeftHand
            Pitch Str)

data Cue
    = Cue
    {
        cuePart      :: Part,
        cueDoubling  :: Doubling,
        cueDynamics  :: Dynamics,  -- time is 0 to 1 for the duration of the cue
        cueTechnique :: Technique
    }
    deriving ( Eq, Show )

instance PartFunctor Cue where
    mapPart     f (Cue p d n t) = Cue (f p) d n t
    mapDoubling f (Cue p d n t) = Cue p (f d) n t

instance PitchFunctor Cue where
    mapPitch f (Cue p d n t) = Cue p d n (mapPitch f t)

instance LevelFunctor Cue where
    mapLevel f    (Cue p d n t) = Cue p d (mapLevel f n) t
    mapDynamics f (Cue p d n t) = Cue p d (mapDynamics f n) t

instance (Time t, PartFunctor a) => PartFunctor (Score t a) where
    mapPart f = fmap (mapPart f)
    mapDoubling f = fmap (mapDoubling f)

instance (Time t, PitchFunctor a) => PitchFunctor (Score t a) where
    mapPitch f = fmap (mapPitch f)

instance (Time t, LevelFunctor a) => LevelFunctor (Score t a) where
```

```
mapLevel    f = fmap (mapLevel f)
mapDynamics f = fmap (mapDynamics f)
```

Midi rendering

We are going to compose the piece as a score of cues. In order to hear the piece and make musical decisions, we need to define a rendering function that renders a cue to a score of Midi notes, which is the object of this chapter.

The `MidiNote` type is imported from `Music.Render.Midi`, but we define some extra type synonyms to make the rendering functions somewhat more readable:

```
type MidiChannel    = Int
type MidiInstrument = Maybe Int
type MidiPitch      = Int
type MidiBend       = Semitones
type MidiDynamic    = Int
```

Channel

A caveat is that the Midi representation does not handle simultaneous tunings well. We must therefore separete the music into different Midi channels based on part, section and intontation.

```
midiChannel :: Cue -> MidiChannel
midiChannel (Cue part doubling dynamics technique) =
    midiChannel' part section intonation'
    where
          section     = partSection part
          intonation' = intonation doubling technique

midiChannel'  ( Violin _)   High   Tuning      =  0
midiChannel'  ( Viola  _ )  High   Tuning      =  1
midiChannel'  ( Cello  _ )  High   Tuning      =  2
midiChannel'  ( Violin _)   Low    Tuning      =  3
midiChannel'  ( Viola  _ )  Low    Tuning      =  4
midiChannel'  ( Cello  _ )  Low    Tuning      =  5
midiChannel'  ( Violin _)   _      Common      =  6
midiChannel'  ( Viola  _ )  _      Common      =  7
midiChannel'  ( Cello  _ )  _      Common      =  8
```

```
midiChannel'  DoubleBass    _      _           =  10

midiChannel'  ( Violin _)   _      Raised      =  11
midiChannel'  ( Viola  _ )  _      Raised      =  11
midiChannel'  ( Cello  _ )  _      Raised      =  13
midiChannel'  ( Violin _)   High   Individual  =  0
midiChannel'  ( Viola  _ )  High   Individual  =  1
midiChannel'  ( Cello  _ )  High   Individual  =  2
midiChannel'  ( Violin _)   Low    Individual  =  3
midiChannel'  ( Viola  _ )  Low    Individual  =  4
midiChannel'  ( Cello  _ )  Low    Individual  =  5
```

Instrument

Instrument rendering is simple: if the technique is *pizzicato*, use the pizzicato strings program, otherwise use the program representing the current instrument.

(The standard programs give us solo sounds. We could mix in the *string ensemble* program based on the doubling attribute. I am not sure this is a good idea though.)

```
midiInstrument :: Cue -> MidiInstrument
midiInstrument (Cue part doubling dynamics technique) =
    case technique of
        (Pizz _ _) -> Just 45
        _          -> midiInstrument' part

midiInstrument' ( Violin _ )  =  Just 40
midiInstrument' ( Viola _ )   =  Just 41
midiInstrument' ( Cello _ )   =  Just 42
midiInstrument' DoubleBass    =  Just 43
```

Pitch and bending

Table of open string pitches.

```
openStringPitch :: Part -> Str -> MidiPitch
openStringPitch ( Violin _ ) I    =  55
openStringPitch ( Violin _ ) II   =  62
```

```
openStringPitch ( Violin _ ) III  =  69
openStringPitch ( Violin _ ) IV   =  76
openStringPitch ( Viola _ )  I    =  48
openStringPitch ( Viola _ )  II   =  55
openStringPitch ( Viola _ )  III  =  62
openStringPitch ( Viola _ )  IV   =  69
openStringPitch ( Cello _ )  I    =  36
openStringPitch ( Cello _ )  II   =  43
openStringPitch ( Cello _ )  III  =  50
openStringPitch ( Cello _ )  IV   =  57
openStringPitch DoubleBass   I    =  28
openStringPitch DoubleBass   II   =  33
openStringPitch DoubleBass   III  =  38
openStringPitch DoubleBass   IV   =  43

naturalHarmonicPitch :: Part -> Str -> Int -> MidiPitch
naturalHarmonicPitch part str tone =
    fundamental + overtone
    where
        fundamental = openStringPitch part str
        overtone = scaleFromSteps [0,12,7,5,4,3,3,2,2,2] `step` tone
```

We determine amount of pitch bend from the part, doubling and technique. Note that the `cents` function converts a frequency to cents, so by subtracting the reference pitch from the intonation, we get the amount of bending in cents. Then divide this by 100 to get the amount in semitones.

For harmonics, we add a compensation for the difference between just and twelve-tone equal temperament. Unfortunately this does not work for harmonic tremolos.

```
midiBend :: Cue -> MidiBend
midiBend (Cue part doubling dynamics technique) =
    midiBend' (intonation', cents') + just
    where
        intonation'  =  intonation doubling technique
        tuning'      =  partTuning part
        cents'       =  cents tuning' - cents 440
```

```
        just         =  midiBendJust (head . leftHand $ technique)

midiBend' ( Raised, c )      =  getCent (c + raisedIntonation) / 100
midiBend' ( Tuning, c )      =  getCent c / 100
midiBend' ( Common, c )      =  0
midiBend' ( Individual, c )  =  0

midiBendJust :: LeftHand Pitch Str -> MidiBend
midiBendJust ( NaturalHarmonic x s )  =  midiBendJust' x
midiBendJust _                        = 0

midiBendJust' 0  =  0
midiBendJust' 1  =  0
midiBendJust' 2  =  0.0196
midiBendJust' 3  =  0
midiBendJust' 4  =  -0.1369
midiBendJust' 5  =  0.0196
midiBendJust' 6  =  -0.3117
midiBendJust' 7  =  0
midiBendJust' 8  =  0.0391
```

Left hand

The `renderLeftHand` function returns a score of duration one, possibly containing tremolos. This property is formalized by the use of a `TremoloScore`, i.e. a score containing either notes or tremolos.

Note: Glissandos are not supported yet.

```
renderLeftHand :: Part -> LeftHand Pitch Str -> TremoloScore Dur MidiNote
renderLeftHand part ( OpenString           s )      =  renderLeftHandSingle (openStringPitch
renderLeftHand part ( NaturalHarmonic      x s )    =  renderLeftHandSingle (naturalHarmonic
renderLeftHand part ( NaturalHarmonicTrem  x y s )  =  renderLeftHandTrem (naturalHarmonicPi
renderLeftHand part ( NaturalHarmonicGliss x y s )  =  renderLeftHandGliss
renderLeftHand part ( QuarterStoppedString s )      =  renderLeftHandSingle (openStringPitch
renderLeftHand part ( StoppedString        x s )    =  renderLeftHandSingle x
renderLeftHand part ( StoppedStringTrem    x y s )  =  renderLeftHandTrem x y
```

```
renderLeftHand part ( StoppedStringGliss   x y s )  =  renderLeftHandGliss

renderLeftHandSingle x    =  note . Left  $ renderMidiNote x
renderLeftHandTrem   x y  =  note . Right $ tremoloBetween tremoloInterval (renderMidiNote x
renderLeftHandGliss       =  error "Gliss not implemented"

renderMidiNote x = MidiNote 0 Nothing x 0 60
tremoloInterval = 0.08
```

Right hand

```
renderRightHand :: Part -> Technique -> TremoloScore Dur MidiNote
renderRightHand part ( Pizz   articulation leftHand )  = renderLeftHand part leftHand
renderRightHand part ( Single articulation leftHand )  = renderLeftHand part leftHand
renderRightHand part ( Phrase phrasing leftHand )      = renderLeftHands part leftHand
renderRightHand part ( Jete   phrasing leftHand )      = renderLeftHands part (zip bounceDur

renderLeftHands :: Part -> [(Dur, LeftHand Pitch Str)] -> TremoloScore Dur MidiNote
renderLeftHands part  =  stretchTo 1 . concatSeq . map leftHands
    where
        leftHands (d, x)  =  stretch d $ renderLeftHand part x
```

Cues

This section needs some cleanup…

```
setMidiChannel :: MidiChannel -> TremoloScore Dur MidiNote -> TremoloScore Dur MidiNote
setMidiChannel c = fmapE f g
    where f = (\(MidiNote _ i p b n) -> MidiNote c i p b n)
          g = fmap (\(MidiNote _ i p b n) -> MidiNote c i p b n)

setMidiInstrument :: MidiInstrument -> TremoloScore Dur MidiNote -> TremoloScore Dur MidiNot
setMidiInstrument i = fmapE f g
    where f = (\(MidiNote c _ p b n) -> MidiNote c i p b n)
          g = fmap (\(MidiNote c _ p b n) -> MidiNote c i p b n)

setMidiBend :: MidiBend -> TremoloScore Dur MidiNote -> TremoloScore Dur MidiNote
```

```
setMidiBend b = fmapE f g
    where f = (\(MidiNote c i p _ n) -> MidiNote c i p b n)
          g = fmap (\(MidiNote c i p _ n) -> MidiNote c i p b n)

setMidiDynamic :: Dynamics -> TremoloScore Dur MidiNote -> TremoloScore Dur MidiNote
setMidiDynamic (Dynamics n) = tmapE f g
    where f = (\t (MidiNote c i p b _) -> MidiNote c i p b (round $ n t * 63 + 63))
          g = (\t x -> tmap (\t (MidiNote c i p b _) -> MidiNote c i p b (round $ n t * 63 +
```

Some constants used for the rendering of jeté strokes.

```
bounceDur :: [Dur]
bounceDur  =  [ (2 ** (-0.9 * x)) / 6 | x <- [0, 0.1..1.2] ]

bounceVel :: [Double]
bounceVel  =  [ abs (1 - x) | x <- [0, 0.08..]]
```

Render each cue to a score of `MidiNote` elements. Each generated score has a duration of one, so this function can be used with >>= to render a score of cues to Midi (see below.)

```
renderCue :: Cue -> TremoloScore Dur MidiNote
renderCue cue =
    renderRest $ renderRightHand (cuePart cue) (cueTechnique cue)
    where
        channel     =  midiChannel cue
        instr       =  midiInstrument cue
        bend        =  midiBend cue

        renderRest  =  setMidiChannel channel
                    .  setMidiInstrument instr
                    .  setMidiBend bend
                    .  setMidiDynamic (cueDynamics cue)
```

This instance makes it possible to use the `play` function on scores of cues:

```
instance Render (Score Dur Cue) Midi where
    render  =  render
```

```
.  restAfter 5
.  renderTremoloEvents
.  (>>= renderCue)
```

High-level constructors

Allthough the *cues* defined in the previous chapters is a flexible representation for an orchestral piece, they are somewhat cubersome to construct. This is easily solved by adding some higher-level constructors.

The constructors all create *standard cues* with the following definitions:

```
standardCue           =  note . Cue (Violin 1) Tutti mf
standardArticulation  =  Straight
standardPhrasing      =  Phrasing
```

These can be overriden using the methods of the type classes `Temporal`, `Timed`, `Delayed`, `PartFunctor`, `PitchFunctor` and `LevelFunctor` respectively.

Open Strings

```
openString :: Str -> Score Dur Cue
openString x = standardCue
    $ Single standardArticulation
    $ OpenString x

openStringPizz :: Str -> Score Dur Cue
openStringPizz x = standardCue
     $ Pizz standardArticulation
     $ OpenString x

openStringJete :: [Str] -> Score Dur Cue
openStringJete xs = standardCue
     $ Jete standardPhrasing
     $ map OpenString xs

openStrings :: [(Dur, Str)] -> Score Dur Cue
openStrings xs = standardCue
     $ Phrase standardPhrasing
     $ map (\(d,x) -> (d, OpenString x)) xs
```

Natural harmonics

```
naturalHarmonic :: Str -> Pitch -> Score Dur Cue
naturalHarmonic s x = standardCue
     $ Single standardArticulation
     $ NaturalHarmonic x s

naturalHarmonicPizz :: Str -> Pitch -> Score Dur Cue
naturalHarmonicPizz s x = standardCue
     $ Pizz standardArticulation
     $ NaturalHarmonic x s

naturalHarmonicJete :: Str -> [Pitch] -> Score Dur Cue
naturalHarmonicJete s xs = standardCue
     $ Jete standardPhrasing
     $ map (\x -> NaturalHarmonic x s) xs

naturalHarmonics :: Str -> [(Dur, Pitch)] -> Score Dur Cue
naturalHarmonics s xs = standardCue
     $ Phrase standardPhrasing
     $ map (\(d,x) -> (d, NaturalHarmonic x s)) xs
```

Quarter stopped strings

```
quarterStoppedString :: Str -> Score Dur Cue
quarterStoppedString x = standardCue
     $ Single standardArticulation
     $ QuarterStoppedString x

quarterStoppedStrings :: [(Dur, Str)] -> Score Dur Cue
quarterStoppedStrings xs = standardCue
     $ Phrase standardPhrasing
     $ map (\(d,x) -> (d, QuarterStoppedString x)) xs
```

Stopped strings

```
stoppedString :: Pitch -> Score Dur Cue
```

```
stoppedString x = standardCue
     $ Single standardArticulation
     $ StoppedString x I

stoppedStringPizz :: Pitch -> Score Dur Cue
stoppedStringPizz x = standardCue
     $ Pizz standardArticulation
     $ StoppedString x I

stoppedStringJete :: [Pitch] -> Score Dur Cue
stoppedStringJete xs = standardCue
     $ Jete standardPhrasing
     $ map (\x -> StoppedString x I) xs

stoppedStrings :: [(Dur, Pitch)] -> Score Dur Cue
stoppedStrings xs = standardCue
     $ Phrase standardPhrasing
     $ map (\(d,x) -> (d, StoppedString x I)) xs
```

Tremolo

```
stoppedStringTrem :: Pitch -> Pitch -> Score Dur Cue
stoppedStringTrem x y = standardCue
     $ Single standardArticulation
     $ StoppedStringTrem x y I

naturalHarmonicTrem :: Str -> Pitch -> Pitch -> Score Dur Cue
naturalHarmonicTrem s x y = standardCue
     $ Single standardArticulation
     $ NaturalHarmonicTrem x y s
```

Final composition

In this chapter we will assemble the final piece.

Pitches

The pitch material is based on a 15-tone symmetric scale. The lower half is mixolydian and the upper half is aeolian (i.e. the inverse of mixolydian).

We will represent melodies in a relative fashion, using 0 to represent a reference pitch. This simplifies inversion and similar techniques. We use A3 as the reference pitch (corresponding to step 0). The `tonality` function applies the major-minor scale at the reference pitch, so pitch operations applied *before* this function is diatonic, while operations applied *after* it is chromatic.

```
minScale    = scaleFromSteps [0, 2, 1, 2, 2, 1, 2, 2]
majMinScale = Scale $ getScale lower ++ getScale upper
    where
        lower = retrograde . invert $ minScale
        upper = Scale . tail . getScale $ minScale

tonality :: PitchFunctor f => f -> f
tonality = mapPitch $ offset . scale . tonic
    where
        tonic  = (+ 7)
        scale  = (majMinScale `step`)
        offset = (+ 57)
```

The `tonalSeq` function generates a binary musical sequence, in which the second operand is transposed the given amount of steps. The `tonalConcat` function is the same on higher arities.

The `fifthUp`, `fifthDown` etc are handy shortcuts for transposition.

```
tonalSeq :: (Time t, PitchFunctor a) => Pitch -> Score t a -> Score t a -> Score t a
tonalConcat :: (Time t, PitchFunctor a) => Pitch -> [Score t a] -> Score t a

tonalSeq    p x y  =  x >>> mapPitch (+ p) y
tonalConcat p      =  List.foldr (tonalSeq p) instant
```

```
duodecDown, octaveDown, fifthDown :: PitchFunctor a => a -> a
fifthUp, octaveUp, duodecUp       :: PitchFunctor a => a -> a

duodecDown  =  mapPitch (+ (-19))
octaveDown  =  mapPitch (+ (-12))
fifthDown   =  mapPitch (+ (-7))
fifthUp     =  mapPitch (+ 7)
octaveUp    =  mapPitch (+ 12)
duodecUp    =  mapPitch (+ 19)
```

Melody

Melodic patterns that may work well in the symmetric scale.

```
type Pattern =  [(Dur, Pitch)]

pattern :: Int -> Pattern
pattern = (patterns !!)

-- Play using
--     play . tonality . patternMelody $ pattern 0
patterns =
    [
        zip [ 3, 3 ]
            [ 0, 1 ],
        zip [ 1, 1, 1, 1, 3, 3 ]
            [ 0, 1, 1, 2, 0, 1 ],
        zip [ 1, 1, 1, 2, 1, 3, 3 ]
            [ 0, 1, 1, 2, 3, 0, 1 ],
        zip [] [],
        zip [] [],

        -- 5
        zip [ 1, 1, 1, 1, 3, 3 ]
            [ 0, 2, 1, 2, 0, -1 ],
```

```
        zip [ 1, 1, 1, 1, 4 ]
            [ 0, 2, 1, 2, 3 ]
    ]

patternMelody :: Pattern -> Score Dur Cue
patternMelody x = stretch (scaling x / 2) . stoppedStrings $ x
    where
        scaling = sum . map fst

patternSequence :: Pitch -> [Pattern] -> Score Dur Cue
patternSequence p = tonalConcat p . map patternMelody
```

Harmony

```
secondChord :: [Int] -> [Str] -> [Int] -> Score Dur Cue
secondChord xs ss ps =  instant
    ||| (setDynamics p . setPart (Cello  (xs !! 0)) $ naturalHarmonic (ss !! 0) (ps !! 0))
    ||| (setDynamics p . setPart (Viola  (xs !! 0)) $ naturalHarmonic (ss !! 1) (ps !! 1))
    ||| (setDynamics p . setPart (Violin (xs !! 0)) $ naturalHarmonic (ss !! 2) (ps !! 2))
    ||| (setDynamics p . setPart (Violin (xs !! 0)) $ naturalHarmonic (ss !! 3) (ps !! 3))

secondChordTrem :: [Int] -> [Str] -> [Int] -> Score Dur Cue
secondChordTrem xs ss ps =  instant
    ||| (setDynamics p . setPart (Cello  (xs !! 0)) $ naturalHarmonicTrem (ss !! 0) 0 (ps !!
    ||| (setDynamics p . setPart (Viola  (xs !! 0)) $ naturalHarmonicTrem (ss !! 1) 0 (ps !!
    ||| (setDynamics p . setPart (Violin (xs !! 0)) $ naturalHarmonicTrem (ss !! 2) 0 (ps !!
    ||| (setDynamics p . setPart (Violin (xs !! 0)) $ naturalHarmonicTrem (ss !! 3) 0 (ps !!


scs = concatSeq $ do
    ss <- return [IV,III,II,I]
    ps <- List.permutations [1,2,3,4]
    -- xs <- List.permutations [2,1,2,1]
    -- return $ secondChord xs ss ps
    return $ secondChord [1,1,1,1] ss ps ||| secondChord [2,2,2,2] ss ps
```

```
-- g, a, d, e
-- scs = concatSeq $ do
--     ss <- return [IV,III,II,I]
--     xs <- List.permutations [2,1,2,1]
--     ps <- List.permutations [1,2,2,3]
--     return $ secondChord xs ss ps

-- g, a, d, e
-- scs = concatSeq $ do
--     xs <- return [1,1,1,1]
--     ss <- return [IV,III,II,I]
--     ps <- List.permutations [1,1,1,2]
--     return $ secondChord xs ss ps
```

```
ch  =  instant
--    ||| (setDynamics p . setPart DoubleBass  $ naturalHarmonic I 4)
    ||| (setDynamics p . setPart (Cello 2)   $ naturalHarmonic IV 2)
    ||| (setDynamics p . setPart (Viola 2)   $ naturalHarmonic III 1)
    ||| (setDynamics p . setPart (Violin 2)  $ naturalHarmonic II 3)
    ||| (setDynamics p . setPart (Violin 2)  $ naturalHarmonic IV 1)
```

```
a  =  instant
    ||| (setDynamics ppp . setPart DoubleBass  $ naturalHarmonic IV 4)
    ||| (setDynamics ppp . setPart (Cello 1)   $ naturalHarmonic IV 3)
    ||| (setDynamics ppp . setPart (Viola 1)   $ naturalHarmonic III 2)
    ||| (setDynamics ppp . setPart (Violin 1)  $ naturalHarmonic II  1)
    ||| (setDynamics ppp . setPart (Violin 2)  $ naturalHarmonic II  1)
```

```
b  =  instant
    ||| (setDynamics ppp . setPart DoubleBass  $ naturalHarmonic IV 4)
    ||| (setDynamics ppp . setPart (Cello 2)   $ naturalHarmonic IV 3)
    ||| (setDynamics ppp . setPart (Viola 2)   $ naturalHarmonic III 2)
    ||| (setDynamics ppp . setPart (Violin 2)  $ naturalHarmonic II  1)
    ||| (setDynamics ppp . setPart (Violin 1)  $ naturalHarmonic II  1)

d  =  instant
    ||| (setDynamics pp . setPart DoubleBass $ naturalHarmonic I 3)
    ||| (setDynamics ppp . setPart (Cello 2)  $ naturalHarmonic II 2)
    ||| (setDynamics ppp . setPart (Cello 2)  $ naturalHarmonic III 3)
    ||| (setDynamics ppp . setPart (Viola 2)  $ naturalHarmonic III 1)
    ||| (setDynamics ppp . setPart (Viola 2)  $ naturalHarmonic II 2)

mm = instant
    ||| (setDynamics pp . delay 0   . stretch 2 . mapPitch (+ 7) . tonality . setPart (Cello
nn = instant
    ||| (setDynamics pp . delay 0   . stretch 2 . mapPitch (+ 7) . tonality . setPart (Cello

m = instant
    ||| (setDynamics pp . delay 0.0 . stretch 2.1 . mapPitch (+ 0) . tonality . setPart (Cel
    ||| (setDynamics pp . delay 1.1 . stretch 2.2 . mapPitch (+ 0) . tonality . invert . set

n = instant
    ||| (setDynamics pp . delay 5   . stretch 5 . mapPitch (+ 7) . tonality . setPart (Cello
    ||| (setDynamics pp . delay 0.0 . stretch 4 . mapPitch (+ 0) . tonality . setPart (Cello
```

Sections

```
introHarm :: Int -> Score Dur Cue
introHarm sect = stretch 1 $ instant
    >>> stretch 3 a >>> stretch 2.2 g >>> stretch 3.4 a >>> introHarm sect
    where a  = setPart (Cello sect) . setDynamics ppp $ naturalHarmonic IV 1
          g  = setPart (Viola sect) . setDynamics ppp $ naturalHarmonic II 1
```

```
introHarmTrem :: Int -> Score Dur Cue
introHarmTrem sect = stretch 2 $ instant
    >>> stretch 3 a >>> stretch 2.4 g
    where a  = setPart (Cello sect) . setDynamics ppp $ naturalHarmonicTrem IV 0 1
          g  = setPart (Viola sect) . setDynamics ppp $ naturalHarmonicTrem II 0 1

introHarmVln :: Int -> Score Dur Cue
introHarmVln sect = stretch 1 $ instant
    >>> stretch 3 d >>> stretch 2.2 d2 >>> stretch 3.4 a >>> introHarm sect
    where d  = setPart (Violin sect) . setDynamics ppp $ naturalHarmonic I  2
          d2 = setPart (Violin sect) . setDynamics ppp $ naturalHarmonic II 1
          a  = setPart (Violin sect) . setDynamics ppp $ naturalHarmonic III 1

intro2 = instant
    ||| (before 30 $ introHarm 1)
    ||| (delay 15  . before 30 $ introHarm 2)
    ||| (delay 25  . stretch 5 $ db)
    ||| (delay 35  . before 35 $ introHarm 1)
    ||| (delay 50  . before 35 $ introHarm 2)
    ||| (delay 60  . stretch 5 $ db2)
    ||| (delay 80  . before 15 $ introHarmTrem 1)
    ||| (delay 90  . before 30 $ introHarm 2)
    ||| (delay 100 . stretch 5 $ db)
    ||| (delay 110 . before 30 $ introHarm 1)
    ||| (delay 125 . before 15 $ introHarmTrem 2)

    ||| (delay 75 . before 25 $ introHarmVln 1)
    ||| (delay 95 . before 30 $ introHarmVln 1)

    where
        db  = setPart DoubleBass . setDynamics ppp . stretch 4  $ naturalHarmonic III 4
        db2 = setPart DoubleBass . setDynamics ppp . stretch 4  $ naturalHarmonic IV 4
```

```
-- TODO redo completely
intro = instant
    ||| (before 40 . stretch 4 . loopOverlayAll $ [a, b])
    ||| rest 5 >>> mm >>> rest 5 >>> nn


-- TODO expand
middle = instant
    ||| stretch 10 d
    ||| (setPart (Cello  1) . setDynamics p . octaveDown . tonality . patternMelody) (patter




middle2 = compress 1.1 . reverse $ instant
    ||| (setDynamics mf . {-delay 0.3 . -}stretch 2.1 . octaveUp . tonality . setPart (Violi
    ||| (setDynamics mf . {-delay 0.2 . -}stretch 2.2 . octaveUp . tonality . setPart (Violi
    ||| (setDynamics mf . {-delay 0.1 . -}stretch 2.5 . fifthUp  . tonality . setPart (Violi
    ||| (setDynamics mf . {-delay 0.4 . -}stretch 2.9 . fifthUp  . tonality . setPart (Violi
    ||| (setDynamics mf . {-delay 0.6 . -}stretch 3.5 . id       . tonality . setPart (Violi
    ||| (setDynamics mf . {-delay 0.5 . -}stretch 4.1 . id       . tonality . setPart (Viola

middle2b = middle2 >>> (before 40 . stretch 0.6 . reverse $ middle2)

end = compress 1.1 $ instant
    ||| (setDynamics f . delay 0.3 . stretch 2.1 . duodecUp . tonality . setPart (Violin 1)
    ||| (setDynamics f . delay 0.2 . stretch 2.2 . duodecUp . tonality . setPart (Violin 2)
    ||| (setDynamics f . delay 0.1 . stretch 2.5 . octaveUp . tonality . setPart (Violin 3)
    ||| (setDynamics f . delay 0.4 . stretch 2.9 . octaveUp . tonality . setPart (Violin 4)
    ||| (setDynamics f . delay 0.6 . stretch 3.5 . fifthUp . tonality . setPart (Violin 2) $
    ||| (setDynamics f . delay 0.5 . stretch 4.1 . fifthUp . tonality . setPart (Viola 1) $
    ||| (setDynamics mf . concatSeq $ map (\x -> stretch 20 . setPart (Cello 1)  $ stoppedSt
    ||| (setDynamics mf . concatSeq $ map (\x -> stretch 30 . setPart (Cello 1)  $ stoppedSt
    ||| (setDynamics mf . stretch 80 . setPart DoubleBass $ openString IV)
```

```
test = intro2 >>> middle >>> middle2b >>> end

-- TODO coda?
```

Test

```
test :: Score Dur Cue
--test = instant

allHarmonics :: Score Dur Cue
allHarmonics = stretch (1/3) $ instant
    >>> concatSeq [ setPart DoubleBass $ naturalHarmonic str pos | str <- enumFrom I, pos <-


allOpenStrings :: Score Dur Cue
allOpenStrings = stretch (1/3) $ instant
    >>> concatSeq [ setPart DoubleBass   $ openString str | str <- enumFrom I ]
    >>> concatSeq [ setPart (instr part) $ openString str | instr <- [Cello, Viola, Violin]
                                                          , part  <- [2,1]
                                                          , str   <- enumFrom I ]

main = writeMidi "Passager.mid" (render test)
```