gpcsets:
Pitch Class Sets for Haskell
Library Documentation

Bruce H. McCosar

May 9, 2009

Contents

Chapter 1

Data.PcSets

1.1 Introduction

1.1.1 The Module Export List

```
{-|
  The basic module for working with Pitch Class Sets of all kinds,
  including Tone Rows.  The broadest datatypes ('GenSet' and 'GenRow')
  can model any equal temperament system; the standard datatypes
  ('StdSet' and 'StdRow') model /12 Tone Equal Temperament/ (12-TET).
-}
module Data.PcSets
  (
    -- * Classes
      PcSet (modulus, elements, pMap)
    , Selective (complement)
    , Inclusive (reconcile)
    -- * Types
    -- ** Selective (Sets)
    , GenSet
    , StdSet
    -- ** Inclusive (Rows)
    , GenRow
    , StdRow
    -- * Constructors
    -- ** Selective (Sets)
    , genset
    , stdset
    -- ** Inclusive (Rows)
    , genrow
    , stdrow
    -- * General Operations (All Sets)
```

```
    -- ** Transformations
    , transpose
    , invert
    , invertXY
    , zero
    -- ** Permutations
    , retrograde
    , rotate
    -- * Selective Set Operations
    -- ** Systematically Equivalent Forms
    , sort
    , normal
    , reduced
    , prime
    -- ** Scalar Quantities
    , cardinality
    , binaryValue
    -- ** Vector Quantities
    , avec
    , cvec
    , ivec
    -- * Inclusive Set (Tone Row) Operations
    , rowP
    , rowR
    , rowI
    , rowRI
  )
where
```

1.1.2 The Module Import List

```
import qualified Data.List (nub, sort, sortBy, elemIndices)
```

1.2 Classes

1.2.1 PcSet

```
{-|
  The broadest class of Pitch Class Set. All members of this class
  have a 'modulus' which restricts their 'elements' in some way. They
  also have 'pMap', a method for lifting integer list functions to act
  on set elements.  The 'modulus' corresponds to the underlying system
  of equivalent pitch classes, for example, 12-TET = modulus 12.
-}
class PcSet a where
```

```
  -- | Determines the range of possible 'elements' of the set,
  --   from 0 to (m-1).  If m = 0, the set can only be empty.
  modulus  :: a -> Int
  -- | Returns the elements of the set as a list.
  elements :: a -> [Int]
  -- | Maps an integer list function across the members of the set,
  --   and returns the results in a new set of the same type.
  pMap     :: ([Int] -> [Int]) -> a -> a
```

1.2.2 Selective PcSets (Pitch Class Sets)

```
{-|
  Selective Pitch Class Sets can have 'elements' in a range of values
  permitted by their 'modulus'. They can have as few as 0 (the empty
  set) or as many as all. The set 'complement' operation only makes
  sense for 'Selective' sets.
-}
class PcSet a => Selective a where
  -- | Returns a new PcSet which is the complement of the original:
  --   it contains all the 'elements' which the original does not.
  complement :: a -> a
```

1.2.3 Inclusive PcSets (Tone Rows)

```
{-|
  Inclusive Pitch Class Sets, or Tone Rows, have all the possible
  'elements' permitted by their 'modulus'. The most important
  characteristic of a Tone Row is not its 'elements', but the
  /ordering/ of its 'elements'.
-}
class PcSet a => Inclusive a where
  -- | Transposes the 'elements' of a Tone Row so that the first
  --   element is /n/.
  reconcile :: Int -> a -> a
  reconcile n ps = transpose r ps
    where
      firstElement = head . elements $ ps
      r = n - firstElement
```

1.3 Types

1.3.1 GenSet: General Pitch Class Sets

```
{-|
  General Pitch Class Set.  This represents a Pitch Class Set that
  can have a 'modulus' of any positive integer value, representing
  the number of equivalent pitch classes in a given system; for
  example, 19-TET would be a modulus 19 set.  The members of a the
  set can be as few as zero and as many as all possible values.
-}
data GenSet = GenSet Int [Int]
  deriving (Eq,Ord,Show)
```

text

```
instance PcSet GenSet where
  modulus  (GenSet m _ ) = m
  elements (GenSet _ es) = es
  pMap f   (GenSet m es) = genset m . f $ es
```

text

```
instance Selective GenSet where
  complement (GenSet 0 _ ) = GenSet 0 []
  complement (GenSet m es) = GenSet m cs
    where cs = filter (`notElem` es) [0..(m-1)]
```

1.3.2 StdSet: Standard Pitch Class Sets

```
{-|
  Standard Pitch Class Set.  This represents the traditional
  definition of a pitch class set, based on 12-TET, with the
  pitch classes numbered C = 0, C#/Db = 1, D = 2, and so on
  up to B = 11.  This set can have anywhere from zero to 12
  members (the empty set vs. the chromatic scale).
-}
data StdSet = StdSet [Int]
  deriving (Eq,Ord,Show)
```

text

```
instance PcSet StdSet where
  modulus  (StdSet _ ) = 12
  elements (StdSet es) = es
  pMap f   (StdSet es) = stdset . f $ es
```

text

```
instance Selective StdSet where
  complement (StdSet es) = StdSet cs
    where cs = filter (`notElem` es) [0..11]
```

1.3.3 GenRow: General Tone Rows

```
{-|
  General Tone Row.  A /Tone Row/ is a collection of all possible
  Pitch Class Set 'elements' within a given 'modulus'.  Since it
  contains all elements, the significant information in this type
  of set is the ordering of the 'elements'.  This set always has
  a length equal to its 'modulus'.
-}
data GenRow = GenRow [Int]
  deriving (Eq,Ord,Show)
```

text

```
instance PcSet GenRow where
  modulus  (GenRow es) = length es
  elements (GenRow es) = es
  pMap f   (GenRow es) = genrow (length es) . f $ es
```

text

```
instance Inclusive GenRow
```

1.3.4 StdRow: Standard Tone Rows

```
{-|
  Standard Tone Row.  This is the traditional Tone Row, a collection
  of all the elements @[0..11]@, based on 12-TET.  As with 'GenRow',
  the most significant information in this type of set is the ordering
  of the elements.  Since this is always a complete set, this set
  always has a length of 12.
-}
data StdRow = StdRow [Int]
  deriving (Eq,Ord,Show)
```

text

```
instance PcSet StdRow where
  modulus  (StdRow _ ) = 12
  elements (StdRow es) = es
  pMap f   (StdRow es) = stdrow . f $ es
```

text

```
instance Inclusive StdRow
```

1.4 Constructors

1.4.1 genset

```
{-|
  Constructor for General Pitch Class Sets.  This constructor accepts
  any @Int@ value for 'modulus', and any @[Int]@ values for an input
  list.  Zero 'modulus' always returns an empty set; a negative 'modulus'
  is always taken as positive (since the number represent the /absolute/
  size of the equivalence class).
-}
genset :: Int -> [Int] -> GenSet
genset 0 _ = GenSet 0 []
genset m_in es = GenSet m (f es)
  where
    m = abs m_in
    f = Data.List.nub . map (`mod` m)
```

1.4.2 stdset

```
{-|
  Constructor for Standard Pitch Class Sets.  This constructor accepts
  any @[Int]@ values for elements.  The 'modulus' is always 12 (12-TET).
-}
stdset :: [Int] -> StdSet
stdset es = StdSet ps
  where ps = elements $ genset 12 es
```

1.4.3 genrow

```
{-|
  Constructor for General Tone Rows.  This constructor accepts any @Int@
  value for 'modulus', and any @[Int]@ values for an input list.  Zero
  'modulus' always returns an empty set; a negative 'modulus' is always
  taken as positive (see 'GenSet').  If the input list of 'elements' is
  incomplete, the remaining 'elements' are filled in at the end, in order.
-}
genrow :: Int -> [Int] -> GenRow
genrow m es = GenRow (os ++ cs)
  where
    ps = genset m es
    os = elements ps
    cs = elements $ complement ps
```

1.4.4 stdrow

```
{-|
  Constructor for Standard Tone Rows.  This constructor accepts any @[Int]@
  values for an input list.  The 'modulus' is always 12 (12-TET).  If the
  input list of 'elements' is incomplete, the remaining 'elements' are filled
  in at the end, in order.
-}
stdrow :: [Int] -> StdRow
stdrow es = StdRow ts
  where ts = elements $ genrow 12 es
```

1.5 General Operations (All Sets)

1.5.1 Transformations

1.5.1.1 transpose

```
-- | Returns a new 'PcSet' which is the original transposed by /n/.
transpose :: PcSet a => Int -> a -> a
transpose = pMap . map . (+)
```

1.5.1.2 invert

```
{-|
  Returns a new 'PcSet' which is the /standard inverse/ of the original,
  that is, about an axis containing pitch class 0.
-}
invert :: PcSet a => a -> a
invert ps = pMap (map (m -)) ps
  where m = modulus ps
```

1.5.1.3 invertXY

```
{-|
  Inversion around an axis specified by pitch classes /x/ and /y/.
  This inverts the set in such a way that /x/ becomes /y/ and /y/
  becomes /x/.
-}
invertXY :: PcSet a => Int -> Int -> a -> a
invertXY x y = transpose (x + y) . invert
```

1.5.1.4 zero

```
{-|
  Returns a new 'PcSet' in which the elements have been transposed
  so that the first element is zero.
-}
zero :: PcSet a => a -> a
zero ps = transpose (-n) ps
  where n = head . elements $ ps
```

1.5.2 Permutations

1.5.2.1 retrograde

```
-- | Returns a new 'PcSet' with the elements of the original reversed.
retrograde :: PcSet a => a -> a
retrograde = pMap reverse
```

1.5.2.2 rotate

```
-- | Returns a new 'PcSet' with the elements shifted /n/ places to the left.
rotate :: PcSet a => Int -> a -> a
rotate n ps = pMap nShift ps
  where
    nShift = take sameLength . drop offset . cycle
    sameLength = (length . elements) ps
    offset = n `mod` sameLength
```

1.6 Selective Set Operations

1.6.1 Systematically Equivalent Forms

1.6.1.1 sort

```
{-|
  Returns a 'Selective' 'PcSet' in which the elements of the original
  have been sorted in ascending order. (Note this is restricted to Sets,
  as sorting a Tone Row produces only an ascending chromatic scale.)
-}
sort :: (PcSet a, Selective a) => a -> a
sort = pMap Data.List.sort
```

1.6.1.2 normal

```
{-|
  Returns a 'Selective' 'PcSet' in which the elements of the original have
  been put into /normal form/.  This can be defined as an ascending order
  in which the elements fit into the smallest overall interval. In the event
  of a tie, the arrangement with the closest leftward packing is chosen.
-}
normal :: (PcSet a, Selective a) => a -> a
normal = nform . bestPack . pcsArrangements
```

1.6.1.3 reduced

```
{-|
  Returns a 'Selective' 'PcSet' in which the elements of the original
  have been put into /reduced form/.  This can be thought of as the
  'normal' form, transposed so that the first element starts on 'zero'.
-}
reduced :: (PcSet a, Selective a) => a -> a
reduced = rform . bestPack . pcsArrangements
```

1.6.1.4 prime

```
{-|
  Returns a 'Selective' 'PcSet' in which the elements of the original
  have been put into /prime form/.  A prime form is able to generate
  all the members of its set family through the some combination of the
  operations 'transpose', 'invert', and simple permutation.
-}
prime :: (PcSet a, Selective a) => a -> a
prime ps = if i_val < o_val then inversion else original
  where
    original = reduced ps
    inversion = reduced $ invert ps
    o_val = binaryValue original
    i_val = binaryValue inversion
```

1.6.2 Scalar Quantities

1.6.2.1 cardinality

```
-- | Returns the number of elements in a 'Selective' 'PcSet'.
cardinality :: (PcSet a, Selective a) => a -> Int
cardinality = length . elements
```

1.6.2.2 binaryValue

```
{-|
  Binary Value. For a given 'Selective' 'PcSet', this returns a
  /unique/ number relating to the elements of the set -- a measure
  of the "leftward packing" of the sorted set (overall closeness
  of each element to zero).
-}
binaryValue :: (PcSet a, Selective a) => a -> Integer
binaryValue = sum . map (2 ^) . elements
```

1.6.3 Vector Quantities

1.6.3.1 avec

```
{-|
  Ascending Vector. If the elements of a 'Selective' 'PcSet' are
  taken to be in strictly ascending order, the ascending vector is
  the interval difference between each element.
-}
avec :: (PcSet a, Selective a) => a -> [Int]
avec ps = map (`mod` m) $ zipWith (-) rs os
  where
    m = modulus ps
    os = elements ps
    rs = elements . rotate 1 $ ps
```

1.6.3.2 cvec

```
{-|
  Common Tone Vector: finds the number of common tones for each possible
  value of /n/ in the operation 'transpose' /n/ . 'invert'. Returns a list
  where element 0 is the number of common tones with /n/=0, element 1 is
  with /n/=1, and so on.
-}
cvec :: (PcSet a, Selective a) => a -> [Int]
cvec ps = count . concatMap f $ es
  where
    m = modulus ps
    es = elements ps
    count cs = map (\n ->
      length (Data.List.elemIndices n cs)) [0..(m-1)]
    f x = map (\y -> (x + y) `mod` m) es
```

1.6.3.3 ivec

```
{-|
  Interval Vector.  Each element of the interval vector represents
  the number of intervals in the set for that particular interval
  class.  Element 0 measures the number of 1-interval leaps;
  element 1 measures the number of 2-interval leaps, and so on,
  up to half of the modulus /m/.
-}
ivec :: (PcSet a, Selective a) => a -> [Int]
ivec ps = if m == 0 then []
    else pivotguard . spacefold . count . intervals . elements $ ps
  where
    m = modulus ps
    -- pivotguard: compensates for even lists, where the largest possible
    -- interval is equal to its inverse (and thereby counted twice, here).
    pivotguard es = if odd m then es
      else init es ++ [last es `div` 2]
    -- spacefold: wraps interval list to interval classes
    spacefold = take (m `div` 2) . flipSum
    flipSum es = zipWith (+) es (reverse es)
    -- count: counts each occurrence of each possible diff
    count ivs = map (g ivs) [1..(m-1)]
    g ivs n = length (Data.List.elemIndices n ivs)
    -- intervals: returns recursive list of diffs
    intervals [] = []
    intervals (e:es) = diffs e es ++ intervals es
    -- diffs: interval difference between pitches
    diffs = map . f
    f a b = (b - a) `mod` m
```

1.7 Inclusive Set (Tone Row) Operations

1.7.1 Permutation-Transformations

1.7.1.1 rowP

```
{-|
  Returns a new Tone Row in which the elements are /Prograde/
  (in their original order) and transposed so that the first
  element is /n/.
-}
rowP :: (PcSet a, Inclusive a) => Int -> a -> a
rowP = reconcile
```

1.7.1.2 rowR

```
{-|
  Returns a new Tone Row in which the elements are /Retrograde/
  (reversed compared to their original order) and transposed so
  that the first element is /n/.
-}
rowR :: (PcSet a, Inclusive a) => Int -> a -> a
rowR = (. retrograde) . reconcile
```

1.7.1.3 rowI

```
{-|
  Returns a new Tone Row in which the elements have been /Inverted/
  (see 'invert') and transposed so that the first element is /n/.
-}
rowI :: (PcSet a, Inclusive a) => Int -> a -> a
rowI = (. invert) . reconcile
```

1.7.1.4 rowRI

```
{-|
  Returns a new Tone Row in which the elements are both /Retrograde/
  and /Inverted/, and transposed so that the first element is /n/.
-}
rowRI :: (PcSet a, Inclusive a) => Int -> a -> a
rowRI = (. (invert . retrograde)) . reconcile
```

1.8 Not Exported

1.8.1 Related to Normal, Reduced, and Prime

```
data (PcSet a, Selective a) => Candidate a = Candidate
  {
    idx :: Integer,
    nform :: a,
    rform :: a
  }
```

```
interview :: (PcSet a, Selective a) => a -> Candidate a
interview ps = Candidate
  {
    idx = binaryValue zs,
```

```
    nform = ps,
    rform = zs
  }
  where zs = zero ps
```

```
sortFunction :: (PcSet a, Selective a) =>
  Candidate a -> Candidate a -> Ordering
sortFunction a b = compare (idx a) (idx b)
```

```
bestPack :: (PcSet a, Selective a) => [a] -> Candidate a
bestPack arrs = head (Data.List.sortBy sortFunction candidates)
  where candidates = [interview ps | ps <- arrs]
```

```
pcsArrangements :: (PcSet a, Selective a) => a -> [a]
pcsArrangements ps = if n == 0
  then [ps] -- only one possible arrangement for nothing.
  else take n $ iterate f (sort ps)
  where
    n = cardinality ps
    f = rotate 1
```

Chapter 2

Data.PcSets.Catalog

text

```
module Data.PcSets.Catalog where
```

text

```
test :: Int
test = 0
```

final

Chapter 3

Data.PcSets.Compact

text

```
module Data.PcSets.Compact where
```

text

```
test :: Int
test = 0
```

final

Chapter 4

Data.PcSets.Notes

text

```
module Data.PcSets.Notes where
```

text

```
test :: Int
test = 0
```

final

Chapter 5

Data.PcSets.Svg

text

```
module Data.PcSets.Svg where
```

text

```
test :: Int
test = 0
```

final