gpcsets:
Pitch Class Sets for Haskell
Test Suite Documentation

Bruce H. McCosar

May 14, 2009

Contents

Chapter 1

Data.PcSets

1.1 Introduction

1.1.1 Structure of the Test Suite

In the first section, "Standard Candles" (1.2), some basic properties of the functions provided in this module are tested against a wide range of possible sets. Since a general pitch class set can have any integer modulus (and thereby almost any length in that range), these tests have to be simple to avoid excessive run times.

In the second section, "Arbitrary Sets" (1.3), more complicated tests are applied against the four different types in the Data.PcSets module. Samples from these types are generated using QuickCheck, and have specifically limited ranges for modulus and set membership.

1.1.2 Imports

The library gpcsets was developed using GHC version 6.10.2 on Ubuntu 9.04. If you have a reasonably modern version of GHC, you should be able to use this library and run the examples.

The test suite, however, uses QuickCheck 2.1.0.1:

```
import Test.QuickCheck
```

Some additional help is needed from the standard Data.List library:

```
import qualified Data.List (elemIndex, permutations, sort)
import Data.Function (on)
```

I'm importing Data.PcSets "qualified" to make it clear, in the test code, which of the module's functions are involved in each test.

```
import qualified Data.PcSets as P
```

1.2 Standard Candles

This section also serves as a demonstration for each of the functions in the module.

1.2.1 By Function

1.2.1.1 modulus

An arbitrary Int modulus should produce no error. The modulus of the set is always the absolute value of the input modulus. (Note that arbitrary Tone Rows will be tested in a later section.)

```
propArbitraryIntModulus :: Int -> [Int] -> Bool
propArbitraryIntModulus m es =
    P.modulus ps == abs m
  where ps = P.genset m es
```

1.2.1.2 elements

An input list of arbitrary integers should produce no error. The final list will always have a cardinality less than or equal to the modulus; the elements will always be between zero and the modulus minus one. (Note Haskell's lazy "all" evaluation will safely pass the empty set.)

```
propArbitraryElements :: Int -> [Int] -> Bool
propArbitraryElements m_in es =
    P.cardinality ps `inRange` m && all (`inRange` (m - 1)) (P.elements ps)
  where
    m = abs m_in
    ps = P.genset m_in es
    x `inRange` y = x >= 0 && x <= y
```

1.2.1.3 complement

The complement of the chromatic spectrum in a given modulus should be the empty set, and vice versa.

```
propChromaticEmptyComplement :: Int -> Bool
propChromaticEmptyComplement m_in =
    P.complement cs == ns && P.complement ns == cs
  where
    -- The modulus has to be limited, or this test will take too long.
    m = abs m_in `mod` 144
    cs = P.genset m [0..m]
    ns = P.genset m []
```

1.2.1.4 reconcile

"Reconciling" a Tone Row should transpose the row so that the first element corresponds to the input value (relative to the set modulus). This should work for any Int.

Note: negative integers count as steps down the list, eg, an input value of -1 in a modulus 12 set corresponds to element 11.

```
propReconciliation :: Int -> [Int] -> Int -> Bool
propReconciliation m_in es n =
    (head . P.elements . P.reconcile n) tr == n `mod` m
  where
    -- The modulus has to be limited, or this test will take too long.
    m = (abs m_in `mod` 143) + 1 -- Also, always a non-empty row.
    tr = P.genrow m es
```

1.2.1.5 pMap

This function is tested indirectly, as it is the basis for many of the fundamental operations that follow.

1.2.1.6 transpose

Transposing any set by 0 should yield the same set.

```
propZeroTransposition :: Int -> [Int] -> Bool
propZeroTransposition m es =
    P.transpose 0 ps == ps
  where ps = P.genset m es
```

Transposing any set by some multiple of the modulus should give the same set.

```
propModulusTransposition :: Int -> [Int] -> Bool
propModulusTransposition m es =
    all (== ps) [P.transpose (k * m) ps | k <- [0..10]]
  where ps = P.genset m es
```

Transposing by $+n$ and then $-n$ should give the same set.

```
propTranspositionReversal :: Int -> [Int] -> Int -> Bool
propTranspositionReversal m es n =
    (P.transpose (-n) . P.transpose n) ps == ps
  where ps = P.genset m es
```

If m is the modulus, transposing by n and then $m - n$ should give the same set.

```
propTranspositionCompletion :: Int -> [Int] -> Int -> Bool
propTranspositionCompletion m es n =
    (P.transpose (m - n) . P.transpose n) ps == ps
  where ps = P.genset m es
```

1.2.1.7 invert

Inverting twice should produce the original set.

```
propDoubleInversion :: Int -> [Int] -> Bool
propDoubleInversion m es =
    (P.invert . P.invert) ps == ps
  where ps = P.genset m es
```

Standard invert should leave any occurrences of zero unchanged. Also, if zero doesn't occur in the original set, then zero should not appear in the inverse.

```
propStandardInversionZero :: Int -> [Int] -> Bool
propStandardInversionZero m es =
    -- True for Just n == Just n and Nothing == Nothing.
    zFind ps == (zFind . P.invert) ps
  where
    ps = P.genset m es
    zFind = Data.List.elemIndex 0 . P.elements
```

1.2.1.8 invertXY

In this section, invertXY is tested indirectly, through invert, above. More thorough tests occur in "Arbitrary Sets" (Section 1.3).

1.2.1.9 zero

Zero should transpose the set so that the first element is zero.

```
propOperationZero :: Int -> [Int] -> Bool
propOperationZero m es =
    (Data.List.elemIndex 0 . P.elements . P.zero) ps == expectedAnswer
  where
    ps = P.genset m es
    expectedAnswer = if P.cardinality ps > 0 then Just 0 else Nothing
```

1.2.1.10 retrograde

Applying the retrograde operation twice should produce the original set.

```
propDoubleRetrograde :: Int -> [Int] -> Bool
propDoubleRetrograde m es =
    (P.retrograde . P.retrograde) ps == ps
  where ps = P.genset m es
```

1.2.1.11 rotate

Rotating by zero should produce the original set.

```
propZeroRotation :: Int -> [Int] -> Bool
propZeroRotation m es =
    P.rotate 0 ps == ps
  where ps = P.genset m es
```

Rotating by the set cardinality should produce the original set.

```
propLengthRotation :: Int -> [Int] -> Bool
propLengthRotation m es =
    P.rotate (P.cardinality ps) ps == ps
  where ps = P.genset m es
```

Rotating by $+n$ and $-n$ should produce the original set.

```
propRotationReversal :: Int -> [Int] -> Int -> Bool
propRotationReversal m es n =
    (P.rotate (-n) . P.rotate n) ps == ps
  where ps = P.genset m es
```

Rotating by $+n$ and $c - n$, where c is the cardinality, should produce the original set.

```
propRotationCompletion :: Int -> [Int] -> Int -> Bool
propRotationCompletion m es n =
    (P.rotate (c - n) . P.rotate n) ps == ps
  where
    ps = P.genset m es
    c = P.cardinality ps
```

1.2.1.12 sort

Sorting a sorted set should produce the same sorted set.

```
propDoubleSortEquivalence :: Int -> [Int] -> Bool
propDoubleSortEquivalence m es =
    (P.sort . P.sort) ps == P.sort ps
  where ps = P.genset m es
```

1.2.1.13 normal

More thorough testing of the normal function occurs in "Arbitrary Sets" (Section 1.3). Here the "Standard Candles" are particular, known examples of the normal operation.

The normal of a major triad in any inversion should be the same major triad in first inversion.

```
propNormalOfMajorTriad :: Int -> Bool
propNormalOfMajorTriad n =
    all (== maj) $ map P.normal perms
  where
    maj = P.transpose n (P.stdset [0,4,7]) -- some major triad, 1st inv.
    perms = map P.stdset . Data.List.permutations . P.elements $ maj
```

The normal of a *minor* triad in any inversion should be the same minor triad in first inversion.

```
propNormalOfMinorTriad :: Int -> Bool
propNormalOfMinorTriad n =
    all (== minor) $ map P.normal perms
  where
    minor = P.transpose n (P.stdset [0,3,7]) -- some minor triad, 1st inv.
    perms = map P.stdset . Data.List.permutations . P.elements $ minor
```

1.2.1.14 reduced

Like normal, reduced is tested more thoroughly in the "Arbitrary Sets" Section (1.3). Here some specific examples are tested.

The reduced form of a major triad—in any transposition, and in any permutation—should be [0,4,7].

```
propReducedMajorTriad :: Int -> Bool
propReducedMajorTriad n =
    all (== [0,4,7]) $ map (P.elements . P.reduced) perms
  where
    maj = P.transpose n (P.stdset [0,4,7]) -- some major triad, 1st inv.
    perms = map P.stdset . Data.List.permutations . P.elements $ maj
```

The reduced form of a *minor* triad—in any transposition, and in any permutation—should be [0,3,7].

```
propReducedMinorTriad :: Int -> Bool
propReducedMinorTriad n =
    all (== [0,3,7]) $ map (P.elements . P.reduced) perms
  where
    minor = P.transpose n (P.stdset [0,3,7]) -- some minor triad, 1st inv.
    perms = map P.stdset . Data.List.permutations . P.elements $ minor
```

1.2.1.15 prime

In this section, the prime operation is tested against known prime forms.

The prime form for both major and minor triads is [0,3,7]. This should not change with any transposition or permutation of the major and minor triads.

```
propMajorMinorPrimes :: Int -> Int -> Bool
propMajorMinorPrimes m n =
    all (== [0,3,7]) $ map (P.elements . P.prime) perms
  where
    ma = P.transpose m (P.stdset [0,4,7])
    mi = P.transpose n (P.stdset [0,3,7])
    perms = concatMap f [ma,mi]
    f = map P.stdset . Data.List.permutations . P.elements
```

The prime form for the major scale [0,2,4,5,7,9,11] is [0,1,3,5,6,8,10]. This should not change with any transposition or permutation of the major scale elements. (Permutation isn't tested here, only rotation and reversal ... testing the 7! permutations for the major scale would be rather computation-intensive.)

```
propMajorScalePrime :: Int -> Int -> Bool
propMajorScalePrime m n =
    all (== [0,1,3,5,6,8,10]) [f ms | f <- ops]
  where
    ms = P.transpose m (P.stdset [0,2,4,5,7,9,11])
    ops = [P.elements . P.prime . g . h |
           g <- [id, P.rotate n],
           h <- [id, P.retrograde]]
```

1.2.1.16 cardinality

Cardinality is fundamental to many of the other set operations; it was specifically tested in an earlier section (1.2.1.2).

1.2.1.17 binaryValue

Because the binaryValue function involves an exponential, tests on sets of arbitrarily large modulus would give unacceptable run times for the test suite. Therefore, full testing of this function will occur in "Arbitrary Sets" (Section 1.3).

1.2.1.18 avec

Two specific cases will be tested in this section. The 12-tone chromatic scale (sorted ascending) should give an interval vector of [1,1,1,1,1,1,1,1,1,1,1,1], regardless of rotation or transposition.

```
propAscendingChromaticAvec :: Int -> Int -> Bool
propAscendingChromaticAvec m n =
    all (all (== 1)) [(P.avec . f) ps | f <- ops]
  where
    ps = P.stdset [0..11]
    ops = [g . h | g <- [id,P.rotate m], h <- [id,P.transpose n]]
```

When the same set is sorted descending, the intervals should all be "11":

```
propDescendingChromaticAvec :: Int -> Int -> Bool
propDescendingChromaticAvec m n =
    all (all (== 11)) [(P.avec . f) ps | f <- ops]
  where
    ps = (P.retrograde . P.stdset) [0..11]
    ops = [g . h | g <- [id,P.rotate m], h <- [id,P.transpose n]]
```

1.2.1.19 cvec

One specific case will be tested in this section. The 12-tone chromatic scale should have a combination vector of all 12's – no matter the rotation, transposition, or permutation, all 12 of the notes should map to other notes of the set under transpose n . invert.

```
propChromaticCombination :: Int -> Int -> Bool
propChromaticCombination m n =
    all (all (== 12)) [(P.cvec . f) ps | f <- ops]
  where
    ps = P.stdset [0..11]
    ops = [g . h . i | g <- [id,P.rotate m],
                       h <- [id,P.transpose n],
                       i <- [id,P.retrograde] ]
```

1.2.1.20 ivec

Two known interval vector cases are tested at once here. For the two sets below, the same interval vector (all 1's) should be returned no matter the transposition or permutation of the set. These are the "All Interval Tetrachords", [0,1,4,6] and [0,1,3,7] (in 12-TET).

```
propAllIntervalTetrachordIvec :: Int -> Bool
propAllIntervalTetrachordIvec n =
    all (all (== 1)) [(P.ivec . f) ps | f <- ops, ps <- series]
  where
    tets1 = map P.stdset (Data.List.permutations [0,1,4,6])
    tets2 = map P.stdset (Data.List.permutations [0,1,3,7])
    series = tets1 ++ tets2
    ops = [id, P.transpose n]
```

1.2.1.21 rowP, rowR, rowI, rowRI

Since these tests involve Tone Rows, these functions will be tested in Section 1.3. A Tone Row can have any integer modulus. However, a Tone Row has *all* the possible elements in the set. This means if a row is generated randomly from a purely arbitrary Int, it could lead to an extraordinarily long list. In section 1.3, the modulus ranges are limited.

1.2.2 General Truths

1.2.2.1 Empty Sets In, Empty Sets Out

None of the functions tested here should fail when given an empty set as input. This includes the null modulus set, GenSet 0 []. The functions tested below should simply return an empty set of the same modulus.

Arbitrary Empty Sets:

```
propEmptySetBulletproof :: Int -> Int -> Bool
propEmptySetBulletproof m n =
    all (== emptyset) [f emptyset | f <- ops]
  where
    emptyset = P.genset m []
    ops = [P.transpose n, P.invert, P.zero, P.retrograde, P.rotate n,
           P.sort, P.normal, P.reduced, P.prime]
```

Null Modulus Sets:

```
propNullModulusBulletproof :: [Int] -> Int -> Bool
propNullModulusBulletproof es n =
    all (== nullset) [f nullset | f <- ops]
  where
    nullset = P.genset 0 es
    ops = [P.transpose n, P.invert, P.zero, P.retrograde, P.rotate n,
           P.sort, P.normal, P.reduced, P.prime]
```

The Null Row (this is the only possible null row; for all other modulus values, the row is nonempty):

```
propNullRowBulletproof :: [Int] -> Int -> Bool
propNullRowBulletproof es n =
    all (== nullrow) [f nullrow | f <- ops]
  where
    nullrow = P.genrow 0 es
    ops = [P.transpose n, P.invert, P.zero, P.retrograde, P.rotate n,
           P.rowP n, P.rowR n, P.rowI n, P.rowRI n]
```

1.2.2.2 Empty Sets to Scalars

The cardinality of any empty set should always be 0 (note that cardinality is not defined for Tone Rows, since they should always be at their maximum length, or modulus):

```
propEmptyCardinalityZero :: Int -> [Int] -> Bool
propEmptyCardinalityZero m es =
    all (== 0) (map P.cardinality emptysets)
  where
    emptysets = [P.genset m [], P.genset 0 es]
```

The binary value of any empty set should always be 0 (note that binary value is not defined for Tone Rows, since all those of the same modulus would be equivalent):

```
propEmptyBinaryValueZero :: Int -> [Int] -> Bool
propEmptyBinaryValueZero m es =
    all (== 0) (map P.binaryValue emptysets)
  where
    emptysets = [P.genset m [], P.genset 0 es]
```

1.2.2.3 Empty Sets to Vectors

The ascending vector for any empty set should always be an empty list:

```
propEmptyAvecOperations :: Int -> [Int] -> Bool
propEmptyAvecOperations m es =
    all (== []) (map P.avec emptysets)
  where
    emptysets = [P.genset m [], P.genset 0 es]
```

The interval vector for any empty set should always be a list half the length of the modulus (rounded down), with all of the entries equal to zero:

```
propEmptyIvecOperations :: Int -> [Int] -> Bool
propEmptyIvecOperations m_in es =
    P.ivec emptyset == zeroline && P.ivec nullset == []
  where
    -- To prevent excessively long tests, the random modulus
    -- is restricted to the range 0 to 144.
    m = abs m_in `mod` 144
    emptyset = P.genset m []
    nullset = P.genset 0 es
    zeroline = replicate (m `div` 2) 0
```

The common tone vector for any empty set should be a list with the same length as the modulus, with all of the entries equal to zero:

```
propEmptyCvecOperations :: Int -> [Int] -> Bool
propEmptyCvecOperations m_in es =
    P.cvec emptyset == zeroline && P.cvec nullset == []
  where
    -- To prevent excessively long tests, the random modulus
    -- is restricted to the range 0 to 144.
    m = abs m_in `mod` 144
    emptyset = P.genset m []
    nullset = P.genset 0 es
    zeroline = replicate m 0
```

1.2.2.4 Empty Row Operations

Any Permutation-Transformation operation on the null row should produce only the null row:

```
propNullRowOperations :: [Int] -> Int -> Bool
propNullRowOperations es n =
    all (== nullrow) [f nullrow | f <- ops]
  where
    nullrow = P.genrow 0 es
    ops = [P.rowP n, P.rowR n, P.rowI n, P.rowRI n]
```

1.3 Arbitrary Sets

1.3.1 QuickCheck

As was discussed in the introduction (Section 1.1.1), the tests in this section involve arbitrary sets generated by QuickCheck. These are generated using only a limited range of possible values for the set modulus and cardinality. This ensures that, even though the tests are complicated, they do not become exponentially long.

However, the price of limiting range is limiting the scope of the test suite. It is hoped the approach of having simple tests over a large input range and complicated tests over a smaller input range provides sufficient coverage.

1.3.1.1 Random General Sets

A random General Pitch Class set should have a modulus between 0 and 144, and a size somewhere between 0 and the maximum value allowed by the modulus.

```
instance Arbitrary P.GenSet where
  arbitrary = do
    m <- choose (-144,144)
    es <- listOf (choose(-144,144))
```

```
    s <- choose (0, abs m)
    let ps = if length es < abs m then es else take s es
    return $ P.genset m ps
```

1.3.1.2 Random Standard Sets

Care is taken, here, so that the input of an arbitrarily large list of integers doesn't always collapse the set to the chromatic scale.

```
instance Arbitrary P.StdSet where
  arbitrary = do
    es <- listOf (choose(-144,144))
    s <- choose (0, 12)
    let ps = if length es < 12 then es else take s es
    return $ P.stdset ps
```

1.3.1.3 Random General Rows

Just as with the General Sets, the General Rows are limited in modulus to the range 0 – 144.

```
instance Arbitrary P.GenRow where
  arbitrary = do
    m <- choose (-24,24)
    es <- listOf (choose(-144,144))
    s <- choose (0, abs m)
    let ps = if length es < abs m then es else take s es
    return $ P.genrow m ps
```

1.3.1.4 Random Standard Rows

Here, the limitation is not so critical. A Standard Tone Row will always have modulus 12 and 12 elements.

```
instance Arbitrary P.StdRow where
  arbitrary = do
    es <- listOf (choose(-144,144))
    s <- choose (0, 12)
    let ps = if length es < 12 then es else take s es
    return $ P.stdrow ps
```

1.3.2 By Function

Some of these tests are duplicates of those that came earlier. However, they ensure that the arbitrary sets aren't "playing by their own rules."

1.3.2.1 modulus

Arbitrary General Set. The cardinality of a general set (of arbitrary modulus) should be between 0 and the modulus.

```
propGenSetModulus :: P.GenSet -> Bool
propGenSetModulus ps =
    0 <= c && c <= P.modulus ps
  where c = P.cardinality ps
```

Arbitrary Standard Set. The cardinality of a standard set should be between 0 and 12.

```
propStdSetModulus :: P.StdSet -> Bool
propStdSetModulus ps =
    0 <= c && c <= 12
  where c = P.cardinality ps
```

Arbitrary General Row. A general Tone Row should always have a number of elements equal to the modulus.

```
propGenRowModulus :: P.GenRow -> Bool
propGenRowModulus tr = (length . P.elements) tr == P.modulus tr
```

Arbitrary Standard Row. A standard Tone Row should always have 12 elements.

```
propStdRowModulus :: P.StdRow -> Bool
propStdRowModulus tr = (length . P.elements) tr == 12
```

1.3.2.2 elements

Arbitrary General Set. The elements of a general set should all be between 0 and $m-1$, where m is the modulus.

```
propGenSetElements :: P.GenSet -> Bool
propGenSetElements ps =
    all (\e -> 0 <= e && e <= m - 1) (P.elements ps)
  where m = P.modulus ps
```

Arbitrary Standard Set. The only elements in a general set should be in the range 0 to 11.

```
propStdSetElements :: P.StdSet -> Bool
propStdSetElements = all (\e -> 0 <= e && e <= 12) . P.elements
```

Arbitrary General Row. A general row should contain every element from 0 to $m-1$, where m is the modulus.

```
propGenRowElements :: P.GenRow -> Bool
propGenRowElements tr =
    (Data.List.sort . P.elements) tr == expected
  where
    m = P.modulus tr
    expected = if m == 0 then [] else [0..(m-1)]
```

Arbitrary Standard Row. A standard row should have all of the elements from 0 to 11.

```
propStdRowElements :: P.StdRow -> Bool
propStdRowElements tr = (Data.List.sort . P.elements) tr == [0..11]
```

1.3.2.3 complement

Arbitrary General Set. The complement of a general set, when added to the original set, should give the chromatic spectrum for that modulus.

```
propGenSetComplement :: P.GenSet -> Bool
propGenSetComplement ps =
    (Data.List.sort . concatMap P.elements) [ps,cs] == expected
  where
    m = P.modulus ps
    expected = if m == 0 then [] else [0..(m-1)]
    cs = P.complement ps
```

Arbitrary Standard Set. The complement of a standard set, when added to the original set, should give the chromatic scale.

```
propStdSetComplement :: P.StdSet -> Bool
propStdSetComplement ps =
    (Data.List.sort . concatMap P.elements) [ps,cs] == [0..11]
  where cs = P.complement ps
```

Arbitrary General or Standard Row. Complement is not defined for this class, since it would make little sense.

1.3.2.4 reconcile

"Reconcile" was tested in "Standard Candles" (Section 1.2). Here it will be tested as part of the Tone Row Operations (rowP, rowI, rowR, rowRI).

1.3.2.5 transpose

Arbitrary General Set. The difference between the elements of the original and transposed set should be the transposition amount (with respect to the modulus).

```
propGenSetTransposeDiff :: Int -> P.GenSet -> Bool
propGenSetTransposeDiff n ps =
    all (== nmod) (zipWith (\a b -> (b - a) `mod` m) os ts)
  where
    m = P.modulus ps
    nmod = n `mod` m
    os = P.elements ps
    ts = (P.elements . P.transpose n) ps
```

Arbitrary Standard Set. The difference between the elements of the original and transposed set should be the transposition amount (mod 12).

```
propStdSetTransposeDiff :: Int -> P.StdSet -> Bool
propStdSetTransposeDiff n ps =
    all (== nmod) (zipWith (\a b -> (b - a) `mod` 12) os ts)
  where
    nmod = n `mod` 12
    os = P.elements ps
    ts = (P.elements . P.transpose n) ps
```

Arbitrary General Row. The difference between the elements of the original and transposed row should be the transposition amount (with respect to the modulus).

```
propGenRowTransposeDiff :: Int -> P.GenRow -> Bool
propGenRowTransposeDiff n tr =
    all (== nmod) (zipWith (\a b -> (b - a) `mod` m) os ts)
  where
    m = P.modulus tr
    nmod = n `mod` m
    os = P.elements tr
    ts = (P.elements . P.transpose n) tr
```

Arbitrary Standard Row. The difference between the elements of the original and transposed row should be the transposition amount (mod 12).

```
propStdRowTransposeDiff :: Int -> P.StdRow -> Bool
propStdRowTransposeDiff n tr =
    all (== nmod) (zipWith (\a b -> (b - a) `mod` 12) os ts)
  where
    nmod = n `mod` 12
    os = P.elements tr
    ts = (P.elements . P.transpose n) tr
```

1.3.2.6 invert

Arbitrary General Set. The sum for each element of a set with its inverse should be zero (within the modulus of the set).

```
propGenSetInverseSum :: P.GenSet -> Bool
propGenSetInverseSum ps =
    all (== 0) (zipWith (\a b -> (a + b) `mod` m) os ts)
  where
    m = P.modulus ps
    os = P.elements ps
    ts = (P.elements . P.invert) ps
```

Arbitrary Standard Set. The sum for each element of a set with its inverse should be zero (mod 12).

```
propStdSetInverseSum :: P.StdSet -> Bool
propStdSetInverseSum ps =
    all (== 0) (zipWith (\a b -> (a + b) `mod` 12) os ts)
  where
    os = P.elements ps
    ts = (P.elements . P.invert) ps
```

Arbitrary General Row. The sum for each element of a row with its inverse should be zero (within the modulus of the set).

```
propGenRowInverseSum :: P.GenRow -> Bool
propGenRowInverseSum tr =
    all (== 0) (zipWith (\a b -> (a + b) `mod` m) os ts)
  where
    m = P.modulus tr
    os = P.elements tr
    ts = (P.elements . P.invert) tr
```

Arbitrary Standard Row. The sum for each element of a row with its inverse should be zero (mod 12).

```
propStdRowInverseSum :: P.StdRow -> Bool
propStdRowInverseSum tr =
    all (== 0) (zipWith (\a b -> (a + b) `mod` 12) os ts)
  where
    os = P.elements tr
    ts = (P.elements . P.invert) tr
```

1.3.2.7 invertXY

General Rules—In each case below, random values are chosen for x and y. These are to be interpreted with respect to the modulus of the set, that is, an arbitrary integer input of 147 is pitch class 3 in modulus 12 space.

The operation invertXY x y should transform all occurrences of x into y, and vice versa. If x does not appear in the original set, then y should not appear in the inverted set; the same applies for y going to x.

Arbitrary General Set.

```
propGenSetIXY :: Int -> Int -> P.GenSet -> Bool
propGenSetIXY x y ps =
    loc xmod os == loc ymod ts && loc ymod os == loc xmod ts
  where
    m = P.modulus ps
    xmod = x `mod` m
    ymod = y `mod` m
    os = P.elements ps
    ts = (P.elements . P.invertXY x y) ps
    loc e es = e `Data.List.elemIndex` es
```

Arbitrary Standard Set.

```
propStdSetIXY :: Int -> Int -> P.StdSet -> Bool
propStdSetIXY x y ps =
    loc xmod os == loc ymod ts && loc ymod os == loc xmod ts
  where
    xmod = x `mod` 12
    ymod = y `mod` 12
    os = P.elements ps
    ts = (P.elements . P.invertXY x y) ps
    loc e es = e `Data.List.elemIndex` es
```

Arbitrary General Row.

```
propGenRowIXY :: Int -> Int -> P.GenRow -> Bool
propGenRowIXY x y tr =
    loc xmod os == loc ymod ts && loc ymod os == loc xmod ts
  where
    m = P.modulus tr
    xmod = x `mod` m
    ymod = y `mod` m
    os = P.elements tr
    ts = (P.elements . P.invertXY x y) tr
    loc e es = e `Data.List.elemIndex` es
```

Arbitrary Standard Row.

```
propStdRowIXY :: Int -> Int -> P.StdRow -> Bool
propStdRowIXY x y tr =
    loc xmod os == loc ymod ts && loc ymod os == loc xmod ts
  where
    xmod = x `mod` 12
    ymod = y `mod` 12
    os = P.elements tr
    ts = (P.elements . P.invertXY x y) tr
    loc e es = e `Data.List.elemIndex` es
```

1.3.2.8 zero, retrograde, rotate, sort

These functions are more fully tested in "Standard Candles" (Section 1.2).

1.3.2.9 normal

Three properties of the normal will be tested here—first, on General Sets. After this, it will be tested against Standard Sets. (Normal is not defined for Tone Rows, since it would only give the ascending chromatic scale in that modulus.)

Arbitrary General Set.

The normal of a normal set should be the same normal set.

```
propDoubleNormalEquivalence :: P.GenSet -> Bool
propDoubleNormalEquivalence ps =
    (P.normal . P.normal) ps == P.normal ps
```

The normal of a set should contain the same elements as the original set. (Verified here by sorting the elements on either side of the equality.)

```
propNormalPreservesElements :: P.GenSet -> Bool
propNormalPreservesElements ps =
    (P.sort . P.normal) ps == P.sort ps
```

The normal form should be the same for any set containing a particular set of elements, regardless of the permutation of these elements. (Here, to avoid enormous testing times for large sets, I'm only considering permutations through retrograde and rotate.)

```
propNormalResilience :: P.GenSet -> Int -> Bool
propNormalResilience ps n =
    all (== ns) [f ps | f <- ops]
  where
    ns = P.normal ps
    ops = [P.normal . g . h |
           g <- [id, P.rotate n],
           h <- [id, P.retrograde]]
```

Arbitrary Standard Set. Now, the same three properties are tested against Standard Sets.

The normal of a normal set should be the same normal set.

```
propStdDoubleNormalEquivalence :: P.StdSet -> Bool
propStdDoubleNormalEquivalence ps =
    (P.normal . P.normal) ps == P.normal ps
```

The normal of a set should contain the same elements as the original set.

```
propStdNormalPreservesElements :: P.StdSet -> Bool
propStdNormalPreservesElements ps =
    (P.sort . P.normal) ps == P.sort ps
```

The normal form should be the same for any set containing a particular set of elements, regardless of the permutation of these elements.

```
propStdNormalResilience :: P.StdSet -> Int -> Bool
propStdNormalResilience ps n =
    all (== ns) [f ps | f <- ops]
  where
    ns = P.normal ps
    ops = [P.normal . g . h |
           g <- [id, P.rotate n],
           h <- [id, P.retrograde]]
```

1.3.2.10 reduced

Two properties of the reduced operator will be tested here—first, on General Sets. After this, it will be tested against Standard Sets. (Reduced is not defined for Tone Rows, since it would only give the ascending chromatic scale in that modulus.)

Arbitrary General Set.

The reduced form of a reduced set should be the same reduced set.

```
propDoubleReducedEquivalence :: P.GenSet -> Bool
propDoubleReducedEquivalence ps =
    (P.reduced . P.reduced) ps == P.reduced ps
```

The reduced form should be the same for any set containing a particular set of elements, regardless of the permutation *or transposition* of those elements. (Here, to avoid enormous testing times for large sets, I'm only considering permutations through retrograde, rotate, and transpose.)

```
propReducedResilience :: P.GenSet -> Int -> Int -> Bool
propReducedResilience ps m n =
    all (== rs) [f ps | f <- ops]
  where
    rs = P.reduced ps
    ops = [P.reduced . g . h . i |
           g <- [id, P.rotate m],
           h <- [id, P.retrograde],
           i <- [id, P.transpose n] ]
```

Arbitrary Standard Set.

The reduced form of a reduced set should be the same reduced set.

```
propStdDoubleReducedEquivalence :: P.StdSet -> Bool
propStdDoubleReducedEquivalence ps =
    (P.reduced . P.reduced) ps == P.reduced ps
```

The reduced form should be the same for any set containing a particular set of elements, regardless of the permutation *or transposition* of those elements.

```
propStdReducedResilience :: P.StdSet -> Int -> Int -> Bool
propStdReducedResilience ps m n =
    all (== rs) [f ps | f <- ops]
  where
    rs = P.reduced ps
    ops = [P.reduced . g . h . i |
           g <- [id, P.rotate m],
           h <- [id, P.retrograde],
           i <- [id, P.transpose n] ]
```

1.3.2.11 prime

Two properties of the prime set operator will be tested here—first, on General Sets. After this, it will be tested against Standard Sets. (Prime is not defined for Tone Rows, since it would only give the ascending chromatic scale in that modulus.)

Arbitrary General Set.

The prime form of a set in prime form should be the same prime set.

```
propDoublePrimeEquivalence :: P.GenSet -> Bool
propDoublePrimeEquivalence ps =
    (P.prime . P.prime) ps == P.prime ps
```

The prime form should be the same for any set containing a particular set of elements, regardless of the permutation, transposition, *or inversion* of those elements. (Here, to avoid enormous testing times for large sets, I'm only considering permutations through retrograde, rotate, transpose, and invert.)

```
propPrimeResilience :: P.GenSet -> Int -> Int -> Bool
propPrimeResilience ps m n =
    all (== ps') [f ps | f <- ops]
  where
    ps' = P.prime ps
    ops = [P.prime . g . h . i . j |
           g <- [id, P.rotate m],
           h <- [id, P.retrograde],
           i <- [id, P.transpose n],
           j <- [id, P.invert] ]
```

Arbitrary Standard Set.

The prime form of a set in prime form should be the same prime set.

```
propStdDoublePrimeEquivalence :: P.StdSet -> Bool
propStdDoublePrimeEquivalence ps =
    (P.prime . P.prime) ps == P.prime ps
```

The prime form should be the same for any set containing a particular set of elements, regardless of the permutation, transposition, *or inversion* of those elements. (Here, to avoid enormous testing times for large sets, I'm only considering permutations through retrograde, rotate, transpose, and invert.)

```
propStdPrimeResilience :: P.StdSet -> Int -> Int -> Bool
propStdPrimeResilience ps m n =
    all (== ps') [f ps | f <- ops]
  where
    ps' = P.prime ps
    ops = [P.prime . g . h . i . j |
           g <- [id, P.rotate m],
           h <- [id, P.retrograde],
           i <- [id, P.transpose n],
           j <- [id, P.invert] ]
```

1.3.2.12 cardinality

Cardinality is indirectly tested by a number of other tests in this suite.

1.3.2.13 binaryValue

The binary value of a pitch class set can be thought of as a "norm" operation on the set, with the lowest values going to sets which are packed closest in toward the lower numbers. It is only defined for Pitch Class Sets; it would make little sense for Tone Rows, which always contain all the possible elements, and therefore would all have the same binary value.

Below, the following two properties are tested for General and Standard Sets:

- If two sets have the same modulus and the same binary value, they should have all of the same elements.
- The binary value should be independent of the arrangement of the elements in the set.

Arbitrary General Set.

Sets with equal modulus and binary value should have the same elements. To test this, the set will be compared against all its possible transpositions.

> **Technical Note:** the sets are sorted before they are compared. A random transposition can generate exactly same elements, but in a different order; this evens the playing field and makes the two directly comparable as lists.

```
propSameBinaryValue :: P.GenSet -> Bool
propSameBinaryValue ps =
    all (`proposition` ps) tps
  where
    m = P.modulus ps
    -- generate a list of transpositions 0 to m
    tps = take m $ iterate (P.transpose 1) ps
    -- if the transpositions have the same binary
    -- value, they must be the same set.
    sameBV = (==) `on` P.binaryValue
    a `proposition` b = if a `sameBV` b
      then P.sort a == P.sort b
      else P.sort a /= P.sort b
```

The binary value should be independent of the permutation of elements in the set. (To avoid enormous testing times for large sets, I'm only considering permutations through retrograde and rotate.)

```
propBinaryResilience :: P.GenSet -> Int -> Bool
propBinaryResilience ps n =
    all (== b) [f ps | f <- ops]
  where
    b = P.binaryValue ps
    ops = [P.binaryValue . g . h |
           g <- [id,P.retrograde],
           h <- [id,P.rotate n] ]
```

Arbitrary Standard Set.

Sets with equal modulus and binary value should have the same elements. To test this, the set will be compared against all its possible transpositions.

```
propSameStdBinaryValue :: P.StdSet -> Bool
propSameStdBinaryValue ps =
    all (`proposition` ps) tps
  where
    -- generate a list of transpositions 0 to 12
    tps = take 12 $ iterate (P.transpose 1) ps
    -- if the transpositions have the same binary
    -- value, they must be the same set.
    sameBV = (==) `on` P.binaryValue
    a `proposition` b = if a `sameBV` b
      then P.sort a == P.sort b
      else P.sort a /= P.sort b
```

The binary value should be independent of the permutation of elements in the set.

```
propStdBinaryResilience :: P.StdSet -> Int -> Bool
propStdBinaryResilience ps n =
    all (== b) [f ps | f <- ops]
```

```
  where
    b = P.binaryValue ps
    ops = [P.binaryValue . g . h |
           g <- [id,P.retrograde],
           h <- [id,P.rotate n] ]
```

1.3.2.14 avec

Given the first element of a set and its ascending vector, it should be possible to reproduce the entire set. (The ascending vector is only defined for Pitch Class Sets; for Tone Rows, the ordering of the elements provides the same information.)

Arbitrary General Set.

```
propGenSetAvecRegeneration :: P.GenSet -> Bool
propGenSetAvecRegeneration ps =
    regeneratedSet == ps
  where
    m = P.modulus ps
    ascent = P.avec ps
    first = (head . P.elements) ps
    regeneratedSet = if P.cardinality ps == 0
      then P.genset m []
      else P.genset m (init newseries)
    newseries = scanl (\a b -> (a + b) `mod` m) first ascent
```

Arbitrary Standard Set.

```
propStdSetAvecRegeneration :: P.StdSet -> Bool
propStdSetAvecRegeneration ps =
    regeneratedSet == ps
  where
    ascent = P.avec ps
    first = (head . P.elements) ps
    regeneratedSet = if P.cardinality ps == 0
      then P.stdset []
      else P.stdset (init newseries)
    newseries = scanl (\a b -> (a + b) `mod` 12) first ascent
```

1.3.2.15 cvec

For a given pitch class set, the common tone vector should be the number of common tones under the operation (transpose n . invert) for each entry n in the vector.

Arbitrary General Set.

```
propGenSetCvecDefinition :: P.GenSet -> Bool
propGenSetCvecDefinition ps =
    actualCombinations == expectedCombinations
  where
    m = P.modulus ps
    expectedCombinations = P.cvec ps
    actualCombinations = map tryCom possibleRange
    possibleRange = if m == 0 then [] else [0..(m-1)]
    tryCom n = commonTones ps ((P.transpose n . P.invert) ps)
    commonTones a b = length (filter (`elem` (P.elements b)) (P.elements a))
```

Arbitrary Standard Set.

```
propStdSetCvecDefinition :: P.StdSet -> Bool
propStdSetCvecDefinition ps =
    actualCombinations == expectedCombinations
  where
    expectedCombinations = P.cvec ps
    actualCombinations = map tryCom [0..11]
    tryCom n = commonTones ps ((P.transpose n . P.invert) ps)
    commonTones a b = length (filter (`elem` (P.elements b)) (P.elements a))
```

1.3.2.16 ivec

For a given pitch class set, the interval vector should be consistent and independent of the permutation of the elements of the set. (Here, to avoid excessive test times, the only permutations considered are rotate and retrograde).

Arbitrary General Set.

```
propGenSetIvecResilience :: P.GenSet -> Int -> Bool
propGenSetIvecResilience ps n =
    all (== ivals) [f ps | f <- ops]
  where
    ivals = P.ivec ps
    ops = [P.ivec . g . h | g <- [id,P.retrograde], h <- [id,P.rotate n] ]
```

Arbitrary Standard Set.

```
propStdSetIvecResilience :: P.StdSet -> Int -> Bool
propStdSetIvecResilience ps n =
    all (== ivals) [f ps | f <- ops]
  where
    ivals = P.ivec ps
    ops = [P.ivec . g . h | g <- [id,P.retrograde], h <- [id,P.rotate n] ]
```

1.3.2.17 rowP

Application of the row operation "P" should return a new row in it's *primary* form (same interval pattern as the original), with the first note transposed to n. (Here, n is understood to be relative to the modulus of the set.)

Arbitrary General Row.

```
propGenRowPrimary :: P.GenRow -> Int -> Bool
propGenRowPrimary tr n =
    -- this test ignores the null modulus row
    m == 0 || (first == nmod && intervals == same)
  where
    m = P.modulus tr
    nmod = n `mod` m
    pr = P.rowP n tr
    first = (head . P.elements) pr
    intervals = diffs pr
    same = diffs tr
    diffs r = zipWith f ((P.elements . P.rotate 1) r) (P.elements r)
    f a b = (a - b) `mod` m
```

Arbitrary Standard Row.

```
propStdRowPrimary :: P.StdRow -> Int -> Bool
propStdRowPrimary tr n =
    first == nmod && intervals == same
  where
    nmod = n `mod` 12
    pr = P.rowP n tr
    first = (head . P.elements) pr
    intervals = diffs pr
    same = diffs tr
    diffs r = zipWith f ((P.elements . P.rotate 1) r) (P.elements r)
    f a b = (a - b) `mod` 12
```

1.3.2.18 rowR

Application of the row operation "R" should return a new row in it's *retrograde* form (reversed interval pattern compared to the original), with the first note transposed to n. (Here, n is understood to be relative to the modulus of the set.)

Arbitrary General Row.

```
propGenRowRetrograde :: P.GenRow -> Int -> Bool
propGenRowRetrograde tr n =
    -- this test ignores the null modulus row
    m == 0 || (first == nmod && intervals == reversed)
  where
```

```
    m = P.modulus tr
    nmod = n `mod` m
    rr = P.rowR n tr
    first = (head . P.elements) rr
    intervals = diffs rr
    reversed = diffs (P.retrograde tr)
    diffs r = zipWith f ((P.elements . P.rotate 1) r) (P.elements r)
    f a b = (a - b) `mod` m
```

Arbitrary Standard Row.

```
propStdRowRetrograde :: P.StdRow -> Int -> Bool
propStdRowRetrograde tr n =
    first == nmod && intervals == reversed
  where
    nmod = n `mod` 12
    rr = P.rowR n tr
    first = (head . P.elements) rr
    intervals = diffs rr
    reversed = diffs (P.retrograde tr)
    diffs r = zipWith f ((P.elements . P.rotate 1) r) (P.elements r)
    f a b = (a - b) `mod` 12
```

1.3.2.19 rowI

Application of the row operation "I" should return a new row in it's *inverse* form (inverted interval pattern compared to the original), with the first note transposed to n. (Here, n is understood to be relative to the modulus of the set.)

Arbitrary General Row.

```
propGenRowInverse :: P.GenRow -> Int -> Bool
propGenRowInverse tr n =
    -- this test ignores the null modulus row
    m == 0 || (first == nmod && intervals == opposite)
  where
    m = P.modulus tr
    nmod = n `mod` m
    ir = P.rowI n tr
    first = (head . P.elements) ir
    intervals = diffs ir
    opposite = map (f m) (diffs tr)
    diffs r = zipWith f ((P.elements . P.rotate 1) r) (P.elements r)
    f a b = (a - b) `mod` m
```

Arbitrary Standard Row.

```
propStdRowInverse :: P.StdRow -> Int -> Bool
propStdRowInverse tr n =
```

```
    first == nmod && intervals == opposite
  where
    nmod = n `mod` 12
    ir = P.rowI n tr
    first = (head . P.elements) ir
    intervals = diffs ir
    opposite = map (f 12) (diffs tr)
    diffs r = zipWith f ((P.elements . P.rotate 1) r) (P.elements r)
    f a b = (a - b) `mod` 12
```

1.3.2.20 rowRI

Application of the row operation "RI" should return a new row in it's *retrograde inverse* form (reversed and inverted interval pattern compared to the original), with the first note transposed to n. (Here, n is understood to be relative to the modulus of the set.)

Arbitrary General Row.

```
propGenRowRetroInverse :: P.GenRow -> Int -> Bool
propGenRowRetroInverse tr n =
    -- this test ignores the null modulus row
    m == 0 || (first == nmod && intervals == opposite)
  where
    m = P.modulus tr
    nmod = n `mod` m
    rir = P.rowRI n tr
    first = (head . P.elements) rir
    intervals = diffs rir
    opposite = map (f m) (diffs (P.retrograde tr))
    diffs r = zipWith f ((P.elements . P.rotate 1) r) (P.elements r)
    f a b = (a - b) `mod` m
```

Arbitrary Standard Row.

```
propStdRowRetroInverse :: P.StdRow -> Int -> Bool
propStdRowRetroInverse tr n =
    first == nmod && intervals == opposite
  where
    nmod = n `mod` 12
    rir = P.rowRI n tr
    first = (head . P.elements) rir
    intervals = diffs rir
    opposite = map (f 12) (diffs (P.retrograde tr))
    diffs r = zipWith f ((P.elements . P.rotate 1) r) (P.elements r)
    f a b = (a - b) `mod` 12
```

1.3.3 General Truths

Here the tests become quite complex, involving multiple combinations and interactions of the operations above.

1.3.3.1 Minimal Ascending Vector

Putting a set in normal or reduced form implies that it is in "closest packed" form. Therefore, the ascending vector for a set should be at a minimum compared to other permutations of the same set.

In practice, it is difficult to test every possible permutation. Therefore, here, each set will be tested in original (random) form vs. normal and reduced forms. The sum of the interval vectors for normal and reduced should be less than or equal to that of the random set—never greater.

Arbitrary General Set.

```
propMinimalAscendingVector :: P.GenSet -> Bool
propMinimalAscendingVector ps =
    all (<= randomCase) [f ps | f <- ops']
  where
    randomCase = sum . P.avec $ ps
    ops = [P.normal,P.reduced]
    ops' = map ((sum . P.avec) .) ops
```

Arbitrary Standard Set.

```
propStdMinimalAscendingVector :: P.StdSet -> Bool
propStdMinimalAscendingVector ps =
    all (<= randomCase) [f ps | f <- ops']
  where
    randomCase = sum . P.avec $ ps
    ops = [P.normal,P.reduced]
    ops' = map ((sum . P.avec) .) ops
```

1.4 Runtime Code

```
confess :: (Testable prop) => String -> prop -> IO ()
confess s p = putStr r >> quickCheck p
  where
    t = 38 - length s
    pad = if t < 1 then "␣" else replicate t ' '
    r = "␣␣" ++ s ++ pad
```

```
main :: IO ()
main = do
```

```
putStrLn "Standard Candles -- By Function"
--
confess "Arbitrary Integer Modulus" propArbitraryIntModulus
confess "Arbitrary Element Ranges" propArbitraryElements
confess "Chromatic / Empty Complement" propChromaticEmptyComplement
confess "Tone Row Reconciliation" propReconciliation
confess "Zero Transposition" propZeroTransposition
confess "Modulus Transposition" propModulusTransposition
confess "Transposition Reversal" propTranspositionReversal
confess "Transposition Completion" propTranspositionCompletion
confess "Double Inversion Identity" propDoubleInversion
confess "Zero Unchanged by Inversion" propStandardInversionZero
confess "Operation Zero" propOperationZero
confess "Double Retrograde Identity" propDoubleRetrograde
confess "Zero Rotation" propZeroRotation
confess "Length Rotation" propLengthRotation
confess "Rotation Reversal" propRotationReversal
confess "Rotation Completion" propRotationCompletion
confess "Double Sort Equivalence" propDoubleSortEquivalence
confess "Normal of Major Triad" propNormalOfMajorTriad
confess "Normal of Minor Triad" propNormalOfMinorTriad
confess "Reduced Major Triad Equivalence" propReducedMajorTriad
confess "Reduced Minor Triad Equivalence" propReducedMinorTriad
confess "Major and Minor Prime Form" propMajorMinorPrimes
confess "Major Scale Prime Form" propMajorScalePrime
confess "Ascending Chromatic Avec" propAscendingChromaticAvec
confess "Descending Chromatic Avec" propDescendingChromaticAvec
confess "Chromatic Combination Vector" propChromaticCombination
confess "All Interval Tetrachord Ivec" propAllIntervalTetrachordIvec
--
putStrLn "Standard Candles -- General Truths"
--
confess "Empty Set Bulletproof" propEmptySetBulletproof
confess "Null Modulus Bulletproof" propNullModulusBulletproof
confess "Null Row Bulletproof" propNullRowBulletproof
confess "Empty Cardinality Zero" propEmptyCardinalityZero
confess "Empty Binary Value Zero" propEmptyBinaryValueZero
confess "Empty Ascending Vector" propEmptyAvecOperations
confess "Empty / Zero Interval Vector" propEmptyIvecOperations
confess "Empty / Zero Common Tone Vector" propEmptyCvecOperations
confess "Null Row Operations" propNullRowOperations
--
putStrLn "Arbitrary Input -- By Function"
--
confess "General Set Modulus" propGenSetModulus
confess "Standard Set Modulus" propStdSetModulus
confess "General Row Modulus" propGenRowModulus
confess "Standard Row Modulus" propStdRowModulus
confess "General Set Elements" propGenSetElements
confess "Standard Set Elements" propStdSetElements
```

```
confess "General Row Elements" propGenRowElements
confess "Standard Row Elements" propStdRowElements
confess "General Set Complement" propGenSetComplement
confess "Standard Set Complement" propStdSetComplement
confess "General Set Transpose Difference" propGenSetTransposeDiff
confess "Standard Set Transpose Difference" propStdSetTransposeDiff
confess "General Row Transpose Difference" propGenRowTransposeDiff
confess "Standard Row Transpose Difference" propStdRowTransposeDiff
confess "General Set Inverse Sum" propGenSetInverseSum
confess "Standard Set Inverse Sum" propStdSetInverseSum
confess "General Row Inverse Sum" propGenRowInverseSum
confess "Standard Row Inverse Sum" propStdRowInverseSum
confess "General Set InvertXY Exchange" propGenSetIXY
confess "Standard Set InvertXY Exchange" propStdSetIXY
confess "General Row InvertXY Exchange" propGenRowIXY
confess "Standard Row InvertXY Exchange" propStdRowIXY
confess "Double Normal Equivalence" propDoubleNormalEquivalence
confess "Normal Preserves Elements" propNormalPreservesElements
confess "Normal Resilience" propNormalResilience
confess "Standard Double Normal Equivalence" propStdDoubleNormalEquivalence
confess "Standard Normal Preserves Elements" propStdNormalPreservesElements
confess "Standard Normal Resilience" propStdNormalResilience
confess "Double Reduced Equivalence" propDoubleReducedEquivalence
confess "Reduced Resilience" propReducedResilience
confess "Standard Double Reduced Equivalence"
  propStdDoubleReducedEquivalence
confess "Standard Reduced Resilience" propStdReducedResilience
confess "Double Prime Equivalence" propDoublePrimeEquivalence
confess "Prime Resilience [long]" propPrimeResilience
confess "Standard Double Prime Equivalence" propStdDoublePrimeEquivalence
confess "Standard Prime Resilience" propStdPrimeResilience
confess "Same Binary Value" propSameBinaryValue
confess "Binary Resilience" propBinaryResilience
confess "Same Standard Binary Value" propSameStdBinaryValue
confess "Standard Binary Resilience" propStdBinaryResilience
confess "General Set Avec Regeneration" propGenSetAvecRegeneration
confess "Standard Set Avec Regeneration" propStdSetAvecRegeneration
confess "General Set Cvec Definition" propGenSetCvecDefinition
confess "Standard Set Cvec Definition" propStdSetCvecDefinition
confess "General Set Ivec Resilience" propGenSetIvecResilience
confess "Standard Set Ivec Resilience" propStdSetIvecResilience
confess "General Row Primary" propGenRowPrimary
confess "Standard Row Primary" propStdRowPrimary
confess "General Row Retrograde" propGenRowRetrograde
confess "Standard Row Retrograde" propStdRowRetrograde
confess "General Row Inverse" propGenRowInverse
confess "Standard Row Inverse" propStdRowInverse
confess "General Row Retrograde Inverse" propGenRowRetroInverse
confess "Standard Row Retrograde Inverse" propStdRowRetroInverse
--
```

```
putStrLn "Arbitrary Input -- General Truths"
--
confess "Minimal Ascending Vector" propMinimalAscendingVector
confess "Standard Minimal Ascending Vector" propStdMinimalAscendingVector
--
putStrLn "All tests complete."
```

Chapter 2

Data.PcSets.Svg

```
import Data.PcSets.Svg
```

```
main = putStrLn "ok"
```

final

Chapter 3

Data.PcSets.Compact

```
import Data.PcSets.Compact
```

```
main = putStrLn "ok"
```

final

Chapter 4

Data.PcSets.Notes

```
import Data.PcSets.Notes
```

```
main = putStrLn "ok"
```

final

Chapter 5

Data.PcSets.Catalog

```
import Data.PcSets.Catalog
```

```
main = putStrLn "ok"
```

final