1 Tutorial 1: The simplest case.

This is the simplest example of using the GenCheck framework to test an implementation against a specification. This module contains a revList (reverse) function that reverses the order of the elements in a list, and two properties for this function: reversing a list twice returns the original list, and reversing a singleton list has no effect.

```
module TestReverseList where

import Test.SimpleCheck
```

The properties that the revList function must satisfy:

- propRevUnitEq - the reverse of a singleton list is the original
- propRevRevEq - reversing the reverse is the original list (involutive)

Note that the first is a property of individual values and the second is a property of lists, so different test suites and test case generators are required.

```
propRevUnitEq :: (Eq a) ⇒ Property a
propRevUnitEq x = (revList [x]) == [x]

propRevRevEq :: (Eq a) ⇒ Property [a]
propRevRevEq xs = (revList.revList) xs == xs
```

The implementation to test.

```
revList :: [a] → [a]
revList xs  = rev xs []
  where rev [] xs' = xs'
        rev (y:ys) xs' = rev ys (y:xs')
```

To evaluate whether the properties are satisfied, we need to generate a collection of test cases, evaluate them and report the results of the test. To do this, we pick one of the test programs provided by GenCheck, and the best place to start is the friendly sounding module SimpleCheck. There are a number of test programs here that provide variations on how test cases are generated and results are reported. The name of the test program consists of three parts:

testsuite Describes the kind of test suite generated: std, deep, base

reporting Defines whether the first failure (Check), all failures (Test) or all results (Report) are printed to the screen

arguments functions ending in Args have additional parameters; those not ending in Args provide default values for these parameters and default test case generators (so must be instances of Testable, more on this class coming up soon).

The first property takes a value, puts it in a list, and tests that reversing the list doesn't change the value. This value could be of any Haskell type, but we have to pick one to actually run the tests, so we'll start with Char. Char is a "base" type (as opposed to a structure) so all of the elements of this type are the same size. SimpleCheck's baseTest, baseCheck, and baseReport are all appropriate as they don't distinguish test elements by size. We'll pick baseTest so that all of the test cases are evaluated, but only the failure cases are reported. Char is an instance of the Testable class, so we don't need to use the parameterized versions (baseTestArgs, etc.) and will settle for the default settings for now.

Note that the GenCheck test programs generally request the desired number of test cases as input, but then only uses that as a guide in developing the test suite. The actual number of test cases will vary according to the number of possible test cases in the test domain.

```
testUnitEq_1 = baseTest (propRevUnitEq :: Property Char) 100
```

The next property requires a list of values to reverse, so baseTest is no longer appropriate as it is only for scalar values. The test suite must have lists of a variety of different lengths; in GenCheck parlance the size of a structure is referred to as it's "rank". stdTest and deepTest both generate test suites for structures, with deepTest including much higher ranked test cases, but fewer at each rank. For lists, there is only one list structure for each rank, so we'll use deepTest (deepReport and deepCheck would produce the same test cases).

Setting the property to a list of Ints seems a good starting point, but in order to use the SimpleCheck test functions we need an instance of Testable. Is —[Int]— an instance of Testable or do we need to do something? Lists are instances of the Structure class, which provides the method to populate a structure with elements from another generator, and also an instance of Testable, so there are standard generators available. A default instance of Testable is provided for lists of any Testable type, and Ints are Testable, so we don't have to worry about how lists are generated - yet.

```
testRevIdem_1 = deepTest (propRevRevEq :: Property [Int]) 100
```

The tests are passing, but somewhat unsatisfying that we can't see the test cases. So, we switch to the baseReport / deepReport functions that report all of the results, flagging any failed test cases with "FAILED: ". These are the same cases that were used for baseTest / deepTest, so we already know they will be successful, but using the Report versions will allow an evaluation of the quality of the test suite. The report dumps a lot of data onto the screen, so it made sense to verify the properties worked first with the Test functions so we didn't miss any of the FAILED flags.

```
testUnitEq_2  = baseReport (propRevUnitEq :: Property Char) 100
testRevIdem_2 = deepReport (propRevRevEq :: Property [Int]) 100
```

What happens when one or more test cases fail? Let's create a failing property, such as comparing a reversed list with the original. The very first test case with two elements in the list fails.

```
propRevEqNot :: (Eq a) ⇒ Property [a]
propRevEqNot xs =  revList xs == xs

testRevEqNot_1 =  deepTest (propRevEqNot :: Property [Int]) 100 -- should fail
```

We also see that *every* other case failed too - maybe we shouldn't print out all the failures! This is where using deepCheck comes in handy - it stops after the first failure case.

```
-- isn't implemented yet, so this is redundant
testRevEqNot_2 =  deepCheck (propRevEqNot :: Property [Int]) 100
```

The SimpleCheck module provides a useful collection of simple test programs that start with default test parameters and test case generators, but allow the tester to make a few simple choices to customize the testing if desired. The simpleTest, simpleReport and simpleCheck functions are the glue to connect the reporting, test execution and test suite options for a test program.

The examples in this module have properties over standard Haskell types, where the default generators were already available. How are test cases generated when the module contains a new type? How are the test cases chosen for inclusion in the test suite, and how can that be controlled? Surely there is a better way to display the results? These questions and more are answered in the next part of the tutorial.

2 Tutorial 2: Testing with New Data Types

The first tutorial covered testing properties over standard Haskell types. Default test case generators were already supplied by GenCheck as instances of the Testable class, but what about new data types? In this tutorial, we explain how to use the type constructor definition and GenCheck enumeration combinators to construct the standard test generators based on the type constructor definitions.

There are three steps in testing a module against a specification using the GenCheck framework:

1. define and code the properties that make up the specification
2. select the test system that provides the most appropriate test scheduling and reporting schema for the current phase of the development
3. define the test suite for those properties, including the test case generators for each.

In the previous section the details of the test system and test suite generation were hidden by the SimpleCheck API. In this section, we look at each of these steps in detail and discuss how to generate new data types and customize test suites for a particular set of properties.

For this tutorial, we'll look at a list zipper implementation, taken from the ListZipper package (1.2.0.2). Only snippets of the code will be provided as the

package is rather large (see tutorial/list_zipper/ListZipper.hs for the module source). The Zipper data type is

```
data Zipper a = Zip ![a] ![a] deriving (Eq,Show)
```

where the "current" element or "cursor" is the head of the second list. The specification for ListZipper includes empty, cursor, start, end, left, right, foldrz, foldlz, and many more functions. The example below shows how to test (some) of the specification for this module, with a focus on the generators for the new type.

2.1 Specifications and Properties

A GenCheck specification is the collection of properties that any implementation of the specification must satisfy. Properties are univariate Boolean valued Haskell functions; this is not a restriction as the input value may be of an arbitrarily complicated type to support uncurried predicates. The GenCheck test programs accept a single property and test it over the defined suite of test cases. Only one property is tested per call because different properties may have different input types or require different test suites The specification module would normally be separate from the implementation, but may include functions that test the properties using GenCheck.

2.1.1 List Zipper Properties

A small set of properties from the ListZipper package specification.

```
module PropListZip where

import Test.GenCheck (Property)
import ListZipper as Zipper
```

A ziplist is empty iff the cursor is both at the beginning and the end

```
propEmptyP :: Property (Zipper a)
propEmptyP z = emptyp z == beginp z && endp z
```

Folding an operation over the zipper with the cursor at the start should be identical to folding it over the unzipped list. Note that the zipper fold folds the cursor through the zipper, but always passes the zipper to the function. (cursor safety is supposed to be guaranteed by the foldz' functions)

```
propFoldrz :: (Eq b) ⇒ (a → b → b) → b → Property (Zipper a)
propFoldrz  f y0 zxs =
   let xs = Zipper.toList zxs
   in (foldr f y0 xs == foldrz f' y0 (start zxs))
   where f' zs y = f (cursor zs) y
```

```
propFoldlz :: (Eq b) ⇒ (b → a → b) → b → Property [a]
propFoldlz  f y0 xs =
   let zxs = Zipper.fromList xs
       fval = foldl f y0 xs
       zxs' = start zxs
   in (fval == foldlz f' y0 zxs') && (fval == foldlz' f' y0 zxs')
   where f' y zs = f y (cursor zs)
```

Replace is equivalent to deleting the current element and inserting a new one.

```
propInsDelRepl :: (Eq a) ⇒ Property (a, Zipper a)
propInsDelRepl (x, xs) = Zipper.insert x (Zipper.delete xs) == Zipper.replace x xs
```

2.2 Generators and Enumerations

A GenCheck generator is any function from a rank (positive integer) to a list of values. Generators may be finite or infinite, may include duplicates of the same value, and may be missing values from the total population of the type. Generators can be manually coded or built from a GenCheck enumeration of the type using a sampling strategy called an "enumerative strategy". These strategies can be applied to any data type that can be enumerated, i.e. ordered, and indexed by a finite rank.

One way to build generators is to start with an enumeration of the type to be generated. An enumeration of the type is a total order over the type's values. For base types, any list of values is a de facto (base) enumeration, as is any type that is an instance of Haskell's Enum and Bounded classes. Regular recursive algebraic data types are enumerated by first partitioning the values by size (i.e. the number of elements) and then indexing each partition. Base and structure type enumerations are distinguished in GenCheck as the structure enumerations use the rank, or size, as the first part of the index, but base type enumerations have no rank. A collection of common base type enumerations is supplied in the —Generator.BaseEnum—. —Generator.Enumeration— contains a collection of enumeration combinators that mirror the sum and product construction of the Haskell algebraic data types, allowing structure enumerations to be created mechanically.

An enumerative strategy is a type independent approach to sampling the values of an enumeration to create a list of values for a generator. These strategies are implemented as lists of —Integer— indices; the selector function from the enumeration is mapped across the strategy, at a specified rank, to get the ordered values to be returned by the generator. For example, —EnumStrat.uniform— takes every k^th value from an enumeration, so the strategy for — uniform 5 -¿ [1, 6, 11, 16, ...] —, and if this were applied to an enumeration of — Char — would produce:

```
genUni5LowChar = enumGenerator (uniform 5) enumLowChar
```

```
→ ['a', 'f', 'k', ○ ..]
```

The so called standard generators are just four different sampling strategies that can be applied to an enumeration of the generated type. These are:

exhaustive Produces all of the values of the specified rank in a sequence,

extreme Alternating lowest and highest values in the enumeration,

uniform Select the specified number elements at uniform intervals through the enumeration,

random Use the random generator to pick an infinite list of values.

2.2.1 List Zipper Enumeration and Generators

The GenCheck class Structure and Enumeration instances for Zipper. It is typical of what would be mechanically generated for the type, but has been converted to literate Haskell and documented. There is no special significance to the ${}_GC$ suffix in the name, that is just a convention so the GenCheck instances module is easily found.

```
{-# LANGUAGE FlexibleInstances,FlexibleContexts  #-}
module ListZipper_GC where

import Control.Monad (liftM2)
‘
import Test.GenCheck
import Test.GenCheck.Generator.Enumeration

--import Test.GenCheck.Generator.Generator (Testable(..))
--import Test.GenCheck.Generator.StructureGens()
--import Test.GenCheck.Generator.Substitution (Structure(..))

import ListZipper
```

The Structure class has a single method called substitute. It replaces the elements in the structure with elements from a list and returns a newly populated instance of the structure and the remaining elements; if there are insufficient elements in the list to populate the structure Nothing is returned along with the original list of elements.

A default substitution method can be mechanically derived for algebraic data types, by interpreting the type as a sum of products, and using substitute methods for any substructures. Alternate substitutions could be provided instead.

```
instance Structure Zipper where
  substitute = substZipper

substZipper :: Zipper a → [b] → (Maybe (Zipper b), [b])
substZipper (Zip x1 x2) ys =
```

```
  let (mx1, ys')  = substitute x1 ys
      (mx2, ys'') = substitute x2 ys'
      zip' = liftM2 Zip mx1 mx2
  in (zip', ys'')
```

The Zipper is just a product of two lists of like elements. The enumeration is built from the combinators eProd and eList (list gets a special combinator due to it's frequent appearance and special syntax).

The Enumeration consists of two functions that, given a rank, produce the number of structures and a selector function that generates a structure given an index. These functions are highly recursive, so are memoized using the Memoize Hackage (eMemoize just applies memoize to both the count and selector functions in the Enumeration records).

The selector function from the Enumeration creates instances of the structure. The Label type is used as a way to instantiate those structures and provide a label for the parametric type variables in the structure as a convenience.

```
instance Testable (Zipper Label) where
  stdTestGens = stdEnumGens

instance Enumerated Zipper where
  enumeration = eZipper

eZipper :: Enumeration Zipper Label
eZipper = eMemoize e
  where
    e = eProd Zip (eList A) (eList A)
```

Gordon says: This should be in Test.GenCheck.Generator.Enumeration ??? *(* ???)

```
eList :: a → Enumeration [] a
eList x = mkEnum c s
  where c _ = 1  -- there is only one list of any length
        s r k | k == 1 = take r $ repeat x
        s _ _ | otherwise = error "Selector error"
```

2.3 Test Suites and Standard Generators

The module System.TestSuite contains functions that assemble test suites from one or more test case generators. These suite builders take a collection of "standard" generators and a set of "generator instructions", a list of rank, count pairs. The allocation of test cases to ranks through these instructions is called a "test strategy" and can be used independently of the type of data being generated, assuming that the standard generators are available for that type.

The basis for the test suites used by SimpleCheck test programs are the standard generator set described above. These are not the only valuable sampling strategies that could be used, but these were selected as the "standard" generator strategies to be used by the GenCheck API. A type is an instance of

Testable if a set of standard generators is available for it; these generators can be constructed for any type that has an Enumeration associated with it.

A test suite is a collection of test cases selected from the ranks of one or more generators. Each test suite uses a sampling "strategy" to decide which generators and at what rank to select the test cases; this mix is called the test strategy. The SimpleCheck test strategies draw test cases from the four standard generator strategies to provide a good sampling of the test cases relative to the number of tests performed. The strategies are heuristic of course, and the optimal composition of a test strategy depends on the nature of the structures and properties being tested, so these provided test suite strategies are really just a starting point. The TestSuite module contains the test suite building functions:

stdSuite exhaustive testing for smaller structures, then the boundary elements and a mix of uniformly and randomly selected values to the specified maximum rank

deepSuite similar to standard but for "narrow" structures with fewer elements per rank (i.e. a binary tree or list), so the exhaustive testing goes to a much higher rank

baseSuite a test suite for base types, where the rank is always 1

The System.SimpleCheck module test suites are test cases stored in a Haskell map (as in Data.Map) indexed by rank, called a MapRankSuite.

2.4 The Testing Process

Typically a GenCheck user would start with the default test suites and only reporting failures during the development cycle. Nearing completion, s/he would switch to the test programs that provide more control over the test suite generation, and report all of the test cases in the results to ensure good coverage.

2.4.1 Testing the List Zipper Module

```
module Test_ListZipper where

import System.Random(mkStdGen)

import Test.GenCheck.System.SimpleCheck
           (stdTest, stdTestArgs, stdReport, stdReportArgs, simpleReport, SimpleResults)

-- because we're doing more than just the usual stuff
import Test.GenCheck
import Test.GenCheck.Generator.BaseGens (genIntAll, genLowCharAll, genUpperCharRnd)

-- the specification, module, and GenCheck instances for the Zipper data type
import PropListZip
```

```
import ListZipper as Zip
import ListZipper_GC()

genZip_Char :: StandardGens (Zipper Char)
genZip_Char = substStdGenAll (stdTestGens :: StandardGens (Zipper Label))
                             genLowCharAll

genValZip_Char :: Rank → [(Char, Zipper Char)]
genValZip_Char r =
  let gc = (genUpperCharRnd (mkStdGen 657985498)) -- random value generator
      gz = genAll genZip_Char -- exhaustive zipper generator
  in zip (gc 1) (gz r) -- (Char, Zipper Char) generator with random Char, exhaustive zippers

testFoldrz_1, testFoldrz_2 :: Rank → Count → IO (SimpleResults (Zipper Char))
testFoldrz_1 r n =
  let propFoldrz_str = propFoldrz (:) "" :: Property (Zipper Char)
      lbl = "Compare concatenating strings with right fold, up to rank " ++
(show r)
  in  stdTestArgs genZip_Char lbl r propFoldrz_str n
testFoldrz_2 r n =
  let propFoldrz_str = propFoldrz (:) "" :: Property (Zipper Char)
      lbl = "Compare concatenating strings with right fold, up to rank " ++
(show r)
  in  stdReportArgs genZip_Char lbl r propFoldrz_str n

testFoldlz_1, testFoldlz_2 :: Rank → Count → IO (SimpleResults [Int])
testFoldlz_1 r n =
  let propFoldlz_sum = propFoldlz (+) 0 :: Property [Int]
      lbl = "Compare sum over integer list to left fold, up to rank " ++ (show r)
  in  stdTest propFoldlz_sum n
testFoldlz_2 r n =
  let propFoldlz_sum = propFoldlz (+) 0 :: Property [Int]
      lbl = "Compare sum over integer list to left fold, up to rank " ++ (show r)
  in  stdReport propFoldlz_sum n

testInsDelRepl_1 :: Rank → IO (SimpleResults (Char, Zipper Char))
testInsDelRepl_1 r =
  let lbl = "Compare insert/delete to replace, up to rank " ++ (show r)
      -- do up to maxr tests for zippers up to rank maxr
      instruct maxr = [(rk, toInteger maxr) | rk ← [1..maxr]]
      suite = genSuite genValZip_Char (instruct r)
  in  simpleReport lbl propInsDelRepl suite
```

2.5 Generating Simple Structures with Significant Elements

Generating test cases is generally decomposed into two phases: generating structures, and then populating them with base type elements. For some properties, such as the ListZipper example above, the structure of the type is the most significant aspect of the values. However, in other situations, the values of the

elements in the structure are very important to the properties being tested. Of course, some properties just require a single base type, but if there are even two base type inputs to the property the test cases are at least a product of the two. GenCheck provides a rich API to control the generation of both the structural and content aspects of the test cases.

The pair tuple is an example of a type where the content is more important than the structure. There is only one "shape" of a pair, namely (A, B), so the structure generator consists of a single value at rank 2 and A and B are labels that represent the sorts of the elements of the pair. The pair is the simplest instance of the Structure2 class, which identifies the substitution2 method for populating pairs from two generators, one of each type of elements for the pair. The implementation of the tuple in — Generator.StructureGens — is shown below:

```
instance Structure2 (,) where
  substitute2 _ (x:xs) (y:ys) = (Just (x,y), xs, ys)
  substitute2 _ xs ys = (Nothing, xs, ys)

genTpl :: Generator (Label, Label)
genTpl r | r == 2    = [(A,B)]
genTpl _ | otherwise = []

genABTpl = subst2N
```

The tuple template is populated by substituting values for the Labels using the substitution combinators from the — Generator.Substitution — module. These combinators provide several different strategies to combine the two sorts of elements. An example the two is given here, where :

genABTpl_20x20 is a generator of up to 400 elements taken as the Cartesian product of the first 20 elements of each generator, or as many exist,

genABTpl_All assuming genB at least is finite, provides all of the pairs in A dominant order

```
genABTpl_20x20 = subst2N 20 genA genB

genABTpl_All = subst2All genA genB
```

The choice and result of the substitution strategies are clearly impacted by the nature of the element generators, specifically whether they are finite and the order they impose upon their values.

Note that the elements substituted are drawn from rank 1 of the generator. This could be because they are base types or through a generator combinator that flattens the ranks of the values arbitrarily to 1; either way they are treated as "flat" or rank-less values. If the rank of the elements was relevant, composition combinators from the —Generator.Composition— module should be used instead of substitution. In principle, composed structures may still receive substituted elements but substituted structures are complete.

2.5.1 Decimal Example

Create generators and an instance of Testable for an adapted version of th DecimalRaw data structure from the Decimal package. The local copy of DecimalRaw drops the Integral constraint from the data statement to avoid the need for the DatatypeContexts extension and makes the precision a type variable to clearly identify it as a two sorted structure.

Two different approaches for generating decimals will be used. The first uses substitution for both the precision and mantissa values. The second creates a single enumeration for the entire set of decimal values, and then standard generators over the entire set.

```
{-# LANGUAGE FlexibleInstances #-}

module Decimal_GC where

-- import Data.Decimal
import Data.Word (Word8)
import System.Random (StdGen, split)

import Test.GenCheck (Structure(..), Structure2(..), subst2N, Label(..),
                      Enumerated(..), Enumeration)
import Test.GenCheck.Generator.Enumeration (mkEnumeration)
import Test.GenCheck.Generator.BaseEnum
-- (BaseEnum, EnumGC(base), enumList,
--   baseEnumGCStdGens     beMemoize, beProd)
import Test.GenCheck.Generator.Generator (Generator, stdEnumGens,
                      enumGens, StandardGens(..), Testable(..))
import Test.GenCheck.Generator.BaseGens (baseEnumGCGens,baseEnumGCStdGens)
import Test.GenCheck.Generator.Substitution( substStdGenStd )
```

The DecimalRaw structure represents a fixed precision decimal expansion, stored as decimal places (precision) and a mantissa. It is a two sorted structure, so an instance of Structure2.

```
data DecimalRaw w8 i = Decimal {  -- (Integral i) constraint dropped
      decimalPlaces :: ! w8,
      decimalMantissa :: ! i}

instance (Integral w8, Integral i, Show w8, Show i) ⇒ Show (DecimalRaw w8 i) where
   showsPrec _ (Decimal e n)
       | e == 0     = (concat [signStr, strN] ++)
       | otherwise  = (concat [signStr, intPart, ".", fracPart] ++)
       where
         strN = show $ abs n
         signStr = if n < 0 then "-" else ""
         len = length strN
         padded = replicate (fromIntegral e + 1 - len) '0' ++ strN
         (intPart, fracPart) = splitAt (max 1 (len - fromIntegral e)) padded

{-
```

```
instance (Show w8, Show i) ⇒ Show (DecimalRaw w8 i) where
  show (Decimal p m) = (show m) ++ " x10^" ++ (show p)
-}
instance Structure2 DecimalRaw where
  substitute2 = subDec
subDec :: DecimalRaw a b → [j] → [i] → (Maybe (DecimalRaw j i), [j], [i])
subDec (Decimal _ _) (p:ps) (m:ms) = (Just (Decimal p m), ps, ms)
subDec _ ps ms = (Nothing, ps, ms)
```

There is only one possible Decimal structure: the labelled product — Decimal _ _ —. The generator for these structures is therefor a single value of rank 2; the Label type provides constants to identify that the sorts are different.

```
genDecStruct :: Generator (DecimalRaw Label Label)
genDecStruct r | r == 2    = [Decimal A B]
genDecStruct _ | otherwise = []
```

For testing, fix the precision as a Word8 type and create a base enumeration from the list of values, then define the standard GenCheck (base) generators.

```
enumW8 = enumList ([minBound..maxBound]::[Word8])
stdGensW8 = baseEnumGCGens enumW8
```

Instead of fixing the mantissa type, require it to be an instance of EnumGC. This guarantees a base enumeration is available to define standard generators. Integer mantissa's can also be used - an arbitrary range of double the min / max bounds of the Ints is used to set the bounds on the enumeration (see Test.GenCheck.Generator.BaseEnum for details).

The full generators for populated DecimalRaw values are given by substituting the mantissa and precision generators into the Decimal structure generator. The subst2N function creates n x n instances of each two sort structure populated, in this case 100 instances of the unique DecimalRaw structure.

The mantissa and precision generators have independent sampling strategies, so the DecimalRaw generator can have hybrid sampling strategies. The three below mix the random and extreme (boundary) strategies, but that is arbitrary.

```
genDecXtrmXtrm ::  (EnumGC i) ⇒ Generator (DecimalRaw Word8 i)
genDecXtrmXtrm  = subst2N 10 genDecStruct gW8x gix
   where gix    = genXtrm baseEnumGCStdGens
         gW8x   = genXtrm stdGensW8

genDecRndXtrm ::  (EnumGC i) ⇒ StdGen → Generator (DecimalRaw Word8 i)
genDecRndXtrm s  = subst2N 10 genDecStruct (gW8r s) gix
   where gix      = genXtrm baseEnumGCStdGens
         gW8r s'  = genRand stdGensW8 s'

genDecXtrmRnd ::  (EnumGC i) ⇒ StdGen → Generator (DecimalRaw Word8 i)
genDecXtrmRnd s  = subst2N 10 genDecStruct gW8x (gir s)
   where gir  s' = genRand baseEnumGCStdGens s'
         gW8x    = genXtrm stdGensW8
```

```
genDecRndRnd ::  (EnumGC i) ⇒ StdGen → StdGen → Generator (DecimalRaw Word8 i)
genDecRndRnd s1 s2 = subst2N 10 genDecStruct (gW8r s1) (gir s2)
   where gir  s'   = genRand baseEnumGCStdGens s'
         gW8r s'   = genRand stdGensW8 s'
```

As an alternative, we can define an enumeration over the entire set of precisions for the DecimalRaw values, creating the standard generators over those and then incorporating the generators for the mantissa. This approach is useful when the precision has few choices, for example, the 256 valued Word8 type.

The standard generators for the DecimalRaw structure will be built pairing the strategies for both the precision and mantissa values.

Note that in this situation, all of the decimals are considered to be of rank 1, because only the mantissa is a type variable.

```
type DecimalW8 = DecimalRaw Word8

enumDecW8 :: Enumeration (DecimalW8) Label
enumDecW8 = mkEnumeration c s
  where c r | r == 1 = baseCounter enumW8
        c _ | otherwise = 0
        s r i | r == 1 = (λp → Decimal p A) (baseSelector enumW8 i)
        s _ _ | otherwise = undefined

instance Functor (DecimalW8) where
  fmap f (Decimal p m) = Decimal p (f m)

instance Enumerated (DecimalW8) where
  enumeration = enumDecW8

instance Structure (DecimalW8) where
  substitute (Decimal p _) (x:xs) = (Just (Decimal p x), xs)
  substitute      _           []  = (Nothing, [])

stdGensDecLbl = stdEnumGens :: StandardGens ((DecimalW8) Label)

instance (Integral i, Show i, EnumGC i) ⇒ Testable (DecimalW8 i) where
  stdTestGens = substStdGenStd stdGensDecLbl baseEnumGCStdGens
```

Comparing the results of the two approaches clarifies when a two sort substitution is more or less appropriate than first enumerating the first sort, then the second.

For example, the extreme mantissa / extreme precision generator will be useful for testing Decimal arithmetic functions as errors are most likely in extreme values.

Note that the arbitrary choice of the range of Integers to test comes up here.

```
*Decimal_GC> take 100 $ genDecXtrmXtrm 2 :: [DecimalRaw Word8 Integer]
[-4294967297,4294967293,-4294967296,4294967292,-4294967295,4294967291,
-4294967294,4294967290,-4294967293,4294967289,-0.0000000...0004294967292,
0.000000000000...0000004294967288,-0.000...0004294967291,
```

```
...

-429496728.7,429496728.3,-429496728.6,429496728.2,-429496728.5,
429496728.1,-429496728.4,429496728.0,-429496728.3,429496727.9,
-0.000...000004294967282, 0.000...000004294967278,

...
```

The values are grouped in 10's because of the arbitrary choice made in the subst2N function. The first 10 have precision 0, the second 10 precision 255, the third 10 precision 1, fourth 10 precision 254, etc. (some of the high precision values been ellided for legibility).

The random mantissa / extreme precision generator might be used to supplement testing to ensure that there are no dependencies on extreme integer values.

```
*Decimal_GC> take 100 $ genDecXtrmRnd (mkStdGen 20398423) 2 :: [DecimalRaw Word8 Integer]
3297478285,-2103007409,-1570044525,3853764760,-553397225,4078554636,
-994370392,-3197401073,-2259414665,-4098040193,-0.000...0003663556986,
-0.000...00000000004161385473,0.000000...00002455986108,

...

292414849.3,57118104.6,293910192.6,-173351886.7,-351335950.1,-242506370.5,
-100176922.4,333869685.3,-1741099.5,359802003.0,0.0000...002669858400,
0.00000000...000003178902858,

...
```

For completeness, there should be some tests on intermediate precision values, such as provided by the random mantissa / random precision generator.

```
*Decimal_GC> take 40 $ genDecRndRnd (mkStdGen 873486) (mkStdGen 20398423) 2 :: [DecimalRaw W
[0.0000... 000003297478285,-0.0000...00002103007409,-0.0000...00001570044525,
0.00000...0000003853764760,
...
```

It is worth noting that this test suite is quite unlike that which would be prepared by a human test writer, which is why the automatic testing is so well suited to complement manual test cases.