Less Arbitrary waiting time

Short paper

Michał J. Gajda

Abstract

Property testing is the cheapest and most precise way of building up a test suite for your program. Especially if the datatypes enjoy nice mathematical laws. But it is also the easiest way to make it run for an unreasonably long time. We prove connection between deeply recursive data structures, and epidemic growth rate, and show how to fix the problem, and make Arbitrary instances run in linear time with respect to assumed test size.

1 Introduction

Property testing is the cheapest and most precise way of building up a test suite for your program. Especially if the datatypes enjoy nice mathematical laws. But it is also the easiest way to make it run for an unreasonably long time. We show that connection between deeply recursive data structures, and epidemic growth rate can be easily fixed with a generic implementation. After our intervention the Arbitrary instances run in linear time with respect to assumed test size. We also provide a fully generic implementation, so error-prone coding process is removed.

2 Motivation

Typical arbitrary instance just draws a random constructor from a set, possibly biasing certain outcomes.

Generic arbitrary instance looks like this:

```
data Tree          α =
      Leaf         α
   |  Branch [Tree α ]
   deriving   ( Eq,Show ,Generic.Generic)
instance Arbitrary            α
      => Arbitrary ( Tree  α ) where
   arbitrary = oneof   [ Leaf    <$> arbitrary
                       , Branch <$> arbitrary
                       ]
```

Assuming we run QuickCheck with any size parameter greater than 1, it will fail to terminate!

List instance is a wee bit better, since it tries to limit maximum list length to a constant option:

```
instance Arbitrary α
      => Arbitrary [ α ] where
   lessArbitrary =     sized $ \ size do
     len <-   choose   (1,size  )
     vectorOf len lessArbitrary
```

Indeed QuickCheck manual[10], suggests an error-prone, manual method of limiting the depth of generated structure by dividing size by reproduction factor of the structure[1] :

```
data Tree = Leaf Int | Branch Tree Tree

instance Arbitrary  Tree where
   arbitrary = sized      tree'
      where tree'  0   = Leaf <$> arbitrary
         tree' n   |  n > 0  =
            oneof  [ Leaf   <$> arbitrary,
                 Branch <$>   subtree <*>    subtree]
            where subtree  = tree'   (n  `div` 2)
```

Above example uses division of size by maximum branching factor to decrease coverage into relatively deep data structures, whereas dividing by average branching factor of ~2 will generate both deep and very large structures.

This fixes non-termination issue, but still may lead to unpredictable waiting times for nested structures. The depth of the generated structure is linearly limited by dividing the n by expected branching factor of the recursive data structure. However this does not work very well for mutually recursive data structures occuring in compilers[3], which may have 30 constructors with highly variable[2] branching factor just like GHC's HSExpr data types.

Now we have a choice of manual generation of these data structures, which certainly introduces bias in testing, or abandoning property testing for real-life-sized projects.

3 Complexity analysis

We might be tempted to compute average size of the structure. Let's use reproduction rate estimate for a single rewrite of arbitrary function written in conventional way.

We compute a number of recursive references for each constructor. Then we take an average number of references among all the constructors. If it is greater than 1, any **non-lazy** property test will certainly fail to terminate. If it is slightly smaller, we still can wait a long time.

What is an issue here is not just non-termination which is fixed by error-prone manual process of writing own instances that use explicit size parameter.

The much worse issue is unpredictability of the test runtime. Final issue is the poor coverage for mutually recursive data structure with multitude of constructors.

[1]We changed liftM and liftM2 operators to <$> and <*> for clarity and consistency.

[2]Due to list parameters.

GPCE, November, 2020, Illinois, USA
2020.

Given a *maximum size* parameter (as it is now called) to QuickCheck, would we not expect that tests terminate within linear time of this parameter? At least if our computation algorithms are linear with respect to input size?

Currently for any recursive structure like Tree a, we see some exponential function. For example $size^n$, where n is a random variable.

4 Solution

We propose to replace implementation with a simple state monad[7] that actually remembers how many constructors were generated, and thus avoid limiting the depth of generated data structures, and ignoring estimation of branching factor altogether.

```
newtype Cost = Cost Int
  deriving ( Eq , Ord, Enum ,Bounded,Num)
newtype CostGen α =
        CostGen {
          runCostGen :: State.StateT Cost QC.Gen α }
  deriving (Functor, Applicative, Monad, State.MonadFix)
```

We track the spending in the usual way:

```
spend :: Cost -> CostGen ()
spend γ = do
  CostGen $ State.modify (-γ+)
  checkBudget
```

To make generation easier, we introduce budget check operator:

```
($$$?) :: HasCallStack
       => CostGen α
       -> CostGen α
       -> CostGen α
cheapVariants $$$? costlyVariants = do
  budget <- CostGen State.get
  if | budget > (0 :: Cost) -> costlyVariants
     | budget > −10000      -> cheapVariants
     | otherwise            -> error $
       "Recursive structure with no loop breaker."
checkBudget :: HasCallStack => CostGen ()
checkBudget = do
  budget <- CostGen State.get
  if budget < −10000
    then error "Recursive structure with no loop breaker."
    else return ()
```

In order to conveniently define our budget generators, we might want to define a class for them:

```
class LessArbitrary α where
  lessArbitrary :: CostGen α
  default lessArbitrary :: ( Generic α
                           , GLessArbitrary (Rep α ))
                        => CostGen α
  lessArbitrary = genericLessArbitrary
```

Then we can use them as implementation of arbitrary that should have been always used:

```
fasterArbitrary :: LessArbitrary α => QC.Gen α
fasterArbitrary = sizedCost lessArbitrary

sizedCost :: CostGen α -> QC.Gen α
sizedCost gen = QC.sized ( `withCost` gen)
```

Then we can implement Arbitrary instances simply with:

```
instance _
      => Arbitrary α where
  arbitrary = fasterArbitrary
```

Of course we still need to define LessArbitrary, but after seeing how simple was a Generic defintion Arbitrary we have a hope that our implementation will be:

```
instance LessArbitrary where
```

That is - we hope that the the generic implementation will take over.

5 Introduction to GHC generics

Generics allow us to provide default instance, by encoding any datatype into its generic Representation:

```
instance Generics ( Tree α ) where
  to :: Tree α -> Rep ( Tree α )
  from :: Rep (Tree α) -> Tree α
```

The secret to making a generic function is to create a set of instance declarations for each type family constructor.

So let's examine Representation of our working example, and see how to declare instances:

1. First we see datatype metadata D1 that shows where our type was defined:

```
type instance Rep (Tree α) =
  D1
   ( 'MetaData "Tree"
               "Test.Arbitrary"
               "less-arbitrary" 'False)
```

2. Then we have constructor metadata C1:

```
    (C1
        ('MetaCons "Leaf" 'PrefixI 'False)
```

3. Then we have metadata for each field selector within a constructor:

```
        (S1
            ('MetaSel
                'Nothing
                'NoSourceUnpackedness
                'NoSourceStrictness
                'DecidedLazy)
```

4. And reference to another datatype in the record field value:

```
            (Rec0 α))
```

5. Different constructors are joined by sum type operator:

```
    :+:
```

6. Second constructor has a similar representation:

```
C1
      ( 'MetaCons "Branch" 'PrefixI 'False)
      (S1
         ('MetaSel
            'Nothing
            'NoSourceUnpackedness
            'NoSourceStrictness
            'DecidedLazy )
         (Rec0 [Tree  α ])))
          ignored
```

7. Note that Representation type constructors have additional parameter that is not relevant for our use case.

For simple datatypes, we are only interested in three constructors:

- :+: encode choice between constructors
- :*: encode a sequence of constructor parameters
- M1 encode metainformation about the named constructors, C1, S1 and D1 are actually shorthands for M1 C, M1 S and M1 D

There are more short cuts to consider: * U1 is the unit type (no fields) * Rec0 is another type in the field

5.1 Example of generics

This generic representation can then be matched by generic instances. Example of Arbitrary instance from [5] serves as a basic example[3]

1. First we convert the type to its generic representation:

```
genericArbitrary :: (Generic α
                   , Arbitrary (Rep α ))
                  => Gen α
genericArbitrary = to <$> arbitrary
```

2. We take care of nullary constructors with:

```
instance Arbitrary G.U1 where
  arbitrary = pure G.U1
```

3. For all fields arguments are recursively calling Arbitrary class method:

```
instance Arbitrary γ => Arbitrary (G.K1 i γ) where
  gArbitrary = G.K1 <$> arbitrary
```

4. We skip metadata by the same recursive call:

```
instance Arbitrary f
      => Arbitrary (G.M1 i γ f) where
  arbitrary = G.M1 <$> arbitrary
```

5. Given that all arguments of each constructor are joined by :*:, we need to recursively delve there too:

```
instance ( Arbitrary α,
         , Arbitrary β )
      => Arbitrary ( α G.:*: β ) where
  arbitrary = (G.:*: ) <$> arbitrary <*> arbitrary
```

6. In order to sample all constructors with the same probability we compute a number of constructor in each representation type with SumLen type family:

```
type family SumLen α :: Nat where
  SumLen ( α G.:+: β ) = (SumLen α) + (SumLen β)
  SumLen α             = 1
```

Now that we have number of constructors computed, we can draw them with equal probability:

```
instance ( Arbitrary α
         , Arbitrary β
         , KnownNat (SumLen α)
         , KnownNat (SumLen β)
         )
      => Arbitrary (α G.:+: β) where
  arbitrary = frequency
    [ ( lfreq , G.L1 <$> arbitrary )
    , ( rfreq , G.R1 <$> arbitrary ) ]
    where
      lfreq = fromIntegral
            $ natVal (Proxy :: Proxy (SumLen α))
      rfreq = fromIntegral
            $ natVal (Proxy :: Proxy (SumLen β))
```

Excellent piece of work, but non-terminating for recursive types with average branching factor greater than 1 (and non-lazy tests, like checking Eq reflexivity.)

5.2 Implementing with Generics

It is apparent from our previous considerations, that we can reuse code from the existing generic implementation when the budget is positive. We just need to spend a dollar for each constructor we encounter.

For the Monoid the implementation would be trivial, since we can always use mempty and assume it is cheap:

```
genericLessArbitraryMonoid :: (Generic α
                             , GLessArbitrary (Rep α )
                             , Monoid α )
                            => CostGen α
genericLessArbitraryMonoid =
  pure ∅ $$$? genericLessArbitrary
```

However we want to have fully generic implementation that chooses the cheapest constructor even though the datatype does not have monoid instance.

5.2.1 Class for budget-conscious

When the budget is low, we need to find the least costly constructor each time.

So to implement it as a type class GLessArbitrary that is implemented for parts of the Generic Representation type, we will implement two methods:

1. gLessArbitrary is used for normal random data generation
2. cheapest is used when we run out of budget

[3] We modified class name to simplify.

```
class GLessArbitrary datatype where
  gLessArbitrary :: CostGen (datatype p)
  cheapest       :: CostGen (datatype p)

genericLessArbitrary :: (Generic α
                        , GLessArbitrary (Rep α))
                     => CostGen α
genericLessArbitrary = G.to <$> gLessArbitrary
```

5.2.2 Helpful type family

First we need to compute minimum cost of the in each branch of the type representation. Instead of calling it *minimum cost*, we call this function Cheapness.

For this we need to implement minimum function at the type level:

```
type family Min m n where
  Min m n = ChooseSmaller (CmpNat m n) m n

type family ChooseSmaller (o :: Ordering)
                          (m :: Nat)
                          (n :: Nat) where
  ChooseSmaller 'LT m n = m
  ChooseSmaller 'EQ m n = m
  ChooseSmaller 'GT m n = n
```

so we can choose the cheapest^[We could add instances for :

```
type family Cheapness α :: Nat where
  Cheapness (α :*: β) =
        Cheapness α + Cheapness β
  Cheapness (α :+: β) =
    Min (Cheapness α) (Cheapness β)
  Cheapness U1 = 0
  ≪ flat-types≫
  Cheapness (K1 α other) = 1
  Cheapness (C1 α other) = 1
```

Since we are only interested in recursive types that can potentially blow out our budget, we can also add cases for flat types since they seem the cheapest:

```
Cheapness (S1 α (Rec0 Int       )) = 0
Cheapness (S1 α (Rec0 Scientific)) = 0
Cheapness (S1 α (Rec0 Double    )) = 0
Cheapness (S1 α (Rec0 Bool      )) = 0
Cheapness (S1 α (Rec0 Text.Text )) = 1
Cheapness (S1 α (Rec0 other     )) = 1
```

5.2.3 Base case for each datatype

For each datatype, we first write a skeleton code that first spends a coin, and then checks whether we have enough funds to go on expensive path, or we are beyond our allocation and need to generate from among the cheapest possible options.

```
instance GLessArbitrary f
      => GLessArbitrary (D1 m f) where
  gLessArbitrary = do
    spend 1
    M1 <$> (cheapest $$$? gLessArbitrary)
  cheapest = M1 <$> cheapest
```

5.2.4 Skipping over other metadata

First we safely ignore metadata by writing an instance:

```
instance GLessArbitrary f
      => GLessArbitrary (G.C1 γ f) where
  gLessArbitrary = G.M1 <$> gLessArbitrary
  cheapest       = G.M1 <$> cheapest

instance GLessArbitrary f
      => GLessArbitrary (G.S1 γ f) where
  gLessArbitrary = G.M1 <$> gLessArbitrary
  cheapest       = G.M1 <$> cheapest
```

5.2.5 Counting constructors

In order to give equal draw chance for each constructor, we need to count number of constructors in each branch of sum type :+: so we can generate each constructor with the same frequency:

```
type family SumLen α :: Nat where
  SumLen (α G.:+: β) = SumLen α + SumLen β
  SumLen α           = 1
```

5.2.6 Base cases for GLessArbitrary

Now we are ready to define the instances of GLessArbitrary class.

We start with base cases GLessArbitrary for types with the same representation as unit type has only one result:

```
instance GLessArbitrary G.U1 where
  gLessArbitrary = pure G.U1
  cheapest       = pure G.U1
```

For the product of, we descend down the product of to reach each field, and then assemble the result:

```
instance (GLessArbitrary α
         , GLessArbitrary β)
      => GLessArbitrary (α G.:*: β) where
  gLessArbitrary = (G.:*:) <$> gLessArbitrary
                           <*> gLessArbitrary
  cheapest = (G.:*:)       <$> cheapest
                           <*> cheapest
```

We recursively call instances of LessArbitrary for the types of fields:

```
instance LessArbitrary γ
      => GLessArbitrary (G.K1 i γ) where
  gLessArbitrary = G.K1 <$> lessArbitrary
  cheapest       = G.K1 <$> lessArbitrary
```

5.2.7 Selecting the constructor

We use code for selecting the constructor that is taken after[5].

```
instance  ( GLessArbitrary α
          , GLessArbitrary β
          , KnownNat (SumLen α)
          , KnownNat (SumLen β)
          , KnownNat (Cheapness α)
          , KnownNat (Cheapness β)
          )
      =>  GLessArbitrary (α Generic.:+: β) where
  gLessArbitrary =
    frequency
      [ ( lfreq  ,  L1 <$> gLessArbitrary )
      , ( rfreq  ,  R1 <$> gLessArbitrary ) ]
    where
      lfreq  =  fromIntegral
             $  natVal (Proxy :: Proxy (SumLen α))
      rfreq  =  fromIntegral
             $  natVal (Proxy :: Proxy (SumLen β))
  cheapest =
      if  lcheap  ≤  rcheap
          then  L1  <$>  cheapest
          else  R1  <$>  cheapest
    where
      lcheap,  rcheap :: Int
      lcheap     =  fromIntegral
                 $  natVal (Proxy :: Proxy (Cheapness α))
      rcheap     =  fromIntegral
                 $  natVal (Proxy :: Proxy (Cheapness β))
```

6 Conclusion

We show how to quickly define terminating test generators using generic programming. This method may be transferred to other generic programming regimes like Featherweight Go or Featherweight Java.

We recommend it to reduce time spent on making test generators.

7 Bibliography

[1] Claessen, K. and Hughes, J. 2000. QuickCheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.* 35, 9 (Sep. 2000), 268–279. DOI:https://doi.org/10.1145/357766.351266.

[2] Claessen, K. and Hughes, J. 2000. QuickCheck: A lightweight tool for random testing of haskell programs. *ICFP '00: Proceedings of the fifth acm sigplan international conference on functional programming* (New York, NY, USA, 2000), 268–279.

[3] Day, L.E. and Hutton, G. 2013. Compilation à la Carte. *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages* (Nijmegen, The Netherlands, 2013).

[4] EnTangleD: A bi-directional literate programming tool: 2019. *https://blog.esciencecenter.nl/entangled-1744448f4b9f.*

[5] generic-arbitrary: Generic implementation for QuickCheck's Arbitrary: 2017. *https://hackage.haskell.org/package/generic-arbitrary-0.1.0/docs/src/Test-QuickCheck-Arbitrary-Generic.html#genericArbitrary.*

[6] genvalidity-property: Standard properties for functions on 'Validity' types: 2018. *https://hackage.haskell.org/package/generic-arbitrary-0.1.0/docs/src/Test-QuickCheck-Arbitrary-Generic.html#genericArbitrary.*

[7] Jones, M.P. and Duponcheel, L. 1993. *Composing monads.*

[8] Knuth, D.E. 1984. Literate programming. *Comput. J.* 27, 2 (May 1984), 97–111. DOI:https://doi.org/10.1093/comjnl/27.2.97.

[9] Pandoc: A universal document converter: 2000. *https://pandoc.org.*

[10] QuickCheck: An Automatic Testing Tool for Haskell: *http://www.cse.chalmers.se/~rjmh/QuickCheck/manual_body.html#16.*

[11] stack 0.1 released:.

Appendix: Module headers

```
{-# language DefaultSignatures #-}
{-# language FlexibleInstances #-}
{-# language FlexibleContexts #-}
{-# language GeneralizedNewtypeDeriving #-}
{-# language Rank2Types #-}
{-# language PolyKinds #-}
{-# language MultiParamTypeClasses #-}
{-# language MultiWayIf #-}
{-# language ScopedTypeVariables #-}
{-# language TypeApplications #-}
{-# language TypeOperators #-}
{-# language TypeFamilies #-}
{-# language TupleSections #-}
{-# language UndecidableInstances #-}
{-# language AllowAmbiguousTypes #-}
{-# language DataKinds #-}
module Test.LessArbitrary(
    LessArbitrary(..)
  , oneof
  , choose
  , budgetChoose
  , CostGen (..)
  , ( <$$$> )
  , ( $$$?)
  , currentBudget
  , fasterArbitrary
  , genericLessArbitrary
  , genericLessArbitraryMonoid
  , flatLessArbitrary
  , spend
  , withCost
  , elements
  , forAll
  , sizedCost
  ) where

import qualified Data.HashMap.Strict as Map
import qualified Data.Set as Set
import qualified Data.Vector as Vector
import qualified Data.Text as Text
import Control.Monad(replicateM)
import Data.Scientific
import Data.Proxy
import qualified Test.QuickCheck.Gen as QC
import qualified Control.Monad.State.Strict as State
import Control.Monad.Trans.Class
import System.Random (Random)
import GHC.Generics as G
import GHC.Generics as Generic
import GHC.TypeLits
import GHC.Stack
import qualified Test.QuickCheck as QC
import Data.Hashable

import Test.LessArbitrary.Cost

≪ costgen ≫
```


Appendix: lifting classic Arbitrary functions

Below are functions and instances that are lightly adjusted variants of original implementations in QuickCheck[1]

```
instance LessArbitrary α
      => LessArbitrary [ α ] where
  lessArbitrary = pure [ ] $$$? do
    budget <- currentBudget
    len <- choose ( 1,fromEnum budget)
    spend $ Cost len
    replicateM len lessArbitrary

instance QC.Testable α
      => QC.Testable (CostGen α ) where
  property = QC.property
           . sizedCost
```

Remaining functions are directly copied from QuickCheck[2], with only adjustment being their types and error messages:

```
forAll :: CostGen α -> (α -> CostGen β) -> CostGen β
forAll gen prop = gen >>= prop

oneof :: HasCallStack
      => [CostGen α] -> CostGen α
oneof [] = error
           ”LessArbitrary.oneof used with empty list”
oneof gs = choose (0,length gs - 1) >>= (gs !!)

elements :: [ α ] -> CostGen α
elements gs = ( gs!! ) <$> choose (0,length gs - 1)

choose :: Random α
       => (α , α)
       -> CostGen α
choose (α,β) = CostGen $ lift $ QC.choose (α, β)

-- | Choose but only up to the budget (for array and list sizes)
budgetChoose :: CostGen Int
budgetChoose = do
  Cost β <- currentBudget
  CostGen $ lift $ QC.choose (1, β)

-- | Version of ‘suchThat‘ using budget instead of sized generators.
cg ‘suchThat‘ pred = do
  result <- cg
  if pred result
    then return result
    else do
      spend 1
      cg ‘suchThat‘ pred
```

This key function, chooses one of the given generators, with a weighted random distribution. The input list must be non-empty. Based on QuickCheck[1].

```
frequency          :: HasCallStack
                   => [(Int, CostGen α)] -> CostGen α
frequency      []  =
  error   $  "LessArbitrary.frequency "
          ++ "used with empty list"
frequency      xs
  | any (     <  0 ) (map fst xs) =
    error      $  "LessArbitrary.frequency: "
               ++ "negative weight"
  | all (      ==  0) (map fst xs) =
    error     $   "LessArbitrary.frequency: "
              ++  "all weights were zero"
frequency      xs0 = choose (1, tot) >>=      (`pick` xs0)
 where
  tot    = sum (map fst xs0)

  pick     n  (  ( k ,x): xs )
    |    n  ≤    k     =  x
    |    otherwise     =  pick (n-k) xs
  pick    _ _ = error
    "LessArbitrary.pick used with empty list"
```

Appendix: test suite

As observed in [6], it is important to check basic properties of Arbitrary instance to guarantee that shrinking terminates:

```
shrinkCheck ::    ∀             term .
                 ( Arbitrary  term
                 , Eq         term  )
              =>              term
              ->  Bool
shrinkCheck  term    =
  term `notElem`     shrink term

arbitraryLaws ::      ∀           ty .
                     ( Arbitrary  ty
                     , Show       ty
                     , Eq         ty )
                  => Proxy        ty
                  -> Laws
arbitraryLaws    (Proxy     :: Proxy ty) =
  Laws  "arbitrary"
          [( "does not shrink to itself"   ,
             property (shrinkCheck :: ty   -> Bool))]
```

For LessArbitrary we can also check that empty budget results in choosing a cheapest option, but we need to provide a predicate that confirms what is actually the cheapest:

```
otherLaws   ::   [Laws]
otherLaws   =  [ lessArbitraryLaws isLeaf]
  where
    isLeaf    :: Tree     Int    -> Bool
    isLeaf    ( Leaf      _ )  =  True
    isLeaf    ( Branch _ )  =  False

lessArbitraryLaws  ::        LessArbitrary  α
                   =>        (α -> Bool )  ->    Laws
lessArbitraryLaws   cheapestPred         =
      Laws  "LessArbitrary"
            [( "always selects cheapest",
               property $
                 prop_alwaysCheapest cheapestPred)]

prop_alwaysCheapest      ::   LessArbitrary  α
                         =>   (α -> Bool )  ->    Gen Bool
prop_alwaysCheapest      cheapestPred     =
  cheapestPred <$>          withCost 0 lessArbitrary
```

Again some module headers:

```
{-# language DataKinds #-}
{-# language FlexibleInstances #-}
{-# language Rank2Types #-}
{-# language MultiParamTypeClasses #-}
{-# language ScopedTypeVariables #-}
{-# language TypeOperators #-}
{-# language UndecidableInstances #-}
{-# language AllowAmbiguousTypes #-}
module  Test.Arbitrary(
          arbitraryLaws
      )   where

import   Data.Proxy
import   Test.QuickCheck
import   Test.QuickCheck.Classes
import   qualified Data.HashMap.Strict  as Map
import            Data.HashMap.Strict ( HashMap)

≪ arbitrary-laws≫
```

And we can compare the tests with LessArbitrary (which terminates fast, linear time):

```
≪ test - file - header ≫
≪ test - less - arbitrary-version ≫

≪ test - file-laws          ≫
≪ less - arbitrary      -check≫
```

Appendix: non-terminating test suite

Or with a generic Arbitrary (which naturally hangs):

```
≪ test - file    - header≫
≪ tree - type - typical-   arbitrary ≫
otherLaws =     []
≪ test-file     - laws≫
```

Here is the code:

```
{-# language FlexibleInstances #-}
{-# language Rank2Types #-}
{-# language MultiParamTypeClasses #-}
{-# language ScopedTypeVariables #-}
{-# language TypeOperators #-}
{-# language UndecidableInstances #-}
{-# language AllowAmbiguousTypes #-}
{-# language DeriveGeneric #-}
module Main where

import Data.Proxy
import Test.QuickCheck
import qualified GHC.Generics as Generic
import Test.QuickCheck.Classes

import Test.LessArbitrary
import Test.Arbitrary

<<tree-type>>
instance LessArbitrary           α
      => LessArbitrary (Tree α ) where

instance LessArbitrary    α
      => Arbitrary (Tree α ) where
  arbitrary = fasterArbitrary

main :: IO ()
main = do
  lawsCheckMany
    [( "Tree",
      [ arbitraryLaws (Proxy :: Proxy (Tree Int ))
      , eqLaws        (Proxy :: Proxy (Tree Int ))
      ] ◇ otherLaws) ]
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Test.LessArbitrary.Cost where

<<cost>>
```

Appendix: convenience functions provided with the module

Then we limit our choices when budget is tight:

```
currentBudget :: CostGen Cost
currentBudget = CostGen State.get
-- unused: loop breaker message type name
type family ShowType k where
  ShowType (D1 ('MetaData name _ _ _) _)  = name
  ShowType other                          = "unknown type"

showType ::   ∀                           α .
              (Generic                    α
              , KnownSymbol (ShowType (Rep α )))
         =>   String
showType =    symbolVal (Proxy :: Proxy (ShowType (Rep α)))
```

Acknowledgments

The text was prepared with great help of bidirectional literate programming[8] tool[4], Pandoc[9] markdown publishing system and live feedback from Stack file watching option[11].