




Property Testing (based on notes by Stephanie Weirich)



1




Schedule

	Project presentations			
Tue Nov 26	Property Testing and QuickCheck		Quiz 5 on Tue Lecture Week13 Lecture13.hs	PS8 due on Tuesday
Tue Dec 3 Fri Dec 6	Monad Transformers		Quiz 6 on Fri	Checkpoint #2: attend office hours this week (or earlier)
Tue Dec 10	Project presentations			Project due 5-8 min presentation in class

Programming in Haskell, A Milanova 2

2




Outline

- Property-based testing with QuickCheck
 - QuickCheck properties
 - quickCheck function
 - Conditional testing
- Generating data with QuickCheck
 - Generator combinators
 - Generator monad Gen a
 - Arbitrary type class
 - Shrinking

Programming in Haskell, A Milanova 3

3




Property-based Testing

- QuickCheck is a technique that exploits type classes and monads to deliver a powerful automatic testing methodology
- QuickCheck is based on the idea of property-based testing. Instead of writing unit tests, one writes properties of a function; QuickCheck then automatically generates random tests that run and verify (or falsify) that function implements those properties
- Properties are specifications

Programming in Haskell, A Milanova 4

4




- QuickCheck emphasizes specifications. Therefore:
 1. Developer is forced to think about what the code should do
 2. Tool finds corner cases that leads to either code or specification getting fixed (property specification can be incorrect!)
 3. Specifications live on as rich, machine-checkable documentation of how the code should behave

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

5

5



Installation

- cabal install QuickCheck
- Download Lecture14.hs and code along

Programming in Haskell, A Milanova

6

6



Properties

- A QuickCheck property is a function returning a `Bool`. E.g.:

```
prop_revapp :: [Int] -> [Int] -> Bool
prop_revapp xs ys = reverse (xs ++ ys) ==
                    reverse xs ++ reverse ys
```

- Property is an assertion about the function (`reverse` in our case) that the programmer believes is true
- QuickCheck convention is to use prefix `prop_` for properties
- Note the monomorphic types: `[Int] -> [Int] -> Bool` rather than `[a] -> [a] -> Bool`. This is necessary for QuickCheck to generate test cases; naturally, it can generate tests for a concrete type (e.g. `Int`)

7

7



- To check a property, we invoke `quickCheck` with that property:


```
quickCheck :: (Testable prop) => prop -> IO ()
-- Defined in Test.QuickCheck.Test
```

- Only properties that are in the `Testable` type class can be checked. `[Int] -> [Int] -> Bool` is a “Testable” property so we can try `quickCheck` on the `prop_revapp` example
- `quickCheck` runs in the IO monad. Therefore, we’ll run it in `ghci` to see the result

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

8

8



- Back to our property:

```
prop_revapp :: [Int] -> [Int] -> Bool
prop_revapp xs ys = reverse (xs ++ ys) ==
                    reverse xs ++ reverse ys
```


- So, what happens when we run quickCheck on our property?

```
> :l Lecture14.hs
> import Test.QuickCheck
> quickCheck prop_revapp
```

Programming in Haskell, A Milanova (based on lecture by
Stephanie Weirich)

9

9



- Let's run with the counter example suggested by QuickCheck:

```
> prop_revapp [0] [1]
```


- So, what went wrong?
- Well, QuickCheck fails if either code is wrong, or property is wrong. In this case, our property is wrong. Should be:

```
prop_revapp_ok :: [Int] -> [Int] -> Bool
prop_revapp_ok xs ys = reverse (xs ++ ys) ==
                      reverse ys ++ reverse xs
```

Programming in Haskell, A Milanova (based on lecture by
Stephanie Weirich)

10

10



- Running the new property should pass:

```

> quickCheck prop_revapp_ok
+++ Ok, passed 100 tests.
```

- We can ask QuickCheck to generate and test with more than 100 tests:


```

> quickCheckN n = QC.quickCheck . QC.withMaxSuccess n
> :t quickCheckN
(Testable p) => Int -> p -> IO ()
> quickCheckN 1000 prop_revapp_ok
```

Programming in Haskell, A Milanova

11

11



QuickCheck QuickSort

- Here is a version of QuickSort:

```

qsort :: forall a. Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort lhs ++ (x : qsort rhs)
  where lhs = [y | y <- xs, y < x]
        rhs = [z | z <- xs, z > x]
```

- Testing qsort in ghci:

```

> qsort [10,9..1]
```

```


> qsort $ [2,4..20] ++ [1,3..11]
```

- Looks good, but something's not quite right. Can you spot it?

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

12

12



- What are some properties that we might want to check?
- Let's start with simplest one, whether result list is ordered


```
isOrdered (x:y:zs) =  
isOrdered ...  
isOrdered ...  
  
prop_qsort_isOrdered :: ...  
prop_qsort_isOrdered xs =
```

```
> quickCheckN 1000 prop_qsort_isOrdered
```

Programming in Haskell, A Milanova (based on lecture by
Stephanie Weirich)

13

13



- More properties. Idempotency is the property that applying qsort more than once does not change result:

```
prop_qsort_idemp :: ...  
prop_qsort_idemp xs = qsort (qsort xs) ==  
qsort xs
```

```
> quickCheckN 1000 prop_qsort_idemp
```

- Passed again, so far so good!

Programming in Haskell, A Milanova (based on lecture by
Stephanie Weirich)

14

14



Conditional Properties

- More properties. Minimal element should be head of the list and maximal element should be last of the list:

```
prop_qsort_min :: [Int] -> Bool
prop_qsort_min xs = head (qsort xs) ==
                    minimum xs
```

```
> quickCheck prop_qsort_min
*** Failed! Exception: 'Prelude.head: empty list' (after 1 test):
[]
```

- These properties make sense only if the list is non-empty

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

15

15



- We need a conditional property where output satisfied property only if the input meets the precondition (non-empty in this case):

```
prop_qsort_nn_min :: [Int] -> Property
prop_qsort_nn_min xs = not (null xs) ==>
    head (qsort xs) == minimum xs
```


```
> quickCheckN 1000 prop_qsort_nn_min
+++ OK, passed 1000 tests; 170 discarded.
```

- What does 170 discarded mean?
- Note: result type is now Property, not Bool. ==> is a QuickCheck combinator that allows us to write more complex property checks

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

16

16



- Another way to test is that our implementation is behaviorally equivalent to an existing one:

```
prop_qsort_sort :: [Int] -> Bool
prop_qsort_sort xs = qsort xs == List.sort xs
```

- Testing against implementation of sort in `List.sort`:

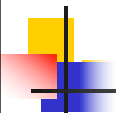
```
> quickCheckN 1000 prop_qsort_sort
```

- And here we go, a counterexample:

```
> qsort [0,0] -- [0,0] is a counterexample
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 17

17




- Looking back at `qsort`:

```
qsort :: forall a. Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort lhs ++ (x : qsort rhs)
  where lhs = [y | y <- xs, y < x]
        rhs = [z | z <- xs, z > x]
```

- The problem is duplicate elements; either `qsort` assumes a single occurrence of `x` as precondition, or it intentionally discards duplicates
- Is this a bug or a feature? Maybe the developer wanted `qsort` to leave only distinct elements. Let's assume this is the case and test

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 18

18



- Test qsort assuming it produces only distinct elements by design:

```

isDistinct :: Eq a => [a] -> Bool
isDistinct (x:ys) = ...
isDistinct [] = ...

prop_is_distinct :: [Int] -> Bool
prop_is_distinct = isDistinct . qsort

```

- Now write a conditional property testing qsort against List.sort

```

prop_qsort_distinct_sort :: [Int] -> Property
prop_qsort_distinct_sort xs = ...

```

```


> quickCheckN 1000 prop_qsort_distinct_sort

```

Program

19

19



- QuickCheck managed to generate enough tests in our case because the probability of a random list having no duplicates is relatively high
- But what if we have a “rare” condition?
- Suppose that we want to test insert of isort:

```


insert :: forall a. Ord a => a -> [a] -> [a]
insert x = aux where
  -- aux :: [a] -> [a]
  aux [] = [x]
  aux (y:ys) | x <= y = x : y : ys
              | otherwise = y : aux ys

```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

20

20



- Property to test whether `insert` leaves list sorted:

```
prop_insert_ordered' :: Int -> [Int] -> Bool
prop_insert_ordered' x xs = isOrdered (insert x xs)
```


- But it doesn't go well. What happens?

```
> quickCheck prop_insert_ordered'
*** Failed! Falsified (after 4 tests and 7 shrinks):
0 -- counterexample for x
[0,-1] -- counterexample value for xs
```

- Let's try to fix this

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 21

21



- Property to test whether `insert` leaves list sorted:

```
prop_insert_ordered :: Int -> [Int] -> Property
prop_insert_ordered x xs =
  isOrdered xs ==> isOrdered (insert x xs)
```


- Still doesn't go well. What happens?

```
> quickCheck prop_insert_ordered
*** Gave up! Passed only 80 tests; 1000 discarded tests.
```

- Probability of generating random ordered lists is too low

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 22

22




Outline

- Property-based testing with QuickCheck
 - QuickCheck properties
 - quickCheck function
 - Conditional testing
- **Generating data**
 - Generator combinators
 - Generator monad Gen a
 - Arbitrary type class
 - Shrinking

Programming in Haskell, A Milanova 23

23




Generator Type

- How does QuickCheck generate random data to test with?
- QuickCheck defines a type Gen a, a polymorphic type that stands for “generators of values of type a”. It has powerful mechanisms for creating random data and multiple ways of constructing generators
- Randomness is inherently impure, and therefore, it is encapsulated in the **Gen monad** structure. A bit more on the monad later
- If we want to write useful QuickCheck tests, we need to know how to configure generators. Similarly to the parsers, we use QuickCheck’s combinators to construct larger generators from smaller ones

Programming in Haskell, A Milanova 24

24



- We can define a function that generates small Ints using the Gen type and QuickCheck's function chooseInt:


```
genSmallInt :: Gen Int
genSmallInt = QC.chooseInt (1, 10)
```

- To see what the generator type yields, use QuickCheck's sample:

```
>:t sample
sample :: Show a => Gen a -> IO ()
>:t sample'
sample' :: Show a => Gen a -> IO [a]
> sample' genSmallInt
...
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 25

25



Generator Combinators

- choose takes an interval and produces a generator of elements in the interval. Type class System.Random.Random describes type that can be sampled:


```
choose :: System.Random.Random a => (a, a) -> Gen a
> sample' $ choose (0,3)
```

- elements takes an input list and returns a generator of elements drawn randomly from that list:

```
elements :: [a] => Gen a
> sample' $ elements [10,20,30]
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 26

26



- oneof randomly chooses between multiple generators:

```
oneof :: [Gen a] -> Gen a
```

```
> sample' $ oneof [choose (0,3), elements [10,20,30]]
[20,0,10,3,30,3,3,1,10,30,10]
```


- Related listOf produces a generator of random lists where elements are generated by the argument generator

```
listOf :: Gen a => Gen [a]
```

```
> sample' $ listOf (elements [1,2,3])
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 27

27



- frequency allows us to produce weighted combinations of multiple generators:

```
frequency :: [(Int,Gen a)] -> Gen a
```

```
> sample' $ frequency [ (1,choose (0,3))
                        , (5,elements [10,20,30]) ]
[30,10,20,10,30,10,2,20,10,20,10]
```

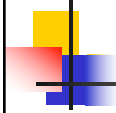
- and compare to

```
oneof :: [Gen a] -> Gen a
```

```
> sample' $ oneof [choose (0,3), elements [10,20,30]]
[20,0,10,3,30,3,3,1,10,30,10]
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 28

28



Generator Monad

- As we expected, Gen a is a Monad! We won't be defining return and bind the way we did for other recent monads (State and Parser). For practice, let's still write their signatures:

```
instance Monad Gen where
  -- return :: ...
  ...
  -- (>>=) :: ...
  ...
```

- You can look at the QuickCheck source code for the definitions
- So, what's the point of these functions?



- return takes a value and returns a generator of this value. E.g.,


```
genThree :: Gen Int
genThree = return 3

> sample' genThree
[3,3,3,3,3,3,3,3,3,3,3]
```

- And bind takes a generator and a function and applies the function to the value returned by the generator. E.g.,

```
genFive :: Gen Int
genFive = genThree >>= \x -> return (x + 2)

> sample' genFive
...
```



- A more interesting generator, generates 3 or 5:

```
genThreeOrFive :: Gen Int
genThreeOrFive = QC.choose (True,False) >>= (\x ->
    return (if x then 3 else 5))

> sample' genThreeOrFive
[5,3,5,3,5,3,5,5,5,5,3]
```


- Bind allows us to define larger generators from smaller ones:

```
genPair :: Gen a -> Gen b -> Gen (a,b)
genPair = ...

-- or just liftM2 (,)
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 31

31



Aside: liftM functions


- `liftM`, `liftM2`, and `liftM3` are defined in `Control.Monad` and are available to any instance of `Monad`, including `Gen`
- We saw how to define `liftM2` (essentially) in `genPair`
- In the case of the `Gen` monad, types specialize as follows:

```
liftM :: (a -> b) -> Gen a -> Gen b
liftM2 :: (a -> b -> c) -> Gen a -> Gen b -> Gen c
liftM3 :: (a -> b -> c -> d) ->
    Gen a -> Gen b -> Gen c -> Gen D
```

- We've seen `liftM` many times before. What is it?

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 32

32



Exercise

- As we are versed with the monad bind, define these functions:


```
genBool :: Gen Bool
genBool = ...
```

```
genTriple :: Gen a -> Gen b -> Gen c -> Gen (a,b,c)
genTriple = ...
```

```
-- generates Nothing or values from ga
genMaybe :: Gen a -> Gen (Maybe a)
genMaybe ga = QC.oneof [return Nothing, fmap Just ga]
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 33

33



Arbitrary Type Class

- To keep track of generators QuickCheck defines a type class containing types for which random values can be generated

```
class Arbitrary a where
  arbitrary :: Gen a
```

- Thus, we make a type an instance of Arbitrary by defining the arbitrary function. This tells QuickCheck how to generate random values for our type
- As expected, QuickCheck knows how to generate Ints, Floats, Booleans, etc. instances, and we can do these:

```
> sample' (arbitrary :: Gen Int)
...
> sample' (arbitrary :: Gen (Int,Float,Bool))
...
Programr ...
```

34

34



Generating Trees

- To define generation for a user-defined data types, e.g., `Tree a`, we make the data type an instance of the `Arbitrary` type class
- Tree data type with values in nodes:

```
data Tree a = Empty | Branch a (Tree a) (Tree a)
  deriving (Show, Foldable)
```

- First try. It doesn't work. Why goes wrong here?

```
genTree1 :: (Arbitrary a) => Gen (Tree a)
genTree1 = liftM3 Branch arbitrary genTree1 genTree1

> sample' (genTree1 :: Gen (Tree Int))
...
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

35

35



- Try again. This is better, but still undesirable. Try in `ghci`. What is happening?

```
genTree2 :: (Arbitrary a) => Gen (Tree a)
genTree2 = QC.oneof [ return Empty
  , liftM3 Branch arbitrary genTree2 genTree2 ]
```

- Try again. This one fixes the problem (though still need tweaks):

```
genTree3 :: (Arbitrary a) => Gen (Tree a)
genTree3 = QC.frequency
  [ (1, return Empty)
  , (2, liftM3 Branch arbitrary genTree3 genTree3) ]
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

36

36

- genTree2 generated too many Empty trees; distribution is undesirable. Since Tree is Foldable, we can use length to see distribution of sizes:

```
> map length <$> sample' (genTree2 :: Gen (Tree Int))
```

- genTree3 is too slow, it generated too many large trees
- QuickCheck's function sized is a higher-order function that takes a generator with a size parameter and progressively increases size
- sized is essential for generating trees (and possibly other types)

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

37

37

- Final version of "arbitrary" function is tricky. Try this out!

```
genTree :: forall a. (Arbitrary a) => Gen (Tree a)
genTree = QC.sized gen where
  gen :: (Arbitrary a) => Int -> Gen (Tree a)
  gen n = QC.frequency [ (1, return Empty)
                        , (n, liftM3 Branch arbitrary
                          (gen (n `div` 2))
                          (gen (n `div` 2))) ]
```


Notes:

- Need forall a. for Haskell to infer a type for gen
- We let QuickCheck determine frequency with sized and for each recursive call we decrease value with div
- sized runs gen function first with small n increasing n progressively

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

38

38



Shrinking


- One last thing before we can instantiate Arbitrary for Trees
- When properties fail QuickCheck provides a counter example, e.g.,

```
> quickCheck prop_qsort_sort --- qsort vs List.sort
*** Failed! Falsified (after 7 tests and 2 shrinks):
[-1,-1] --- the counter example
```

- It is important to produce as small a counter example as possible. It is a lot easier to debug!

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 39

39



- The following function has a glaring bug, can you spot it?

```
treeSum :: Tree Int -> Int
treeSum = aux where
  aux Empty = 0
  aux (Branch x l r) = if x == 0 then 0 else aux l + x + aux r
```


- Let's define a property and ask QuickCheck to find it too. What happens in example below?

```
prop_treeSum :: Tree Int -> Bool
prop_treeSum t = treeSum t == sum t --- Tree is Foldable!

> quickCheck prop_treeSum
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich) 40

40

- 
- It didn't work, but if we make Tree an Arbitrary instance it will. We need to tell QuickCheck how to generate random trees

```
instance (Arbitrary a) => Arbitrary (Tree a) where
  arbitrary = genTree
```


- What happens now when you run prop_treeSum?

```
prop_treeSum :: Tree Int -> Bool
prop_treeSum t = treeSum t == sum t --- Tree is Foldable!

> quickCheck prop_treeSum
```

- 
- A heuristic to shrink the tree

```
shrinkTree :: Arbitrary a => Tree a -> [Tree a]
shrinkTree = aux where
  aux Empty = [] -- empty trees cannot be shrunk
  aux (Branch x l r) = [l, r]
    -- left and right subtrees are smaller
  ++ map (\l' -> Branch x l' r) (shrinkTree l)
    -- shrink left subtree
  ++ map (\r' -> Branch x l r') (shrinkTree r)
    -- shrink right subtree
  ++ map (\x' -> Branch x' l r) (shrink x)
    -- shrink the value
```

- 
- Because shrinking is so important, QuickCheck includes a shrinking function as optional member of Arbitrary

```
instance (Arbitrary a) => Arbitrary (Tree a) where
  arbitrary :: ...
  arbitrary = genTree
  shrink :: ...
  shrink = shrinkTree
```

- Test what happens now when you run prop_treeSum!

```
prop_treeSum :: Tree Int -> Bool
prop_treeSum t = treeSum t == sum t --- Tree is Foldable!

> quickCheck prop_treeSum
```

Programming in Haskell, A Milanova (based on lecture by Stephanie Weirich)

43

43




Exercises

- Write generators and shrinking functions for Lists. QuickCheck does have library functions of these of course, but it is a good exercise to try. Then test a bunch of list functions
- Write generators and shrinking functions for trees with values at leaves (as opposed to our Tree a which had values at nodes)
- Write generators and shrinking functions for the AVL trees
- USE QuickCheck in projects, as much as you can!

Programming in Haskell, A Milanova

44

44



- Happy Thanksgiving!
- Week after Thanksgiving we'll cover monad transformers
- Quiz on QuickCheck and monad transformers

Programming in Haskell, A Milanova 45