CIS552: Advanced Programming

Handout 10

QuickCheck

Overview

QuickCheck is a lightweight tool for random testing of Haskell programs, developed by Koen Claessen and John Hughes.

- Based on specifications of desired properties, expressed as Haskell functions
- Properties are verified on randomly generated test data.
- The class system is used in clever ways to make everything look simple.

A Simple Property of Lists

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

A Simple Property of Lists

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

```
Prelude Main> Quickcheck.quickCheck prop_RevApp
OK, passed 100 tests.
```

A Simple Property of Lists

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

```
Prelude Main> Quickcheck.quickCheck prop_RevApp
OK, passed 100 tests.
```

N.b.: the type declaration on the property is required here, because we need to restrict its type to a particular instance — only monomorphic properties can be checked by QuickCheck.

A Bad Property

Suppose we mess up the specification:

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

A Bad Property

Suppose we mess up the specification:

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


```
Prelude Main> Quickcheck.quickCheck prop_BadRevApp
Falsifiable, after 4 tests:
[-3,-4,-4]
[-4,-1,1,1]
```

Conditional Properties

Many properties are not true universally (for all inputs of appropriate types), but only for inputs satisfying some conditions.

```
ins :: Ord a => a -> [a] -> [a]
ins a [] = [a]
ins a (a':as) = if a < a'
                then a:a':as
                else a':(ins a as)

ordered :: Ord a => [a] -> Bool
ordered (a:a':as) = (a<=a') && (ordered (a':as))
ordered _ = True

prop_BadIns :: Int -> [Int] -> Bool
prop_BadIns a as = ordered (ins a as)
```

Conditional Properties

```
Prelude Main> Quickcheck.quickCheck prop_BadIns
Falsifiable, after 9 tests:
4
[5,-3]
```

Conditional Properties

We can make a property conditional by writing it as `<condition> ==> <property>`:

```
prop_Ins :: Int -> [Int] -> Property
prop_Ins a as  =  (ordered as) ==> (ordered (ins a as))
```

```
Prelude Main> Quickcheck.quickCheck prop_Ins
OK, passed 100 tests.
```

Conditional Properties

We can make a property conditional by writing it as `<condition> ==> <property>`:

```
prop_Ins :: Int -> [Int] -> Property
prop_Ins a as  =  (ordered as) ==> (ordered (ins a as))
```

```
Prelude Main> Quickcheck.quickCheck prop_Ins
OK, passed 100 tests.
```


Note that the result type of `prop_Ins` has changed from `Bool` to `Property`. This is because the “testing semantics” of conditional properties is a little more tricky than for simple properties.

A Pitfall of Conditional Properties

```
insWrong :: Ord a => a -> [a] -> [a]

insWrong a [] = [a]
insWrong a as
  | (length as) == 6  =  as ++ [a]
  | otherwise         =  ins a as

prop_InsWrong :: Int -> [Int] -> Property
prop_InsWrong a as  =
  (ordered as) ==> (ordered (insWrong a as))
```

```
Prelude Main> Quickcheck.quickCheck prop_InsWrong
OK, passed 100 tests.
```

```
prop_InsWrong' :: Int -> [Int] -> Property
prop_InsWrong' a as  =
  (ordered as) ==>
    collect (length as) $
    classify (ordered (a:as)) "at-head" $
    classify (ordered (as++[a])) "at-tail" $
    (ordered (insWrong a as))
```

Fixing the distribution — First try

We can try to fix the distribution by adding another condition:

```
prop_InsWrong'' :: Int -> [Int] -> Property
prop_InsWrong'' a as  =
  (ordered as) && (length as >= 5) ==>
    (ordered (insWrong a as))
```

What Went Wrong?

QuickCheck provides combinators for investigating the distribution of test cases.

```
collect :: Show a => a -> Property -> Property
classify :: Bool -> String -> Property -> Property
trivial :: Bool -> Property -> Property
```

To see information about distribution, use `verboseCheck` instead of `quickCheck`.

```
Prelude Main> Quickcheck.verboseCheck prop_InsWrong'
...
OK, passed 100 tests.
42% 0, at-head, at-tail.
12% 1, at-tail.
11% 2, at-tail.
9% 2, at-head.
7% 2.
7% 1, at-head.
6% 1, at-head, at-tail.
2% 3, at-tail.
2% 3.
1% 4, at-head.
1% 3, at-head.
```

Fixing the distribution — First try

We can try to fix the distribution by adding another condition:

```
prop_InsWrong'' :: Int -> [Int] -> Property
prop_InsWrong'' a as  =
  (ordered as) && (length as >= 5) ==>
    (ordered (insWrong a as))
```

However:

```
Prelude Main> Quickcheck.quickCheck prop_InsWrong''
Arguments exhausted after 0 tests.
```

Generating Random Test Data

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

Generating Random Test Data

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

QuickCheck provides generators for most base types such as `Int`, `Char`, `Float`, and lists.

QuickCheck also provides combinators for building custom generators...

Generating Random Test Data

```
newtype Gen a = Gen (Rand -> a)
-- (roughly!)

choose :: (Int,Int) -> Gen Int

oneof :: [Gen a] -> Gen a
oneof [return Heads, return Tails]

frequency :: [(Int, Gen a)] -> Gen a
frequency [(1, return Heads), (2, return Tails)]

etc...
```

N.b.: The `return`s here are because `Gen` is a monad.

Generating Random Test Data

We can use these primitives to build generators for a variety of types. E.g. ...

```
instance Arbitrary Int where
  arbitrary = choose (-20,20)

instance (Arbitrary a, Arbitrary b)
           => Arbitrary (a,b) where
  arbitrary = liftM2 (,) arbitrary arbitrary
```

(Actually, both of these are predefined.)

A Custom Generator for Ordered Lists

```
orderedList :: Gen [Int]
orderedList =
  do a <- frequency
             [(1, return []),
              (7, liftM2 (:) arbitrary arbitrary)]
     return (sort a)
```

Using Custom Generators

The `forAll` combinator uses a specified custom generator instead of the default one.

```
prop_InsWrong''' :: Int -> Property
prop_InsWrong''' a =
  forAll orderedList $ \ as -> ordered (insWrong a as)
```

```
Prelude Main> Quickcheck.quickCheck prop_InsWrong'''
Falsifiable, after 19 tests:
0
[-5,0,3,5,7,8]
```

Whew.

Generators for Recursive Types

Here is a naive definition of arbitrary lists:

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary =
    oneof [return [],
           liftM2 (:) arbitrary arbitrary]
```

Why is this not what we want?

Generators for Recursive Types

Here is a naive definition of arbitrary lists:

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary =
    oneof [return [],
           liftM2 (:) arbitrary arbitrary]
```

Why is this not what we want?

Better:

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary =
    frequency [(1, return []),
               (7, liftM2 (:) arbitrary arbitrary)]
```

Generators for Trees

However, in some cases we need to be even more careful...

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving Show

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary =
    frequency
      [(1, liftM Leaf   arbitrary),
       (2, liftM2 Branch arbitrary arbitrary)]
```

What goes wrong?

Generators for Trees

However, in some cases we need to be even more careful...

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving Show

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary =
    frequency
      [(1, liftM Leaf   arbitrary),
       (2, liftM2 Branch arbitrary arbitrary)]
```

What goes wrong?

```
Prelude Main> Quickcheck.quickCheck prop_SomeTreeProperty
Stack space overflow: current size 1048576 bytes.
```

The Problem

Given our definition, an arbitrary tree has only a 50% chance of being finite!

Intuition: If the first few choices yield `Branch`es, then the only way for the tree to be finite is for many subtrees to choose (with 1/3 probability each time) to be leaves.

Sized Generators

We need to be able to control the size of the generated data.

This is accomplished by changing the definition of the `Gen` monad:

```
newtype Gen a = Gen (Int -> Rand -> a)
```

Sized Generators

The `sized` combinator allows the programmer to access the "current size bound."

```
sized :: (Int -> Gen a) -> Gen a

sized f = Gen (\n r -> m n r
                       where Gen m = f n)
```

The `Gen` Monad

```
instance Monad Gen where
  return a    = Gen (\n r -> a)
  Gen m >>= k =
    Gen (\n r0 -> let (r1,r2) = split r0
                      Gen m'  = k (m n r1)
                  in m' n r2)
```

Intuition

We want to build a function generator of type `Gen (a->b)`.

Unpacking the definition of `Gen`, we find that this is `Int->Rand->a->b`.

But this type is isomorphic to `a->Int->Rand->b`, which is the representation of `a -> Gen b`.

Using Sized Generators

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbTree


arbTree 0 = liftM Leaf arbitrary
arbTree n =
    frequency
      [(1, liftM Leaf    arbitrary),
       (4, liftM2 Branch (arbTree (n `div` 2))
                         (arbTree (n `div` 2)))]
```

Generating Random Functions

Since Haskell encourages higher-order programming, we may well want to use QuickCheck to test functions that take other functions as parameters.

To do so, we need to be able to generate random functions.

Surprisingly, this is possible.

Promote

I.e., we can define a function

```
promote :: (a -> Gen b) -> (Gen (a->b))

promote f = Gen (\n r ->
                   \a ->
                     m n r
                     where Gen m = f a)
```

We can now use `promote` to build a generator for a function type `a->b`, given a function that takes an `a` and uses it to construct a `b` generator that depends in some way on the `a` argument.

Where do such functions come from?

Arbitrary Class, Take 2

```
class Arbitrary a where
  arbitrary   :: Gen a
  coarbitrary :: a -> Gen b -> Gen b
```

I.e., `coarbitrary` takes a value of `a` and yields a generator transformer that takes a `b` generator and yields a new `b` generator whose behavior depends on the `a` argument.

Generating Random Functions

We can now use `arbitrary` and `coarbitrary`, together with `promote`, to generate random functions as needed:

```
instance (Arbitrary a, Arbitrary b) => Arbitrary (a -> b)
where
  arbitrary         = promote (`coarbitrary` arbitrary)
  coarbitrary f gen = ...later...
```

All we need to do now is to define appropriate `coarbitrary` functions for each instance of the `Arbitrary` class.

Defining `coarbitrary`

Recall that all our generators were ultimately based on the `choose` function (which generates uniformly distributed integers from a given range).

Similarly, the foundation of all our generator transformers is a function

```
variant :: Int -> Gen a -> Gen a
```

defined in such a way that

```
(variant i1 . variant i2 . ··· . variant im) g
                    ≠
(variant j1 . variant j2 . ··· . variant jn) g
```

(with high probability) whenever `[i1,...,im]` ≠ `[j1,...,jn]`.

`coarbitrary` for Booleans

```
instance Arbitrary Bool where
  arbitrary     = ...
  coarbitrary b = if b then variant 0 else variant 1
```

`coarbitrary` for Ints

```
instance Arbitrary Int where
  arbitrary     = ...
  coarbitrary n = variant (if n >= 0
                           then 2*n
                           else 2*(-n) + 1)
```

`coarbitrary` for pairs

```
instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b)
where
  arbitrary          = ...
  coarbitrary (a, b) = coarbitrary a . coarbitrary b
```

Note how function composition (`.`) is used to combine the generator transformers for types `a` and `b`.

`coarbitrary` for lists

The same idea can be applied to lists:

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary          = ...
  coarbitrary []     = variant 0
  coarbitrary (a:as) = coarbitrary a
                     . variant 1
                     . coarbitrary as
```

Defining `variant`

Here is the actual definition of the `variant` function:

```
variant :: Int -> Gen a -> Gen a

variant v (Gen m) =
   Gen (\n r -> m n (rands r !! (v+1)))
 where rands r0 = r1 : rands r2
                  where (r1, r2) = Random.split r0
```

`coarbitrary` for functions

```
instance (Arbitrary a, Arbitrary b) => Arbitrary (a -> b)
where
  arbitrary          = ...
  coarbitrary f gen = arbitrary >>=
                      (coarbitrary (f a) gen)
```

All we need to do now is to define appropriate `coarbitrary` functions for each instance of the `Arbitrary` class.

Closing Thoughts

- Thinking about properties (specifications) of functions is useful even when no errors are found by testing them.
- Indeed, many users report that, when errors are found by QuickCheck, they are just as often errors in the properties as in the code!
- The properties make excellent documentation, in part because they can be re-verified automatically as part of regression testing.

Acknowledgment

These slides are partly based on a nice presentation of QuickCheck by Jue Wang.