EECS 776
Functional Programming

Dr. Andy Gill

University of Kansas

November 20th, 2015

KU

QuickCheck

QuickCheck is a Haskell library for testing Haskell programs.

- It is a Domain Specific Language.
- Allows tests to have a declarative feel.
- The Haskell type system is used for finding good test cases.

KU

Properties of reverse

reverse 是检查一个list是否reverse

reverse :: [Int] -> [Int]

Good properties

```
reverse [x]              = [x]

reverse (xs ++ ys)       = reverse ys ++ reverse xs

reverse (reverse xs) = xs
```

A bad property

```
reverse (xs ++ ys)       = reverse xs ++ reverse ys
```

KU

```
reverse [x]             = [x]
```

```
prop_rev1 x = reverse [x] == [x]
  where
        types = (x :: Int)
```

```
> quickCheck prop_rev1
OK, passed 100 tests.
```

prop_sth :: type -> Bool

```
reverse (xs ++ ys)   = reverse ys ++ reverse xs
```

```
prop_rev2 xs ys
     = reverse (xs ++ ys) == reverse ys ++ reverse xs
  where
        types = (xs :: [Int],ys :: [Int])
```

```
> quickCheck prop_rev2
OK, passed 100 tests.
```

KU

```
reverse (reverse xs) = xs
```

```
prop_rev3 xs = reverse (reverse xs) == xs
  where
        types = (xs :: [Int])
```

```
> quickCheck prop_rev3
OK, passed 100 tests.
```

KU

```
reverse (xs ++ ys)   = reverse xs ++ reverse ys
```

```
prop_rev4 xs ys
        = reverse (xs ++ ys) == reverse xs ++ reverse ys
  where
        types = (xs :: [Int],ys :: [Int])
```

```
> quickCheck prop_rev4
Falsifiable, after 5 tests:
[0]
[2,-2]
```

KU

```
> :t quickCheck
quickCheck :: (Testable a) => a -> IO ()
> :t prop_rev1
prop_rev1 :: Int -> Bool
```

```
prop_insert x xs =
        ordered xs ==> ordered (insert x xs)
  where 在ordered xs的前提条件下， insert x到xs中也要为有序的
        types = (x :: Int, xs :: [Int])
```

ordered :: [Int] -> Bool

```
ordered xs = and $ zipWith (<=) xs (tail xs)
```

```
> quickCheck prop_insert
OK, passed 100 tests.
```

KU

```
> :t (==>)
(==>) :: (Testable a) => Bool -> a -> Property
```

```
prop_insert2 x xs =
        collect (length xs) $
          ordered xs ==> ordered (insert x xs)
  where
        types = (x :: Int, xs :: [Int])
```

```
> quickCheck prop_insert2
OK, passed 100 tests.
47% 0.
21% 1.
19% 2.
10% 3.
3% 4.
```

```
> :t collect
collect :: (Show a, Testable b) => a -> b -> Property
```

KU

```
prop_insert3 x xs =
        (length xs > 2) ==>
        ordered xs ==>
                ordered (insert x xs)
  where
  types = (x :: Int, xs :: [Int])
```

```
> quickCheck prop_insert3
Arguments exhausted after 24 tests.
```

KU

```
prop_insert4 x =
        forAll largerOrderedLists $ \ xs ->
        (length xs > 2) ==>
        ordered xs ==>
                ordered (insert x xs)
  where
        types = (x :: Int)

largerOrderedLists :: Gen [Int]
largerOrderedLists = do
  n <- choose (3,10)
  vs <- vector n
  return (sort vs)
```

```
> quickCheck prop_insert4
OK, passed 100 tests.
```

```
> :t forAll
forAll :: (Show a, Testable b) => Gen a -> (a -> b) -> Propert
> :t choose
choose :: (System.Random.Random a) => (a, a) -> Gen a
> :t vector
vector :: (Arbitrary a) => Int -> Gen [a]
```

```
class Arbitrary a where
  arbitrary :: Gen a
  --- also coarbitrary, ignore for now
```

KU

```
instance Arbitrary Bool where
  arbitrary = elements [True, False]
```

```
> :t elements
elements :: [a] -> Gen a
```

KU

- One DSL for testing properties.
  - This DSL uses the `Arbitrary` instance by default.
  - We can use other generators, via `forAll`.
  - We can examine the test cases being generated.
- One DSL for building data generators.
  - This DSL is monadic.
  - DSL primitive for generating random values in a ranges.
- QuickCheck is the combination of both DSLs.

KU

Testing

Four styles of functional testing, for testing f.

- $f\ x = y$, for fixed x and y.
  - Requires enumerating lots of examples.
- $g(f\ x) = \mathit{True}$, perhaps with precondition.
  - g is a property of f.
  - neatest use of QuickCheck.
- $f\ x = f'\ x$, perhaps with precondition.
  - Requires a f' to exist.
- $f^{-1}(f\ x) = x$, perhaps with precondition.
  - Requires a f^{-1} to exist.

KU

Test cases for reverse

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Test cases

```
test_rev1 = reverse [] == []
test_rev2 = reverse [1] == [1]
test_rev3 = reverse [1,2] == [2,1]
...
```

KU

Properties for reverse

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Properties

```
prop_rev1 x = reverse [x] == [x]
  where
        types = (x :: Int)
prop_rev2 xs ys
     = reverse (xs ++ ys) == reverse ys ++ reverse xs
  where
        types = (xs :: [Int],ys :: [Int])
```

KU

Using a trusted reverse

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

trusted_reverse :: [a] -> [a]
trusted_reverse xs = foldl (flip (:)) [] xs
```

Properties

```
prop_t_rev xs
     = reverse xs == trusted_reverse xs
  where
        types = (xs :: [Int])
```

KU

Using a inverse reverse

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

trusted_reverse :: [a] -> [a]
trusted_reverse xs = foldl (flip (:)) [] xs
```

Properties

```
prop_inv_rev_and_rev xs = reverse (reverse xs) == xs
  where
        types = (xs :: [Int])
prop_inv_rev_and_rev2 xs
     = trusted_reverse (reverse xs) == xs
  where
        types = (xs :: [Int])
```

```
> :t quickCheck
quickCheck :: (Testable a) => a -> IO ()

> :i Testable
class Testable a where
   property :: a -> Property
instance Testable Bool
instance Testable Property
instance (Arbitrary a, Show a, Testable b)
      => Testable (a -> b)
```

KU

```
prop_inv_rev_and_rev :: [Int] -> Bool
prop_inv_rev_and_rev xs = reverse (reverse xs) == xs
  where
        types = (xs :: [Int])


prop_example :: [Int] -> Property
prop_example xs =
   (length xs > 4) ==> reverse (reverse xs) == xs
  where
        types = (xs :: [Int])

-- (==>)   :: (Testable a) => Bool -> a -> Property
```

KU

```
(==)      :: (Eq a) => a -> a -> Bool
($)       :: (a -> b) -> a -> b

(==>)     :: (Testable a) => Bool -> a -> Property
label     :: (Testable a) => String -> a -> Property
classify  :: (Testable a) => Bool -> String -> a -> Property

forAll    :: (Show a, Testable b)
             => Gen a -> (a -> b) -> Property

collect   :: (Show a, Testable b) => a -> b -> Property
collect a b = label (show a) b
```

Nothing here about `Arbitrary`, this is all about `Bool` and `Property`.

Revisiting quickCheck.

```
> :t quickCheck
quickCheck :: (Testable a) => a -> IO ()

> :i Testable
class Testable a where
   property :: a -> Property
instance Testable Bool
instance Testable Property
instance (Arbitrary a, Show a, Testable b)
      => Testable (a -> b)
```

KU

This is the second (monadic) DSL, for generating arbitrary values.

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

choose    :: (Random a) => (a, a) -> Gen a
sized     :: (Int -> Gen a) -> Gen a
oneof     :: [Gen a] -> Gen a
frequency :: [(Int, Gen a)] -> Gen a
elements  :: [a] -> Gen a
vectorOf  :: Int -> Gen a -> Gen [a]
vector    :: Arbitrary a => Int -> Gen [a]
```

KU