QuickCheck: Beyond the Basics

Dave Laing

May 6, 2014

Basic QuickCheck

Some properties

```haskell
reverse_involutive :: [Int] -> Bool
reverse_involutive xs =
    xs == (reverse . reverse) xs

sort_idempotent :: [Int] -> Bool
sort_idempotent xs =
    sort xs == (sort . sort) xs

map_reverse :: Blind (Int -> String) -> [Int] -> Bool
map_reverse (Blind f) xs =
    (reverse . map f) xs == (map f . reverse) xs
```

Potential pitfall 1 - exhaustion

```
mod2_precondition :: [Int] -> Property
mod2_precondition xs =
  all ((== 0) . (`mod` 2)) xs ==>
    xs == map ((* 2) . (`div` 2)) xs

> mod2_precondition
*** Gave up! Passed only 62 tests.
```

Workaround

```
newtype EvenList a = EvenList { getList :: [a] }
                     deriving (Eq, Show)

instance (Arbitrary a, Integral a) =>
  Arbitrary (EvenList a) where
    arbitrary = EvenList <$> oneof [
                  return []
                , liftM2 (:)
                    (liftM (* 2) arbitrary)
                    (liftM getList arbitrary)
                ]
```

Workaround

```
mod2_precondition :: EvenList Int -> Bool
mod2_precondition (EvenList xs) =
    xs == map ((* 2) . (`div` 2)) xs

> quickCheck mod2_precondition
+++ OK, passed 100 tests.
```

Potential pitfall 2 - coverage

```
mod2_precondition :: EvenList Int -> Property
mod2_precondition (EvenList xs) =
    collect (length xs) $
    xs == map ((* 2) . (`div` 2)) xs

> quickCheck mod2_precondition
+++ OK, passed 100 tests.
41% 0
28% 1
12% 3
 9% 2
 4% 4
 2% 7
 2% 6
 2% 5
```

Workaround

```
instance (Arbitrary a, Integral a)
    => Arbitrary (EvenList a) where
      arbitrary = EvenList <$> frequency [
                    (1, return [])
                  , (4, liftM2 (:)
                      (liftM (* 2) arbitrary)
                      (liftM getList arbitrary))
                  ]
```

Workaround

```
> quickCheck mod2_precondition
+++ OK, passed 100 tests.
17% 1
16% 2
15% 0
12% 4
12% 3
 6% 5
 5% 6
 4% 7
 3% 9
 3% 12
 3% 11
 2% 8
 1% 20
 1% 14
```

Workaround

```
mod2_precondition :: EvenList Int -> Property
mod2_precondition (EvenList xs) =
    cover ((> 2) . length $ xs) 50 "non-trivial" $
      xs == map ((* 2) . (`div` 2)) xs

> quickCheck mod2_precondition
+++ OK, passed 100 tests (only 49% non-trivial; not 50%).

> quickCheck mod2_precondition
+++ OK, passed 100 tests.
```

Potential pitfall 3 - infinite structures

```haskell
data EvenTree a = Leaf
                | Node (EvenTree a) a (EvenTree a)
                deriving (Eq, Show)

instance (Arbitrary a, Integral a)
  => Arbitrary (EvenTree a) where
    arbitrary = frequency [
                (1, return Leaf)
              , (3, liftM3 Node
                       arbitrary
                       ((* 2) <$> arbitrary)
                       arbitrary)
              ]
```

Workaround

```
instance (Arbitrary a, Integral a) =>
  Arbitrary (EvenTree a) where
    arbitrary = sized arbTree

arbTree :: (Arbitrary a, Integral a)
        => Int -> Gen (EvenTree a)
arbTree 0 = return Leaf
arbTree n = frequency [
              (1, return Leaf)
            , (3, liftM3 Node
                    shrub
                    ((* 2) <$> arbitrary)
                    shrub)
            ]
  where
   shrub = arbTree (n `div` 2)
```

Testing from specifications

An abstract queue

```haskell
empty   :: Queue a
isEmpty :: Queue a -> Bool
peek    :: Queue a -> a
add     :: a -> Queue a -> Queue a
remove  :: Queue a -> Queue a
```

An algebraic specification

```
isEmpty empty                = True
isEmpty (add x xq)           = False

peek (add x empty)           = x
peek (add x (add y yq))      = peek (add y yq)

remove (add x empty)         = empty
remove (add x (add y yq)) = add x (remove (add y yq))
```

A list-based queue

```haskell
type Queue a = [a]

empty = []                  -- empty :: Queue a

isEmpty = null              -- isEmpty :: Queue a -> Bool

peek = head                 -- peek :: Queue a -> a

add x xs = xs ++ [x]        -- add :: a -> Queue a -> Queue a

remove = tail               -- remove :: Queue a -> Queue a
```

The algebraic specification in QuickCheck

```haskell
-- isEmpty empty              = True
-- isEmpty (add x xq)         = False

emptyAssert :: Assertion
emptyAssert =
    isEmpty empty @?= True

nonEmptyProp :: Int -> Queue Int -> Bool
nonEmptyProp x =
    not . isEmpty . add x
```

The algebraic specification in QuickCheck

```haskell
-- peek (add x empty)        = x
-- peek (add x (add y yq))   = peek (add y yq)

peekAddEmptyProp :: Int -> Bool
peekaddEmptyProp x =
    peek (add x empty) == x

peekAddNonEmptyProp :: Int -> Queue Int -> Property
peekAddNonEmptyProp x xs =
    (not . isEmpty) xs ==>
        peek (add x xs) == peek xs
```

The algebraic specification in QuickCheck

```
-- remove (add x empty)       = empty
-- remove (add x (add y yq)) = add x (remove (add y yq))

removeAddEmptyProp :: Int -> Bool
removeAddEmptyProp x =
    remove (add x empty) == empty

removeAddNonEmptyProp :: Int -> Queue Int -> Property
removeAddNonEmptyProp x xs =
    (not . isEmpty) xs ==>
        remove (add x xs) `equiv` add x (remove xs)
```

Model-based testing

A faster queue

```
data Queue a = Queue [a] [a]

empty = Queue [] []

isEmpty (Queue xs ys) = null xs

peek (Queue (x : _) _) = x

remove (Queue (_ : xs) ys) = mkValid xs ys

add x (Queue xs ys) = mkValid xs (x : ys)

mkValid [] ys = Queue (reverse ys) []
mkValid xs ys = Queue xs ys
```

The setup

```haskell
import qualified FastQueue as F
import qualified ListQueue as L

toBasic :: F.Queue a -> L.Queue a
valid :: F.Queue a -> Bool
```

Testing the model

```
emptyAssertion =
  toBasic F.empty @?= L.empty

isEmptyProperty q =
    F.isEmpty q == (L.isEmpty . toBasic) q
```

Testing the model

```
peekProperty q =
  (not . F.isEmpty) q ==>
    F.peek q == (L.peek . toBasic) q

addProperty x q =
    (toBasic . F.add x) q == (L.add x . toBasic) q

removeProperty q =
  (not . F.isEmpty) q ==>
    (toBasic . F.remove) q == (L.remove . toBasic) q
```

Testing monadic code

General points

- We want the ability to
  - generate an arbitrary list of actions that we can perform
  - carry out those actions in a context
- We can model actions using the free monad version of the monad under test
- We can use Arbitrary to generate a list of these actions
  - requires some thinking to make sure that we remain in a valid state

General points

```haskell
data QueueF a k = Empty k
                | IsEmpty (Bool -> k)
                | Peek (a -> k)
                | Remove k
                | Add a k
                deriving (Functor)

type Queue a r = Free (QueueF a) r
```

General points

```haskell
import qualified Queue.Free as F
type Action a = F.Queue a ()

actions :: Arbitrary a => Int -> Gen [Action a]
actions n = oneof $ [
            return []
          , liftM2 (:)
             (liftM F.add arbitrary)
             (actions (n + 1))
          ] ++ if n == 0 then [] else [
            liftM (F.remove :) (actions (n - 1))
          , liftM (void F.peek :) (actions n)
          ]
```

General points

```
type Queue s a = STRef s (V.Vector a)

perform :: Queue s a -> [Action a] -> ST s [a]
perform sq = execWriterT . mapM_ step
  where
    step (Free (F.Peek _)) = do
      x <- lift . peek $ sq
      tell [x]
    step (Free (F.Add x _)) =
      lift . add x $ sq
    step (Free (F.Remove _)) =
    ...
```

Monadic code and specifications

- We want to be able to model a sequence of actions and get a trace of some observations of the internal state
- If two sequences of actions gives the same trace of internal states in all contexts, they are observationally equivalent
- Context matters
  - both before and after the property of interest

Monadic code and specifications

```haskell
observe :: [Action a] -> ST s [a]
observe xs = empty >>= \q -> perform q xs

testEquiv xs ys = do
  forAll (actions 0) $ \prefix ->
  forAll (actions (delta (prefix ++ xs))) $ \suffix ->
  let
    o1 = runST $ observe (prefix ++ xs ++ suffix)
    o2 = runST $ observe (prefix ++ ys ++ suffix)
  in
    o1 == o2

addPeekProp m n =
  testEquiv [F.add m, F.add n, void F.peek]
            [F.add m, void F.peek, F.add n]
```

Monadic code and models

```haskell
addProperty :: Int -> Property
addProperty x =
  forAll (actions 0) $ \as -> runST $ do
  q <- empty
  void $ perform q as
  q' <- toList q
  let l' = L.add x q' -- l' = L.add x . toList
  add x q
  r' <- toList q      -- r' = toList . ST.add x
  return $ l' == r'
```

```haskell
addProperty :: Int -> Property
addProperty x = implements (L.add x) (ST.add x)
```

Monadic QuickCheck API

```
assert :: Monad m => Bool -> PropertyM m ()

pre :: Monad m => Bool -> PropertyM m ()

run :: Monad m => m a -> PropertyM m a

pick :: (Monad m, Show a) => Gen a -> PropertyM m a
```

Monadic QuickCheck API

Before

```
testEquiv xs ys = do
  forAll (actions 0) $ \prefix ->
  forAll (actions (delta (prefix ++ xs))) $ \suffix ->
  let
    o1 = runST $ observe (prefix ++ xs ++ suffix)
    o2 = runST $ observe (prefix ++ ys ++ suffix)
  in
    o1 == o2
```

Monadic QuickCheck API

After

```
testEquiv xs ys = monadicST $ do
  prefix <- pick $ actions 0
  suffix <- pick $ actions (delta (prefix ++ xs))

  equivalent <- run $ liftM2 (==)
    (observe (prefix ++ xs ++ suffix))
    (observe (prefix ++ ys ++ suffix))

  assert equivalent
```

Monadic code and pre-/post-conditions

```
addPeek y = monadicST $ do
  prefix <- pick (actions 0)

  q <- run empty
  run . void $ perform q prefix

  hasElems <- run . liftM not . isEmpty $ q
  pre hasElems

  equivalent <- run $ do
    x <- peek q
    add y q
    x' <- peek q
    return (x == x')
  assert equivalent
```

Conclusion

- Usually an interation of working on code, tests, and spec
- Lots of fun
- Really builds confidence in both code and understanding

Resources

- QuickCheck papers
  - QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs
  - Testing Monadic Code with QuickCheck
- Books
  - Introduction to FP using Haskell by Bird
  - The Fun of Programming by Gibbons and de Moor
- http://github.com/dalaing/ylj-quickcheck