Functional Programming

Systematic Testing with QuickCheck

Andreas Abel, Steffen Jost
LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians-Universität München

11 June 2012

1 Introduction to QuickCheck

QuickCheck is a Haskell library for automatic testing of logical program properties, orginally authored by Koen Claessen and John Hughes (Chalmers University of Technology, 2000).

import *Test.QuickCheck*

1.1 Properties

A logical property is an expression of type *Bool*, it either holds (*True*) or not (*False*).

For example: "appending the empty list changes nothing".

$$propAppendNil0\ xs = xs +\!\!+ [\,] \equiv xs$$
$$t0 = quickCheck\ propAppendNil0$$

Quickcheck answers:

```
+++ OK, passed 100 tests.
```

propAppendNil0 is not of type *Bool* but of type $a \to Bool$, thus it is a predicate over type a. Predicates p are tested by generating random values v of type a and then checking whether $p\ v \equiv True$.

Booleans and predicates are instances of *Testable* properties.

$$quickCheck :: Testable\ prop \Rightarrow prop \to IO\ ()$$

class *Testable prop* **where**
  $property :: prop \to Property$

instance *Testable Bool*

instance $(Arbitrary\ a, Testable\ prop) \Rightarrow Testable\ (a \to prop)$

Making a type an instance of *Arbitrary* amounts to writing a random value generator for this type. More on this below.

1.2 Polymorphic Properties

How come *QuickCheck* knows how to generate values of an unknown type a which occurs in the type of *propAppendNil0*?

$$propAppendNil0 :: [a] \to Bool$$

It turns out that unknown types a are replaced by the unit type.

```
verboseCheck propAppendNil0
Passed:
[]
Passed:
[()]
Passed:
[(),()]
Passed:
[]
...
```

The values of type () are not terribly interesting. *QuickCheck* defines wrapper types A, B, C around *Integer* which we can use instead.

import *Test.QuickCheck.Poly*

$$propAppendNil :: [A] \to Bool$$
$$propAppendNil\ xs = xs +\!\!+ [\,] \equiv xs$$
$$t1 = verboseCheck\ propAppendNil$$

Now we get:

```
Passed:
[]
Passed:
[3]
Passed:
[1,1,2]
...
```

Module *Poly* also defines types *OrdA*, *OrdB*, and *OrdC* which are wrappers of *Integer* implementing an *Ord* instance

1.3 Parametrizing QuickCheck

We can increase the number of test cases.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Args} = \mathit{Args}\ \{\\
&\quad \mathit{replay} \quad\ :: \mathit{Maybe}\ (\mathit{StdGen}, \mathit{Int})\\
&\quad \mathit{maxSuccess} :: \mathit{Int}\\
&\quad \mathit{maxDiscard} :: \mathit{Int}\\
&\quad \mathit{maxSize} \quad :: \mathit{Int}\\
&\}\\
&\mathit{stdArgs} :: \mathit{Args}\\
&\mathit{quickCheckWith} :: \mathit{Testable}\ \mathit{prop} \Rightarrow \mathit{Args} \to \mathit{prop} \to \mathit{IO}\ ()
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{qc1000} = \mathit{quickCheckWith}\ (\mathit{stdArgs}\ \{\mathit{maxSuccess} = 1000\})\\
&\mathit{t2} = \mathit{qc1000}\ \mathit{propAppendNil}
\end{aligned}
$$

Another Example: Testing the associativity of append.

$$
\begin{aligned}
&\mathit{propAppendAssoc} :: [A] \to [A] \to [A] \to \mathit{Bool}\\
&\mathit{propAppendAssoc}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs} = (\mathit{xs} +\!\!+ \mathit{ys}) +\!\!+ \mathit{zs} \equiv \mathit{xs} +\!\!+ (\mathit{ys} +\!\!+ \mathit{zs})\\
&\mathit{t3} = \mathit{quickCheck}\ \mathit{propAppendAssoc}
\end{aligned}
$$

2 Testing Tree Sort

Tree Sort sorts a list by building a search tree from the list and then flatten it out to a list again by infix traversal.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\\
&\quad \mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)
\end{aligned}
$$

```
    deriving (Show, Eq)
insert :: Ord a ⇒ a → Tree a → Tree a
insert a Leaf = Node Leaf a Leaf
insert a (Node l b r) = if a ⩽ b then Node (insert a l) b r
  else Node l b (insert a r)
toList :: Tree a → [a]
toList Leaf = []
toList (Node l a r) = toList l ++ a : toList r
sort :: Ord a ⇒ [a] → [a]
sort = toList ∘ foldl (flip insert) Leaf
```

We can test tree sort always returns a sorted list.

```
sorted :: Ord a ⇒ [a] → Bool
sorted []        = True
sorted [a]       = True
sorted (a : b : l) = a ⩽ b ∧ sorted (b : l)
propTreeSort :: [OrdA] → Bool
propTreeSort = sorted ∘ sort
t4 = quickCheck propTreeSort
```

2.1 Test Case Generators

We would like to test *insert* by itself. If we insert something into a search tree, the result is again a search tree. This is a conditional/implicational property, one with a precondition.

```
(==>) :: Testable prop ⇒ Bool → prop → Property
propSortedInsert :: OrdA → Tree OrdA → Property
propSortedInsert a t = sortedTree t ==>
  sortedTree (insert a t)
```

We need to write a test case generator for trees. That is an expression of type *Gen* (*Tree a*), where *Gen* is the *QuickCheck*'s generator monad.

A generator for a type *a* can be made available under the overloaded identifier *arbitrary*, if the type is made an instance of *Arbitrary*.

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

QuickCheck offers an extensive library of combinators for generators, such as *oneof*.

```
import Control.Monad
liftM  :: Monad m ⇒ (a → b) → (m a → m b)
liftM3 :: Monad m ⇒ (a → b → c → d) → (m a → m b → m c → m d)
oneof  :: [Gen a] → Gen a
```

2.2 Generating Unsorted Trees

Here is a first, naive generator for (unsorted) trees:

```
treeGen0 :: (Arbitrary a) ⇒ Gen (Tree a)
treeGen0 = oneof [return Leaf
                 , liftM3 Node treeGen0 arbitrary treeGen0]
intTreeGen0 :: Gen (Tree Int)
intTreeGen0 = treeGen0
t5 = sample intTreeGen0
```

Unfortunately, it is unusable. *oneof* chooses each alternative with equal probability, thus, one of each subtrees of *Node* can be expected to be non-empty. This leads to very large *Tree* instances.

We can control the size of sampled trees using the *sized* combinator..

```
sized :: (Int → Gen a) → Gen a
```

For size 0 we will now always generate the empty tree, and for size n we restrict potential subtrees to size $n - 1$.

```
treeGen1 :: (Arbitrary a) ⇒ Gen (Tree a)
treeGen1 = sized treeGen1'
treeGen1' :: (Arbitrary a) ⇒ Int → Gen (Tree a)
treeGen1' 0 = return Leaf
treeGen1' n = oneof [return Leaf
                    , liftM3 Node t arbitrary t]
  where t = treeGen1' (n `div` 2)
intTreeGen1 :: Gen (Tree Int)
intTreeGen1 = treeGen1
t6 = sample intTreeGen1
```

Since we generate arbitrary trees, we need to filter out the unsorted ones. To this end, we check the search tree invariant.

```
between :: Ord a ⇒ Maybe a → a → Maybe a → Bool
between Nothing a Nothing = True
between Nothing a (Just r) = a ⩽ r
between (Just l)  a Nothing = l ⩽ a
between (Just l)  a (Just r) = l ⩽ a ∧ a ⩽ r

betweenTree :: Ord a ⇒ Maybe a → Tree a → Maybe a → Bool
betweenTree ml Leaf           mr = True
betweenTree ml (Node l a r) mr = between ml a mr
                                 ∧ betweenTree ml l (Just a)
                                 ∧ betweenTree (Just a) r mr

sortedTree :: Ord a ⇒ Tree a → Bool
sortedTree t = betweenTree Nothing t Nothing
```

Now we can test that *insert* preserves the search tree invariant. Property combinator *forAll* let's us supply our custom test case generator.

```
forAll :: (Show a, Testable prop) ⇒ Gen a → (a → prop) → Property


propSortedInsert :: OrdA → Property
propSortedInsert a = forAll treeGen1 $ λt →
  sortedTree t ==>
    sortedTree (insert a t)
t7 = quickCheck propSortedInsert
```

2.3 Test Case Quality

Analyzing the quality of test cases.

```
classify      -- Conditionally labels test case.
:: Testable prop
⇒ Bool     -- True if the test case should be labelled.
→ String   -- Label.
→ prop
→ Property
```

How often did we test on the empty tree?

```
trivial :: Testable a ⇒ Bool → a → Property
trivial = (`classify` "trivial")

propSortedInsert1 :: OrdA → Property
propSortedInsert1 a = forAll treeGen1 $ λt →
```

```
  sortedTree t ==>
    (t ≡ Leaf) `trivial` sortedTree (insert a t)
t8 = quickCheck propSortedInsert1
```

Collecting statistics about the test data. Here: depth of tested tree.

```
depth :: Tree a → Int
depth Leaf = 0
depth (Node l a r) = 1 + max (depth l) (depth r)

propSortedInsert2 :: OrdA → Property
propSortedInsert2 a = forAll treeGen1 $ λt →
  sortedTree t ==>
    collect (depth t) $ sortedTree (insert a t)
t9 = quickCheck propSortedInsert2
```

Controlling the probability distribution of generated data.

```
frequency :: [(Int, Gen a)] → Gen a


treeGen2 :: (Arbitrary a) ⇒ Gen (Tree a)
treeGen2 = sized treeGen2'

treeGen2' :: (Arbitrary a) ⇒ Int → Gen (Tree a)
treeGen2' 0 = return Leaf
treeGen2' n = frequency [(10, return Leaf)
                        ,(90, liftM3 Node t arbitrary t)]
  where t = treeGen2' (n `div` 2)

intTreeGen2 :: Gen (Tree Int)
intTreeGen2 = treeGen2

t10 = sample intTreeGen2

propSortedInsert3 :: OrdA → Property
propSortedInsert3 a =
  forAll treeGen2 $ λt →
    sortedTree t ==>
      collect (depth t) $ sortedTree (insert a t)
t11 = quickCheck propSortedInsert3
```

2.4 Generating Sorted Trees

The probability that a random tree is a search tree is rather small. Thus, quickcheck needs to generate a huge number of trees to obtain one that is suitable for testing *insert*. This is rather wasteful.

Let us define a generator for sorted trees instead!

```
import System.Random


sortedTreeGen :: (Arbitrary a, Bounded a, Random a) ⇒ Gen (Tree a)
sortedTreeGen = sized $ λn → sortedTreeGen' n (minBound, maxBound)
sortedTreeGen' :: (Arbitrary a, Random a) ⇒ Int → (a, a) → Gen (Tree a)
sortedTreeGen' 0 _      = return Leaf
sortedTreeGen' n (l, r) =
  frequency [(10, return Leaf)
            ,(90, do a ← choose (l, r)
                     let n' = n `div` 2
                     tl ← sortedTreeGen' n' (l, a)
                     tr ← sortedTreeGen' n' (a, r)
                     return $ Node tl a tr)]
sortedIntTreeGen :: Gen (Tree Int)
sortedIntTreeGen = sortedTreeGen
t12 = sample sortedIntTreeGen
propSortedInsert4 :: Int → Property
propSortedInsert4 a =
  forAll sortedTreeGen $ λt →
    collect (depth t) $ sortedTree (insert a t)
t13 = quickCheck propSortedInsert4
```

2.5 Instantiating the *Arbitrary* Class

We make the generator for unordered *Tree*s available as instance of *Arbitrary*.

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a → [a]
```

Class *Arbitrary* suggest to also define a test case shrinker. If *quickCheck* finds a counterexample to a property, it will repeatedly *shrink* it as long as it stays a counterexample. Small counterexamples make bug finding a lot easier! *shrink t* should return all immediate proper subtrees of t.

```
instance (Arbitrary a) ⇒ Arbitrary (Tree a) where
  arbitrary = treeGen2
  shrink (Leaf)       = []
```

```
  shrink (Node l a r) = [l, r]
propSortedInsert5 :: OrdA → Tree OrdA → Property
propSortedInsert5 a t = sortedTree t ==>
  collect (depth t) $ sortedTree (insert a t)
t14 = quickCheck propSortedInsert5
```

We would also like to expose the generator for sorted trees. But there cannot be two instances of a type class for the same type! We help ourselves by a **newtype** wrapper.

```
newtype SortedTree a = SortedTree {tree :: (Tree a)} deriving (Show, Eq)
instance (Arbitrary a, Random a, Bounded a) ⇒ Arbitrary (SortedTree a) where
  arbitrary = liftM SortedTree sortedTreeGen
  shrink (SortedTree (Leaf))        = []
  shrink (SortedTree (Node l a r)) = [SortedTree l, SortedTree r]
propSortedInsert6 :: Int → SortedTree Int → Property
propSortedInsert6 a (SortedTree t) =
  collect (depth t) $ sortedTree (insert a t)
t15 = quickCheck propSortedInsert6
```

3 Generating Random Functions

Testing higher-order functions such as *map* requires the generation of random functions.

```
prop_map_comp :: Blind (A → B) → Blind (B → C) → [A] → Bool
prop_map_comp (Blind f) (Blind g) xs = map g (map f xs) ≡ map (g ∘ f) xs
t16 = quickCheck prop_map_comp
```

Blind means that there is no *Show* instance, i.e., we cannot see on which functions the test case fails.

How does *QuickCheck* generate random functions?

3.1 The *CoArbitrary* Class

A function of type $a \to b$ should not return the same b for all a. It should return different bs for different as, at least in some or most cases.

QuickCheck achieves this by changing the seed of the random generator by composing it with the function argument, before randomly generating a function

result. To this end, the function argument has to be casted into an *Integral* value that can be mixed into the random seed.

The job of perturbing the random seed is done for us by library function *variant*:

$$variant :: Integral\ n \Rightarrow n \rightarrow Gen\ a \rightarrow Gen\ a$$

If we want to generate functions over our own types a, we have to make them instances of *CoArbitrary*.

class *CoArbitrary a* **where**
  coarbitrary :: $a \rightarrow Gen\ c \rightarrow Gen\ c$

QuickCheck offers a standard implementation for "lazy people""

$$coarbitraryShow :: Show\ a \Rightarrow a \rightarrow Gen\ b \rightarrow Gen\ b$$

If we have a *Show* instance for our type T we get the *CoArbitrary* instance simply by.

instance *CoArbitrary T* **where**
  coarbitrary = *coarbitraryShow*

This will *show* values of type T, then cast them to an *Integral* which will then act on the random seed.

3.2 Printable Functions

QuickCheck also offers generation of functions that can be printed as argument-result-tables.

import *Test.QuickCheck.Function*

$$prop\_idempot :: Fun\ Integer\ Integer \rightarrow Integer \rightarrow Bool$$
$$prop\_idempot\ (Fun\ \_\ f)\ x = f\ (f\ x) \equiv f\ x$$
$$t20 = quickCheck\ prop\_idempot$$

Here we might get:

```
*** Failed! Falsifiable (after 4 tests and 7 shrinks):
{0->1, _->0}
0
```

Which means that the function that maps 0 to 1 and everything else to 0 is not idempotent.

The library offers us a type $Fun\ a\ b$ for functions from a to b that have a concrete representation $a :-> b$, a default value b, and the actual function $a \to b$.

$$\textbf{data}\ Fun\ a\ b = Fun\ (a :-> b, b)\ (a \to b)$$

The concrete representation could be just a input-output table. For details, see library documentation.

4 Testing Stateful Code

The paper

> Koen Classen and John Hughes
> *Testing Monadic Code with QuickCheck*
> Haskell Workshop 2002

demonstrates how to test stateful code.