

Model-Based Testing

(DIT848 / DAT260)

Spring 2014

Lecture 5

Property-Based Testing: QuickCheck

Gerardo Schneider

Department of Computer Science and Engineering
Chalmers | University of Gothenburg

Note on this lecture

- QuickCheck has been introduced by Koen Claessen as an interactive lecture
- These slides are **not** from Koen's lecture, but will serve for you to remember the essential of Property-Based Testing and QuickCheck
- See slides of "Lecture 6" on QuickCheck for additional notes on Koen's lecture

Outline

- Property-based testing
- QuickCheck
 - Haskell

Note: All the examples in this lecture has been taken from

- **Chapter 11: *Testing and quality assurance of Real World Haskell* by B. O'Sullivan, D. Stewart, and J. Goerzen**
(Available at <http://book.realworldhaskell.org/read/testing-and-quality-assurance.html>)

Property-Based Testing

- **Property-based testing** is a *kind of* MBT, where test cases are automatically generated from a **property**
- One of the difference with MBT in its classical definition is that test cases are extracted from a **property**, not a **model** of the system!
- Such properties are written in a formal language
 - First-order logic

Property-Based Testing

Automatic testing needs:

- The **system under test** (SUT - as usual)
- A **generator** to automatically generate inputs
- An **oracle** to automatically check that the outputs are correct
- The **oracle** should have the following properties
 1. Executable ("efficient enough")
 2. Simpler than the system
 3. Complete

QuickCheck in short

- **QuickCheck** is a random testing tool
 - Embedded domain-specific language for defining properties (Haskell)
 - Generates and executes random test cases
 - Evaluates outcome of test cases against properties
 - Shrinks counter examples
 - Originally for Haskell
- Commercial version
 - QuviQ (<http://www.quviq.com>)
 - Can test Erlang and C programs

A sorting algorithm: Quicksort

- **Quicksort** is a divide and conquer sorting algorithm
- It first divides a large list into two sub-lists: the low elements and the high elements
 - It then recursively sorts the sub-lists

Algorithm

1. Pick an element, called a **pivot**, from the list
2. Reorder the list so
 - All elements less than the pivot come before the pivot
 - All elements greater than the pivot come after it (equal values can go either way)
 - After the pivot is in its final position (*partition operation*)
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements

Base case: lists of size zero or one, which never need to be sorted

Group exercise

- Write a recursive version of the quicksort algorithm
- You can write it as a mathematical function, or in any functional programming language

Quicksort in Haskell

```
-- file: ch11/QC-basics.hs
import Test.QuickCheck
import Data.List

qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
               where lhs = filter (< x) xs
                     rhs = filter (>= x) xs
```

filter applies the predicate to the list and filters the list with those satisfying the predicate

Not an efficient implementation, but simple and elegant!

A simple property about qsort

```
-- file: ch11/QC-basics.hs
```

```
prop_idempotent xs = qsort (qsort xs) == qsort xs
```

Does this property hold?

```
ghci> prop_idempotent []  
True
```

```
ghci> prop_idempotent [1,1,1,1]  
True
```

```
ghci> prop_idempotent [1..100]  
True
```

```
ghci> prop_idempotent [1,5,2,1,2,0,9]  
True
```

Interesting but tedious: Better to automatically generate random data!

Generating test data with QuickCheck

```
ghci> generate 10 (System.Random.mkStdGen 2) arbitrary :: [Bool]  
[False,False,False,False,False,True]
```

Generates a random
list of boolean values

Shows the type of
QuickCheck

idempotent is polymorphic:
needs to be given a type to
generate data

arbitrary is a
function from the
Arbitrary type
class, to generate
data of each type
(Don't worry about it
for now...)

```
ghci> :type quickCheck  
quickCheck :: (Testable a) => a -> IO ()
```

```
ghci> quickCheck (prop_idempotent :: [Integer] -> Bool)  
00, passed 100 tests.
```

Using QuickCheck to test a property about qsort

```
-- file: ch11/QC-basics.hs  
prop_minimum xs = head (qsort xs) == minimum xs
```

Should the program pass the test? (Does the program satisfy the property?)

```
ghci> quickCheck (prop_minimum :: [Integer] -> Bool)  
O* * Exception: Prelude.head: empty list
```

It fails when sorting an empty list!

Using QuickCheck to test a property about qsort

```
-- file: ch11/minimum.hs
```

```
head :: [a] -> a
```

```
head (x:_) = x
```

```
head [] = error "Prelude.head: empty list"
```

```
minimum :: (Ord a) => [a] -> a
```

```
minimum [] = error "Prelude.minimum: empty list"
```

```
minimum xs = foldl1 min xs
```

head and minimum not defined for empty lists!

foldl1 takes the first 2 items of the list and applies the function to them, then feeds the function with this result and the 3rd argument and so on

```
-- file: ch11/QC-basics.hs
```

```
prop_minimum' xs =
```

```
    not (null xs) ==> head (qsort xs) == minimum xs
```

Property needs to be redefined, filtering invalid data

Property type, not Bool! (Filters non-empty lists before testing them)

```
ghci> quickCheck (prop_minimum' :: [Integer] -> Property)
00, passed 100 tests.
```

Group exercise

- Write 4 more properties about the sorting function
- You might think about “inherent” properties (i.e., what does it mean to be sorted), and/or additional properties (e.g., what happened when you operate on sorted lists)

Group exercise: Some properties

Prop 1: The list should be ordered 😊

```
prop_ordered xs = ordered (qsort xs)  
  where ordered [] = True  
        ordered [x] = True  
        ordered (x:y:xs) = x <= y && ordered (y:xs)
```

Prop 2: The ordered list is a permutation of the original list

```
prop_permutation xs = permutation xs (qsort xs)  
  where permutation xs ys = null (xs \\ ys) && null (ys \\ xs)
```

Group exercise: Some properties

Prop 3: The maximum of the sorted list is the last element

```
prop_maximum xs =  
  not (null xs) ==> last (qsort xs) == maximum xs
```

Prop 4: The minimum of two concatenated sorted lists is the minimum of the minimum of both lists

```
prop_append xs ys =  
  not (null xs) ==>  
  not (null ys) ==>  
  head (qsort (xs ++ ys)) == min (minimum xs) (minimum ys)
```

This is not exactly what is written in the informal spec. Why? Is it a good property anyway?

Testing against a model

- It is possible to compare an implementation with a **reference implementation** (prototype)

```
prop_sort_model xs = sort xs == qsort xs
```

The *reference*
implementation

The implementation
(SUT)

QuickCheck can do more...

- Testing against FSMs
- Testing concurrent systems
- Erlang, C programs (more recently for Java, Python, Scala, Ocaml,...)

Assignment 2

You will have to:

- Write properties and generators in QuickCheck to test Haskell programs

Further Reading

Read the following:

- Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*
 - **Chapter 11: *Testing and quality assurance***
 - Available online at <http://book.realworldhaskell.org/read/testing-and-quality-assurance.html>
- For **assignment 7** you should read the chapter above, in particular the section **"Testing case study: specifying a pretty printer"**
- Also, for the two remaining lectures on QuickCheck read the other listed papers at the course homepage