Papers we love - Utrecht

QuickCheck

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Requirements for a topic

- A paper that I love

Requirements for a topic

- A paper that I love
- A paper that is interesting to academics and developers

Requirements for a topic

- A paper that I love
- A paper that is interesting to academics and developers
- A paper that had significant impact on my own career

QuickCheck: a lightweight tool for random testing of Haskell programs

Full Text: Pdf Get this Article

Authors: Koen Claessen Chalmers University of Technology
John Hughes Chalmers University of Technology

2000 Article

Published in:

- Proceeding
  ICFP '00 Proceedings of the fifth ACM SIGPLAN international conference on Functional programming
  Pages 268 - 279
  ACM New York, NY, USA ©2000
  table of contents ISBN:1-58113-202-6 doi>10.1145/351240.351266

SIGPLAN

- Newsletter
  ACM SIGPLAN Notices Homepage
  Volume 35 Issue 9, Sept. 2000
  Pages 268 - 279
  ACM New York, NY, USA
  table of contents doi>10.1145/357766.351266

Bibliometrics

- Citation Count: 303
- Downloads (cumulative): 1,769
- Downloads (12 Months): 106
- Downloads (6 Weeks): 17

Haskell in 2000

- The *Functional Programming* course at the University of Utrecht was taught using WinHugs.

Haskell in 2000

- The *Functional Programming* course at the University of Utrecht was taught using WinHugs.
- It took a PhD student a week to install GHC.

Haskell in 2000

- The *Functional Programming* course at the University of Utrecht was taught using WinHugs.
- It took a PhD student a week to install GHC.
- There was no such thing as Cabal, Stack, Haddock, Hackage, etc. If you wanted something that wasn't in the base libraries, you needed to download it from the authors homepage.

Haskell in 2000

- The *Functional Programming* course at the University of Utrecht was taught using WinHugs.
- It took a PhD student a week to install GHC.
- There was no such thing as Cabal, Stack, Haddock, Hackage, etc. If you wanted something that wasn't in the base libraries, you needed to download it from the authors homepage.
- There were approximately zero industrial users.

Haskell

A Purely Functional Language

Haskell is a general purpose, purely functional programming language. Haskell compilers are freely available for almost any computer.

There will be another Haskell Workshop this summer in Montreal. Later, a special issue of the Journal of Functional Programming will be devoted to Haskell.

The Haskell Wiki has a new address: http://haskell.org/wiki/wiki.

Site contents:

- A Short Introduction to Haskell and its Advantages
- The Haskell Bookshelf: Books and Papers about Haskell and Functional Programming
- Definition of the Language and the Standard Libraries
- Haskell in Education
- Haskell Compilers and Interpreters
- Libraries and Tools for Haskell
- Haskell in Practice: Experiences and Applications
- The Future of Haskell
- The Haskell Mailing List
- Links to People and Pages Related to Haskell and Functional Programming
- Haskell Humor

This site is maintained by John Peterson and Olaf Chitil. Suggestions and comments welcome. This web site is a service to the Haskell community; new contributions are always welcome. If you wish to add your project, compiler, paper, class, or anything else to this site please contact the authors.

Main site at Yale, USA; Mirrors at Aachen, Germany, St. Andrews, Scotland, and the UK Mirror Service

---

Last update: March 17, 2000.

Wouter in 2000

I started by degree in Mathematics and Computer Science in 1999.

I'd just taken my first course on Functional Programming using Haskell.

Wouter in 2000

I started by degree in Mathematics and Computer Science in 1999.

I'd just taken my first course on Functional Programming using Haskell.

And I really loved it!

Wouter in 2000

I started by degree in Mathematics and Computer Science in 1999.

I'd just taken my first course on Functional Programming using Haskell.

And I really loved it!

Doaitse came back from ICFP '00 and handed me this paper…

The problem

How do we test software?

As a running example, let's assume we're developing a library for implementing queues:

```
enq :: Int -> Queue -> Queue
deq :: Queue -> Maybe Queue
front :: Queue -> Maybe Int
empty :: Queue
toList :: Queue -> List Int
fromList :: List Int -> Queue
```

Writing tests

Typically, we write unit tests by hand:

```
testFrontEnq :: Bool
testFrontEnq =
  front (enq 4 empty) == Just 4

testDeqEmpty :: Bool
testDeqEmpty =
  deq empty == Nothing

...
```

Test framework

And we may want to group these tests in a list and check they are all true:

```
runTests = and [testFrontEnq, testDeqEmpty, ...]
```

Writing tests

But writing these unit tests manually has drawbacks:

- to get good coverage, we need to write many tests;
- to get good coverage, we need to think quite hard about suitable input data.
- when a test fails, the test framework won't help us figure out why.

Queues

In our example, our tests weren't very good:

```
testFrontEnq :: Bool
testFrontEnq =
  front (enq 4 empty) == Just 4
```

This property also holds for a stack!

Types

In our strawman test framework, all of our tests were assertions of type `Bool`…

Oftentimes, there is nothing special about the constants we're using in our tests – we'd like to abstract over them:

```
testFrontEnq :: Queue -> Int -> Bool
testFrontEnq q x =
  front (enq i q) == Just i
```

What we'd really like to test is whether or not this **property** holds for our queue implementation.

QuickCheck

QuickCheck is a Haskell test framework that lets you test these properties:

```
> quickCheck testFrontEnq
Falsifiable, after 4 tests:
(Queue [1], 2)
```

It generates inputs for the `testFrontEnq` function and checks whether the property holds for the generated inputs.

Fix

We can, of course, fix our test to check that we have the desired FIFO behaviour:

```
testFifo :: Queue -> Int -> Bool
testFifo q x =
   last (toList (enq i q)) == Just i
```

Implementing QuickCheck

As a first approximation, it may help to think of the `quickCheck` function being implemented as follows:

```
quickCheck :: (a -> Bool) -> IO ()
quickCheck p = go ...
  where
  go :: [a] -> IO ()
  go []       = print "All tests succeed"
  go (x:xs)   =
    if p x then go xs
    else print ("Falsified " ++ show x)
```

Implementing QuickCheck

As a first approximation, it may help to think of the `quickCheck` function being implemented as follows:

```
quickCheck :: (a -> Bool) -> IO ()
quickCheck p = go ...
  where
  go :: [a] -> IO ()
  go []       = print "All tests succeed"
  go (x:xs)   =
    if p x then go xs
    else print ("Falsified " ++ show x)
```

The only question still open is: how do we generate inputs for our property?

Universiteit Utrecht
Faculty of Science
Information and Computing Sciences

Generating inputs

Basically, we use Haskell's classes (aka ad-hoc polymorphism, traits, protocols,interfaces) to define how to generate input for all the types of data that we wish to test.

- QuickCheck generates random input;
- Other libraries (SmallCheck) enumerate all inputs up to a given size;
- Other hybrid choices also exist.

The idea is the same: generate data for our properties to test if they hold.

Using the random inputs

Suppose we have a type class defined (pretty much) as follows:

```
class Arbitrary a where
  arbitrary :: RandomGenerator -> a
```

This captures the idea of being able to generate elements of type a randomly.

```
-- use arbitary and a rng to generate
--   a list of elements
elements :: Arbitrary a => IO [a]

quickCheck :: Arbitrary a => (a -> Bool) -> IO ()
quickCheck p = do elts <- elements;
                  go elts
```

Testing Queues

There are different ways to implement Queues in Haskell:

- the simple implementation uses lists, but may be inefficient;
- a clever implementation providing amortized O(1) access.

Can we use QuickCheck to show that they're equivalent?

Simple queues

```haskell
newtype Queue = Queue [Int]

enq :: Int -> Queue -> Queue
enq x (Queue xs) = Queue (xs ++ [x])

front :: Queue -> Maybe Int
front (Queue (x:xs)) = Just x
front (Queue []) = Nothing

empty :: Queue
empty = Queue []

...
```

Smarter queues

```haskell
newtype Queue = Queue ([Int],[Int])

enq :: Int -> Queue -> Queue
enq x (Queue (fs,bs)) = Queue (fs, x:bs)

front :: Queue -> Maybe Int
front (Queue (x:fs,bs)) = Just x
front (Queue ([],bs)) = Nothing

empty :: Queue
empty = Queue ([],[])

...
```

Smarter queues

The only interesting thing we need to do is ensure that dequeuing elements maintains the invariant that the first queue is only empty, when the entire queue is empty:

```
deq :: Queue -> Maybe Queue
deq (Queue (x:fs,bs)) = restore (fs,bs)
  where
  restore ([],bs) = (reverse bs, [])
  restore (fs,bs) = (fs,bs)
```

Testing our implementation

This gives us two ways to implement the same spec:

- the reference implementation using lists;
- the more efficient version using queues.

How do we relate the two?

Using QuickCheck

We can convert between our efficient implementation `I` and reference implementation `R` easily enough:

```
convert :: I.Queue -> R.Queue
convert (I.Queue (fs,bs)) =
  R.Queue (fs ++ reverse bs)

testEmpty = convert I.empty == R.empty
testEnq x q = convert (I.enq x q) == R.enq x q
...
```

Using QuickCheck

We can convert between our efficient implementation `I` and reference implementation `R` easily enough:

```
convert :: I.Queue -> R.Queue
convert (I.Queue (fs,bs)) =
  R.Queue (fs ++ reverse bs)

testEmpty = convert I.empty == R.empty
testEnq x q = convert (I.enq x q) == R.enq x q
...
```

Yet these tests fail! We should only consider queues satisfying our invariant.

Revising our tests

```
invariant (Queue (fs,bs)) = not (null fs) || null bs

testEmpty = convert I.empty == R.empty
testEnq x q =
  invariant q ==> convert (I.enq x q) == R.enq x q
```

We add the precondition `invariant q` to our tests.

Any test data that does not satisfy the invariant is discarded.

Nested calls?

This tests our two implementations line up after *one call*.

But what if we want to test that they line up after many calls?

```
testEnqEnq x y q = invariant q ==>
  convert (I.enq x (I.enq y q))
  == R.enq x (R.enq y q)
```

Aren't we trying to automate our tests?

Code is data

We can define an explicit data type capturing the API of our Queue libraries:

```
data QAPI = Enq Int QAPI
  | Front (Maybe Int -> QAPI)
  | Deq (Maybe Queue -> QAPI)
  | ...
```


Universiteit Utrecht
Faculty of Science
Information and Computing Sciences

Code is data

We can define an explicit data type capturing the API of our Queue libraries:

```
data QAPI = Enq Int QAPI
  | Front (Maybe Int -> QAPI)
  | Deq (Maybe Queue -> QAPI)
  | ...
```

And then interpret this data type as a sequence of commands on queues:

```
evaluate :: QAPI -> Queue -> Maybe Queue
evaluate (Enq x c) q = evaluate c (enq x q)
evaluate (Front c) q = evaluate (c (front q)) q
evaluate (Deq c) q   = evaluate (c (deq q)) (deq q)
...
```

Testing reference and implementation

But if we have a data type representing commands on queues:

```
data QAPI = Enq Int QAPI
  | Front (Maybe Int -> QAPI)
  | Deq (Maybe Queue -> QAPI)
  | ...
```

Why not generate random series of commands?

```
instance Arbitary QAPI where
  arbitrary = ...
```

Testing reference and implementation

And test that these two APIs produce the same results

```
referenceTest : QAPI -> I.Queue -> Bool
referenceTest cmds q =
  I.evaluate cmds q ==
    R.evaluate cmds (convert q)
```

What are our tests?

- *Unit tests* test that a property holds for certain values;
- Using QuickCheck, we can test that a property holds for *many* values – QuickCheck is generating unit tests.
- We can even use QuickCheck to generate a series of API calls – QuickCheck is generating completely new tests!

Computing specs

Given an API, what are the properties that hold of its functions?

That seems like an impossible problem to solve automatically…

Yet using QuickCheck you can get pretty close.

QuickSpec

Suppose we have the API for a handful of list functions:

```
(++) :: [a] -> [a] -> [a]
(:) :: a -> [a] -> [a]
[] :: [a]
reverse :: [a] -> [a]
```

What terms can we build using these functions?

Enumerating terms

Given some values x and y, we can enumerate all possible terms built using these functions:

```
[]
[] ++ []
(x : [])
(x : y : [])
reverse (x : [])
...
```

Enumerating terms

Given some values x and y, we can enumerate all possible terms built using these functions:

```
[]
[] ++ []
(x : [])
(x : y : [])
reverse (x : [])
...
```

As a first approximation, assume that all these terms are equal.

Universiteit Utrecht
Faculty of Science
Information and Computing Sciences

Start testing!

Now choose random values for `x` and `y` – which terms can you distinguish?

For example, `reverse (x : [])` and `x:y:[]` are different for any choice of `x` and `y`.

Start testing!

Now choose random values for `x` and `y` – which terms can you distinguish?

For example, `reverse (x : [])` and `x:y:[]` are different for any choice of `x` and `y`.

But no choice of `x`, can distinguish `reverse (x : [])` and `x : []` – hence these must be equal!

QuickSpec

To work well, QuickSpec does quite some work to remove duplicate equations and present a 'minimal' set of equations.

But for many APIs it manages to compute sensible properties from scratch.

QuickCheck in practice

- Functions to classify the data that is generated;
- Functions to define random data generators;
- Controlling size of data generated and number of tests;
- ‘Shrinking’ of counterexamples to facilitate diagnosis.

What makes QuickCheck work?

If you take a step back, what is it that makes QuickCheck work so well?

- Purity – all functions are known to be free of side-effects;
- Types – the types of our properties drive the generation of random inputs.

What makes QuickCheck work?

If you take a step back, what is it that makes QuickCheck work so well?

- Purity – all functions are known to be free of side-effects;
- Types – the types of our properties drive the generation of random inputs.

This is playing exactly to the strengths of Haskell!

Impact

QuickCheck has been ported to 35+ different programming languages.

It forces you to think of *specifications* rather than unit tests.

Bugs may still show up in the specification, the random data generator, or code under test – each requires separate diagnosis.

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Advantages

- The underlying ideas of QuickCheck are very widely applicable to different languages and systems.
- It is particularly good at spotting interactions that conventional test cases miss.
- QuickCheck makes diagnosis simple by shrinking inputs.
- QuickCheck makes it easier to achieve much better test coverage.

It's been used with great success in many different projects.

Why the love?

A simple idea, implemented in about 200 loc. (The original code was included in an appendix of the paper.)

It plays to Haskell's strengths: types and purity.

Changes the way we think about testing.

Useful in practice.

Another useful piece of kit in the formal reasoning toolkit.

Questions?