

QuickCheck from Everywhere

Daisuke IKEGAMI <ikegami@madscientist.jp>

2008-06-27

**Tokyo Society for the Application of Currying
(TSAC) meeting**

25 slides + 2 appendixes + acknowledgment

Note: Underlined sentence is a comment, which is pointed out by a participant at the meeting

Today's Japanese tip

Kata
型



Kata 型

Kata has several meanings in Japanese

- type (in functional programming language)
- style (in Japanese special tea ceremony)
- form (in Japanese flower arrangement)
- fighting pose (in Aikido, Karate, Kendo and Judo)
- trick (in Ninjutsu which Ninja use)

Traditional Japanese is *typed*

Today's topic

QuickCheck

Not only a library in Haskell,
but also a *formal* testing method

“Kata” for testing

What is a test for software?

The final phase of development software, which our client requires and let she be satisfied, isn't it?

No

Test is a
heuristic way
to find a bug

Not to guarantee the quality of software

**Program
testing** can be
used to show
the presence of
bugs, but **never**
**to show their
absence!**



Edsger Wybe Dijkstra

*“Notes on Structured Programming”, 1972, at the end
of section 3 (p.7) On The Reliability of Mechanisms*

How to test a
software, **effectively**?

Interested question:

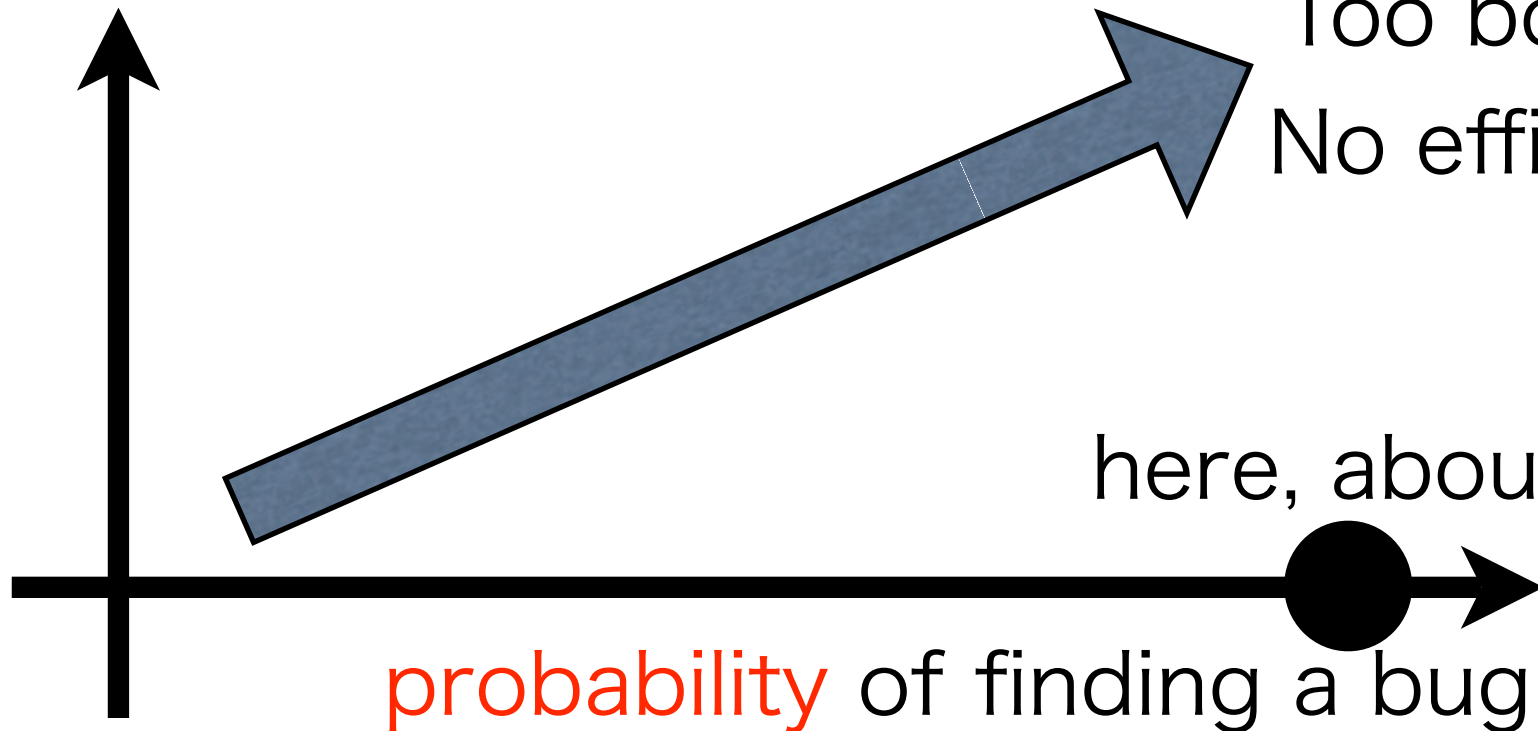
If there is an **effective** testing way,
then MS-Windows may not freeze

*Microsoft Research is one of great laboratories to
study how to development softwares*

Then, how to test a software **efficiently**?

Testing has a trade-off :

number of tests



Why testing is so boring?

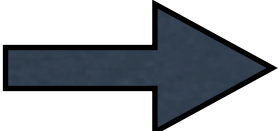
```
test :: TargetProgram → Property → Bool
test p q = case (while running p) of
  (satisfies q) → True
  otherwise → False
```

The reason why testing is boring, because there are too many instances for checking the property!

Key idea of the random testing

**Generate test instances by the
property **randomly** using a
random testing framework, until
you are satisfied (for a week?)**

Hint: We have `rand()` and the type (or, the class)

“Integer”  [0, 10, -24, 3, 6, 101, -98, ...]

QuickCheck

developed by Koen Claessen and John Hughes
in Haskell since 2000

`quickCheck :: Testable a => a → IO ()`

- Random generator can be tuned by user
- Random instances are generated by *type*
- Property is written in the first order logic, without the existential quantifier

Learning QuickCheck is not difficult, and it is a **efficient** testing framework

QuickCheck, more

Two examples using QuickCheck in Haskell

- testing properties of a data
 - a list
 - what are properties of a list?
- testing properties of a function
 - the fibonacci function
 - what are properties of the fibonacci?

Testing a *list* using QuickCheck

1. for all $x :: \text{Int}$, $\text{head } [x] == x$

How about remove type annotation, such as $(\lambda x \rightarrow \dots)$? Haskell has polymorphism.

```
% ghci -package QuickCheck-1.1.0.0
```

```
Prelude> :m + Debug.QuickCheck
```

```
Prelude Debug.QuickCheck> let prop =
```

```
\(x::Int) -> head [x] == x
```

```
prop :: Int -> Bool
```

```
Prelude Debug.QuickCheck> quickCheck
```

```
prop
```

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

```
it :: IO ()
```

Testing a list using QuickCheck, continued

2. for all $xs :: [Int]$ and $x :: Int$, $tail (x :: xs) == \text{tail } xs$ whenever xs is a finite list

```
% ghci -package QuickCheck-1.1.0.0
```

```
Prelude> :m + Debug.QuickCheck
```

```
Prelude Debug.QuickCheck> let prop2 = \  
(xs :: [Int]) -> (x :: Int) -> tail (x:xs)  
== tail xs
```

```
prop2 :: [Int] -> Int -> Bool
```

```
Prelude Debug.QuickCheck> quickCheck prop2
```

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

```
it :: IO ()
```

Error in this slide at the last *tail*

Testing the fibonacci using QuickCheck

For all $n :: \text{Int}$, $\text{fib } (n + 2) - \text{fib } (n + 1) == \text{fib } n$
whenever $n > 1$

At first, we have to define the fibonacci function
in Haskell!

```
% ghci -package QuickCheck-1.1.0.0
Prelude> :m + Debug.QuickCheck
Prelude Debug.QuickCheck> let {fib :: Int
-> Int; fib 0 = 1; fib 1 = 1; fib (n+2) =
fib (n + 1) + fib n}
fib :: Int -> Int
```

Testing the fibonacci using QuickCheck

For all $n :: \text{Int}$, $\text{fib } (n + 2) - \text{fib } (n + 1) == \text{fib } n$
whenever $n > 1$

```
Prelude Debug.QuickCheck> let prop3 = \  
(n::Int) -> (n > 1 ==> fib (n+2)-fib(n+1)  
== fib n)  
prop3 :: Int -> Property  
Prelude Debug.QuickCheck> quickCheck prop3  
(testing for a while...)  
OK, passed 100 tests.  
it :: IO ()
```

How about a counter example? Can we find it ?

Let's see a sequence of fibonacci numbers:

..., 3, 5, 8, 13, 21, ...

You maybe misunderstood that

for all $n :: \text{Int}$, $\text{fib } n < \text{fib } (n + 1)$

```
Prelude Debug.QuickCheck> let prop4 = \  
(n::Int) -> fib n < fib (n + 1)  
prop4 :: Int -> Property  
Prelude Debug.QuickCheck> quickCheck prop4  
Falsifiable, after 0 tests:  
0  
it :: ()
```

Tiny cheat sheet for QuickCheck

- “For all an integer x ...”

- $\lambda(x::Integer) \rightarrow \dots$

How about remove type annotation, such as $\lambda x \rightarrow \dots$? The latest GHC has “Data” type for generics

- “Whenever ***boolean exp*** is **true**, ...”

- $(boolean\ exp) \implies \dots$

- Tuning random generators!!! (no cheat)
- Show statistics of results, etc.

If you have an interest to QuickCheck more, please refer the manual (and *papers*, ugh)

Recent trend

mix some lovely features of functional programming into popular programming language, which you may like

- Ruby 1.9 = Ruby 1.8 + FP
- Scala = Java + FP
- F# = not C# but OCaml + no more FP
- etc... *Thanks to point out about F#*

even the concept of functional programming is not easy to understand

Powered by Ph.D

Summary until here

QuickCheck is a Haskell library, which is a random testing framework

A functional programming technique

- To add a new feature in FP is valued at Ph.D
- To study is not easy
- To use is not difficult
- To enjoy programming with **it** is *really* fun



RushCheck

developed by Ikegami in Ruby since 2006

Key idea of writing QuickCheck in Ruby

- writing ruby program like Haskell
 - use Class instead of *type* in Haskell
 - use Proc for *lazy evaluation*
 - use Exception for failure of a test
 - write state monads for random generators (with a magic)
 - doesn't use any non-functional way in the library

RIP: RushCheck is *not alive* (works only on Ruby 1.6)
because the semantics of *Proc* are changed between 1.6 and 1.8
there are **426 Proc** in the library, oops.

A proposal : Replace Proc into lambda and try {} or do ... end

QuickCheck from Everywhere

What do we need features in the programming language to implement QuickCheck?

- Type or Class for generating random instance
- Closure (or anonymous function) for state monads and lazy evaluation
- Domain Specific Language (DSL) for writing properties

Implement QuickCheck for **C** is challenging!

Though C is poor than Haskell or Ruby, I have an idea (using C++ and GDB!) to implement QuickCheck for C

Summary

- Efficient testing is needed to find a bug
- Random testing is an efficient way
- QuickCheck is based on the random testing
- QuickCheck maybe used from everywhere, we really need it!

References

Look up “QuickCheck” at
<http://en.wikipedia.org/>

Next generations

- QuickCheck version 2
 - <http://darcs.haskell.org/QuickCheck/>
- SparseCheck in Haskell
 - <http://www-users.cs.york.ac.uk/~mfn/sparsecheck/>
- Exhaustive testing instead of random testing
 - not in scope of this talk
 - SmallCheck in Haskell
 - Lazy SmallCheck in Haskell

Appendix A.

QuickCheck version 2.

developed by Bjorn Bringert, who is one of PhD students in Chalmers university: note that John and Koen are professors in Chalmers

- not for academic but for practice
- `quickCheck' :: Testable a => a -> IO Bool`
- many useful features (but yet experimentally)

It is a chance to be restored to life of RushCheck when QuickCheck ver.2 becomes stable.

Appendix B.

SparseCheck

QuickCheck has a dilemma that to let a test be efficient, then the property of the test becomes hard to write; boring! A participant points out this problem.

The SparseCheck library in Haskell assumes that a property that has a *sparse* input domain

SparseCheck uses an idea of logic programming for random generator to overcome the dilemma

logic (or rule) is easy to write

property in functional way may need many guards

Acknowledgments

- Thanks to the organizers of TSAC for giving a chance to talk
- Sorry for taking over 90 minutes
 - but all participants were kind and gave many useful comments and discussion
- There are some mistakes or errors in this slides, but I left them and add discussions as *underlined* comments; for telling you that TSAC meeting is not formal, but warming community, thanks!