A Short Introduction to QuickCheck

Lars-Åke Fredlund
Facultad de Informática, Universidad Politécnica de Madrid

What is QuickCheck

- A tool for *random* testing of programs (and systems) developed by Koen Claessen and John Hughes at Chalmers University, Gothenburg
- First QuickCheck was written in Haskell, but now available for many languages: Java, Perl, Python, Scala…
- Today we will focus on **QuviQ QuickCheck**, a commercial QuickCheck tool written in Erlang
- Why QuviQ QuickCheck? We are collaborating with the QuviQ developers in the EU projects ProTest (finished) and Prowess (ongoing)

What is QuickCheck: basic ideas

- Express *test properties* on a high abstraction level (using a restricted variant of first-order logic)
- Generate test data randomly: from one *test property* many concrete *test cases* can be generated
- When a counterexample (bug) is found, QuickCheck tries to generate a simpler and thus more easily debuggable counterexample, in a constructive manner (*shrinking*)

Example

We want to test the library `sets` that implement sets for Erlang.

API:

```
new() -> set()
from_list(List) -> Set
to_list(Set) -> List

size(Set) -> integer()

is_element(Element,Set) -> boolean()
add_element(Element,Set1) -> Set2
del_element(Element,Set1) -> Set2

union(Set1,Set2) -> Set3
intersection(Set1,Set2) -> Set3

...
```

What to test?

What to test?

We begin with the property

$$\text{for all } X : set, Y : set \;:\; X \cup Y = Y \cup X$$

(commutativity for union)

“Traditional” testing

- We pick a number “representative” sets, and check that the `sets:union` function returns the same result:

```
sets:union(sets:from_list([]),sets:from_list([1]))
  ==
sets:union(sets:from_list([1]),sets:from_list([])),

...

sets:union(sets:from_list([2]),sets:from_list([1]))
  ==
sets:union(sets:from_list([1]),sets:from_list([2])).
```

- We could of course improve on this by using a testing framework for Erlang such as e.g. EUnit

Expressing commutativity in QuickCheck

The abstract property:

$$\text{for all } X : set, Y : set \ : \ X \cup Y = Y \cup X$$

Expressing commutativity in QuickCheck

The abstract property:

$$\text{for all } X : set, Y : set \;:\; X \cup Y = Y \cup X$$

- In QuickCheck:

```
?FORALL(X,set(),
        ?FORALL(Y,set(),
                sets:union(X,Y) == sets:union(Y,X)))
```

- What is `set()`? A ***generator*** for sets.

Generators

- A *generator* for some “type” can be used repeatedly to generate *elements* of that type, **according to some probability distribution**.
- Example: the built-in generator `int()` can generate Erlang integers randomly.

Generators

- A *generator* for some “type” can be used repeatedly to generate *elements* of that type, **according to some probability distribution**.
- Example: the built-in generator `int()` can generate Erlang integers randomly.
- A generator can also “shrink” a generated value to a “simpler” value, in order to try to find a “simpler” failing test case

Generators

- A *generator* for some "type" can be used repeatedly to generate *elements* of that type, **according to some probability distribution**.
- Example: the built-in generator `int()` can generate Erlang integers randomly.
- A generator can also "shrink" a generated value to a "simpler" value, in order to try to find a "simpler" failing test case
- Examples of shrinking:
  - What is simpler than 5?

Generators

- A *generator* for some "type" can be used repeatedly to generate *elements* of that type, **according to some probability distribution**.
- Example: the built-in generator `int()` can generate Erlang integers randomly.
- A generator can also "shrink" a generated value to a "simpler" value, in order to try to find a "simpler" failing test case
- Examples of shrinking:
    - What is simpler than 5? 0

Generators

- A *generator* for some "type" can be used repeatedly to generate *elements* of that type, **according to some probability distribution**.
- Example: the built-in generator `int()` can generate Erlang integers randomly.
- A generator can also "shrink" a generated value to a "simpler" value, in order to try to find a "simpler" failing test case
- Examples of shrinking:
  - What is simpler than 5? 0
  - What is simpler than [5,2]?

Generators

- A *generator* for some “type” can be used repeatedly to generate *elements* of that type, **according to some probability distribution**.
- Example: the built-in generator `int()` can generate Erlang integers randomly.
- A generator can also “shrink” a generated value to a “simpler” value, in order to try to find a “simpler” failing test case
- Examples of shrinking:
  - What is simpler than 5? 0
  - What is simpler than [5,2]? [5], [2], [],

Generators

- A *generator* for some "type" can be used repeatedly to generate *elements* of that type, **according to some probability distribution**.
- Example: the built-in generator `int()` can generate Erlang integers randomly.
- A generator can also "shrink" a generated value to a "simpler" value, in order to try to find a "simpler" failing test case
- Examples of shrinking:
  - What is simpler than 5? 0
  - What is simpler than [5,2]? [5], [2], [], [0,0], [0]

Standard QuickCheck Generators: simple ones

`int()`	integers
`nat()`	natural numbers
`bool()`	`true` or `false`
`choose(M,N)`	a number in the range M..N
...	

Note that `int()` does not return an integer, but a generator for integers:

```
> erl
Erlang R15B02 ...

Eshell V5.9.2  (abort with ^G)
1> eqc_gen:int().
{eqc_gen,#Fun<eqc_gen.27.118839684>}
```

Standard QuickCheck Generators: combinators

`list(G)`	a list of elements constructed from the generator `G`
`oneof([G1,...,GN])`	a value constructed from a randomly selected generator `Gi`
`?LET(Pat,G1,G2)`	Generates a value from `G1`, binds `Pat` to it, and possibly uses `Pat` in `G2`

- For controlling probability distributions:
  `frequency([{Weight1,G1},...,{WeightN,Gn}])`
  A value constructed from a generator Gi chosen according to the probability distribution defined by `Weight1`...`WeightN`.
- Example: `frequency([{1,true},{2,false}])`
  `false` is twice as likely to be generated as `true`

Returning to the example: generators

```
?FORALL(X,set(),
        ?FORALL(Y,set(),
                sets:union(X,Y) == sets:union(Y,X)))
```

- How do we implement the generator `set()`?

Returning to the example: generators

```
?FORALL(X,set(),
        ?FORALL(Y,set(),
                sets:union(X,Y) == sets:union(Y,X)))
```

- How do we implement the generator `set()`?
- Almost:

```
set() ->
  list(nat()).
```

Returning to the example: generators

```
?FORALL(X,set(),
        ?FORALL(Y,set(),
                sets:union(X,Y) == sets:union(Y,X)))
```

- How do we implement the generator `set()`?
- Almost:

  ```
  set() ->
    list(nat()).
  ```

- Better:

  ```
  set() ->
    ?LET(X,list(nat()),
         sets:from_list(X)).
  ```

Using QuickCheck in practice. . .

Demo

More set properties

- We clearly need to test more set properties to gain confidence.
- One particularly interesting properties relates the set to list conversion functions with set membership:

```
set_to_from_list() ->
  ?FORALL({S,N},{set(),nat()},
    sets:is_element(N,S) ==
    sets:is_element(N,sets:from_list(sets:to_list(S))
```

Here the definition of the set generator is recursive:

```
set() -> ?SIZED(Size,set(Size)).

set(0) -> sets:new();
set(N) ->
  oneof([sets:new(),
        ?LET({N,S},{nat(),set(N-1)},
             sets:add_element(N,S))]).
```

What have we achieved?

- Tests written on a higher abstraction level compared to "traditional" testing
- A large number of concrete test cases can be generated automatically from test properties
- Once a bug is found, QuickCheck attempts to shrink the bug so that the cause can more easily be identified
- What about test coverage?
  - Clearly good generators are key; can be difficult