Property Based Testing

Thomas Arts

GENERATORS

Generators

Generators

Generators randomly generate data for the test case and have built-in shrinking behavior

Examples:
`int()` generates a random integer
`bool()` randomly generates `true` or `false`
`list(int())` generates a list of random length with randomly chosen integers

Basic generators are defined in eqc_gen module

3

Generators

Test data generators.
- Define a *set* of values for test data…
- …plus a *probability distribution* over that set.

Test data generators are defined by designers, defined by basic generators with generator constructors

```
-record(person,{name, gender, age}).

person() ->
   #person{name = name(),
           gender = oneof([male,female]),
           age = choose(0,120)}.
```

4

Test data generators.
- Define a *set* of values for test data…
- …plus a *probability distribution* over that set.

Test data generators are defined by designers, defined by basic generators with generator constructors

```
-record(person,{name, gender, age}).

person() ->
   #person{name = name(),
           gender = oneof([male,female]),
           age = choose(0,120)}.
```

User defined generator

Basic generators (oneof / choose)

Generators

5

Test data generators.
- Define a *set* of values for test data…
- …plus a *probability distribution* over that set.

Test data generators are defined by designers, defined by basic generators and generator constructors

```
-record(person,{name, gender, age}).

person() ->
   #person{name = name(),
           gender = oneof([male,female]),
           age = choose(0,120)}.
```

A record with generators is a generator itself

Generators

6

Generators are defined in terms of other generators

For example, positive integers

Wrong:

```
nat() ->
    N = int(), abs(N).
```

Returns a test data generator, not an integer.

Abs function undefined for generators

Generators

7

Generators are defined in terms of other generators

For example, positive integers

Right:

```
nat() ->
    ?LET(N,int(),abs(N)).
```

Bind a **name** to **generated value**.

Convert **value** to **constant generator**

Generators

8

The function eqc_gen:sample(Generator) produces a sample of the given generator

```
1> eqc_gen:sample(eqc_gen:int()).
-9
-1
6
12
0
-6
3
15
6
-1
4
ok
```

sample

9

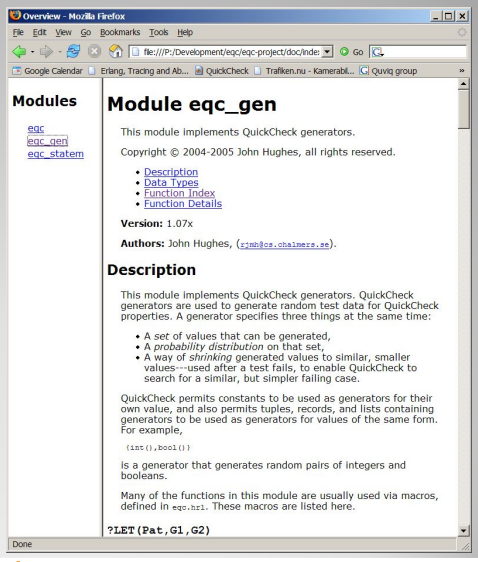
The function eqc_gen:sample(Generator) produces a sample of the given generator

```
1> N = eqc_gen:int().
#Fun<eqc_gen.13.4230413>
2> eqc_gen:sample({N,N}).
{-9,-1}
{5,10}
{0,-5}
{3,11}
{5,-1}
{3,-11}
{-10,7}
{-12,2}
{-11,-2}
{-3,-19}
{3,-1}
ok
```

sample

10

Consult documentation for a set of basic generators



Basic generators

11

An example from the calendar module:

```
day_of_the_week(Date) -> DayNumber

Types:
Date = {Year, Month, Day}
Year = Month = Day = DayNumber = int()
```

This function computes the day of the week given Year, Month and Day. The return value denotes the day of the week as follows:
Monday = 1, Tuesday = 2, ..., Sunday = 7

ear cannot be abbreviated and a value of
93, and not the year 1993. Month is th
January = 1. Day is an integer in the
of days in the month Month of the yea

Let us check for Type Correctness

Calendar Example

12

Straightforward translation (brute force random testing)

```
prop_day_of_the_week1() ->
   ?FORALL(Date,{int(),int(),int()},
          begin
            D = calendar:day_of_the_week(Date),
            (1=<D) and (D=<7)
          end).
```

Calendar Example

13

Run QuickCheck

```
24> eqc:quickcheck(calendar_eqc:prop_day_of_the_week1()).
Failed! Reason:
{'EXIT',{function_clause,[{calendar,date_to_gregorian_days
,[0,0,0]},
                          {calendar,day_of_the_week,3},
                          …]}}
After 1 tests.
{0,0,0}
false
```

Gregorian days function not defined for month 0 and day 0

Calendar Example

14

Read documentation carefully!

```
day_of_the_week(Date) -> DayNumber
```

Types:
Date = {Year, Month, Day}
Year = Month = Day = DayNumber = int()

This function computes the day of the week given Year, Month and Day. The return value denotes the day of the week as follows:
Monday = 1, Tuesday = 2, ..., Sunday = 7
Year cannot be abbreviated and a value of 93 denotes the year 93, and not the year 1993. Month is the month number with January = 1. Day is an integer in the range 1 and the number of days in the month Month of the year Year.

Calendar Example

15

Specify more precise

(guided random testing)

```
prop_day_of_the_week2() ->
   ?FORALL(Date,{int(),choose(1,12),choose(1,31)},
           begin
             D = calendar:day_of_the_week(Date),
             (1=<D) and (D=<7)
           end).
```

Calendar Example

16

Run QuickCheck

```
8>  eqc:quickcheck(calendar_eqc:prop_day_of_the_week2()).
......Failed! Reason:
{'EXIT',{function_clause,[{calendar,last_day_of_the_month,
[-2,2]}, …]}}
After 7 tests.
{-2,2,11}
Shrinking....(4 times)
Reason:
{'EXIT',{function_clause,[{calendar,last_day_of_the_month,
[-1,1]}, …]}}
{-1,1,1}
false
```

Calendar Example

17

Read documentation

```
day_of_the_week(Date) -> DayNumber

Types:
Date = {Year, Month, Day}
Year = Month = Day = DayNumber = int()
```

This function computes the day of the week given Year, Month and Day. The return value denotes the day of the week as follows:

Monday = 1, Tuesday = 2, ..., Sunday = 7

Year cannot be abbreviated and a value of 93 denotes the year 93, and not the year 1993. Month is the month number with January = 1. Day is an integer in the range 1 and the number of days in the month Month of the year Year.

Calendar Example

18

Building your own generators

Several ways of creating a generator for years, i.e., positive integers

```
year() ->
   ?LET(I,int(),abs(I)).
year() ->
   nat().
year() ->
   ?SUCHTHAT(I,largeint(),I>=0).
year() ->
   choose(1800,2100).
```

Many small numbers are generated

19

Calendar Example

Specify more precise
Only Years >= zero

```
prop_day_of_the_week3() ->
   ?FORALL(Date,{year(),choose(1,12),choose(1,31)},
           begin
             D = calendar:day_of_the_week(Date),
             (1=<D) and (D=<7)
           end).
```

20

Run QuickCheck

```
2>  eqc:quickcheck(calendar_eqc:prop_day_of_the_week3()).
.............................Failed! Reason:
{'EXIT',{if_clause,[{calendar,date_to_gregorian_days,3},…]}}
After 31 tests.
{1949,2,29}
Shrinking..(2 times)
Reason:
{'EXIT',{if_clause,[{calendar,date_to_gregorian_days,3},…]}}
{1800,2,29}
false
```

Calendar Example

21

Specify more precise

Verify whether a date is valid before evaluating the function

```
prop_day_of_the_week4() ->
  ?FORALL(Date,{year(),choose(1,12),choose(1,31)},
    ?IMPLIES(calendar:valid_date(Date),
          begin
             D = calendar:day_of_the_week(Date),
             (1=<D) and (D=<7)
          end)).
```

but... we don't like to trust the module we test

Calendar Example

22

Run Quickcheck

```
3>  eqc:quickcheck(calendar_eqc:prop_day_of_the_week4()).
.....................................................x.............
.............................x............
OK, passed 100 tests
true
```

Calendar Example

23

Instead of filtering dates:

A date generator that only generates valid dates

```
prop_day_of_the_week5() ->
   ?FORALL(Date,calendar_date(),
           begin
             D = calendar:day_of_the_week(Date),
             (1=<D) and (D=<7)
           end).
```

Calendar Example

24

A simple generator for dates

```
calendar_date1() ->
  {year(),choose(1,12),choose(1,31)}.
```

```
7> eqc_gen:sample(calendar_eqc:calendar_date1()).
{2074,12,11}
{2172,8,6}
{1942,10,17}
{1959,1,28}
{2200,2,30}
{2029,1,11}
{1977,10,22}
…
```

Building your own generators

25

How to make generator for dates more advanced?

1. Only a few of the generated samples are invalid, use a function to filter them, or
2. Put effort in specifying the number of days per month

Solution 1.

```
calender_date2() ->
  ?SUCHTHAT(Date,
     {year(),choose(1,12),choose(1,31)},
     calendar:valid_date(Date)).
```

Building your own

We trust on calendar implementation

26

Solution 2.

```
calendar_date() ->
?LET({Y,M},{year(),choose(1,12)},
      {Y,M,dayinmonth(Y,M)}).

dayinmonth(Y,M) ->
  oneof(<1,…,28>,<1,…,29>,<1,…,31>,<1,…,30>).
```

Pass values to generator

If Feb and no leap year

If Feb and leap year

If Jan, Mar, May etc

If Apr, Jun, Sep etc

Building your own generators

27

- Problem: we want to include a choice in some cases, but not others
- Trick: list comprehensions with no generator include an element if a condition is true
  - `[1 || true] == [1]`
  - `[1 || false] == []`
- Solution: append (++) such a list comprehension to argument of oneof
  - `oneof([choose(...,...)`
    `|| condition to include it]++`
    `rest)`

Trick: Degenerate List Comprehensions

28

How to make generator for dates more advanced?

```
dayinmonth(Y,M) ->
  oneof(
    [choose(1,28) || (M==2) and not calendar:is_leap_year(Y)] ++
    [choose(1,29) || (M==2) and calendar:is_leap_year(Y)] ++
    [choose(1,30) || lists:member(M,[4,6,9,11]))] ++
    [choose(1,31) || lists:member(M,[1,3,5,7,8,10,12])]).
```

Given that `calendar:is_leap_year` is correct, our `calendar_date()` is a generator for dates.

Building your own generators

29

Idea: test is_leap_year! Look into the manual:

"The notion that every fourth year is a leap year is not completely true. By the Gregorian rule, a year Y is a leap year if either of the following ru

Y is divisible by 4, but not by 100; o

Y is divisible by 400.

Only 3 test cases given. We can do better!

Accordingly, 1996 is a leap year, 1900 is not, but 2000 is."

```
prop_leap_year() ->
  ?FORALL(Y,year(),
          calendar:is_leap_year(Y) ==
          (divisible(Y,4) and not divisible(Y,100))
          or divisible(Y,400)).

divisible(N,M)-> N rem M == 0.
```

Calendar Example

30

Testing calendar module summary:

Fine-tune generators for the basic data type (*date*) in the module
Type correctness is a simple property to formulate
QuickCheck specification precise documentation
Preferably at least one property per function in the module

Summary

31

Objectives

Learn about symbolic test cases
Learn to define recursive generators

Symbolic Test cases

32

We are developing a queue in C

We want to create tests to show that these queues behave as expected

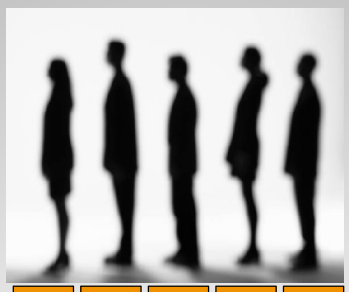
What is “expected” behaviour?

We have a mental model of queues that the software should conform to.

Queues

33

Mental model of a fifo queue



Queue

34

Traditional test cases could look like:

```
Q0 = queue:new(),
Q1 = queue:cons(1,Q0),
Q2 = queue:cons(2,Q1),
1 = queue:head(Q2).
```

We want to check for arbitrary elements that **if we add an element, it's there**.

```
Q0 = queue:new(),
Q1 = queue:cons(8,Q0),
Q2 = queue:cons(0,Q1),
0 = queue:last(Q2),
```

We want to check for arbitrary queues that **last added element is "last"**

Queue

35

We want to know that for any element, when we add it, it's there

```
prop_itsthere() ->
   ?FORALL(I,int(),
           I == queue:last(
                   queue:cons(I,
                      queue:new()))).
```

QuickCheck property

36

Run QuickCheck

```
1> eqc:quickcheck(queue_eqc:prop_itsthere()).
.....................................................
.....................................................
OK, passed 100 tests
true
2>
```

but we want more variation in our test data...

QuickCheck

37

Generating random queues

```
queue() ->
  oneof([queue:new(),
         queue:cons(int(),queue())]).
```

NO GOOD! Why?

- generators as argument of normal function
- infinite recursion

Queue

38

Generating random queues

```
queue() ->
   oneof([queue:new(),
          ?LET({I,Q},{int(),queue()},queue:cons(I,Q))]).
```

Still infinite recursion!

Queue

39

Generating random queues

```
queue() ->


queue(Size) ->
  oneof([queue:new()] ++
        [?LET({I,Q},{int(),queue(Size-1)},queue:cons(I,Q)) ||
         Size>0]).
```

generator for smaller queues

Queue

40

Generating random queues

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

queue(Size) ->
  oneof([queue:new()] ++
        [?LET({I,Q},{int(),queue(Size-1)},queue:cons(I,Q)) ||
         Size>0]).
```

Queue

41

Generating random queues

```
eqc_gen:sample(queue_eqc:queue()).
{[],[-4]}
{[],[]}
{[],[]}
{[],[]}
{[],"\t"}
{[-8],[8,5,-14]}
{"\b",[5]}
{[],[-13]}
{[],[]}
{[5],[5]}
{[],[]}
```

Internal representation of queues

Because of black box testing we do not necessarily understand representation

Queue

42

Generating random queues

```
prop_last_cons () ->
    ?FORALL(Q,queue(), ……).
```

yes... it's a queue, but what to check?

new/0 and cons/2 do not crash, is all we know

Queue

43

Check newly added element is last in queue

```
prop_last_cons () ->
 ?FORALL({I,Q},{int(),queue()},
          queue:last(queue:cons(I,Q)) == I).

eqc:quickcheck(queue_eqc:prop_last_cons()).
...Failed! After 4 tests.
{-1,{[],[1]}}
Shrinking.(1 times)
{0,{[],[1]}}
false
```

counter example hard to read because of internal representation of queues instead of how they were created

Queue

44

Symbolic Queue

Build a symbolic representation for a queue
This representation can be used to both create the queue and to inspect queue creation

```
Q0 = {call,queue,new,[]}
Q1 = {call,queue,cons,[1,Q0]}
Q2 = {call,queue,cons,[2,Q1]}
inspect {call,queue,head,[Q2]}

{[1],[2]} = eval(Q2)
```

eval function provided by QuickCheck in eqc_gen

45

Generating random Queues

Build a symbolic representation for a queue
This representation can be used to both create the queue and to inspect queue creation

Why Symbolic?

1. We want to be able to see how a value is created as well as its result
2. We do not want tests to depend on a specific representation of a data structure
3. We want to be able to manipulate the test itself

46

Generating random symbolic queues

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

queue(Size) ->
  oneof([{call,queue,new,[]}] ++
        [{call,queue,cons,[int(),queue(Size-1)]} ||
         Size>0]).
```

We can now add generators to the arguments

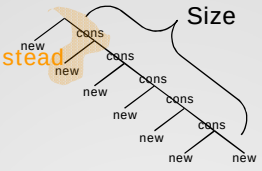
Symbolic Queue

47

Erlang evaluates all arguments first! We compute unnecessarily much

```
?LAZY(
  oneof([{call,queue,new,[]}] ++
        [{call,queue,cons,[int(),queue(Size-1)]} ||
         Size>0])
       )
```

Use lazy evaluation instead



Symbolic Queue

48

Generating random symbolic queues

```
eqc_gen:sample(queue_eqc:queue()).
{call,queue,cons,[-8,{call,queue,new,[]}]}
{call,queue,new,[]}
{call,queue,
      cons,
      [12,
       {call,queue,
             cons,
             [-5,
              {call,queue,
                    cons,
                    [-18,{call,queue,cons,[19,{call,queue,new,[]}]}]}]}]}
{call,queue,
      cons,
      [-18,
       {call,queue,cons,[-11,{call,queue,cons,
                             [-18,{call,queue,new,[]}]}]}]}
```

Symbolic Queue

Generating random symbolic queues

```
prop_cons_tail() ->
 ?FORALL({I,Q},{int(),queue()},
         queue:last(queue:cons(I,eval(Q))) == I).


eqc:quickcheck(queue_eqc:prop_cons_tail()).
...Failed! After 4 tests.
{0,{call,queue,cons,[-1,{call,queue,new,[]}]}}
Shrinking.(1 times)
{0,{call,queue,cons,[1,{call,queue,new,[]}]}}
false
```

clear how queue is created

Symbolic Queue

Symbolic representation helps to understand test data

Symbolic representation helps in manipulating test data (e.g. shrinking)

Symbolic Queue

51

Compare to traditional test cases:

REAL DATA	MODEL
`Q0 = queue:new(),`	`[]`
`Q1 = queue:cons(1,Q0),`	`[1]`
`Q2 = queue:cons(2,Q1),`	`[1,2]`
`1 = queue:head(Q2).`	`↑ (inspect)`
`Q0 = queue:new(),`	`[]`
`Q1 = queue:cons(8,Q0),`	`[8]`
`Q2 = queue:cons(0,Q1),`	`[8,0]`
`0 = queue:last(Q2);.`	`↑ (inspect)`

Model Queue

52

Model Queue

Do we understand queues correctly: what is first and what last?

```
prop_cons() ->
   ?FORALL({I,Q},{int(),queue()},
           model(queue:cons(I,eval(Q))) == model(eval(Q)) ++
  [I]).
```

Write a model function from queues to list

(or use the function queue:to_list, which is already present in the library)

53

Model Queue property

```
eqc:quickcheck(queue_eqc:prop_cons()).
...Failed! After 4 tests.
{0,{call,queue,cons,[1,{call,queue,new,[]}]}}
false
```

54

```
cons(Item, Q1) -> Q2
```

Types: Item = term(), Q1 = Q2 = queue()
Inserts Item at the head of queue Q1. Returns the new queue Q2.

```
head(Q) -> Item
```

Types: Item = term(), Q = queue()
Returns Item from the head of queue Q.
Fails with reason empty if Q is empty.

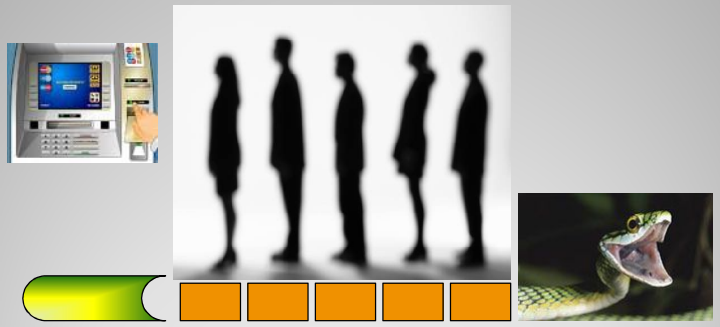
```
last(Q) -> Item
```

Types: Item = term(), Q = queue()
Returns the last item of queue Q. This is the opposite of head(Q).
Fails with reason empty if Q is empty.

Queue manual page

55

Mental model of a fifo queue



Queue

56

Change property to express new understanding

```
prop_cons() ->
   ?FORALL({I,Q},{int(),queue()},
           model(queue:cons(I,eval(Q))) == [I | model(eval(Q))]).


eqc:quickcheck(queue_eqc:prop_cons()).
.....................................................
.....................................................
OK, passed 100 tests
true
```

Model Queue

57

Add properties

```
prop_cons() ->
   ?FORALL({I,Q},{int(),queue()},
           model(queue:cons(I,eval(Q))) == [I | model(eval(Q))]).

prop_head() ->
   ?FORALL(Q,queue(),
     begin
       QVal = eval(Q),
       queue:is_empty(QVal) orelse
                queue:head(QVal) == hd(model(QVal))
     end).
```

similar queue:last(Qval) == lists:last(model(Qval))

Queue

58

There are more constructors for queues, e.g., tail, sonc, in, out, etc. All constructors should respect queue model

Tail removes last added element from the queue

```
queue(Size) ->
  ?LAZY(
    oneof(
        [{call,queue,new,[]}] ++
       [{call,queue,cons,[int(),queue(Size-1)]} || Size>0 ] ++
       [{call,queue,tail,[queue(Size-1)]} || Size>0])).
```

Queue

59

Check properties again

```
eqc:quickcheck(queue_eqc:prop_cons()).
...Failed! Reason:
{'EXIT',{empty,[{queue,tail,[{[],[]}]},
                {queue_eqc,'-prop_cons2/0-fun-0',1},
                 ...
After 4 tests.
{0,{call,queue,tail,[{call,queue,new,[]}]}}
false
```

Queue

60

Only generate well defined queues

```
queue() ->
   ?SIZED(Size,well_defined(queue(Size))).

defined(E) ->
   case catch {ok,eval(E)} of
        {ok,_} -> true;
        {'EXIT',_} -> false
   end.

well_defined(G) ->
  ?SUCHTHAT(X,G,defined(X)).
```

Queue

61

Testing a queue data structure

- symbolic representation make counter examples readable
- recursive generators require size control and lazy evaluation
- Define property for each queue operation: compare result operation on real queue and model

```
model(queue:operator(Q)) == model_operator(model(Q))
```

Summary

62

- Write QuickCheck tests for your queue implementation

Exercises

63

- Implement a queue storing integers in C

queue.c

```
typedef struct queue
{
  int front, rear;
  int buf_size, elements;
  int *buf;
} Queue;
```

- When the queue is full allocate a bigger buffer
- Write Erlang functions to inspect the queue

Exercises

64