Properties and Generators

© Quviq AB

Objectives

Objectives

Get familiar with basic generators and constructing your own generators.

Change your mind about
- value of failing test case
- searching for small test cases

© Quviq AB

Most developers agree that writing unit tests is useful

…. but also quickly gets boring …

An example: the Erlang function lists:seq

 © Quviq AB

Unit tests in Erlang shell:

```
21> lists:seq(1,5).
[1,2,3,4,5]
22> lists:seq(-3,12).
[-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11,12]
23> lists:seq(3,3).
[3]
24> lists:seq(3,2).
[]
```

Manual inspection needed

Some border cases explicitely tested

 © Quviq AB

Automated Unit tests:

```
seq_test() ->
  ?assert([1,2,3,4,5],lists:seq(1,5)),
  ?assert
  ([-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11,12],
     lists:seq(-3,12)),
  ?assert([3],lists:seq(3,3)),
  ?assert([],lists:seq(3,2)).
```

Execution gives test value... Implementation determines what is correct

What is so specific for these values?

How many tests shall we write?

© Quviq AB

Properties... **Try to spot patterns** in your tests

```
seq_test() ->
  ?assert([1,2,3,4,5],lists:seq(1,5)),
  ?assert
  ([-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11,12],
     lists:seq(-3,12)),
  ?assert([3],lists:seq(3,3)),
  ?assert([],lists:seq(3,2)).
```

Length of the created list seems to be 5 = 5 – 1 +1
16 = 12 - -3 +1
1 = 3 – 3 +1
0 = 3 – 2 +1

© Quviq AB

A property for the lists:seq function

```
prop_seq() ->
  ?FORALL({From,To},{int(),int()},
          length(lists:seq(From,To)) ==
                             To – From + 1).
```

int() is a generator for an arbitrary integer value.

 © Quviq AB

From Unit test to Property

A QuickCheck module

```
-module(lists_eqc).

-include_lib("eqc/include/eqc.hrl").

-compile(export_all).

prop_seq() ->
  ?FORALL({From,To},{int(),int()},
          length(lists:seq(From,To)) == To - From + 1).
```

 © Quviq AB

Running QuickCheck

```
1> c(lists_eqc).
{ok,lists_eqc}
2> eqc:quickcheck(lists_eqc:prop_seq()).
....Failed! Reason:
{'EXIT',function_clause}
After 5 tests.
{1,-1}
false
```

3> lists:seq(1,-1).
** exception error: no function clause matching lists:seq
(1,-1)

 © Quviq AB

A property with positive and negative testing

```
prop_seq() ->
  ?FORALL({From,To},{int(),int()},
          try List = lists:seq(From,To),
              length(List) == To - From + 1
          catch
            error:_ ->
              (To - From + 1) < 0
          end).
```

 © Quviq AB

Process

Practical use of QuickCheck

1. Consider which property should hold (not which test should pass)
2. Check the property (100 tests)

false

Error in code or property

Find out failure characteristics:
- Running more tests
- Trying another property
- Tracing the code

true

Check coverage of test data with collect

© Quviq AB

Generators

GENERATORS

© Quviq AB

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

 © Quviq AB

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)}.
```

 © Quviq AB

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)}.
```

User defined generator

Basic generators (oneof / choose)

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 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

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

© Quviq AB

Generators

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**

© Quviq AB

See generated data

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

Eg:

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


© Quviq AB

See generated data

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

Eg:

```
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
```


© Quviq AB

Calendar Example

An example from the calendar module:

day_of_the_week(Date) -> daynum()

Types:

Date = date()

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

Let us check for Type Correctness

© Quviq AB

Calendar Example

Straightforward translation (brute force random testing)

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


© Quviq AB

Calendar Example

We need a generator for date.

date() = {year(), month(), day()}

year() = integer() >= 0

Year cannot be abbreviated. Example: 93 denotes year 93, not 1993. Valid range depends on the underlying OS. The date tuple must denote a valid date.

month() = 1..12

day() = 1..31

© Quviq AB

Building your own generators

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

```
year() ->
    ?SUCHTHAT(I,int(),I>=0).
year() ->
    ?LET(I,int(),abs(I)).
year() -> nat().
year() ->  choose(1586,2100).
year() ->  choose(1800,2200).
```

This may generate a lot of integers that are ignored

Many small numbers are generated

What is the use-case?

© Quviq AB

Specify more precise

(guided random testing)

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

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

 © Quviq AB

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
```

 © Quviq AB

Calendar Example

Specify more precise

Verify whether a date is valid before evaluating the function

```
prop_day_of_the_week() ->
  ?FORALL(Date,date(),
    ?IMPLIES(calendar:valid_date(Date),
          begin
             D = calendar:day_of_the_week(Da
             (1=<D) and (D=<7)
          end)).
```

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

Calendar Example

Run Quickcheck

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

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)).
```

We trust on calendar implementation

Solution 2.

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

Pass values to generator

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

- If Feb and no leap year
- If Feb and leap year
- If Jan, Mar, May etc
- If Apr, Jun, Sep etc

Trick: Degenerate List Comprehensions

- **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)`

© Quviq AB

Building your own generators

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.

© Quviq AB

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 rules is valid:

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

Y is divisible by 400.

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

Only 3 test cases given. We can do better!

```
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.
```


© Quviq AB

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

© Quviq AB

QuviQ

Symbolic Test Cases

 © Quviq AB

Objectives

Objectives

Learn about symbolic test cases
Learn to define recursive generators

 © Quviq AB

Erlang contains a queue data structure (see stdlib documentation)

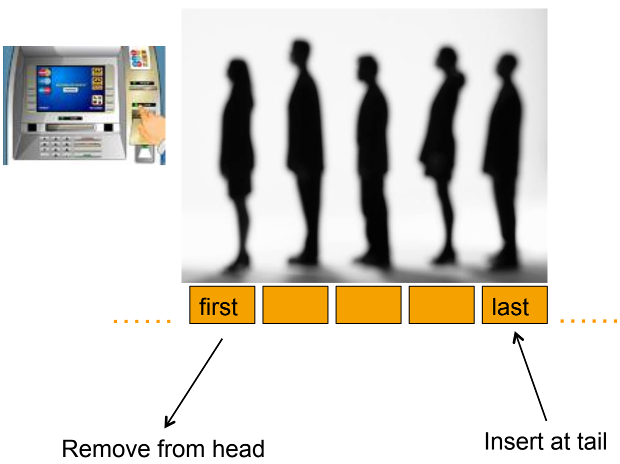
We want to test that these queues behave as expected

What is "expected" behaviour?

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

© Quviq AB

Mental model of a fifo queue



© Quviq AB

Unit tests 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"**

© Quviq AB

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()))).
```


© Quviq AB

Run QuickCheck

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

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

 © Quviq AB

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()))).
```

Any queue, not only a new queue

 © Quviq AB

Generating random queues

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

NO GOOD! Why?

- generators as argument of normal function
- infinite recursion

© Quviq AB

Queue

Generating random queues

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

Still infinite recursion!

© Quviq AB

Generating random queues

```
queue() ->


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

generator for smaller queues

 © Quviq AB

Generating random queues

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

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

 © Quviq AB

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

© Quviq AB

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

© Quviq AB

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]}

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

eval function provided by QuickCheck in eqc_gen

© Quviq AB

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

© Quviq AB

Generating random symbolic queues

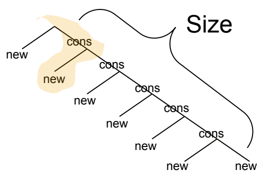
```
queue() ->
   ?SIZED(Size,queue(Size)).

queue(0) ->
  {call,queue,new,[]};
queue(N) ->
  oneof([queue(0),
         {call,queue,cons,[int(),queue(N-1)]}]).
```

We can now add generators to the arguments

 © Quviq AB

Erlang evaluates all arguments first!
We compute unnecessarily much

```
?LAZY(oneof([queue(0),
             {call,queue,cons,[int(),queue(N-1)]}])
      ).
```



Use lazy evaluation instead

 © Quviq AB

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,[]}]}]}]}
```

 © Quviq AB

Generating random symbolic queues

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

```
eqc:quickcheck(queue_eqc:prop_last_cons()).
...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

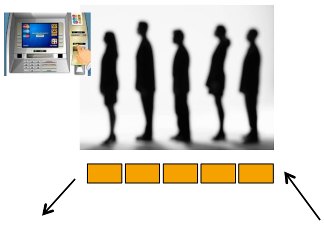
 © Quviq AB

Symbolic Queue

Symbolic representation helps to understand test data

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

But, in order to understand the behaviour,
we need a MODEL



© Quviq AB

Model Queue

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)
```


© Quviq AB

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)

© Quviq AB

Model Queue property

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


© Quviq AB

Queue manual page

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.

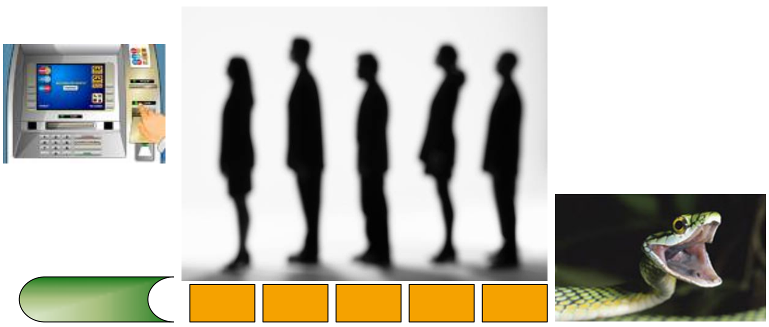
last(Q) -> Item

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.

 © Quviq AB

Queue

Mental model of a fifo queue



 © Quviq AB

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
```

 © Quviq AB

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))

 © Quviq AB

Queue

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


© Quviq AB

Queue

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
```

cause immediately clear: advantage of symbolic representation

© Quviq AB

Queue

Only generate well defined queues (See eqc_symbolic)

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

well_defined part of QuickCheck library

Only generate symbolic terms for which evaluation does NOT crash

© Quviq AB

Summary

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))

© Quviq AB