QuickCheck

Properties and Generators

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

From Unit test to Property

Most developers agree that writing unit tests is useful

…. but also quickly gets boring …

An example: the Erlang function lists:seq

From Unit test to Property

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

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?

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 = 2 – 3 +1

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.

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).
```

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)

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).
```

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

Recursive Generators and Testing Data Types

Thomas Arts

Objectives

Learn about symbolic test cases
Learn to define recursive generators

Data types

- core libraries used by many
- expected to be error free

How to test data types effectively?

Example data type: Decimal

Store "money" as digits before and after the decimal separator

€ M.N → {decimal, M, N}
where M and N are 32 bit integers

Example data type: Decimal

Several "constructors"

new(*integer*) -> *decimal*
new(*float*) -> *decimal*
new(*integer*,*natural*) -> *decimal*
add(*decimal*, *decimal*) -> *decimal*
divide(*decimal*, *decimal*) -> *decimal*
....

Write QuickCheck properties

generator

```
decimal() ->
   ?LET({M,N}, {int(),nat()}, new(M,N)).
```

property

```
prop_add_comm() ->
  ?FORALL({D1,D2}, {decimal(),decimal()},
                  add(D1,D2) == add(D2,D1)).
```


Write QuickCheck properties

Which properties and when enough?

```
decimal() ->
   ?LET({M,N}, {int(),nat()}, new(M,N)).

prop_add_comm() ->
  ?FORALL({D1,D2}, {decimal(),decimal()},
                  add(D1,D2) == add(D2,D1)).
```

Testing data types

Write QuickCheck properties

Compare implementation to a model implementation:
(Arts, Castro, Hughes 2008)

[add(D1,D2)] = [D1] + [D2]
[subs(D1,D2)] = [D1] – [D2]
[mult(D1,D2]) = [D1] * [D2]
[lt(D1,D2)]l = [D1] < [D2]

implemented operation

model operation

Testing data types

How to create the model ?

- Simpler than Subject Under Test
- Correctness verified by incompatibility with implementation

In this case, use Erlang/C floating point implementation as model (based upon IEEE 754-1985 standard)

```
model(Decimal) ->
  decimal:get_value(Decimal).
```

For each operator one property, e.g.:

```
prop_add() ->
  ?FORALL({D1,D2},{decimal(),decimal()},
          model(add(D1,D2)) ==
            model(D1) + model(D2)).
```

Model addition

```
prop_lt() ->
   ?FORALL({D1,D2},{decimal(),decimal()},
           lt(D1,D2) ==
                    model(D1) < model(D2)).
```

returns a boolean

The model is too rough!

```
>eqc:quickcheck(decimal_eqc:prop_add()).
........Failed! After 9 tests.
```

Reason:

model({decimal,0,1},{decimal,0,2}) =/=
 model({decimal,0,1}) + model({decimal,0,2}).

> 0.3 =/= 0.1+0.2
difference: 5.55112e-17

floating point arithmetic

Model versus reality

The model is too rough!

```
equiv(F1,F2) ->
  if (abs(F1-F2) < ?ABS_ERROR) -> true;
     (abs(F1) > abs(F2)) ->
          abs( (F1-F2)/F1 ) < ?REL_ERROR;
     (abs(F1) < abs(F2)) ->
          abs( (F1-F2)/F2 ) < ?REL_ERROR
  end.
```

Adopt model, but do not copy the implementation!

Sufficient testing

One property per operation and a "good enough" model.

Is this sufficient testing? No!

We only test on decimals created by:

decimal() ->
  ?LET({M,N}, {int(),nat()}, new(M,N)).

But the other constructors could break an invariant

Sufficient testing

We only test on decimals created by:

```
decimal() ->
  ?LET({M,N}, {int(),nat()}, new(M,N)).
```

assume

model(add(new(1,0),new(0,1))) →

model(add({decimal,1,0},{decimal,0,1})) →

model(**{decimal,1.1,0}**) →

1.1 == 1.0 + 0.1 ←

model({decimal,1,0}) + model({decimal,0,1})

Not found by the prop_add

Sufficient testing

Subtraction:

model(sub(new(1,1),new(0,1))) →

model(sub({decimal,1,1},{decimal,0,1})) →

model({decimal,1,0}) →

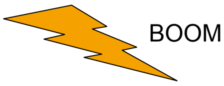
1.0 == 1.1 - 0.1 ←

model({decimal,1,1}) + model({decimal,0,1})

Computed in a smart way with carrier

Sufficient testing

model(sub(add(new(1,0),new(0,1)),new(0,1))) →
model(sub(**{decimal,1.1,0}**, {decimal,0,1})) →



The data structure is corrupted, we only notice if we use it in a specific way!

Sufficient testing

One property per operation and a "good enough" model.

Not enough for sufficient testing!
We need to generate the values in all possible ways!

Improved generator:

base case

```
decimal() ->
   oneof([?LET({M,N}, {int(),nat()}, new(M,N)),
          add(decimal(),decimal()),
          sub(decimal(),decimal())
         ]).
```

NO GOOD! Why?

- generators as argument of normal function
- infinite recursion

Improved generator:

```
decimal() ->
   oneof([?LET({M,N}, {int(),nat()}, new(M,N)),
          ?LET([D1,D2],[decimal(),decimal()],
               add(D1,D2)),
          ?LET([D1,D2],[decimal(),decimal()],
               sub(D1,D2)
         ]).
```

Still infinite recursion

Generate in all possible ways

```
decimal(0) ->
  ?LET({M,N}, {int(),nat()}, new(M,N));
decimal(S) ->
  Smaller = decimal(S div 2),
  oneof([decimal(0),
         ?LET([D1,D2],[Smaller,Smaller],
              add(D1,D2)),
         ?LET([D1,D2],[Smaller,Smaller],
              sub(D1,D2))]).
```

base case

generator for smaller decimals

Generate in all possible ways

```
decimal(0) ->
  ?LET({M,N}, {int(),nat()}, new(M,N));
decimal(S) ->
  Smaller = decimal(S div 2),
  oneof([decimal(0),
         ?LET([D1,D2],[Smaller,Smaller],
              add(D1,D2)),
         ?LET([D1,D2],[Smaller,Smaller],
              sub(D1,D2))]).
```

base case

generator for smaller decimals

unbalanced depth!

Generate in all possible ways

```
decimal(0) ->
  ?LET({M,N}, {int(),nat()}, new(M,N));
decimal(S) ->
  Smaller = decimal(S div 2),
  oneof([decimal(0),
         ?LET([D1,D2],[Smaller,Smaller],
              add(D1,D2)),
         ?LET([D1,D2],[Smaller,Smaller],
              sub(D1,D2))]).

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

base case

generator for smaller decimals

Vary size with test size

Testing data types

Generating data structures using all possible constructors

Create a model with model operations

Have one property per operation comparing the operation with the model:

```
prop_op() ->
  ?FORALL(Xs,vector(X,datatype()),
          model(apply(op,Xs)) ==
               model_op([model(X) || X<-Xs])).
```

Queues

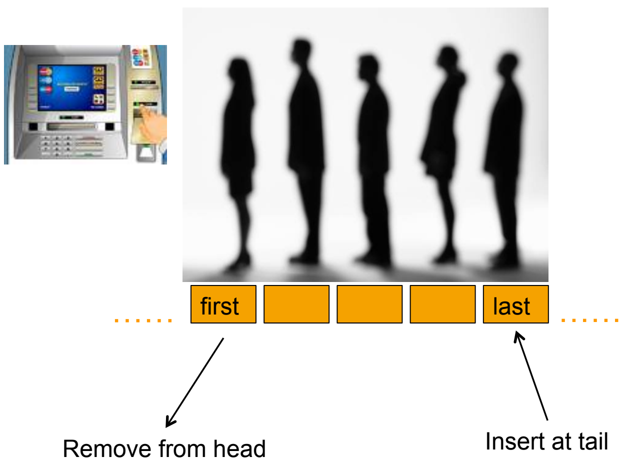
Erlang contains a queue data structure
(see stdlib documentation)

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.

Queue

Mental model of a fifo queue



Queue

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))]).
```


Queue

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

Model the queue by lists

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

Write a model function from queues to list

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


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

Ok, the model is wrong or the code is wrong... but what queue did we construct??

Symbolic Queue

Build a symbolic representation for a queue

This representation can be used to both **create the queue** and to **inspect queue creation**

```
Q = {call,queue,cons,[1, {call,queue,new,[]}]}

{[],[1]} = eval(Q)
```

eval function provided by QuickCheck in eqc_gen

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

Generating random symbolic queues

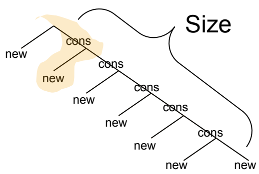
```
queue() ->
   ?SIZED(Size,queue(Size)).

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

Erlang evaluates all arguments first!
We compute unnecessarily much

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



Use lazy evaluation instead

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


Model the queue by lists

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

Model Queue property

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

Ok, the model is wrong.

We know what the queue is!

Symbolic Queue

Symbolic representation helps to understand test data

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

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

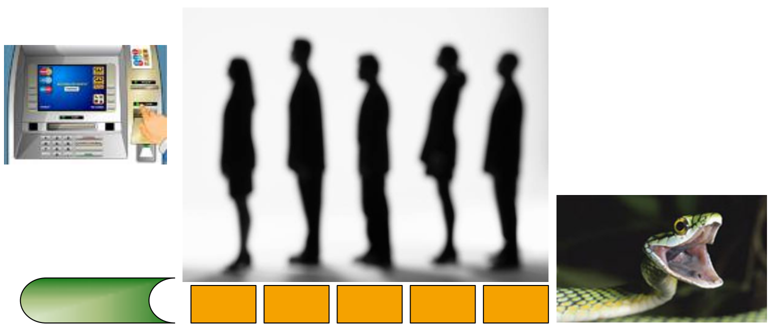
Types: **Item = term(), Q = queue()**
Returns Item from the head of queue Q.
Fails with reason empty if Q is empty.

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.

Queue

Mental model of a fifo queue



tail   head   taihead

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
```


Add properties

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

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

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

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)]}])).
```


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

Only generate well defined queues (See eqc_symbolic)

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

Repeat computation of queue until a non-crashing one is found.

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))