Testing Abstract Data Structures with QuickCheck

Thomas Arts

Quviq AB / Chalmers

Erlang shell

```
21> lists:seq(1,5).
```

Erlang shell

```
21> lists:seq(1,5).
[1,2,3,4,5]
22> lists:seq(-3,12).
```

```
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,-4).
```

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,-4).
** exception error: no function clause
matching lists:seq(3,-4)
24> lists:seq(3,3).
```

```
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,-4).
** exception error: no function clause
matching lists:seq(3,-4)
24> lists:seq(3,3).
[3]
25> lists:seq(3,2).
```

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,-4).
** exception error: no function clause matching
lists:seq(3,-4)
24> lists:seq(3,3).
[3]
25> lists:seq(3,2).
[]
```

The functions lists:seq/1,2 return the empty list in a few cases when they used to generate an exception, for example lists:seq(1, 0).

See lists(3) for details. (Thanks to Richard O'Keefe.)

*** POTENTIAL INCOMPATIBILITY ***

Own Id: OTP-7230

seq(From, To) -> Seq
seq(From, To, Incr) -> Seq

Returns a sequence of integers which starts with From and contains the successive results of adding Incr to the previous element, until To has been reached or passed (in the latter case, To is not an element of the sequence). Incr defaults to 1.

Failure: If To<From-Incr and Incr is positive, or

if To>From-Incr and Incr is negative, or

if Incr==0 and From/=To.

The following equalities hold for all sequences:

length(lists:seq(From, To)) == To-From+1

length(lists:seq(From, To, Incr)) == (To-From+Incr) div Incr

Property

```
prop_seq() ->
  ?FORALL({From,To,Incr},{int(),int(),int()},
     case catch lists:seq(From,To,Incr) of
    {'EXIT',_} ->
      (To < From-Incr andalso Incr > 0) orelse
            (To > From-Incr andalso Incr < 0) orelse
            (Incr==0 andalso From /= To);
    List when Incr /= 0 ->
      is_list(List) andalso
            length(List) == (To-From+Incr) div Incr
      end).
```

QuickCheck property

```
eqc:quickcheck(lists_eqc:prop_seq()).
Failed! Reason:
{'EXIT',{badarith,[{lists_eqc,'-prop_seq/0-fun-0-',1},
                   {eqc_gen,gen,3}]}}
After 1 tests.
{0,0,0}
false
```

of course... you cannot divide by zero, but...

```
> lists:seq(0,0,0).
[0]
```

Property

```
prop_seq() ->
  ?FORALL({From,To,Incr},{int(),int(),int()},
      case catch lists:seq(From,To,Incr) of
     {'EXIT',_} ->
       (To < From-Incr andalso Incr > 0) orelse
            (To > From-Incr andalso Incr < 0) orelse
            (Incr==0 andalso From /= To);
     List when Incr /= 0 ->
       is_list(List) andalso
             length(List) == (To-From+Incr) div Incr;
     [From] when Incr == 0 ->
       true
       end).
```

Moral

1. No matter how well you specify....

   if you only write unit tests to validate your code, then you will forget a test case.

2. Even simple functions are worth checking with QuickCheck.

Data types

QuickCheck for unit testing...

...data types as simple example

Example of Erlang modules defining data types:

queue.erl, dict.erl, sets.erl, digraph.erl, etc.

Implementation of a data type:

- implement a data structure, and
- interface functions to manipulate data structure

A type is defined... an opaque type.

```
-opaque set() :: #set{}.
```

Construct
ing a set

```
-spec new() -> set().
-spec from_list([term()]) -> set().
```

Accessi
ng a
set

```
-spec size(set()) -> non_neg_integer().
-spec to_list(set()) -> [term()].
```

```
-opaque set() :: #set{}.

-spec new() -> set().

-spec from_list([term()]) -> set().

-spec size(set()) -> non_neg_integer().

-spec to_list(set()) -> [term()].
```

This is how one generates a set

Use to create a set generator

From types to generators

From –type and –spec annotations one can retrieve interface for generators

```
7> eqc_types:defining(sets).
Defining type {set,0}
{call,sets,new,[]}
{call,sets,from_list,[list(term())]}
{call,sets,add_element,[term(), set()]}
{call,sets,del_element,[term(), set()]}
{call,sets,union,[set(), set()]}
{call,sets,union,[list(set())]}
{call,sets,intersection,[set(), set()]}
{call,sets,intersection,[non_empty(list(set()))]}
{call,sets,subtract,[set(), set()]}
{call,sets,filter,[function1(bool()), set()]}
```

Generator

Create a generator for the data type by calling a sequence of interface functions.

The result of the generator is a symbolic term, that when evaluated creates the data structure.

E.g.:

```
{call,sets,intersection,
         [[{call,sets,union,
               [{call,sets,new,[]},{call,sets,from_list,[[-14,3,-9,10]]}]},
          {call,sets,union,
               [{call,sets,from_list,[[-18,14,3,-3,3,7]]},
                {call,sets,from_list,[[6,3,1,14]]}]},
          {call,sets,subtract,[{call,sets,new,[]},{call,sets,new,[]}]}]]}
```

Properties

What properties to write and how do we know we have written enough properties?

Use a model interpretation!

⟦ sets:f(Set)⟧ ≃ f_model(⟦Set⟧)

Example:

⟦ sets:union(Set1,Set2)⟧ ≃ ⟦Set1⟧ ∪ ⟦Set1⟧

⟦Set⟧ □ turn the data structure set into a real set

But we have no real sets, that's why we implemented them in Erlang

But we may have

- a reference implementation (in any language)
- a simple correct implementation, but inefficient implementation

We use such implementation as our model

Model

⟦Set⟧

```
model(Set) -> lists:sort(sets:to_list(Set)).


munion(S1,S2) -> lists:usort(S1++S2).

madd_element(E,S) -> lists:usort([E|S]).


etc
```

```
-opaque set() :: #set{}.

-spec new() -> set().
-spec from_list([term()]) -> set().

-spec size(set()) -> non_neg_integer().
-spec to_list(set()) -> [term()].
```

Any function that takes a set as argument defines a property.

Properties

```
prop_add_element(G) ->
  ?FORALL({E,Set},{G,set(G)},
       model(sets:add_element(E,eval(Set))) ==
   madd_element(E,model(eval(Set)))).

prop_union(G) ->
  ?FORALL({Set1,Set2},{set(G),set(G)},
       model(sets:union(eval(Set1),eval(Set2))) ==
   munion(model(eval(Set1)),model(eval(Set2)))).

prop_size(G) ->
  ?FORALL(Set,set(G),
       sets:size(eval(Set)) ==
   msize(model(eval(Set)))).
```

But what about ...

filter(Pred, Set1) -> Set2

Types:

Pred = fun (E) -> bool()

Set1 = Set2 = set()

Filter elements in Set1 with boolean function Fun.

Propeties

```
prop_filter(G) ->
  ?FORALL({Pred,Set},{function1(bool()),set(G)},
       model(sets:filter(Pred,eval(Set))) ==
          mfilter(Pred,model(eval(Set)))).

mfilter(Pred,S) ->
    [ E || E<-S, Pred(E)].
```

Summary

Easy to use QuickCheck for testing data types

Use type signatures to define generators

Create a model interpretation

Create a model function for each interface function

Create a property for each interface function

Guaranteed full test of the data structure