Part 4 – Testing Erlang Programs
How do we know that software works?
  – One commonly used method is to use testing

Let’s do manual testing of Erlang programs first
  – Relatively easy due to the interactive shell

%% my first sort program, inspired by QuickSort
-module(my_sort).
-export([sort/1]).

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
  sort([X || X <- Xs, X < P])
++ [P] ++ sort([X || X <- Xs, P < X]).
• Seems to work!

• However, perhaps it’s not a good idea to execute these tests repeatedly by hand
  – Let’s put them in the file...
  – … and exploit the power of pattern matching
-module(my_sort).
-export([[sort/1, sort_test/0]]).

-spec sort([T]) -> [T].
sort([]) -> []; sort([P|Xs]) ->
    sort([X || X <- Xs, X < P])
    ++ [P] ++ sort([X || X <- Xs, P < X]).

-spec sort_test() -> ok.
sort_test() ->
    [] = sort([]),
    [17,42] = sort([17,42]),
    [17,42] = sort([42,17]),
    [1,2,3,4] = sort([3,1,4,2]),
    ok.

• And now let’s use EUnit to run them automatically
Running tests using EUnit

- EUnit in its simplest form is a test framework to automatically run all _test functions in a module.
- Calling `eunit:test(Module)` was all that was needed here.
- However, EUnit can do much more...
  - Let us, temporarily, change one test to:
    ```
    [1,3,2,4] = sort([3,1,4,2])
    ```
  - and see what happens.
• Reports number of tests that failed and why
  – the report is pretty good, but it can get even better
  – using EUnit macros
%% my first sort program, inspired by QuickSort
-module(my_sort).
-export([sort/1, sort_test/0]).

/include_lib("eunit/include/eunit.hrl").

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
    sort([X || X <- Xs, X < P])
    ++ [P] ++ sort([X || X <- Xs, P < X]).

-spec sort_test() -> ok.
sort_test() ->
    ?assertEqual([], sort([])),
    ?assertEqual([17,42], sort([17,42])),
    ?assertEqual([17,42], sort([42,17])),
    ?assertEqual([1,3,2,4], sort([3,1,4,2])),
    ok.
This report is much more detailed

But, it considers the complete set of tests as one
-module(my_sort).
-export([sort/1]).

-includelib("eunit/include/eunit.hrl").

sort([]) -> ...

sort_test_() -> % notice trailing underscore
    [test_zero(), test_two(), test_four()].

test_zero() ->
    [~, _assertEqual([], sort([]))]. % notice underscores

test_two() ->
    [~, _assertEqual([17,42], sort([17,42])),
                  _assertEqual([17,42], sort([42,17]))].

test_four() ->
    [~, _assertEqual([1,3,2], sort([3,1,4,2]))].
EUnit test generators

• EUnit now reports accurate numbers of passed and failed test cases

• In fact, we can test EUnit generators individually
• This works only for test generator functions
  (not very impressive, as there is only one in this example)

• There are other forms that may come handy (RTFM)
  e.g. \{dir,Path\} to run all tests for the modules in Path
• Let us undo the error in the test_four test
• add one more EUnit generator

```erlang
another_sort_test_() ->
    [test_five()].

test_five() ->
    [_assertEqual([1,2,3,4,5], sort([1,3,2,4,5])),
     _assertEqual([1,2,3,4,5], sort([1,4,5,2,3]))].
```

• and run again: all tests and just the new ones

```erlang
15> c(my_sort).
{ok,my_sort}
16> eunit:test(my_sort).
   All 6 tests passed
ok
17> eunit:test({generator, fun my_sort:another_sort_test_/0}).
   All 2 tests passed
ok
```
There is more to EUnit...

- More macros
  - Utility, assert, debugging, controlling compilation
- Support to run tests in parallel
- Lazy generators
- *Fixtures* for adding scaffolding around tests
  - Allow to define setup and teardown functions for the state that each of the tests may need
  - Useful for testing stateful systems

For more information consult the EUnit manual
Towards automated testing

- Testing accounts for a large part of software cost
- Writing (unit) tests by hand is
  - boring and tedious
  - difficult to be convinced that all cases were covered
- Why not automate the process?
  - Yes, but how?
- One approach is property-based testing
  - Instead of writing test cases, let’s write properties that we would like our software (functions) to satisfy
  - and use a tool that can automatically generate random inputs to test these properties
-module(my_sort).
-export([sort/1]).

-includelib("proper/include/proper.hrl").
-includelib("eunit/include/eunit.hrl").

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
    sort([X || X <- Xs, X < P])++[P]++sort([X || X <- Xs, P < X]).

prop_ordered() ->
    ?FORALL(L, list(integer()), ordered(sort(L))).

ordered([]) -> true;
ordered([_]) -> true;
ordered([A,B|T]) -> A =< B andalso ordered([B|T]).
Testing the ordered property

- Runs any number of “random” tests we feel like
- If all tests satisfy the property, reports that all tests passed

```erl
$ erl -pa /path/to/proper/ebin
Erlang (BEAM) emulator version 5.10.3

Eshell V5.10.3 (abort with ^G)
1> c(my_sort).
{ok,my_sort}
2> proper:quickcheck(my_sort:prop_ordered()).
.......... 100 dots ..........OK: Passed 100 tests
true
3> proper:quickcheck(my_sort:prop_ordered(), 10000).
.......... 10000 dots ..........OK: Passed 10000 tests
true
```
Another property for sorting

-module(my_sort).
-export([sort/1]).

-include_lib("proper/include/proper.hrl").
-include_lib("eunit/include/eunit.hrl").

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
    sort([X || X <- Xs, X < P])
    ++ [P] ++ sort([X || X <- Xs, P < X]).

prop_ordered() ->
    ?FORALL(L, list(integer()), ordered(sort(L))).

prop_same_length() ->
    ?FORALL(L, list(integer()),
        length(L) =:= length(sort(L))).

ordered([]) -> ...
Testing the same length property

4> c(my_sort).
   {ok,my_sort}
5> proper:quickcheck(my_sort:prop_same_length()).
   .............!
Failed: After 14 test(s).
   [1,3,-3,10,-3]

Shrinking (6 time(s))
   [0,0]
   false
6> proper:quickcheck(my_sort:prop_same_length()).
   .............!
Failed: After 13 test(s).
   [2,-8,-3,1,1]

Shrinking .(1 time(s))
   [1,1]
   false
Let us suppose that we actually wanted that our program only sorts lists without duplicates.

How would we have to write the property then?

```erlang
prop_same_length() ->
    ?FORALL(L, list(integer()),
        ?IMPLIES(no_duplicates(L),
            length(L) =:= length(sort(L)))).
```

%% better implementations of no_duplicates/1 exist
no_duplicates([]) -> true;
no_duplicates([A|T]) ->
    not lists:member(A, T) andalso no_duplicates(T).

7> proper:quickcheck(my_sort:prop_same_length()).
...........x.x..................x.xx....xx........xx.xxxxx.....x....xx.xxx
...........xx.x.x........x.x.x.x.x.....xxxxx.xxxxxx....x.x.x.x.x.x.x
OK: Passed 100 tests
Custom generators

- An even better way is to try to generate lists without duplicates in the first place!

```erlang
list_no_dupls(T) ->
  ?LET(L, list(T), remove_duplicates(L)).

%% better versions of remove_duplicates/1 exist
remove_duplicates([]) -> [];
remove_duplicates([A|T]) ->
  case lists:member(A, T) of
    true -> remove_duplicates(T);
    false -> [A|remove_duplicates(T)]
  end.
```

```erlang
prop_same_length() ->
  ?FORALL(L, list_no_dupls(integer()),
  length(L) =:= length(sort(L))).
```

```erlang
7> proper:quickcheck(my_sort:prop_same_length()).
.......... 100 dots ..........
OK: Passed 100 tests```
Testing for stronger properties

- Ok, but the properties we tested were quite weak
- How about ensuring that the list after sorting has the same elements as the original one?
- We can use some ‘obviously correct’ function as reference implementation and test equivalence

```
prop_equiv_usort() ->
   ?FORALL(L, list(integer()),
   sort(L) =:= lists:usort(L)).
```

```
8> proper:quickcheck(my_sort:prop_equiv_usort()).
......... 100 dots ..........
OK: Passed 100 tests
```

- **Note**: PropEr is ideally suited for easily checking equivalence of two functions and gradually refining or optimizing one of them!
Beyond monotypic testing

- But why were we testing for lists of integers?
- We do not have to! We can test for general lists!

```erlang
def proposition(prop_equiv_usort()) ->
  ?FORALL(L, list(), sort(L) =:= lists:usort(L)).
```

```erlang
9> proper:quickcheck(my_sort:prop_equiv_usort()).
........... 100 dots ...........
OK: Passed 100 tests
```
Shrinking general terms

- How does shrinking work in this case?
- Let’s modify the property to a false one and see

```
prop_equiv_sort() ->
    ?FORALL(L, list(), sort(L) =:= lists:sort(L)).
```

```
10> proper:quickcheck(my_sort:prop_equiv_sort()).
.............!
Failed: After 14 test(s)
[[[],[<<54,17,42:7>>]],4],{},-0.05423250622902363,{},{42,<<0:3>>}]
Shrinking ...(3 time(s))
[{},{}]
falsen
11> proper:quickcheck(my_sort:prop_equiv_sort()).
.........................!
Failed: After 28 test(s)
[{},[],6,‘f%Co’,{42},.... A REALLY BIG COMPLICATED TERM HERE
CONTAINING TWO EMPTY LISTS
Shrinking ....(4 time(s))
[[],[]]
falsen
```
Basic generators

- any Erlang term
- atom()
- boolean()
- integer()
- pos_integer(), ...
- range(L,H)
  range(17,42)
- any()
- list(G)
- vector(Len,G)
- union(Gs)
  union([a,b])
- frequency(Gs)
  frequency([[1,a],[4,b]])
## Testing frameworks

<table>
<thead>
<tr>
<th></th>
<th>Unit Testing</th>
<th>Property-Based Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquire a valid input</td>
<td>User-provided inputs</td>
<td>Generated semi-randomly from specification</td>
</tr>
<tr>
<td>Run the program</td>
<td>Automatic</td>
<td>Automatic</td>
</tr>
<tr>
<td>Decide if it passes</td>
<td>User-provided expected outputs</td>
<td>Partial correctness property</td>
</tr>
</tbody>
</table>
More about PropEr

- Homepage: http://proper.softlab.ntua.gr
- GitHub: http://github.com/manopapad/proper