A PropEr Talk

Kostis Sagonas

With PropEr help by

Manolis Papadakis
Eirini Arvaniti
A PropEr announcement

PropEr
A QuickCheck-Inspired Property-Based Testing Tool for Erlang

Contents
- About: The PropEr developers
- API: The PropEr API and its documentation
- Download: With PropEr instructions on how to do this
- FAQ: Frequently Asked Questions with PropEr Answers
- Publications: PropEr papers and talks
- Tips: For the effective use of PropEr
- Tutorials: Showing the tool's PropEr use
A PropEr question

Why did you create PropEr?
How Erlang modules used to look

```erlang
zip_open(Archive) -> zip_open(Archive, []).

zip_open(Archive, Options) ->
    Pid = spawn(fun() -> server_loop(not_open) end),
    request(self(), Pid, {open, Archive, Options}).

zip_get(Pid) when is_pid(Pid) ->
    request(self(), Pid, get).

zip_close(Pid) when is_pid(Pid) ->
    request(self(), Pid, close).
```
How modern Erlang modules look

```erlang
-module(zip).
-export([open/1, get/1, close/1]).
-export([spec/1]).

-spec open(archive()) -> zip_open_return().

zip_open(Archive) -> zip_open(Archive, []).

-spec open(Archive, Options) -> zip_open_return().

zip_open(Archive, Options) ->
    Pid = spawn(fun() -> server_loop(not_open) end),
    request(self(), Pid, {open, Archive, Options}).

-spec get(pid()) -> {ok, [filespec()] | {error, term()}.

zip_get(Pid) when is_pid(Pid) ->
    request(self(), Pid, get).

-spec close(pid()) -> 'ok' | {error, 'eintrval'}.zip_close(Pid) when is_pid(Pid) ->
    request(self(), Pid, close).
```
A PropEr start...

**MANOLIS, HAVE YOU EVER HEARD OF SOMETHING CALLED QUICK-CHECK? IT IS ESSENTIALLY BLABLABLA**

**I WANT TO COMBINE IT WITH ERLANG'S TYPE LANGUAGE! IT'S A GREAT DIPLOMA THESIS TOPIC!**

**I KNOW YOU BY NOW, WHAT'S THE CATCH? WELL, QUICKCHECK IS CLOSED SOURCE, SO WE HAVE TO IMPLEMENT THE WHOLE THING FIRST!**
PropEr progress four months later

IT HAS TAKEN ME A LONG TIME, BUT I FINALLY FINISHED THE IMPLEMENTATION.

GOOD JOB! THIS WILL DO AS A BASIS.

NOW IT’S TIME TO START WORKING ON YOUR THESIS TOPIC.

4 MONTHS OF WORK AND I HAVEN’T EVEN STARTED YET? THAT’S RIGHT.
PropEr: A property-based testing tool

• Inspired by QuickCheck
• Available open source under GPL
• Has support for
  - Writing properties and test case generators
    - ?FORALL/3, ?IMPLIEDS, ?SUCHTHAT/3, ?SHRINK/2,
      - ?LAZY/1, ?WHENFAIL/2, ?LET/3, ?SIZED/2,
      - aggregate/2, choose2, oneof/1, ...
  - Concurrent/parallel “statem” and “fsm” testing
• Fully integrated with the language of types and specs
  - Generators often come for free!
-module(simple Props).

%% Properties are automatically exported.
-include_lib("proper/include/proper.hrl").

%% Functions that start with prop_ are considered properties
prop_t2b_b2t() ->
  ?FORALL(T, term(), T =:= binary_to_term(term_to_binary(T))).

1> c(simple Props).
{ok,simple Props}  
2> proper:quickcheck(simple Props:prop_t2b_b2t()).
............................
............................
OK: Passed 100 test(s)
true
prop_enc_dec() ->
  ?FORALL(Msg, union([binary(), list(range(1,255))])),
  begin
    EncDecMsg = base64:decode(base64:encode(Msg)),
    case is_binary(Msg) of
      true  -> EncDecMsg =:= Msg;
      false -> EncDecMsg =:= list_to_binary(Msg)
    end
  end).
%% Using a user-defined simple type as a generator
-type bl() :: binary() | [1..255].

prop_enc_dec() ->
  ?FORALL(Msg, bl(),
    begin
      EncDecMsg = base64:decode(base64:encode(Msg)),
      case is_binary(Msg) of
        true  -> EncDecMsg =:= Msg;
        false -> EncDecMsg =:= list_to_binary(Msg)
      end
    end).

%% A lists delete implementation
-spec delete(T, list(T)) -> list(T).
delete(X, L) ->
    delete(X, L, []).

delete(_, [], Acc) ->
    lists:reverse(Acc);
delete(X, [X|Rest], Acc) ->
    lists:reverse(Acc) ++ Rest;
delete(X, [Y|Rest], Acc) ->
    delete(X, Rest, [Y|Acc]).

prop_delete() ->
    ?FORALL({X,L}, {integer(),list(integer())},
        not lists:member(X, delete(X, L)))).
41> c(simple_props).
{ok,simple_props}
42> proper:quickcheck(simple_props:prop_delete()).
......................!
Failed: After 42 test(s).
{12,[-36,-1,-2,7,19,-14,40,-6,-8,42,-8,12,12,-17,3]}

Shrinking ...(3 time(s))
{12,[12,12]}
false
PropEr integration with types

- type tree(T) :: 'leaf' | {'node',T,tree(T),tree(T)}.

%%% A tree delete implementation
-spec delete(T, tree(T)) -> tree(T).
delete(X, leaf) ->
    leaf;
delete(X, {node,X,L,R}) ->
    join(L, R);
delete(X, {node,Y,L,R}) ->
    {node,Y,delete(X,L),delete(X,R)}.

prop_delete() ->
    ?FORALL({X,L}, {integer(),tree(integer())},
            not member(X, delete(X, L))).

member(_, leaf) -> false;  % tree member function
member(X, {node,Y,L,R}) ->
    X =:= Y orelse member(X, L) orelse member(X, R).
What one would have to write in EQC

tree(G) ->
    ?SIZED(S, tree(S, G)).

tree(0, _) ->
    leaf;

tree(S, G) ->
    frequency([ 
        {1, tree(0, G)},
        {9, ?LAZY(?LETSHRINK([ 
            [L,R],
            [tree(S div 2, G),tree(S div 2, G)],
            {node,G,L,R}
        )))},
    ]).
What one has to write in PropEr

This slide intentionally left blank
-module(myspecs).

-export([[divide/2, filter/2, max/1]]).

-spec divide(integer(), integer()) -> integer().
divide(A, B) ->
    A div B.

-spec filter(fun((T) -> term()), [T]) -> [T].
filter(Fun, List) ->
    lists:filter(Fun, List).

-spec max([T]) -> T.
max(List) ->
    lists:max(List).
1> c(myspecs).
   {ok,myspecs}
2> proper:check_spec({myspecs,divide,2}).
   !
   Failed: After 1 test(s).
   An exception was raised: error:badarith.
   Stacktrace: [{myspecs,divide,2}].
   [0,0]

Shrinking (0 time(s))
   [0,0]
false

.... AFTER FIXING THE PROBLEMS ....
42> proper:check_specs(myspecs).

PropEr testing of specs
PropEr integration with remote types

• We want to test that `array:new/0` can handle any combination of options
• Why write a custom generator (which may rot)?
• We can use the remote type as a generator!

```
-type array_opt() :: 'fixed' | non_neg_integer()
[311x268]| {'default', term()}
[323x245]| {'fixed', boolean()}
[323x222]| {'size', non_neg_integer()}.

-type array_opts() :: array_opt() | [array_opt()].
```

```
-module(types).
#include_lib("proper/include/proper.hrl").

prop_new_array_opts() ->
  ?FORALL(Opts, array:array_opts(),
          array:is_array(array:new(Opts))).
```
PropEr testing of stateful systems

• PropEr can be used to test these as well
  - We simply have to define a callback for the PropEr `state` or `fsm` behavior

• What are these behaviors?
  - Libraries that can be used to test a system by generating and performing API calls to that system

• The callback module specifies a PropEr abstract model of the system under test
PropEr testing of stateful systems

- PropEr *statem* or *fsm* libraries
  - automatically generate test cases from the model and
  - execute them to test the real implementation against the model

- However, the test cases should be generated strictly *before* they are run
  - otherwise, they are not repeatable and we cannot shrink them
PropEr statem testing of pdict

Intention: test put/2, get/1, erase/1 operations

Test cases are sequences of symbolic API calls

command([]) ->
  {call, erlang, put, [key(), integer()]};
command(_State) ->
  oneof([{call, erlang, put, [key(), integer()]},
         {call, erlang, get, [key()]},
         {call, erlang, erase, [key()]})].

-define(KEYS, [a,b,c,d]).

key() ->
  elements(??KEYS).
PropEr commands

- We have put a rule: first generate, then execute
- What if we need to use the result of a previous call in a subsequent one?

Commands to the rescue!

- PropEr automatically binds the result of each symbolic call to a symbolic variable

```
[{set, {var,1}, {call, erlang, put, [a,42]}}],
{set, {var,2}, {call, erlang, erase, [a]}},
{set, {var,3}, {call, erlang, put, [b,{var,2}]}]}
```
The PropEr model states

• A model of the system's internal state (at least of the useful part of it!)
• We model the process dictionary as a property list

```
initial_state() -> [].

next_state(State, _Result, {call,erlang,put,[Key,Value]}) -> State ++ [{Key,Value}];
next_state(State, _Result, {call,erlang,erase,[Key]}) -> proplists:delete(Key, State);
next_state(State, _Result, {call,erlang,get,[_Key]}) -> State.
```
PropEr pre- and post-conditions

precondition(_, {call,erlang,put,[_Key,_Val]}) ->
    true;
precondition(State, {call,erlang,get,[Key]}) ->
    proplists:is_defined(Key, State);
precondition(State, {call,erlang,erase,[Key]}) ->
    proplists:is_defined(Key, State).

postcondition(State, {call,erlang,put,[Key,_]}, undefined) ->
    not proplists:is_defined(Key, State);
postcondition(State, {call,erlang,put,[Key,_Val]}, Old) ->
    {Key,Old} =:= proplists:lookup(Key, State);
postcondition(State, {call,erlang,get,[Key]}, Val) ->
    {Key,Val} =:= proplists:lookup(Key, State);
postcondition(State, {call,erlang,erase,[Key]}, Val) ->
    {Key,Val} =:= proplists:lookup(Key, State);
postcondition(_, _, _) ->
    false.
A PropEr property for pdict...

```erlang
prop_pdict() ->
    ?FORALL(Cmds, commands(?MODULE),
        begin
            {Hist, State, Res} = run_commands(?MODULE, Cmds),
            clean_up(),
            ?WHENFAIL(io:format("H: ~w
St: ~w
Res: ~w
", [Hist, State, Res]),
                Res =:= ok)
        end).

clean_up() ->
    lists:foreach(fun(Key) -> erlang:erase(Key) end, ?KEYS).
```

- **Random symbolic command sequence generator**
- **Evaluate the command sequence**
- The PropEr thing to do...
- Tests pass when no exception is raised and all postconditions are true
\[42> \text{proper:quickcheck(pdict_statem:prop_pdict()).}
\]

............!

Failed: After 13 test(s).

\[
\{\text{set},\text{var,1}\},\{\text{call,erlang,put,}[a,-12]\}\},\{\text{set,}{\text{var,2}},\{\text{call,erlang,put,}[a,-18]\}\},
\{\text{set,}{\text{var,3}},\{\text{call,erlang,put,}[c,4]\}\},\{\text{set,}{\text{var,4}},\{\text{call,erlang,put,}[b,6]\}\},
\{\text{set,}{\text{var,5}},\{\text{call,erlang,erase,}[b]\}\},\{\text{set,}{\text{var,6}},\{\text{call,erlang,put,}[d,39]\}\},
\{\text{set,}{\text{var,7}},\{\text{call,erlang,get,}[a]\}\}
\]

H: \[
\{[],\text{undefined}\},\{\{a,-12\},-12\},\{\{a,-12\},\{a,-18\},\text{undefined}\},\{\{a,-12\},\{a,-18\},\{c,4\}\},\text{undefined}\},\{\{a,-12\},\{a,-18\},\{c,4\},\{b,6\}\},6\},\{\{a,-12\},\{a,-18\},
\{c,4\}\},\text{undefined}\},\{\{a,-12\},\{a,-18\},\{c,4\},\{d,39\}\},-18\}
\]

St: \[
\{a,-12\},\{a,-18\},\{c,4\},\{d,39\}\]

Res: \{\text{postcondition,false}\}

Shrinking ....(4 time(s))

\[
\{\text{set,}{\text{var,1}},\{\text{call,erlang,put,}[a,-12]\}\},
\{\text{set,}{\text{var,2}},\{\text{call,erlang,put,}[a,-18]\}\},
\{\text{set,}{\text{var,7}},\{\text{call,erlang,get,}[a]\}\}
\]

H: \[
\{[],\text{undefined}\},\{\{a,-12\},-12\},\{\{a,-12\},\{a,-18\},-18\}\}
\]

St: \[
\{a,-12\},\{a,-18\}\]

Res: \{\text{postcondition,false}\}

false
PropEr already used out there!

```erlang
%% compile with
%% erl -pz ebin --make
%% start test with
%% erl -pz ebin -pz test
%% proper:module(ec_dictionary_proper).

-module(ec_dictionary_proper).
-compile(export_all).
-included_lib("proper/include/proper.hrl").

% Properties
%

prop_size_increases_with_new_key() ->
?FORALL({Dict,K}, {my_dict(),integer()},
begin
  Size = ec_dictionary:size(Dict),
  case ec_dictionary:has_key(K,Dict) of
    true ->
      Size == ec_dictionary:size(ec_dictionary:add(K,0,Dict));
    false ->
      (Size + 1) == ec_dictionary:size(ec_dictionary:add(K,0,Dict))
  end
end).
```
“I ran PropEr using statem on a real example which I already had for EQC. It was just to switch include file, recompile and run!”
When I use ordered_set ets over gb_trees it has more than once been due to the fact that you can do wonderful stuff with first, next, prev and last - and gb_trees doesn't have them.

I've made a stab at implementing these functions for the gb_trees data structure, together with a quickcheck spec to verify that they work as expected (you can use eqc mini to run the tests). I think they are reasonably efficient, but perhaps someone can think of a way to optimize them?

Have at it, and pls use the spec to verify that you didn't break them (recalling that an incorrect program can be made arbitrarily fast)
```erlang
-module(gb1).
-compile(export_all).

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

gb_next(K, {_, T}) ->
gb_next_1(K, T).

gb_next_1(K, {K1, _, Smaller, Bigger}) when K < K1 ->
  case gb_next_1(K, Smaller) of
    none ->
      case gb_next_1(K, Bigger) of
        none ->
          {value, K1};
        {value, K2} ->
          {value, erlang:min(K1, K2)}
        end;
        {value, _} = Res ->
          Res
      end;
      {value, _} = Res ->
        Res
    end;
  gb_next_1(K, Bigger);

gb_next_1(K, {_, _, _, Bigger}) ->
  case Bigger of
    nil ->
      none;
    {K1, _, Smaller, _} ->
      case gb_next_1(K, Smaller) of
        none ->
          {value, K1};
        {value, _} = Res ->
          Res
      end;
    end;
  gb_next_1(_, nil) ->
    none.

gb_prev(K, {_, T}) ->
gb_prev_1(K, T).

gb_prev_1(K, {K1, _, Smaller, Bigger}) when K > K1 ->
  case gb_prev_1(K, Bigger) of
    none ->
      case gb_prev_1(K, Smaller) of
        none ->
          {value, K1};
        {value, K2} ->
          {value, erlang:max(K1, K2)}
        end;
        {value, _} = Res ->
          Res
      end;
      {value, _} = Res ->
        Res
    end;
  gb_prev_1(K, Smaller);

gb_prev_1(K, {_, _, Smaller, _}) when K < K1 ->
  gb_prev_1(K, Smaller);

gb_prev_1(K, {_, _, Smaller, _}) ->
  case Smaller of
    nil ->
      none;
    {K1, _, _, Bigger} ->
      case gb_prev_1(K, Bigger) of
        none ->
          {value, K1};
        {value, _} = Res ->
          Res
      end;
    end;
  gb_prev_1(_, nil) ->
    none.
```

first({_, T}) ->
  first_1(T).

first_1({K,_,nil,_}) ->
  {value, K};
first_1({_,_,Smaller,_,}) ->
  first_1(Smaller);
first_1(nil) ->
  none.

last({_, T}) ->
  last_1(T).

last_1({K,_,_,nil}) ->
  {value, K};
last_1({_,_,_,Bigger}) ->
  last_1(Bigger);
last_1(nil) ->
  none.

prop_first() ->
  ?FORALL(L, list(int()),
  begin
  {T, Sorted} = make_tree(L),
  case first(T) of
    none -> Sorted == [];
    {value,X} -> X == hd(Sorted)
  end
  end).

prop_last() ->
  ?FORALL(L, list(int()),
  begin
  {T, Sorted} = make_tree(L),
  case last(T) of
    none -> Sorted == [];
    {value,X} -> X == lists:last(Sorted)
  end
  end).

prop_prev() ->
  ?FORALL(L, list(int()),
  begin
  {T, Sorted} = make_tree(L),
  ok == all_prev(lists:reverse(Sorted), T)
  end).

prop_next() ->
  ?FORALL(L, list(int()),
  begin
  {T, Sorted} = make_tree(L),
  ok == all_prev(lists:reverse(Sorted), T)
  end).

make_tree(L) ->
  T = lists:foldl(fun(X,T) ->
    gb_trees:enter(X,1,T)
  end, gb_trees:empty(), L),
  Sorted = [K || {K,_} <- gb_trees:to_list(T)],
  {T, Sorted}.
-module(gb1).
-compile(export_all).

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

gb_next(K, {_, T}) ->
    gb_next_1(K, T).
-module(gb1).
-export([gb_next/2, gb_prev/2, first/1, last/1]).

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

-spec gb_next(term(), gb_tree()) ->
   'none' | {'value', term()}.

gb_next(K, {_, T}) ->
    gb_next_1(K, T).
-module(gb1).
-export([gb_next/2, gb_prev/2, first/1, last/1]).

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

-spec gb_next(term(), gb_tree()) ->
   'none' | {'value', term()}.

gb_next(K, {_, T}) ->
   gb_next_1(K, T).
prop_next() ->
  ?FORALL(L, list(int())),
  begin
    {T, Sorted} = make_tree(L),
    ok == all_prev(lists:reverse(Sorted), T)
  end).

make_tree(L) ->
  T = lists:foldl(fun(X,T) ->
    gb_trees:enter(X,1,T)
  end, gb_trees:empty(), L),
  Sorted = [K || {K,_} <- gb_trees:to_list(T)],
  {T, Sorted}.
Comments from a guru

From: John Hughes on erlang-questions
Date: 16/3/2011, 20:58

Nice!

Slight typo: you tested prev twice... your prop_next actually tested prev, it's a copy-and-paste of prop_prev without the renaming to next!

One drawback of your approach is that you only test next and prev on gb_trees constructed using empty and enter. Conceivably the other functions could create gb_trees with a different structure that you might fail on.

Here's some code that uses ALL of the constructors to build the test data (no bugs found though!).
%% gb_tree constructors

gb() ->
  ?SIZED(Size,
    frequency([[1, {call, gb_trees, empty, []}],
                [1, {call, gb_trees, from_orddict, [orddict()]}],
                [Size, ?LAZY(compound_gb())]])).
compound_gb() ->
  ?LETSHRINK([GB], [gb()]),
  oneof([\{call,gb_trees,\text{Fun},\text{Args}++[GB]\}
         || \{Fun|\text{Args}\} <-
                lists:map(fun \{call,erlang,element,\[3,\{call,gb_trees,
                                                                 \text{take_smallest},[GB]\}\}\})),
         \{call,erlang,element,
         \[3,\{call,gb_trees,
                 \text{take_largest},[GB]\}\}\})).
Even more code from a guru

From: John Hughes on erlang-questions

```erlang
gb_constructors() ->
    [{balance},
     {delete,key()},
     {delete_any,key()},
     {enter,key(),val()},
     {insert,key(),val()},
     {update,key(),val()}].

dict() ->
    nat().

dict() ->
    int().

gb_constructors() ->
    ?LET(List, list([{key(),val()}]),
        orddict:from_list(List)).
```
The PropEr solution

Why not just write this?

```prolog
prop_next() ->
    ?FORALL(T, gb_tree(key(), val()),
        ok == all_next(gb_trees:keys(T), T)).
```

Compare with:

```prolog
prop_next() ->
    gb_constructors() ->
        ?FORALL(I, gb_constructors(),
            {balance},
            {delete, key()},
            {delete_any, key()},
            {enter, key(), val()},
            {insert, key(), val()},
            {update, key(), val()}).

key() ->
    nat().

val() ->
    int().

orddict() ->
    ?LET(L, list([{key(), val()}]),
        orddict:from_list(L)).
```

```prolog
% gb_tree constructors
gb() ->
    ?SIZED(Size,
        frequency([[1, {call, gb_trees, empty, []}]],
            {1, {call, gb_trees, from_orddict, [orddict()]}},
            ^'ze, ?LAZY(compound_gb()))).

compound_gb() ->
    ?LETSHRINK([GB], [gb()],
        oneof([call, gb_trees, Fun, Args++[GB]],
            Fun|Args <-
            lists:map(fun tuple_to_list/1, gb_constructors())
        ]++)
        ++
        [{call, erlang, element, [3, {call, gb_trees, take_smallest, [GB]}]},
            {call, gb_trees, take_largest, [GB]}]).

gb_constructors() ->
    [{balance},
        {delete, key()},
        {delete_any, key()},
        {enter, key(), val()},
        {insert, key(), val()},
        {update, key(), val()}].
```
Is this really all?

Yes, but we recommend that you also write:

```erlang
-type key() :: integer().
-type val() :: integer().
```

Do I **really** need to write these type declarations?

Well, no. You could write the property as:

```erlang
prop_next() ->
    ?FORALL(T, gb_tree(integer(), integer()),
        ok == all_next(gb_trees:keys(T), T)).
```
I do not believe this...

OK, let’s do a demo...
Thanks from the PropEr developers!