

# A PropEr Talk

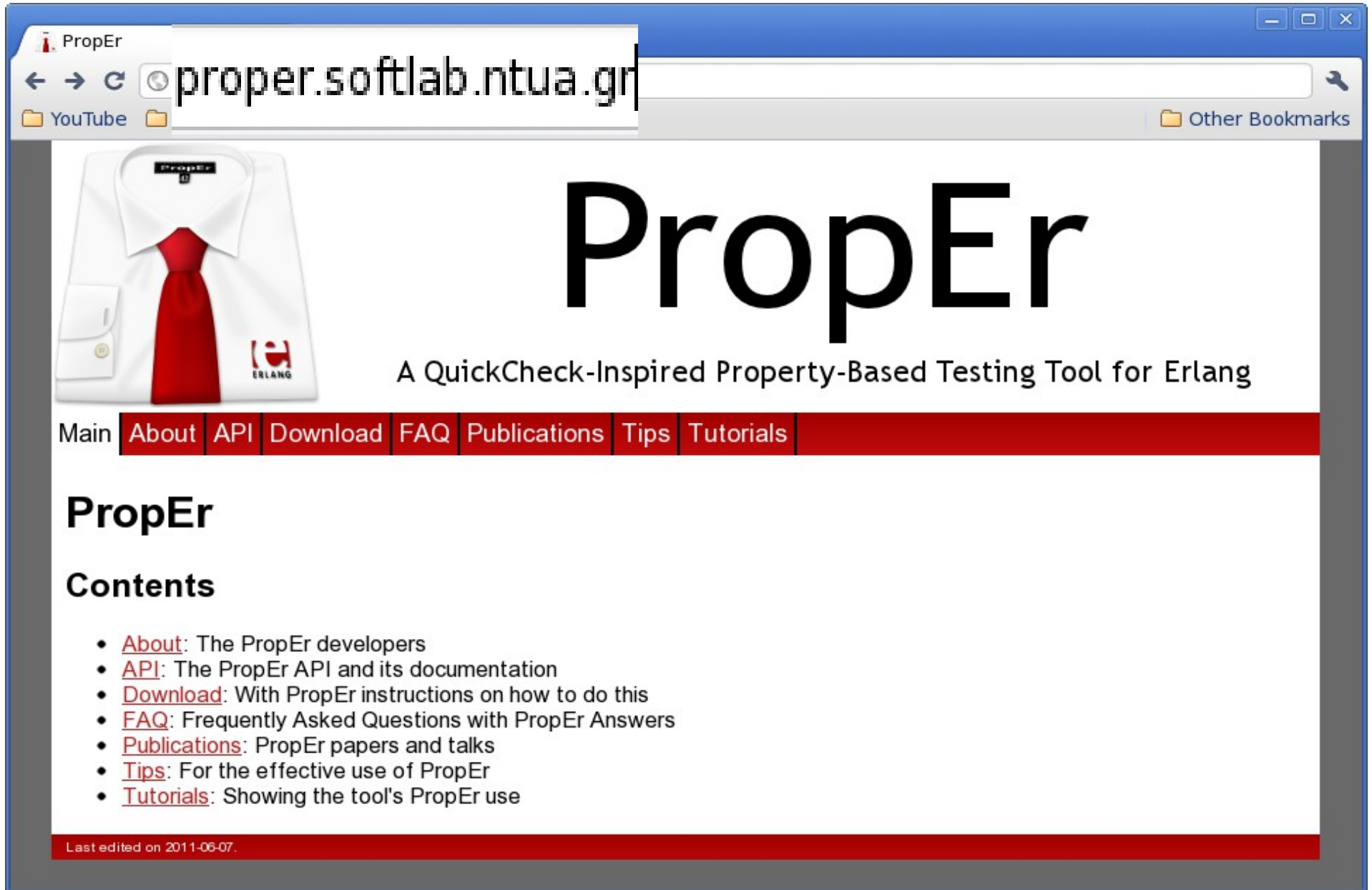
**Kostis Sagonas**



With PropEr help by  
**Manolis Papadakis**  
**Eirini Arvaniti**



# A PropEr announcement



The screenshot shows a web browser window with the address bar containing `proper.softlab.ntua.gr`. The page features a white dress shirt and a red tie on the left, with the Erlang logo below it. The main heading is "PropEr" in a large, bold, black font. Below the heading is the subtitle "A QuickCheck-Inspired Property-Based Testing Tool for Erlang". A navigation bar contains links for "Main", "About", "API", "Download", "FAQ", "Publications", "Tips", and "Tutorials". The "Contents" section lists several items with red underlined links: "About", "API", "Download", "FAQ", "Publications", "Tips", and "Tutorials". At the bottom left, a footer note states "Last edited on 2011-06-07."

PropEr

A QuickCheck-Inspired Property-Based Testing Tool for Erlang

Main [About](#) [API](#) [Download](#) [FAQ](#) [Publications](#) [Tips](#) [Tutorials](#)

## PropEr

### Contents

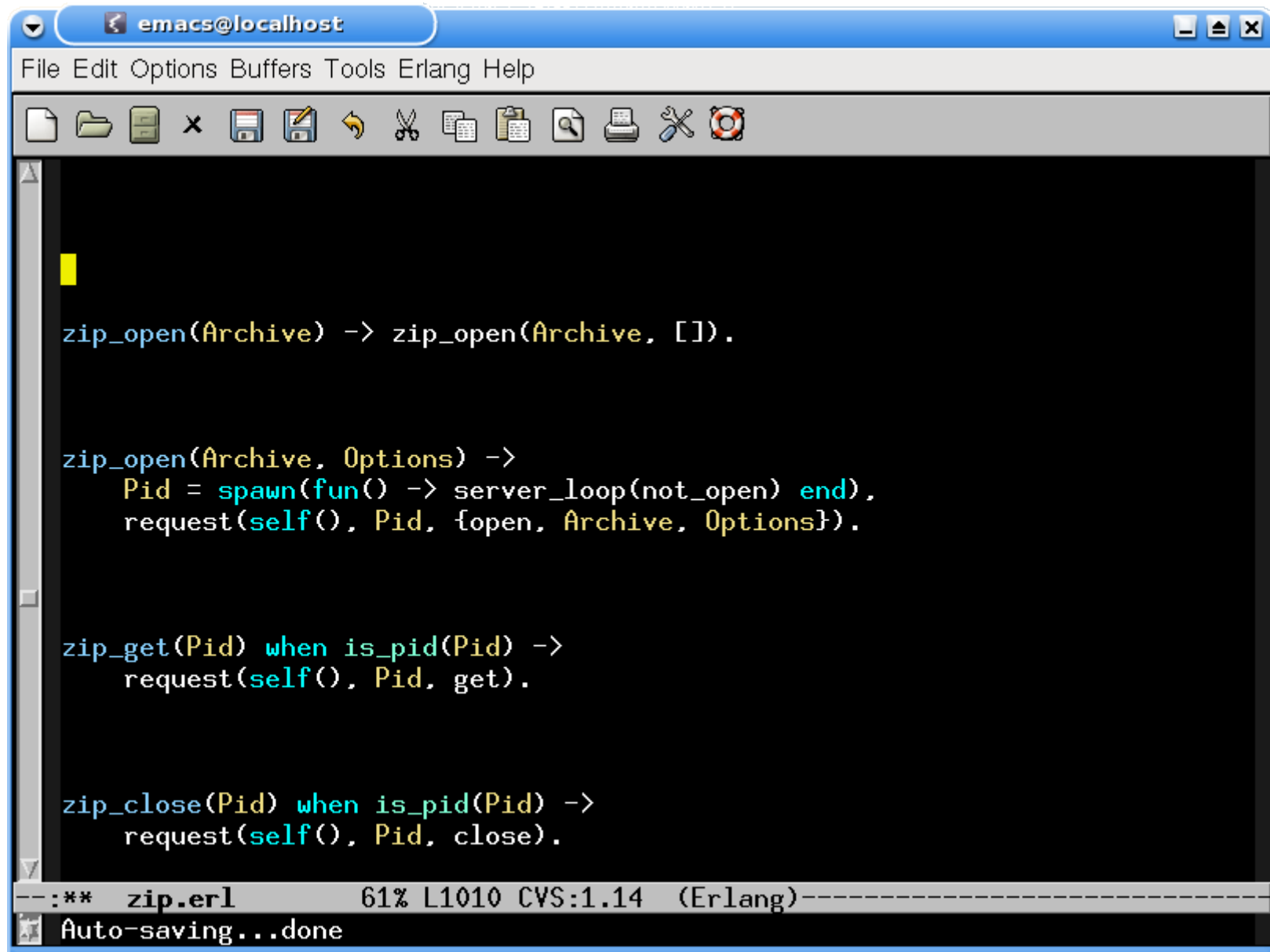
- [About](#): The PropEr developers
- [API](#): The PropEr API and its documentation
- [Download](#): With PropEr instructions on how to do this
- [FAQ](#): Frequently Asked Questions with PropEr Answers
- [Publications](#): PropEr papers and talks
- [Tips](#): For the effective use of PropEr
- [Tutorials](#):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

Last edited on 2011-06-07.

# A PropEr question

Why did you create PropEr?

# How Erlang modules used to look



The image shows a screenshot of an Emacs editor window titled "emacs@localhost". The window contains Erlang code for a module named "zip". The code is as follows:

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

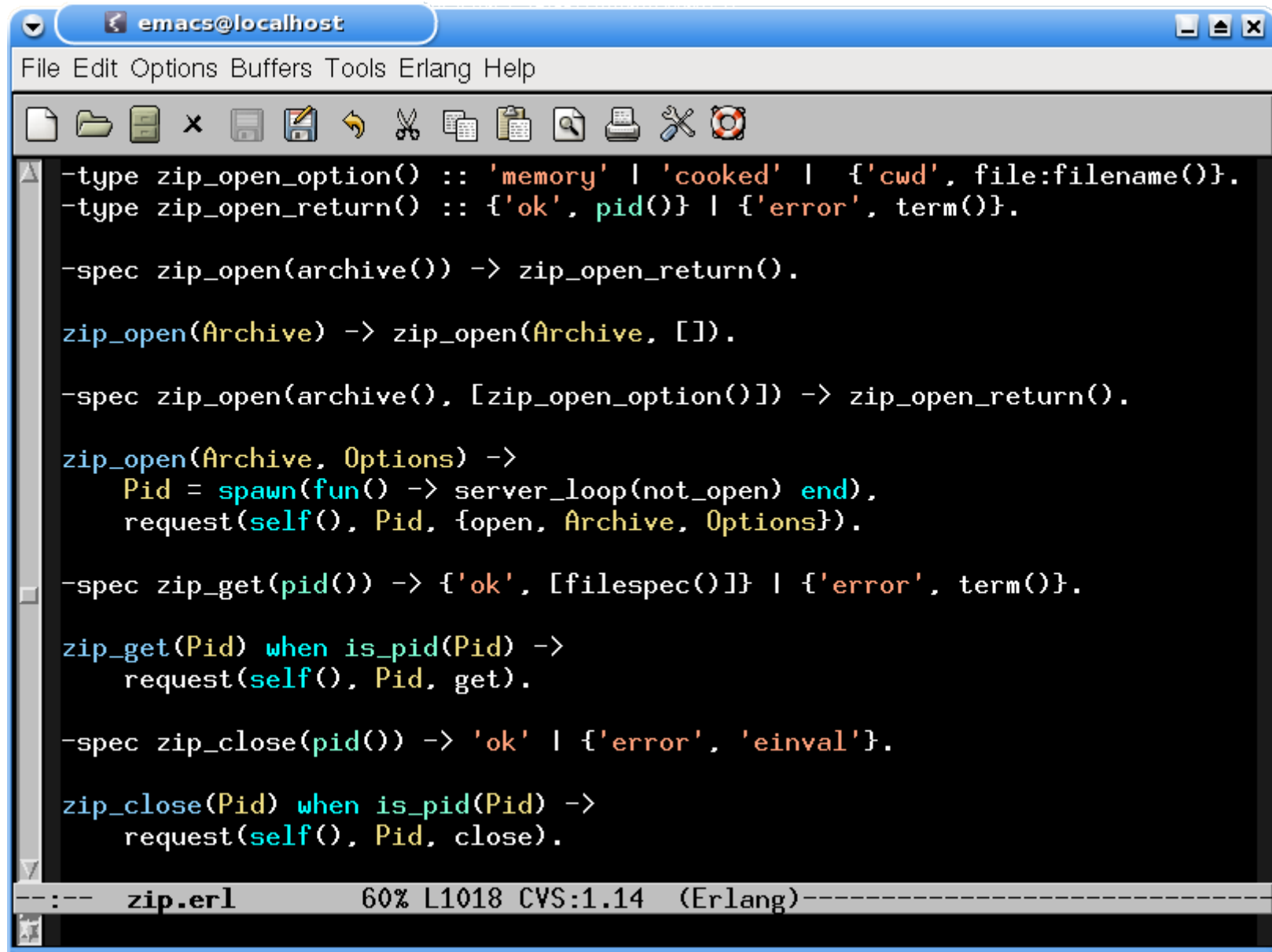
zip_open(Archive, Options) ->
  Pid = spawn(fun() -> server_loop(not_open) end),
  request(self(), Pid, {open, Archive, Options}).

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

zip_close(Pid) when is_pid(Pid) ->
  request(self(), Pid, close).
```

At the bottom of the window, the status bar shows: "--:\*\* zip.erl 61% L1010 CVS:1.14 (Erlang)-----" and "Auto-saving...done".

# How modern Erlang modules look



The image shows a screenshot of an Emacs editor window titled 'emacs@localhost'. The window displays Erlang code for a module named 'zip'. The code includes type definitions, specifications, and function definitions. The code is as follows:

```
-type zip_open_option() :: 'memory' | 'cooked' | {'cwd', file:filename()}.
-type zip_open_return() :: {'ok', pid()} | {'error', term()}.

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

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

-spec zip_open(archive(), [zip_open_option()]) -> zip_open_return().

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

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

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

-spec zip_close(pid()) -> 'ok' | {'error', 'EINVAL'}.

zip_close(Pid) when is_pid(Pid) ->
    request(self(), Pid, close).
```

The status bar at the bottom of the window shows: `--:-- zip.erl 60% L1018 CVS:1.14 (Erlang)-----`

# A PropEr start...



[www.phdcomics.com](http://www.phdcomics.com)

# PropEr progress four months later



WWW.PHDCOMICS.COM

# 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`, `?IMPLIES`, `?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!



# Testing simple properties (1)

```
-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
```

# Testing simple properties (2)

```
%% Testing the base64 module:
%%   encode should be symmetric to decode:

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) .
```

# PropEr integration with simple types

```
%% 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) .
```

# PropEr shrinking

```
%% 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))).
```

# PropEr shrinking

```
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)}.
```

```
join(leaf, T) -> T;  
join({node, X, L, R}, T) ->  
  {node, X, join(L, R), T}.
```

```
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



# PropEr testing of specs

```
-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).
```

# PropEr testing of specs

```
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 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()
                    | {'default', term()}
                    | {'fixed', boolean()}
                    | {'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 `staterm` 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 stateful 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...

random symbolic  
command sequence  
generator

evaluate the command  
sequence

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

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

the PropEr  
thing to do...

tests pass when  
no exception is  
raised and all  
postconditions are true

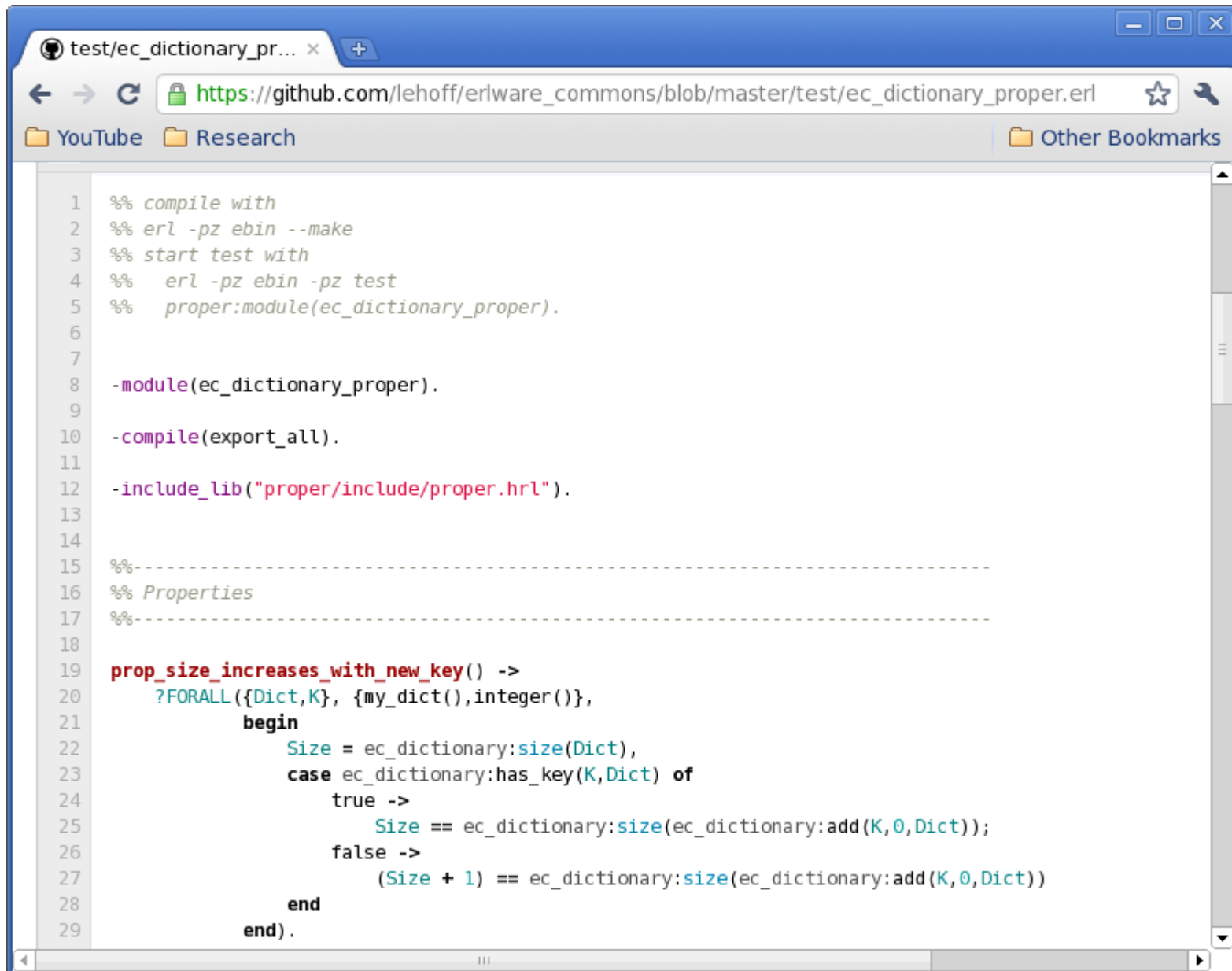
# ...with a PropEr failure

```
42> proper:quickcheck(pdect_statem:prop_pdect()).
.....!
Failed: After 13 test(s).
[{set,{var,1},{call,erlang,put,[a,-12]}},{set,{var,2},{call,erlang,put,[a,-18]}},
 {set,{var,3},{call,erlang,put,[c,4]}},{set,{var,4},{call,erlang,put,[b,6]}},
 {set,{var,5},{call,erlang,erase,[b]}},{set,{var,6},{call,erlang,put,[d,39]}},
 {set,{var,7},{call,erlang,get,[a]}}]
H: [[[],undefined],[[a,-12],-12],[[a,-12],[a,-18],undefined],[[a,-12],[a,-
18],[c,4],undefined],[[a,-12],[a,-18],[c,4],[b,6]],6],[[a,-12],[a,-18],
[c,4],undefined],[[a,-12],[a,-18],[c,4],[d,39]],-18]]
St: [[a,-12],[a,-18],[c,4],[d,39]]
Res: {postcondition,false}
```

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

```
[{set,{var,1},{call,erlang,put,[a,-12]}},
 {set,{var,2},{call,erlang,put,[a,-18]}},
 {set,{var,7},{call,erlang,get,[a]}}]
H: [[[],undefined],[[a,-12],-12],[[a,-12],[a,-18]],-18]]
St: [[a,-12],[a,-18]]
Res: {postcondition,false}
false
```

# PropEr already used out there!



```
1  %% compile with
2  %% erl -pz ebin --make
3  %% start test with
4  %%   erl -pz ebin -pz test
5  %%   proper:module(ec_dictionary_proper).
6
7
8  -module(ec_dictionary_proper).
9
10 -compile(export_all).
11
12 -include_lib("proper/include/proper.hrl").
13
14
15 %%-----
16 %% Properties
17 %%-----
18
19 prop_size_increases_with_new_key() ->
20   ?FORALL({Dict,K}, {my_dict(),integer()}),
21   begin
22     Size = ec_dictionary:size(Dict),
23     case ec_dictionary:has_key(K,Dict) of
24     true ->
25       Size == ec_dictionary:size(ec_dictionary:add(K,0,Dict));
26     false ->
27       (Size + 1) == ec_dictionary:size(ec_dictionary:add(K,0,Dict))
28     end
29   end).
```

# Quote from a PropEr user

“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!”

# Property-based testing by experts

From: **Ulf Wiger** on **erlang-questions**

Date: 16/3/2011, 18:13

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)

# Code from Ulf Wiger

```
-module (gb1).
-compile (export_all).

-include_lib ("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;
gb_next_1(K, {K1, _, _, Bigger}) when K > K1 ->
    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;
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;
gb_prev_1(K, {K1, _, 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;
end;
gb_prev_1(_, nil) ->
    none.
```

# More code from Ulf Wiger

```
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.
```

```
all_next([X], T) ->
  {X,none} = {X,gb_next(X, T)},
  ok;
all_next([A,B|Rest], T) ->
  {A,{value,B}} = {A,gb_next(A, T)},
  all_next([B|Rest], T);
all_next([], _) ->
  ok.
```

```
all_prev([X], T) ->
  {X,none} = {X,gb_prev(X, T)},
  ok;
all_prev([A,B|Rest], T) ->
  {A,{value,B}} = {A,gb_prev(A, T)},
  all_prev([B|Rest], T);
all_prev([], _) ->
  ok.
```

```
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}.
```

```
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).
```

```
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).
```



# A closer look at the code

```
-module (gb1) .  
-compile (export_all) .  
  
-include_lib ("eqc/include/eqc.hrl") .  
  
gb_next (K, {_, T}) ->  
    gb_next_1 (K, T) .
```

# A better version

```
-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) .
```

# A PropEr version

```
-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).
```

# A closer look at the properties

```
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!).

# Code from a guru

From: **John Hughes** on **erlang-questions**

```
%% gb_tree constructors

gb() ->
    ?SIZED(Size,
        frequency([
            {1, {call, gb_trees, empty, []}},
            {1, {call, gb_trees,
                from_orddict, [orddict()]}},
            {Size, ?LAZY(compound_gb())}
        ])).
```

# More code from a guru

From: **John Hughes** on **erlang-questions**

```
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,erlang,element,
            [3,{call,gb_trees,
                take_largest,[GB]}}]}])).
```

# Even more code from a guru

From: **John Hughes** on **erlang-questions**

```
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)).
```



# The PropEr solution

Why not just write this?

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

Compare with:

```
prop_next()
  ?FORALL(I,
    k
    {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)).
```

```
%% gb_tree constructors
```

```
gb() ->
  ?SIZED(Size,
    frequency([
      {1, {call, gb_trees, empty, []}},
      {1, {call, gb_trees, from_orddict, [orddict()]}},
      {Size, ?LAZY(compound_gb())}])).
```

```
1, gb_trees, Fun, Args++[GB]}
Fun|Args] <-
  lists:map(fun tuple_to_list/1, gb_constructors())]
```

```
ll, erlang, element,
, {call, gb_trees, take_smallest, [GB]}}],
ll, erlang, element,
, {call, gb_trees, take_largest, [GB]}}]}}).
```

# Is this really all?

Yes, but we recommend that you also write:

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

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

Well, no. You could write the property as:

```
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!

