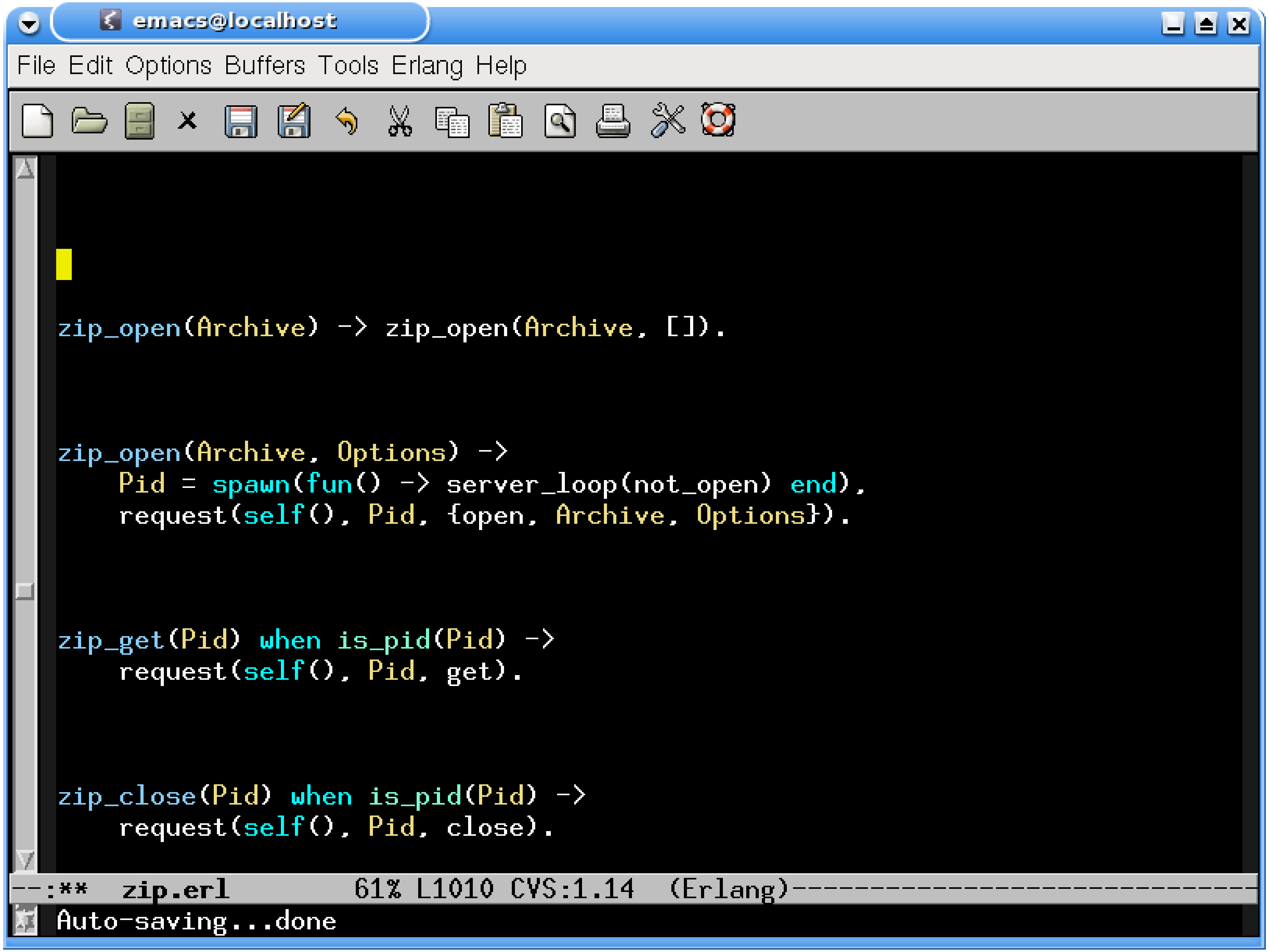
A PropEr Integration of Types and Function Specifications with Property-based Testing

Manolis Papadakis Kostis Sagonas

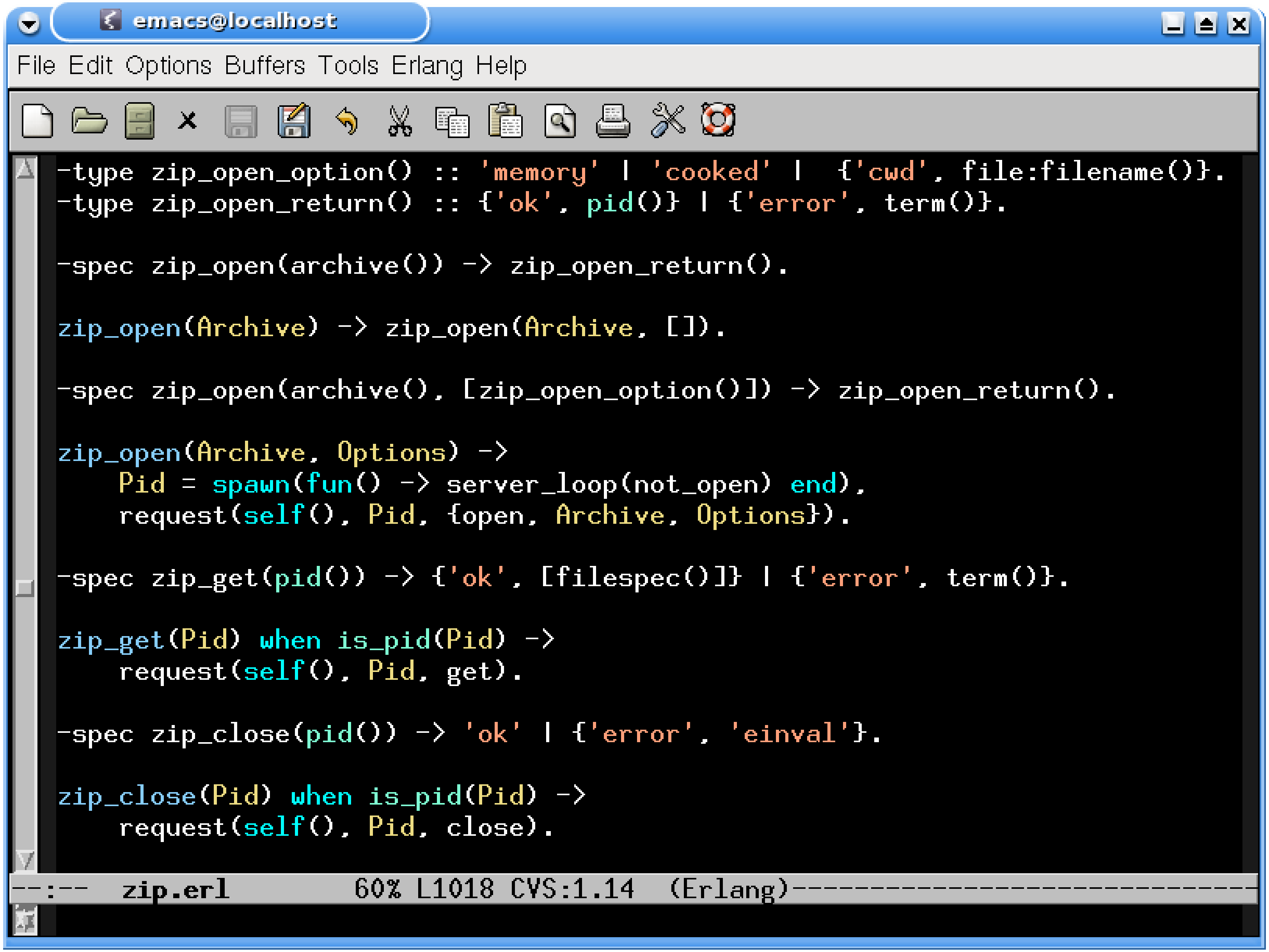
How Erlang modules used to look



```
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).
```

```
--:**  zip.erl         61% L1010 CVS:1.14   (Erlang)--------------------------------
Auto-saving...done
```

How modern Erlang modules look


emacs@localhost
File Edit Options Buffers Tools Erlang Help
--:-- zip.erl   60% L1018 CVS:1.14   (Erlang)------------------------------

```erlang
-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).
```

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
- Full integration with the language of types and function specifications
  - 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 lists:member(X, delete(X, L))).
```

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

Integration with recursive types

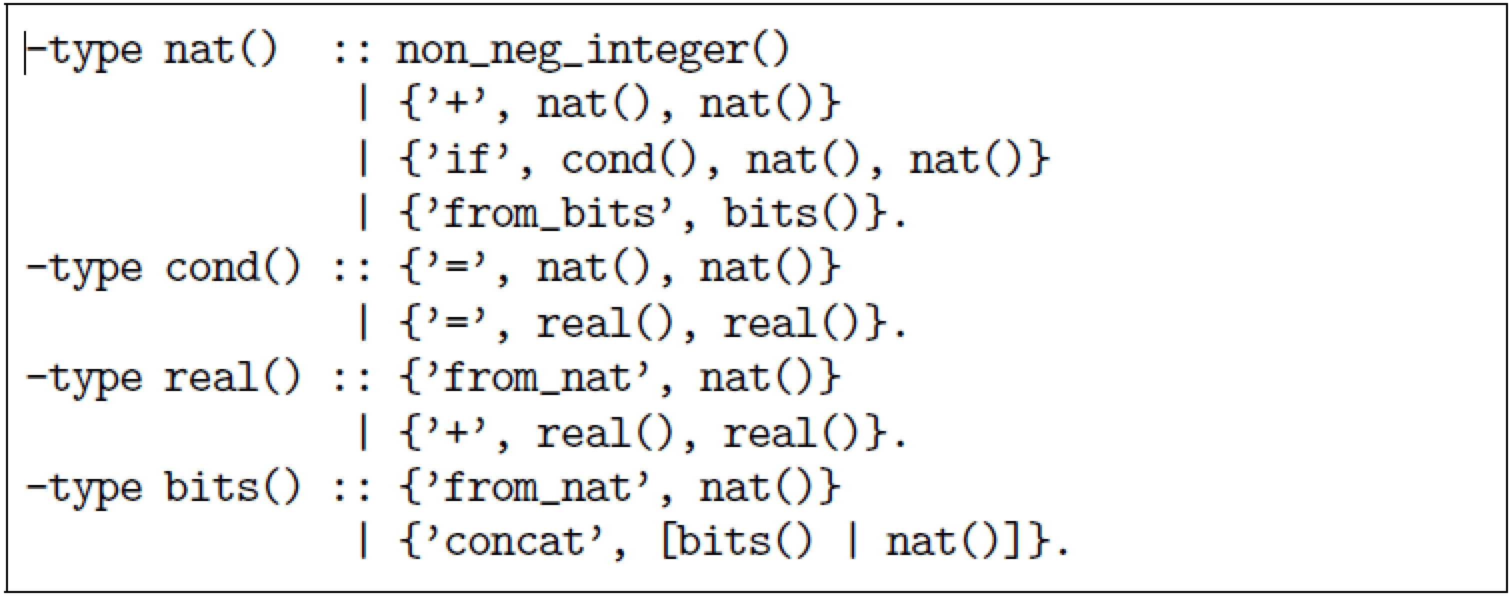
```
41> c(mytrees).
{ok,mytrees}
42> proper:quickcheck(mytrees:prop_delete()).
.......................!
Failed: After 24 test(s).
{6,{node,19,{node,-19,leaf,leaf},
            {node,6,leaf,{node,6,leaf,leaf}}}}

Shrinking .(1 time(s))
{6,{node,6,{node,6,leaf,leaf}}}
false
```

Generators from recursive types

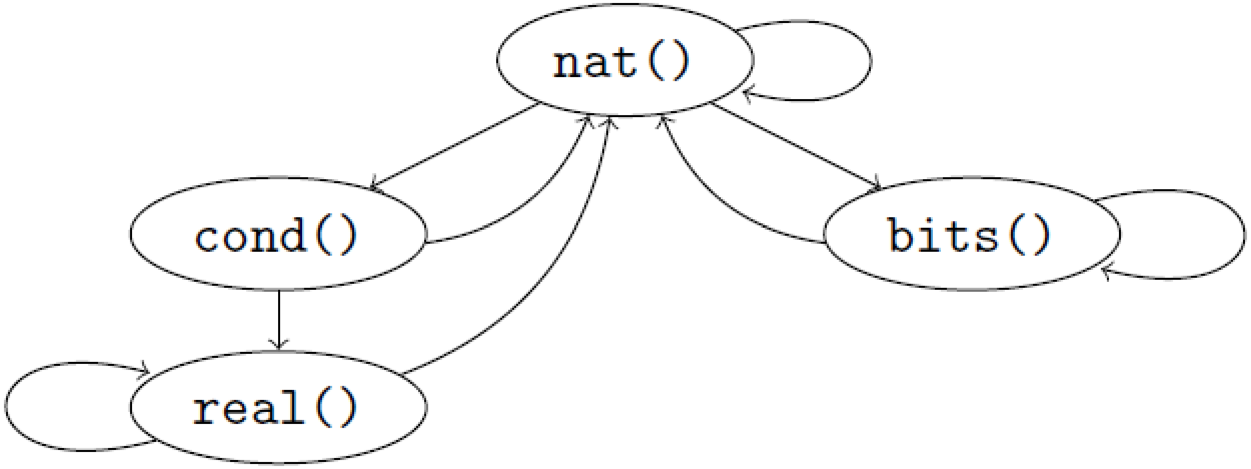
Takes place, roughly, in the following steps

- Detect recursion
- Inline (non-recursive) type definitions
- Normalize by pushing unions to the top level
- Find base cases
- Prepare the recursive calls
- Determine shrinking behavior
- Compose a generator

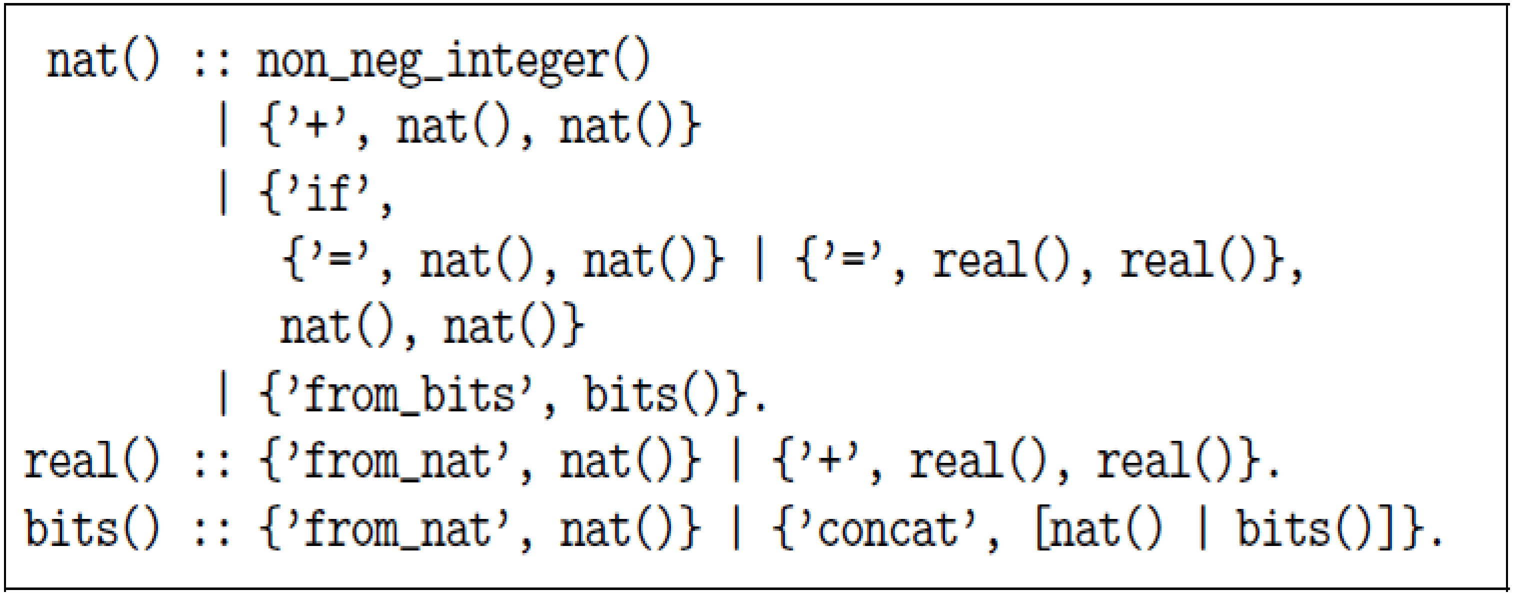
Example: detecting recursion



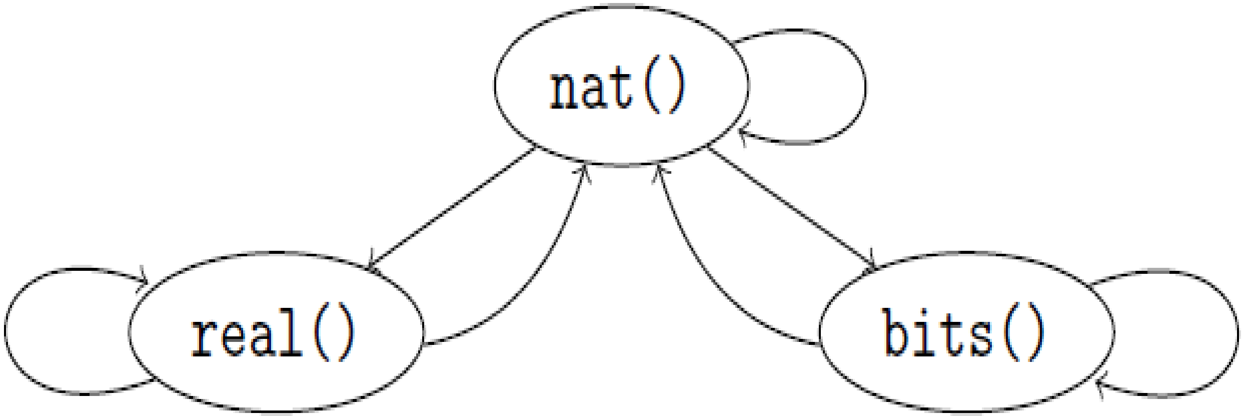
```
-type nat()  :: non_neg_integer()
              | {'+', nat(), nat()}
              | {'if', cond(), nat(), nat()}
              | {'from_bits', bits()}.
-type cond() :: {'=', nat(), nat()}
              | {'=', real(), real()}.
-type real() :: {'from_nat', nat()}
              | {'+', real(), real()}.
-type bits() :: {'from_nat', nat()}
              | {'concat', [bits() | nat()]}.
```



Example: after inlining

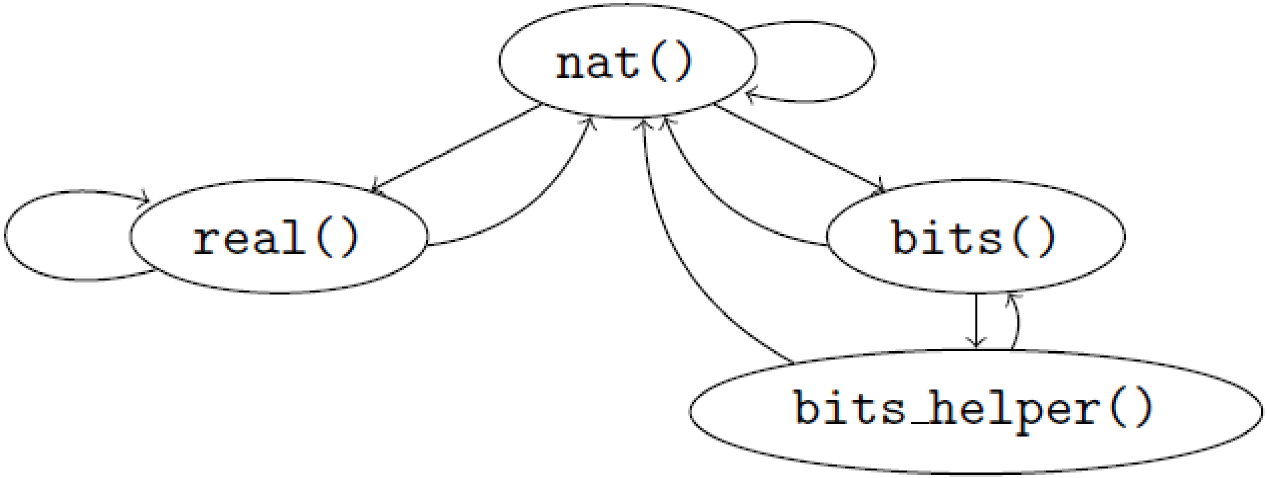


```
 nat() :: non_neg_integer()
        | {’+’, nat(), nat()}
        | {’if’,
           {’=’, nat(), nat()} | {’=’, real(), real()},
           nat(), nat()}
        | {’from_bits’, bits()}.
real() :: {’from_nat’, nat()} | {’+’, real(), real()}.
bits() :: {’from_nat’, nat()} | {’concat’, [nat() | bits()]}.
```

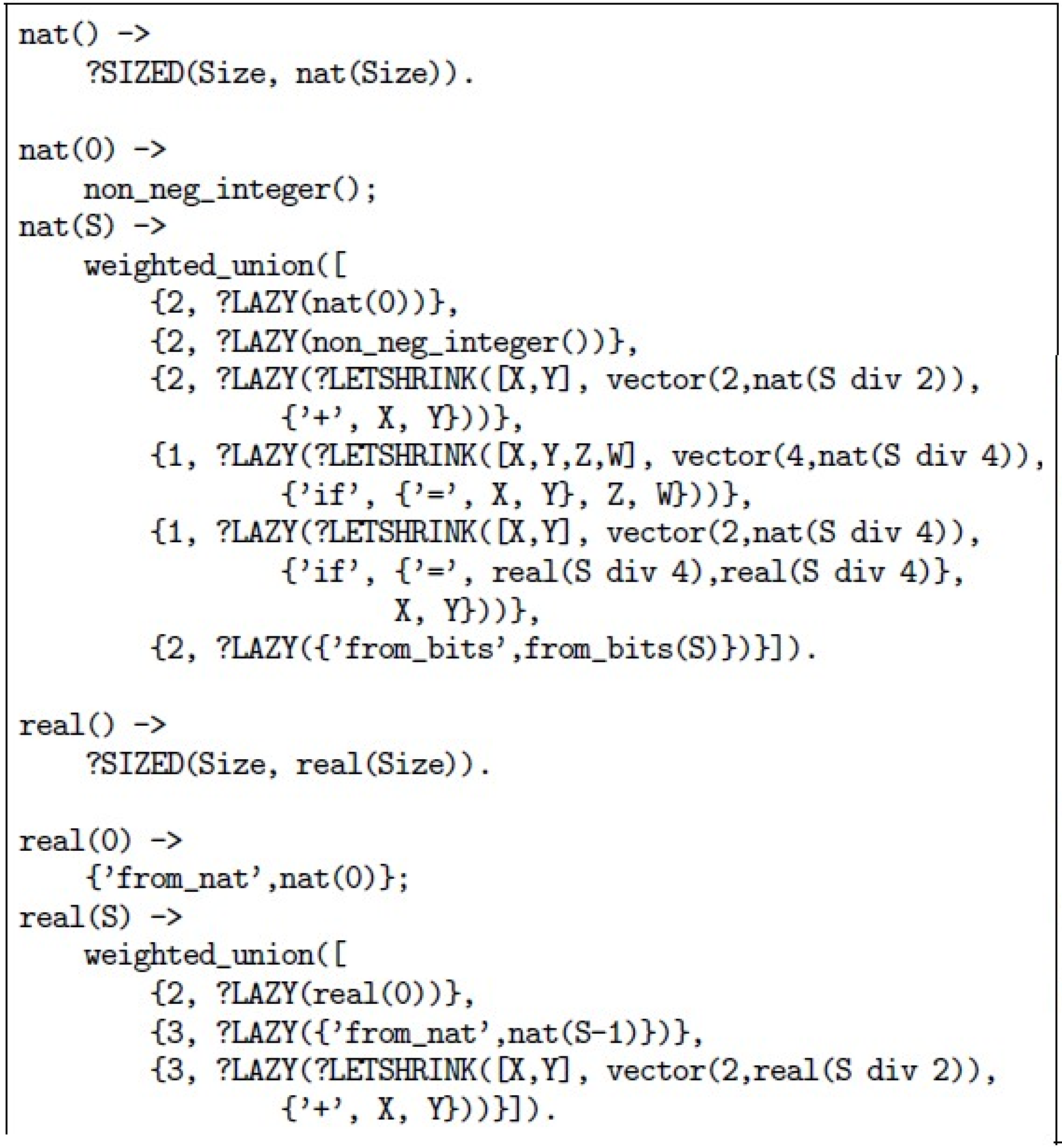


Example: after normalization

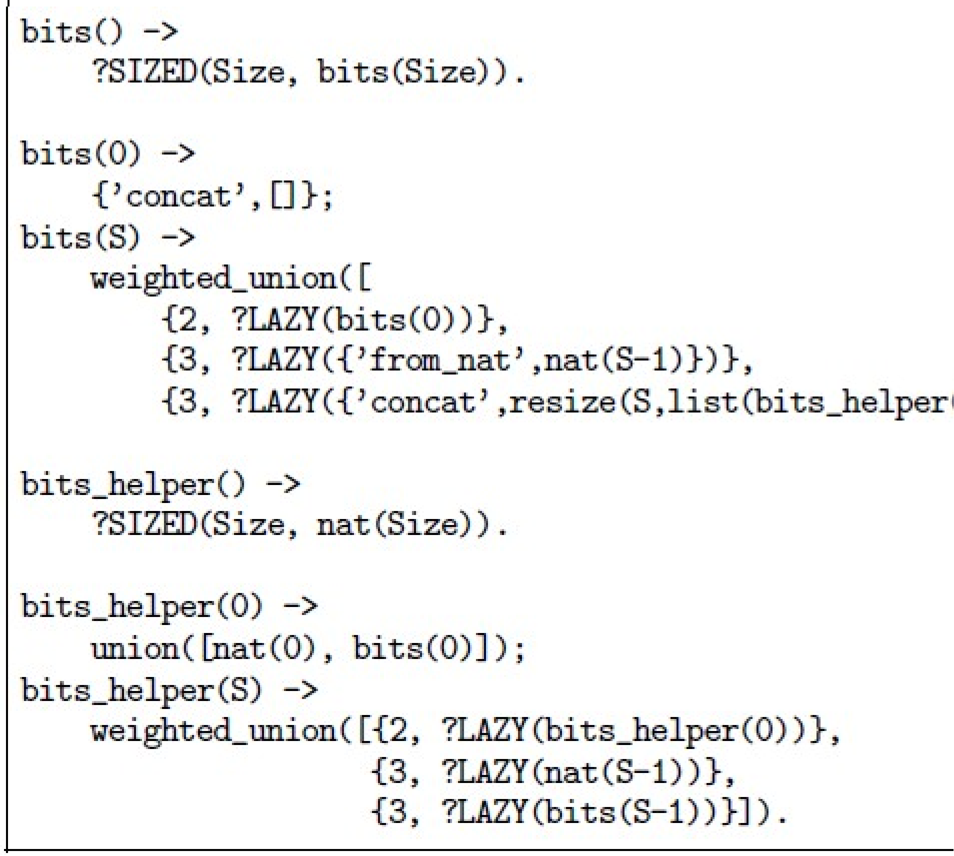
```
nat()  :: non_neg_integer()_2
        | {'+', nat(), nat()}_2
        | {'if', {'=',nat(),nat()}, nat(), nat()}_1
        | {'if', {'=',real(),real()}, nat(), nat()}_1
        | {'from_bits', bits()}_2.
real() :: {'from_nat', nat()}_1 | {'+', real(), real()}_1.
bits() :: {'from_nat', nat()}_1 | {'concat', [bits_helper()]}_1.
bits_helper() :: nat()_1 | bits()_1.
```



Example: the generated generator



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

nat(0) ->
    non_neg_integer();
nat(S) ->
    weighted_union([
        {2, ?LAZY(nat(0))},
        {2, ?LAZY(non_neg_integer())},
        {2, ?LAZY(?LETSHRINK([X,Y], vector(2,nat(S div 2)),
                {’+’, X, Y}))},
        {1, ?LAZY(?LETSHRINK([X,Y,Z,W], vector(4,nat(S div 4)),
                {’if’, {’=’, X, Y}, Z, W}))},
        {1, ?LAZY(?LETSHRINK([X,Y], vector(2,nat(S div 4)),
                {’if’, {’=’, real(S div 4),real(S div 4)},
                       X, Y}))},
        {2, ?LAZY({’from_bits’,from_bits(S)})}]).

real() ->
    ?SIZED(Size, real(Size)).

real(0) ->
    {’from_nat’,nat(0)};
real(S) ->
    weighted_union([
        {2, ?LAZY(real(0))},
        {3, ?LAZY({’from_nat’,nat(S-1)})},
        {3, ?LAZY(?LETSHRINK([X,Y], vector(2,real(S div 2)),
                {’+’, X, Y}))}]).
```



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

bits(0) ->
    {’concat’,[]};
bits(S) ->
    weighted_union([
        {2, ?LAZY(bits(0))},
        {3, ?LAZY({’from_nat’,nat(S-1)})},
        {3, ?LAZY({’concat’,resize(S,list(bits_helper

bits_helper() ->
    ?SIZED(Size, nat(Size)).

bits_helper(0) ->
    union([nat(0), bits(0)]);
bits_helper(S) ->
    weighted_union([{2, ?LAZY(bits_helper(0))},
                    {3, ?LAZY(nat(S-1))},
                    {3, ?LAZY(bits(S-1))}]).
```

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))).
```

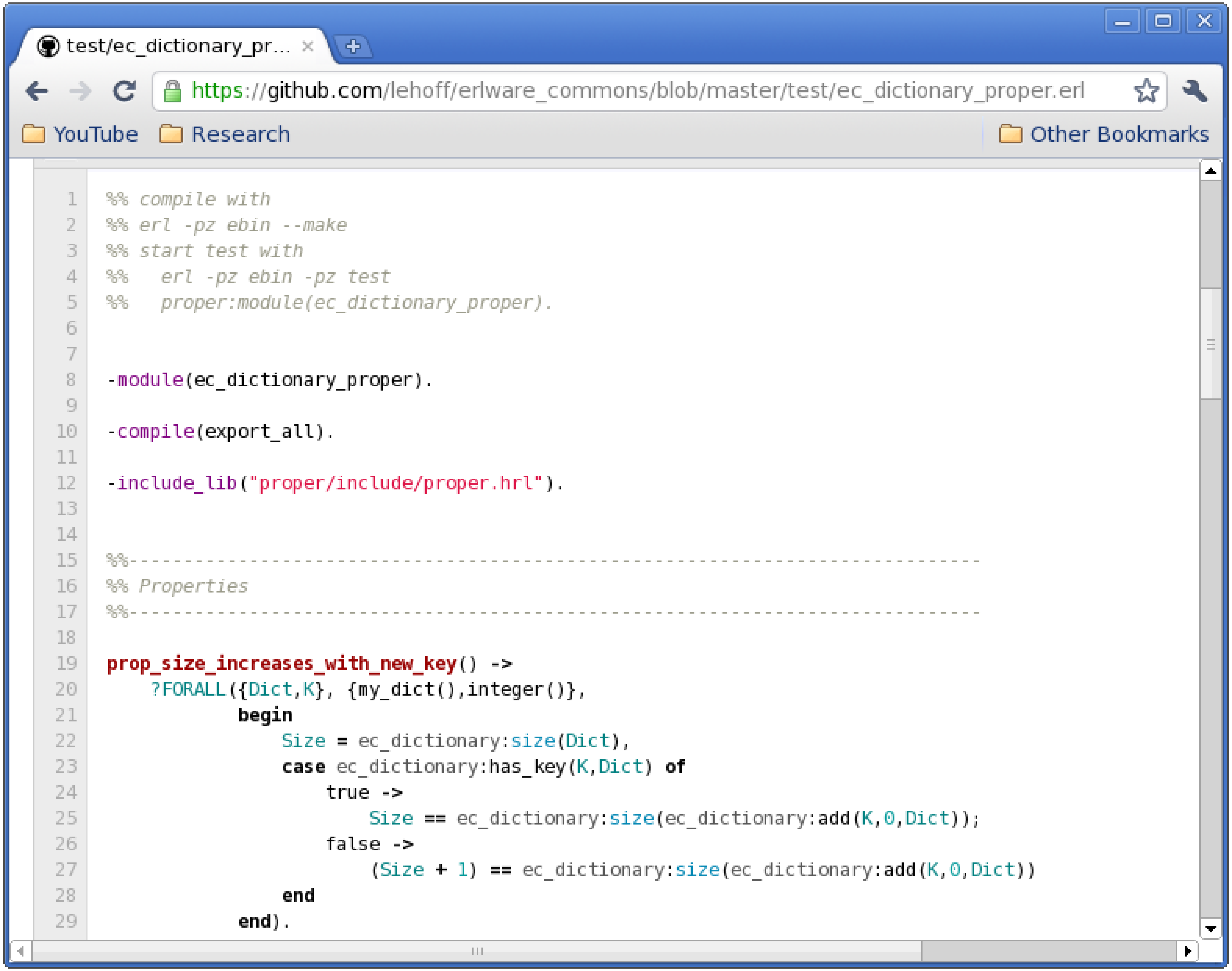
PropEr testing of specs

```
-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 already used out there!



test/ec_dictionary_pr...
https://github.com/lehoff/erlware_commons/blob/master/test/ec_dictionary_proper.erl
YouTube
Research
Other Bookmarks

```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).

-include_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).
```

Some observations from PropEr uses

- Erlang's type language is often less expressive than desired for property-based testing
  - e.g. not possible to specify that binaries should contain valid UTF8 characters
- Function specs cannot express argument dependencies
  - e.g. dependencies between args of **`lists:nth/2`**
- Users often under-specify function domains
- Function signatures can often be used as simple specifications of functions

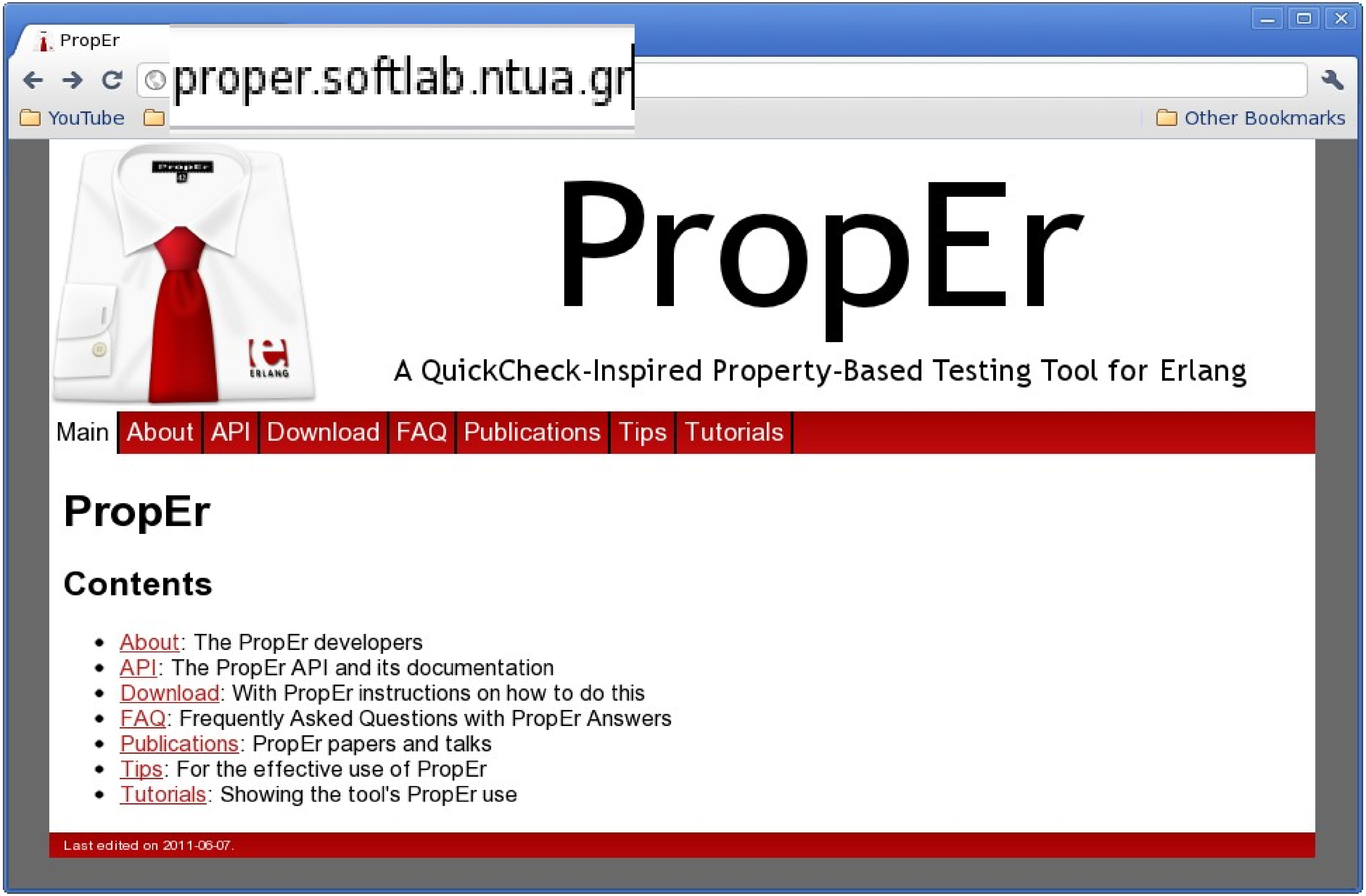
Lessons learned

- Unit testing and property-based testing require different mindsets
  - Difficult to come up with “interesting” properties
  - Tricky to express them
    - often one debugs the property rather than the code
- Writing generators for recursive types is tricky and requires significant time and effort
  - PropEr significantly eases this task

Some PropEr advice

- Start with testing the functional core
- Break the testing into smaller, simpler to express (partial) correctness properties
- Write properties for readability
- For generators of recursive datatypes
  - Just write the data type and rely on PropEr
  - Put a global size bound if the above is not enough
  - Only if the steps above are not enough resort to using **`?LAZY/1, ?LETSHRINK/1, resize, …`**

More info on our PropEr website



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.