

Property-based Testing con PropEr

Miguel Emilio Ruiz Nieto

3 de diciembre de 2021

Contenidos

- 1 Motivación
- 2 Definiciones
- 3 Erlang
- 4 PropEr
- 5 Un caso real
- 6 Conclusiones
- 7 Bibliografía

```
-module(sort_lib).  
  
-export([sort/1]).  
% Implementation of quicksort algorithm  
-spec sort(list(integer())) -> list(integer()).  
sort([]) -> [];  
sort([P|Xs]) ->  
    sort([X || X <- Xs, X < P]) ++  
        [P] ++ sort([X || X <- Xs, P < X]).
```

Motivación. Test unitarios

```
-module(sort_lib_eunit).  
  
-include_lib("eunit/include/eunit.hrl").  
  
sort_test() ->  
    [test_zero(), test_two(), test_four()].  
test_zero() ->  
    ?_assertEqual([], sort_lib:sort([])).  
test_two() ->  
    [?_assertEqual([17,42],  
                    sort_lib:sort([X,Y]))  
      || {X,Y} <- [{17,42}, {42,17}]  
    ].  
test_four() ->  
    ?_assertEqual([1,2,3,4],  
                    sort_lib:sort([3,1,4,2])).
```

Preguntas

- ¿Son buenos estos tests?
- ¿Harían falta más?
- En caso de que sí, ¿cuántos más?

Las metodologías de testing tradicionales son útiles ya que:

- Obliga a los desarrolladores a escribir casos de prueba del software desarrollado
- Para cada input se debe generar un cierto output con el fin de comprobar el correcto funcionamiento del sistema

Pero tienen sus inconvenientes:

- Consumen tiempo (€€€)
- No se garantiza que la batería de tests cubra todos los casos

La alternativa

Property-based testing

Property-based Testing. Definición

- Es una técnica para hacer pruebas sobre las propiedades de nuestro sistema
- Los tests no son sobre casos de uso, sino sobre el comportamiento del propio sistema
- Muy común en lenguajes de programación funcional (i.e Quickcheck)

Property-based Testing. Definición

El proceso de aplicar PBT se divide en dos fases:

Property-based Testing. Definición

El proceso de aplicar PBT se divide en dos fases:

- Generar datos de entrada de manera aleatoria (Generadores)

El proceso de aplicar PBT se divide en dos fases:

- Generar datos de entrada de manera aleatoria (Generadores)
- Hacer verificaciones de las funciones aplicadas a esos datos (Propiedades)

- Son reglas generales que describen el comportamiento de una función o un programa
- Han de ser aplicables a cualquier tipo de entrada y salida del propio programa bajo sus propias condiciones
- La salida debe verificar ciertas características deseadas

Al principio puede resultar no tan trivial como los tests unitarios ya que:

- El desarrollador ha de tener una visión más nítida de los casos de uso y del comportamiento del sistema

No obstante:

- Asegura encontrar un mayor número de “casos esquina” y bugs dentro del código

- Lenguaje de programación desarrollado en Ericsson
- Orientado a sistemas distribuidos:
 - Modelo de actores
 - Paso de mensajes
 - Tolerancia a fallos
 - Alta disponibilidad
 - Filosofía “Let it crash”

```
-module(successions).  
-export([fib/1]).
```

```
fib(0) -> 0;  
fib(1) -> 1;  
fib(N) -> fib(N-1) + fib(N-2).
```



```
1> [First | TheRest] = [1,2,3,4,5].  
2> First.  
1  
3> TheRest.  
[2,3,4,5]
```

```
4> X = 10, Y = 4.
```

```
4
```

```
5> Point = {X,Y}.
```

```
{10,4}
```

```
6> PreciseTemperature = {celsius, 23.213, 45}.
```

Sintaxis. Pattern Matching

```
is_even(N) when N rem 2 == 0 -> true;  
is_even(_) -> false.
```

Sintaxis. Funciones de Orden Superior

```
7> Add_3 = fun(X) -> X + 3 end.  
#Fun<erl_eval.7.126501267>  
8> lists:map(Add_3, [1,2,3]).  
[4,5,6]
```

- Herramienta para realizar property-based testing en Erlang
- Inspirada en Quickcheck
- Completamente integrada con los tipos de Erlang

Nos centraremos en los siguientes aspectos:

- Generadores
- Estructura de las propiedades
- Propiedades sin estado
- Propiedades con estado
- Reducción de contraejemplos (*shrinking*)

- Funciones que *generan* entradas de manera “aleatoria”
- Proporcionan datos en base al tipo del generador y a los filtros dados
- Pueden ser:
 - En base a los tipos de Erlang
 - Customizados por el desarrollador

PropEr. Generadores. Ejemplos

Basados en los tipos de Erlang

Generador	Muestra
<code>integer()</code>	89234
<code>boolean()</code>	true, false
<code>list(Type)</code>	[true, true, false]
<code>tuple()</code>	{true, 13.321123, -67}

Customizados por el desarrollador

```
?LET(InstanceOfType, TypeGenerator, Transform).
```

- **InstanceOfType** : La variable que contendrá los datos generados por el generador del segundo argumento
- **TypeGenerator** : La función generadora que produce los datos que se almacenan en el argumento anterior
- **Transform** : Expresión que transforma los datos de la propia función y los acumula en el argumento anterior

```
% Customized Generator
list_no_dupls(T) ->
    ?LET(L, list(T), remove_duplicates(L)).

% Helper
remove_duplicates([]) -> [];
remove_duplicates([A|T]) ->
    case lists:member(A, T) of
        true -> remove_duplicates(T);
        false -> [A|remove_duplicates(T)]
    end.
```

`?FORALL(InstanceOfType, TypeGenerator,
PropertyExpression).`

- `InstanceOfType` : La variable que contendrá los datos generados por los generadores
- `TypeGenerator` : La función generadora que produce los datos que se almacenan en el argumento anterior
- `PropertyExpression` : Expresión booleana que especifica la propiedad que se desea verificar

PropEr. Estructura de las propiedades. Ejemplos

```
-module(prop_sort_lib).  
  
-include_lib("proper/include/proper.hrl").  
  
% Propiedad con generador con tipo de Erlang  
prop_same_length() ->  
    ?FORALL(L, list(integer()),  
        length(L) == length(sort_lib:sort(L))).  
  
% Propiedad con generador customizado  
prop_same_length_no_dupls() ->  
    ?FORALL(L, list_no_dupls(integer()),  
        length(L) == length(sort_lib:sort(L))).
```

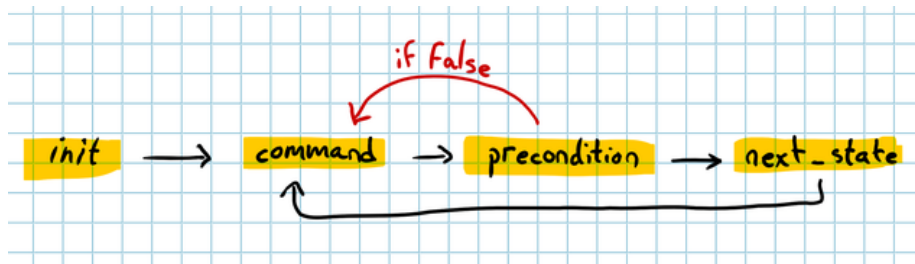
PropEr. Estructura de las propiedades. Ejemplos

```
prop_exists_already_sorted() ->  
  ?EXISTS(L, list(integer()), L := sort_lib:sort(L)).  
  
prop_not_exists_different_sorted_after_double_sort() ->  
  ?NOT_EXISTS(L, list(integer()),  
    sort_lib:sort(L) /= sort_lib:sort(sort_lib:sort(L))).
```

- Hasta ahora los ejemplos que hemos visto eran *sin estado*
- Nos interesa verificar que aquellas aplicaciones que tras realizar una operación y cambian su estado mantienen la consistencia
- Ideal para los behaviours `gen_server` y `gen_statem`

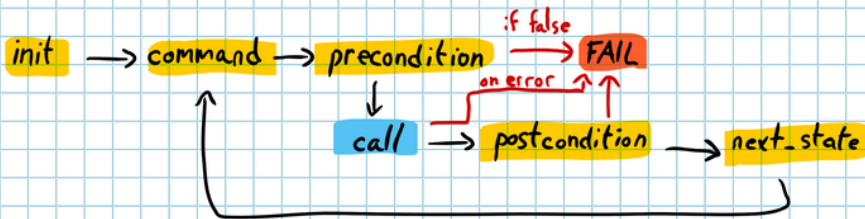
- Los tests con estado se dividen en dos fases:
 - Fase abstracta: se crea el escenario de test (Modelo)
 - Fase real: El sistema donde se aplican los comandos generados para dicho modelo

PropEr. Propiedades con estado



Fase abstracta

PropEr. Propiedades con estado



Fase real

- En algunos casos, puede haber ciertos tests que no cumplan las propiedades que hemos definido
- PropEr es capaz de aplicar *reducciones* hasta llegar al contraejemplo más pequeño posible

PropEr. Reducción de contraejemplos

```
~/sort_lib $ rebar3 proper
==> Verifying dependencies...
==> Compiling sort_lib
==> Testing prop_sort_lib:prop_same_length()
.....!
Failed: After 10 test(s).
[9,-2,8,-2]

Shrinking ..(2 time(s))
[-2,-2]
==>
0/1 properties passed, 1 failed
==> Failed test cases:
prop_sort_lib:prop_same_length() -> false
```

- Coowry. Empresa dedicada a los pagos a través de *airtime*
- Cientos de clientes diariamente realizaban pagos a través del API
 - El formato de las transacciones no siempre era el mismo
 - Caídas de servicio

- Coowry. Empresa dedicada a los pagos a través de *airtime*
- Cientos de clientes diariamente realizaban pagos a través del API
 - El formato de las transacciones no siempre era el mismo
 - Caídas de servicio

La solución

Baleen. Biblioteca de validadores escritos en Erlang

```
-type validator(A, B) :: (fun(A) -> result(B) end).  
-type result(A) :: {ok, A} | {error, binary()}.  
-type str() :: string() | binary().  
  
-spec validate(validator(A,B), A) -> result(B).  
  %% @doc Validates data with a validator.  
  %% `X' is the term to be validated with  
  %% validator `V'.  
validate(V, X) -> V(X).
```

```
-spec valid() -> validator(_,_).  
%% @doc Returns a validator that always validates.  
valid() ->  
    fun(X) -> {ok, X} end.
```

```
-spec to_integer() -> validator(str(), integer()).  
%% @doc Returns a validator that takes a `Value`  
%% and tries to cast to integer. If the cast success,  
%% `{ok, Integer}` is returned, otherwise,  
%% `{error, <<"Value is not an integer">>}` is returned.  
to_integer() ->  
    fun(Value) when is_binary(Value)->  
        try erlang:binary_to_integer(Value) of  
            Integer -> {ok, Integer}  
        catch  
            _:_ -> {error, format("~p is not an integer", [Value])}  
        end;  
    (Value) ->  
        case io_lib:fread("~d",Value) of  
            {ok, [Integer], []} -> {ok, Integer};  
            _ -> {error, format("~p is not an integer", [Value])}  
        end  
    end.  
end.
```



```
prop_valid() ->
  ?FORALL(Data, term(),
    begin
      {ok, Data} == baleen:validate(valid(), Data)
    end).

prop_is_integer_with_integer_strings() ->
  ?FORALL(Integer, integer(),
    {ok, Integer} == baleen:validate(to_integer(),
                                     integer_to_list(Integer))).

prop_is_integer_with_integer_binaries() ->
  ?FORALL(Integer, integer(),
    {ok, Integer} == baleen:validate(to_integer(),
                                     integer_to_binary(Integer))).
```

- Merece la pena usar PBT porque es capaz de generar casos de test muy “rápido”
- No obstante,
 - Hay casos en los que puede merecer la pena más hacer test unitarios
 - A veces cuesta más definir las propiedades que el desarrollo pedido
 - Se usa poco dado que está muy centrado en lenguajes funcionales

- Getting Started with Erlang
https://www.erlang.org/doc/getting_started/intro.html
- PropEr Guide <https://proper-testing.github.io>
- Property-Based Testing with PropEr, Erlang and Elixir
<https://www.propertesting.com/>
- Baleen repo <https://github.com/coowry/baleen>