Property-Based Test-Case Generators for Free

Emanuele De Angelis[1], **Fabio Fioravanti**[1], **Adrián Palacios**[2], **Alberto Pettorossi**[3], and **Maurizio Proietti**[4]

[1] University of Chieti-Pescara "G. d'Annunzio", Italy
[2] MiST, DSIC, Polytechnic University of Valencia, Spain
[3] University of Rome "Tor Vergata", Italy
[4] CNR - Istituto di Analisi dei Sistemi ed Informatica, Italy

Tests and Proofs
(TAP 2019)

Porto, Portugal
11 October 2019

Property-Based Testing (PBT)

Idea behind PBT

instead of designing specific test-cases,
a software test engineer **specifies properties**
of the **program inputs** and **outputs**

Then, a test execution engine

1. randomly generates an **input** that **satisfies** the specification
2. runs the program with that input
3. checks whether or not the **output** **satisfies** the specification

QuickCheck (Haskell, [Classen & Hughes, ICFP '00])

PropEr (Erlang, [Papadakis et al., ACM SIGPLAN WKSH Erlang '11])

Constraint-based PBT

FocalTest [Carlier et al., ICSOFT ’10, TAP ’12]

PBT for Focalize based on Constraint Logic Programming

other **constraint-based** testing methods

ArbitCheck (Java), JML-TT (Java/JML), PathCrawler (C), DART (C), CUTE (C), Euclide (C), PrologCheck (Prolog), GATeL (Lustre), CutEr (Erlang), AUTOFOCUS (MBT XP dev. Proc.), ...

This work

ProSyT: Property-based **Sy**mbolic **T**esting

PBT framework for testing **Erlang** programs
based on Constraint Logic Programming

smart symbolic execution of an **interpreter** for Erlang
to generate input test values
directly from program inputs specifications

Property-Based Testing for Erlang

Erlang: concurrent, higher-order, functional programming language with dynamic, strong typing

Erlang program: sequence of function definitions

```
f(X1,…,Xn)-> e
```

f is a function name, `X1,…,Xn` are variables, and **e** is an expression

A faulty insertion program

```
insert(E,L) ->
  case L of
    [] -> [E];
    [X|Xs] when E=<X -> [X,E|Xs];
    [X|Xs] -> [X] ++ insert(E,Xs)
  end.
```

Erlang PBT: PropEr (Property-based testing tool for Erlang)

`https://github.com/proper-testing/proper`

Specifying PBT tasks with PropEr

Property to be satisfied by the program inputs and outputs

Given any `integer` I and any `ordered list of integers` L,
`insert(I,L)` produces an ordered list

```
prop_ordered_insert() ->
    ?FORALL( {I,L},
             {integer(),ordered_list()},
             ordered(insert(I,L))       ).
```

Specification of the
GENERATOR
of program inputs

Specification of the
PROPERTY
of program outputs

Properties of program outputs

```
prop_ordered_insert() ->
    ?FORALL( {I,L},
             {integer(),ordered_list()},
             ordered(insert(I,L))     ).
```

An Erlang boolean function:

```
ordered(L) -> case L of
  [A,B|T] -> A =< B andalso ordered([B|T]);
  _ -> true
end.
```

Properties of program inputs

```
prop_ordered_insert() ->
    ?FORALL( {I,L},
             {integer(),ordered_list()},
             ordered(insert(I,L))     ).
```

Generators are Erlang expressions built upon

- Predefined types

  `integer(), float(), list(integer()), { integer(), … }`

- User-defined types

  ```
  -type tree() :: ‘leaf’ | {‘node’,tree(T),T,tree(T)}
  ordered_list() ->
     ?SUCHTHAT(L, list(integer()), ordered(L) ).
  ```

Filter of valid lists

Specifying filters

```
prop_ordered_insert() ->
    ?FORALL( {I,L},
             {integer(),ordered_list()},
             ordered(insert(I,L))     ).
```

- without an ad-hoc implementations of `ordered_list()`, the generation of **ordered lists** from

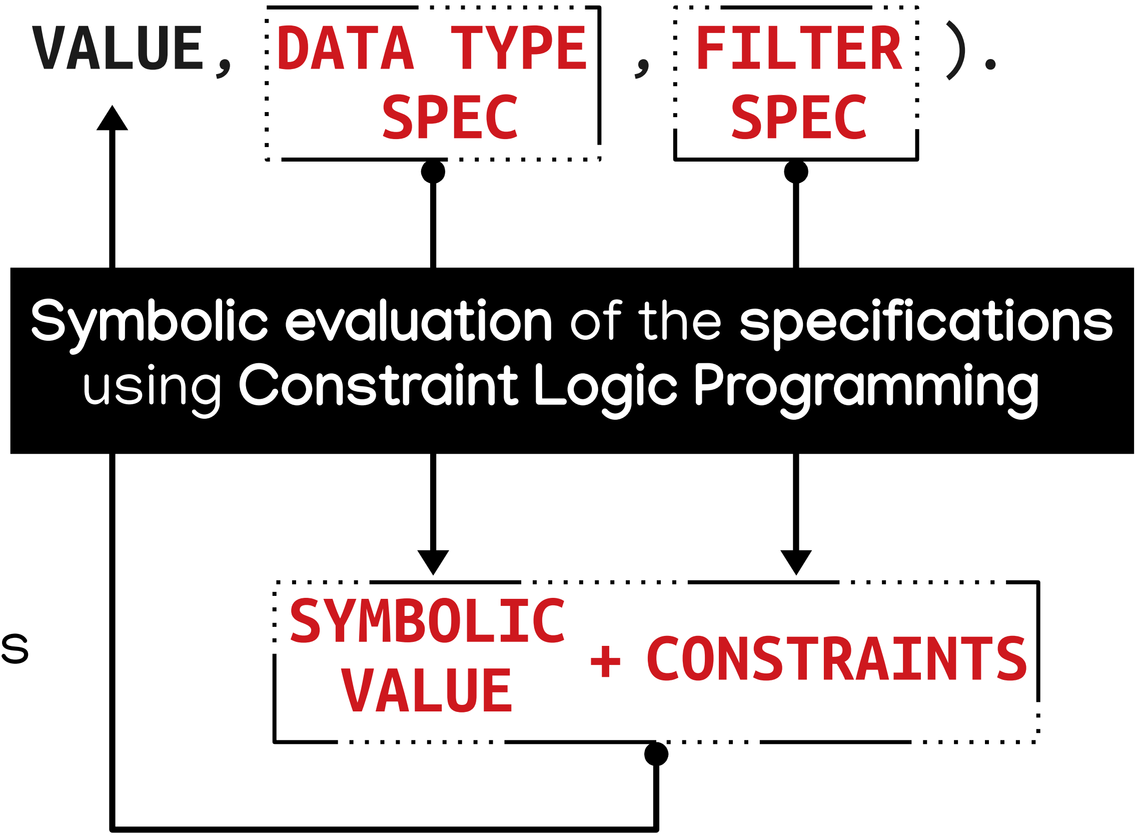
  ```
  ordered_list() ->
      ?SUCHTHAT(L, list(integer()), ordered(L)).
  ```

  is performed in an inefficient way by
  randomly **generating** a list of integers **& testing** if ordered
- implementing ad-hoc generators
  time-consuming & error-prone activity

Our goal & contribution

To relieve developers from implementing ad-hoc input **generators** by **deriving generators** directly from **PropEr specifications**

```
ordered_list() ->
        ?SUCHTHAT( VALUE, DATA TYPE SPEC , FILTER SPEC ).
```



Symbolic evaluation of the **specifications** using **Constraint Logic Programming**

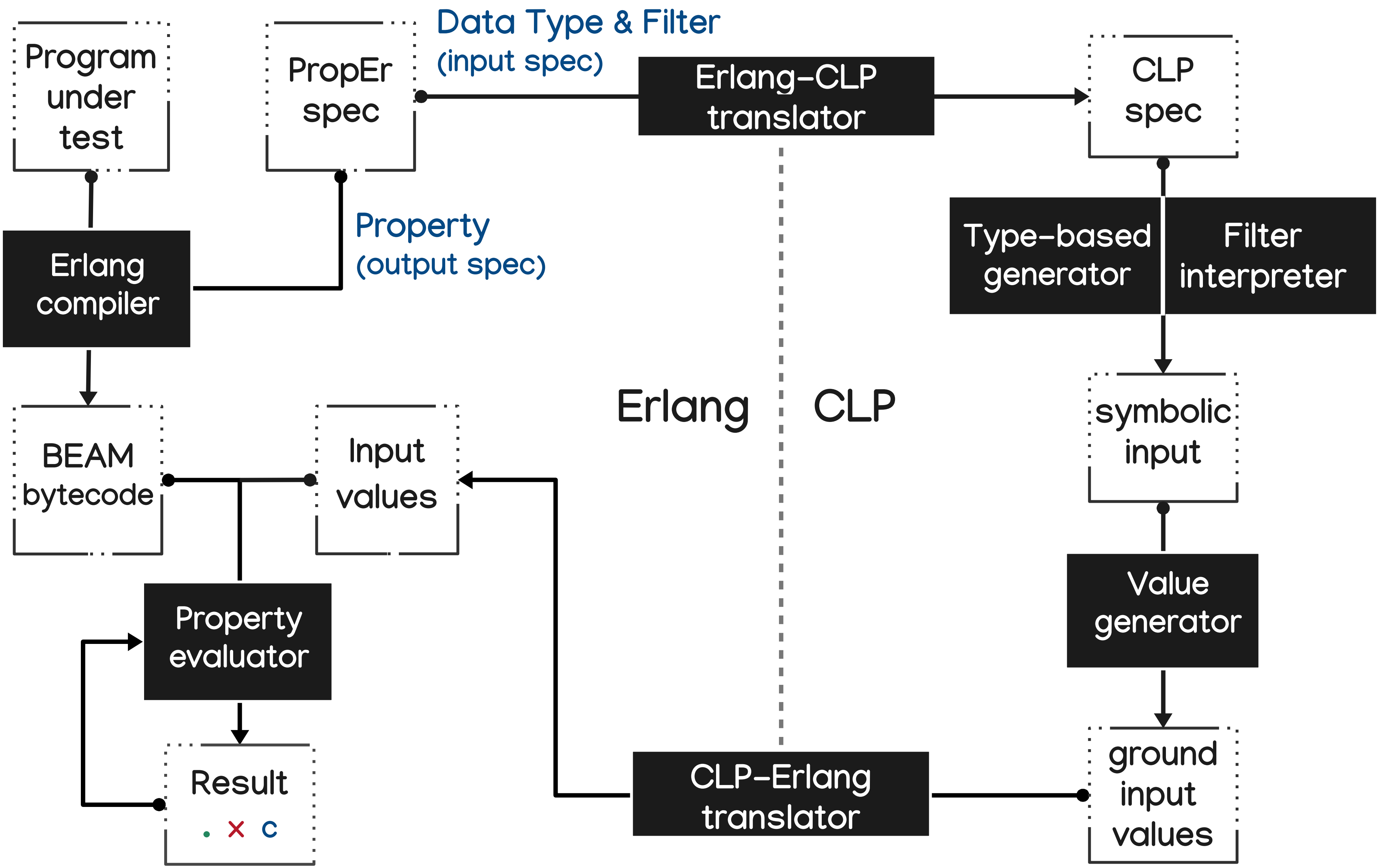
SYMBOLIC VALUE + CONSTRAINTS

Constrain & generate computation pattern:

1. generates **symbolic data structures** with **constraints**
2. generates random values satisfying those constraints
3. translates back concrete data into inputs for the program under test

ProSyT

Property-Based Symbolic Testing



Program under test

PropEr spec

Data Type & Filter (input spec)

Erlang-CLP translator

CLP spec

Erlang compiler

Property (output spec)

Type-based generator

Filter interpreter

BEAM bytecode

Input values

Erlang

CLP

symbolic input

Value generator

Property evaluator

Result

. × c

CLP-Erlang translator

ground input values

Constraint Logic Programming

CLP(X) languages

CLP(FD)	for integers
CLP(R)	for floats
CLP(B)	for booleans

Constraint of CLP(FD)

Quantifier free first-order formula whose variables range over FD

Definition of the predicate `insert/3`**:**

clauses

```
insert(I,[],[I]).
insert(I,[X|Xs],[I,X|Xs]) :- I #=< X.
insert(I,[X|Xs],[X|Ys]) :- I #> X, insert(I,Xs,Ys).
```

query →

```
?- I #>= 3, I #=< 7, L = [2,4,8], insert(I,L,O).
L = [2,4,8], O = [2,I,4,8], I in 3..4 ;
L = [2,4,8], O = [2,4,I,8], I in 5..7 ;
false.
```

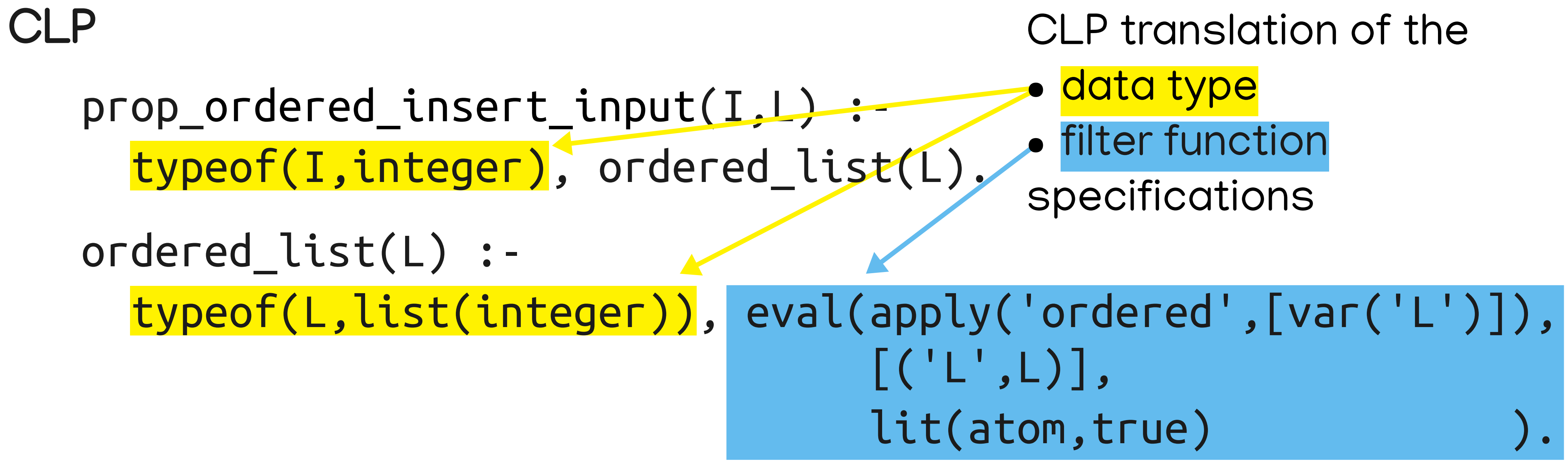
answers

SWI Prolog

Translator from PropEr to CLP

```
prop_ordered_insert() ->
  ?FORALL( {I,L},
           {integer(),ordered_list()},
           ordered(insert(I,L))     ).
 ordered_list() ->
   ?SUCHTHAT( L, list(integer()), ordered(L) ).
```

Erlang



CLP

```
prop_ordered_insert_input(I,L) :-
  typeof(I,integer), ordered_list(L).

ordered_list(L) :-
  typeof(L,list(integer)), eval(apply('ordered',[var('L')]),
                                     [('L',L)],
                                     lit(atom,true)        ).
```

CLP translation of the
- data type
- filter function

specifications

Type-based value generator

Given a data type specification T,
the predicate `typeof(X,T)` holds iff
X is a CLP term encoding an Erlang value of type T.

```
ordered_list() ->
      ?SUCHTHAT(L, list(integer()), ordered(L)).
```

```
typeof(nil,list(T)).
typeof(cons(Hd,Tl),list(T)) :-
    typeof(Hd,T), typeof(Tl,list(T)).
```

```
?- typeof(L,list(integer)).
L = nil ;
L = cons(lit(int,X),nil), X in inf..sup ;
...
```

Size (of terms)
is configurable:

- length of lists
- interval of integers
- ...

Interpreter of filter functions

The CLP interpreter provides the predicate

`eval(In,Env,Out)`

that computes

the **output** expression `Out` from

the **input** expression `In` in the **environment** `Env`
(maps variables to values)

Using **symbolic** expressions (`In` and `Out` are CLP terms possibly with variables) in `eval` enables the

exploration of all program computations

without

explicitly enumerating all concrete inputs

Interpreter of filter functions

```
eval(apply(Func,Exps),Env,Out) :-
    fundef(Func,fun(Pars,Body)),
    eval_args(Exps,Env,Vs),
    zip_binds(Pars,Vs,Binds),
    constrain_output_exp(Func,Out),
    eval(Body,Binds,Out).
```

❶ Retrieves the definition of `Func`

❷ Evaluates the actual parameters `Exps` in the environment `Env`, to get their values `Vs`

❸ Binds the formal parameters `Pars` to the values `Vs`, to get the new environment `Binds`

❹ Enforces constraints derived from function contracts. For instance, from

`-spec listlength(list(any())) -> non_neg_integer().`

we enforce `R #>= 0` upon `Out = lit(int,R)`

❺ Evaluates the `Body` of `Func` in the new environment `Binds`, to get the output expression `Out`

Generating symbolic ordered lists

```
ordered_list(L) :-
 typeof(L,list(integer)),
 eval(apply('ordered',[var('L')]),[('L',L)],lit(atom,true)).
```

generetes a non-ground CLP term representing a list

enforces constraints on the list (ascending order of its elements)

```
?- ordered_list(L).
L = nil ;
L = cons(lit(int,X),nil), X in inf..sup ;
L = cons(lit(int,X),cons(lit(int,Y),nil)),
 Y #>= X ;
L = cons(lit(int,X),cons(lit(int,Y),cons(lit(int,Z),nil))),
 Y #>= X, Z #>= Y
```

Value generator

Random generation of **ground** terms

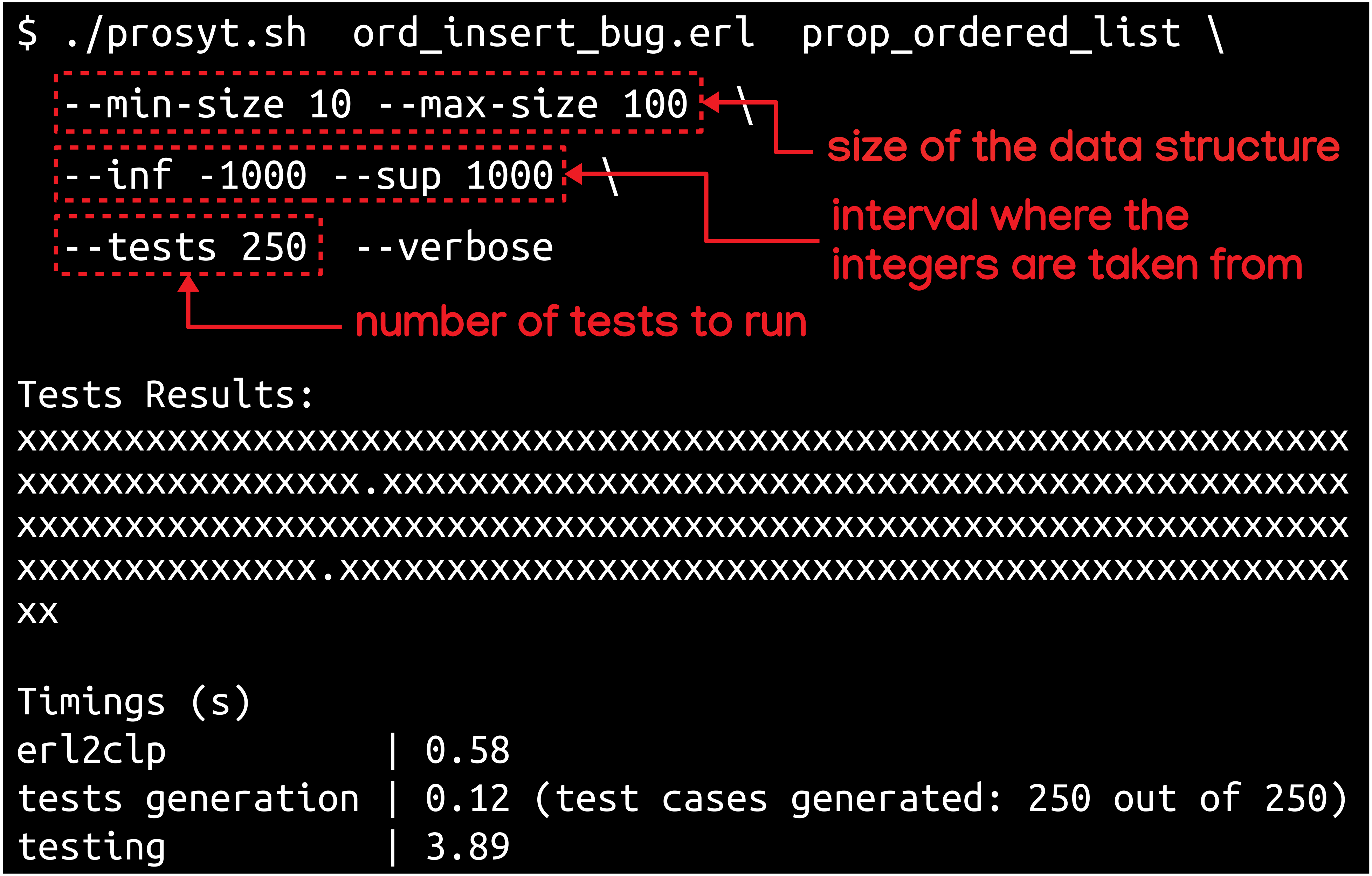
```
rand_elem(nil).
rand_elem(cons(X,L)) :- rand_elem(X), rand_elem(L).
rand_elem(lit(int,V)) :-
  fd_inf(V,Inf), fd_sup(V,Sup), random_between(Inf,Sup,V).
```

```
?- ordered_list(L), rand_elem(L), write_elem(L).
[]
L = nil ;
[10]
L = cons(lit(int,10),nil) ;
[4,8]
L = cons(lit(int,4),cons(lit(int,8),nil)) ;
[2,6,9]
L = cons(lit(int,2),cons(lit(int,6),cons(lit(int,9),nil)))
```

Translate
CLP terms to
Erlang values

Running ProSyT

https://fmlab.unich.it/testing/



```
$ ./prosyt.sh  ord_insert_bug.erl  prop_ordered_list \
   --min-size 10 --max-size 100 \
   --inf -1000 --sup 1000 \
   --tests 250  --verbose

Tests Results:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xx

Timings (s)
erl2clp          | 0.58
tests generation | 0.12 (test cases generated: 250 out of 250)
testing          | 3.89
```

size of the data structure

interval where the integers are taken from

number of tests to run

Generating complex data structures

Symbolic test-case generation performs well when the filter **does not specify constraints on the skeleton** of the data structure, but only on its elements

AVL tree

- binary search tree (constraints on the elements)
- height-balanced (constraints on the skeleton)

```
avl(T) -> case T of
  leaf -> true;
  {node,L,V,R} -> B = height(L) - height(R)   andalso
                  B >= -1   andalso   B =< 1    andalso
                  ltt(L,V)  andalso   gtt(R,V)  andalso
                  avl(L)    andalso   avl(R)    ;
  _ -> false
 end.
```

Symbolic generation of AVL trees

```
?- typeof(X,tree(integer)),                                    ❶
   eval(apply(‘avl’,[var(’T’)]),[(’T’,X)],lit(atom,true)).     ❷
```

❶ generates a symbolic **binary tree** X

❷ applies the filter to X

- makes X a **search tree**
  by enforcing constraints on the values of the nodes
- makes X **height-balanced**
  how? Can’t enforce constraints on the skeleton of X
  which is determined by ❶

Among the answers of ❶ just a few are height-balanced trees

For trees of size 10 (number of nodes) ❷ finds 10 AVL trees out of 9000 binary trees generated by ❶

Data-driven generation: coroutining

Interleaving the execution of

- the type-based generator `typeof`, and
- the interpreter of filter functions `eval`

enables **enforcing constraints while generating** data

```
eval(apply(‘avl’,[var(’T’)]),
   [(’T’,X)],lit(atom,true)), typeof(X,tree(integer))
```

`typeof` and `eval` cooperate through `X`
during the generation of the AVL tree

Implemented using the **coroutining** mechanism provided by SWI-Prolog through the primitive

`when(Cond,Conj)`

that **suspends** the execution of `Conj` until `Cond` becomes true

Coroutining typeof and eval

```
?- typeof(…), eval(…).
```

```
eval(case(CExps,Cls),Env,Exp) :-
  eval(CExps,Env,EExps),
  suspend_on(Env,EExps,Cls,Cond),
  when(Cond, (
    match(Env,Eexps,Cls,MEnv,Cl),
    eval(Cl,MEnv,Exp)
  )).
```

selects the variables that would get bound to **lists** or **tuples** while matching `EExpr` against the clauses `Cls` of the `case-of` expression

suspends the evaluation of the `match` until all the variables get bound non-variable terms

```
?- eval(…)coroutining, typeof(…).
```

Generating AVL trees

```
?- eval(apply('avl',[var('T')]),
        [('T',X)],lit(atom,true)), typeof(X,tree(integer))
```

coroutining

height-balanced (BST) ?

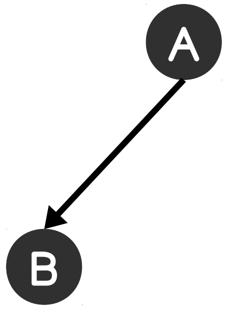
The evaluation of the filter adds constraints on X

$$-1\ \#{=}{<}\ \overbrace{\text{height}(A_{left})-\text{height}(A_{right})}^{1}\ \#{=}{<}\ 1,$$

$$-1\ \#{=}{<}\ \underbrace{\text{height}(B_{left})-\text{height}(B_{right})}_{0}\ \#{=}{<}\ 1,$$

$$A\ \#{<}\ B$$

the constraints on X restrict the possible ways in which its left and right subtrees can be further expanded



As soon as the typeof (partially) instantiates X to a binary tree

Generating AVL trees

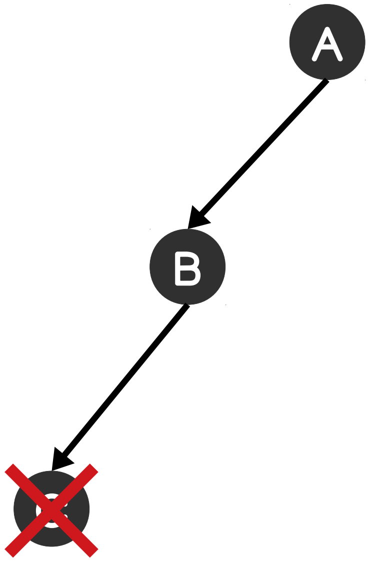
```
?- eval(apply(‘avl’,[var(’T’)]),
                                          coroutining
      [(’T’,X)],lit(atom,true)), typeof(X,tree(integer))
```

height-balanced (BST) ?

$$-1 \ \#=< \overbrace{\text{height}(A_{\text{left}})-\text{height}(A_{\text{right}})}^{2 \quad \text{unsat}} \ \#=< 1,$$

$$-1 \ \#=< \underbrace{\text{height}(B_{\text{left}})-\text{height}(B_{\text{right}})}_{1} \ \#=< 1,$$

$$A \ \#< B$$



Generating AVL trees

```
?- eval(apply(‘avl’,[var(’T’)]),
      [(’T’,X)],lit(atom,true)), typeof(X,tree(integer))
```

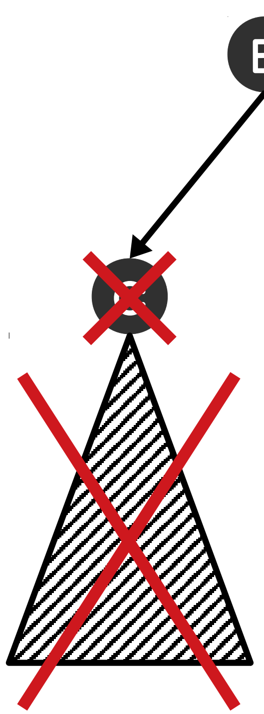
coroutining

height-balanced (BST) ?

$$-1 \;\#{=}{<}\; \overbrace{\mathtt{height}(A_{left})\mathtt{-height}(A_{right})}^{2\quad \text{unsat}} \;\#{=}{<}\; 1,$$

$$-1 \;\#{=}{<}\; \underbrace{\mathtt{height}(B_{left})\mathtt{-height}(B_{right})}_{1} \;\#{=}{<}\; 1,$$

$$A \;\#{<}\; B$$



Generating AVL trees

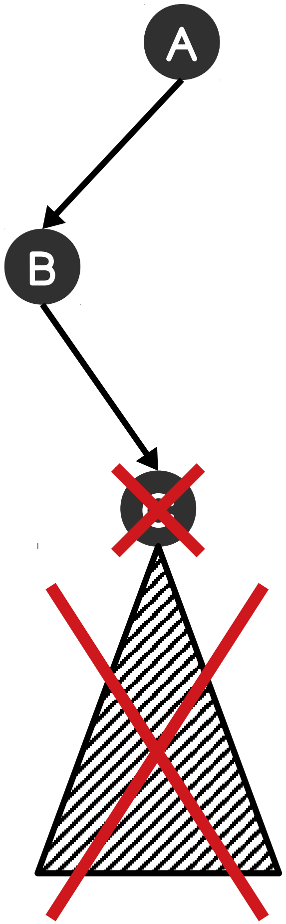
```
?- eval(apply(‘avl’,[var(’T’)]),
                                      coroutining
       [(’T’,X)],lit(atom,true)), typeof(X,tree(integer))
```

height-balanced (BST) ?

$$-1 \;\#\!=\!< \overbrace{\texttt{height}(A_{left})\texttt{-height}(A_{right})}^{2\quad \text{unsat}} \#\!=\!<\; 1,$$

$$-1 \;\#\!=\!< \underbrace{\texttt{height}(B_{left})\texttt{-height}(B_{right})}_{-1} \#\!=\!<\; 1,$$

$$A \;\#\!<\; B$$



Generating AVL trees

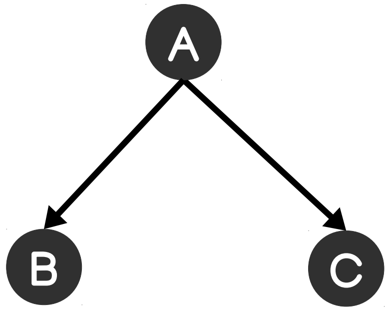
```
?- eval(apply(‘avl’,[var(’T’)]),
                                                coroutining
     [(’T’,X)],lit(atom,true)), typeof(X,tree(integer))
```

height-balanced (BST) ?

$$-1 \ \#{=}{<} \ \overbrace{\mathtt{height}(A_{left})-\mathtt{height}(A_{right})}^{0} \ \#{=}{<} \ 1,$$

$$-1 \ \#{=}{<} \ \underbrace{\mathtt{height}(B_{left})-\mathtt{height}(B_{right})}_{0} \ \#{=}{<} \ 1,$$

$A \ \#{<} \ B$



Experimental evaluation

Program	PropEr		ProSyT	
	Time	N	Time	N
ord_insert	300.00	0	300.00	67,083
up_down_seq	300.00	0	300.00	22,500
n_up_seqs	300.00	0		24,000
delete	300.00	0	9.21	100,000
stack	143.71	100,000	19.57	100,000
matrix_mult	300.00	0	300.00	76,810
det_tri_matrix	300.00	304	32.28	13,500
balanced_tree	300.00	121	21.54	100,000
binomial_tree_heap	300.00	0	43.45	4,500
avl_insert	300.00	0	300.00	23,034

Time reports the seconds needed to generate N (≤ 100,000) test cases of size in the interval [10,100] within the time limit of 300s.
(Intel® Core™ i7-8550U with 16GB of memory running Ubuntu 18.04.2 LTS)

Conclusions

ProSyT

a PBT framework that relieves developers from writing generators of input values for testing Erlang programs

- ✓ based on a **constrain & generate** computation pattern implemented using a **CLP interpreter** that makes the generation process efficient in many cases
- ✓ CLP is fully **transparent** to users
  - translator from PropEr/Erlang specifications to CLP
  - translator from CLP test-cases to Erlang

Future work

- provide developers with explicit **shrinking** mechanism
- apply the approach to other programming languages, the **interpreter** makes it independent of the prog. lang.