Automatic Synthesis of Random Generators for Numerically Constrained Algebraic Recursive Types*

Ghiles Ziat[1], Vincent Botbol[2], Matthieu Dien[3], Arnaud Gotlieb[4], Martin Pépin[1][0000−0003−1892−3017], and Catherine Dubois[5]

[1] Université Paris Cité, IRIF, 75205 Paris Cedex 13, France, {ghiles.ziat, martin.pepin}@irif.fr,
[2] Nomadic Labs, vincent.botbol@nomadic-labs.com
[3] Normandie Université, UNICAEN, ENSICAEN, CNRS, GREYC, 14000 Caen, France, matthieu.dien@unicaen.fr
[4] SIMULA RESEARCH LAB, arnaud@simula.no
[5] Ecole Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise, Samovar, catherine.dubois@ensiie.fr

Abstract. In program verification, constraint-based random testing is a powerful technique which aims at generating random test cases that satisfy functional properties of a program. However, on recursive constrained data-structures (e.g., sorted lists, binary search trees, quadtrees), and, more generally, when the structures are highly constrained, generating uniformly distributed inputs is difficult. In this paper, we present Testify: a framework in which users can define algebraic data-types decorated with high-level constraints. These constraints are interpreted as membership predicates that restrict the set of inhabitants of the type. From these definitions, Testify automatically synthesises a partial specification of the program so that no function produces a value that violates the constraints (*e.g.* a binary search tree where nodes are improperly inserted). Our framework augments the original program with tests that check such properties. To achieve that, we automatically produce uniform random samplers that generate values which satisfy the constraints, and verifies the validity of the outputs of the tested functions. By generating the shape of a recursive data-structure using *Boltzmann sampling* and generating evenly distributed finite domain variable values using constraint solving, our framework guarantees size-constrained uniform sampling of test cases. We provide use-cases of our framework on several key data structures that are of practical relevance for developers. Experiments show encouraging results.

* this research was partially supported by the ANR PPS project ANR-19-CE48-0014 and the "DYNNET" project, co-funded by the Normandy County Council and the European Union (ERDF-ESF 2014-2020).

1 Introduction

Software Testing is one of the most widespread program verification techniques, and is also one of the most practical to implement. One interesting instance of it is Property-based Testing (PBT), where programs are tested by generating random inputs and results of the output are compared against software specifications. This technique has been extensively studied, for testing correctness [20,31], exhaustiveness [34], complexity [11] etc. However, this technique requires the developer to manually write the tests, that is the properties to be checked and the random generators. The latters can be particularly complicated to design, especially in the case of complex and constrained algebraic data structures.

In this field, *constraint-based random testing* [23] (commonly used in PBT [26,13]) is a powerful technique which aims at generating random test cases that satisfy functional properties of a program under test. By specifying a property that a program has to satisfy and by using uniformly-distributed inputs generators, it is possible to uncover subtle robustness faults that may be not be discovered otherwise. For instance, [1] explored the usage of PBT for testing a steam boiler, [28] explored its usage for wireless sensor network applications. It is worth noticing that generating inputs according to a uniform probability distribution is crucial to ensure that all the distinct program behaviours have the same chance to be triggered, even those which are the most constrained. The technique has been successfully applied in the field of unit testing for imperative programs [22] as well as various programming languages including Haskell [15], Prolog [3] and proof-assistant methodologies and tools such as Coq [32] or Isabelle [10]. Sampling constraint systems solutions according to a uniform distribution is a well-known difficult problem. Initially studied in the context of hardware testing [30], the problem has been studied in [21] and more extensively in [2]. Other random generation schemes are either not uniform, or very slow *e.g.* rejection sampling is generally uniform by construction, but fits very poorly with generation under constraints.

Recently, in [38] the authors introduced an automated framework capable of providing tests for functions that manipulate constrained values without requiring manual input from the programmer. The framework introduces a type language, with algebraic data-types, and constrained types *i.e.* types augmented with a membership predicate that is used to filter invalid representations. To verify that a function does not create invalid representations, the authors opted for a random testing approach. The main interest of the framework is that both the generators and the specifications are automatically extracted from the constraints specified by the user, which greatly alleviate the user's workload. Generators are uniform random value samplers used to provide input for functions, and specifications that are predicates that verify that a given value satisfies the constraint attached to its type, are used to check whether a function's output violates the constraint or not. Their tool is implemented as a pre-processor for OCaml programs, *i.e.* before compiling, programs are rewritten into augmented programs where a test suite has been added. During the pre-processing step, from each constrained type declaration τ is extracted a CSP p. Then, each p is solved only once, that is to say that a characterisation of the set of solutions of p, called coverage, is calculated. Each coverage is then compiled into code which uniformly generates solutions which are then converted back into values of the type τ. However, to be able to solve a CSP only once per constrained type, the authors limit themselves to types involving a fixed number of numerical atoms (*e.g.* tuples), which automatically excludes recursive types. This makes it impractical as, for instance, in OCaml, real-world programs rely heavily on recursive data-types (lists, trees, sets, etc.).

This paper investigates the automatic synthesis of uniform pseudo-random generators, as in [38], but for recursive constrained types.

1.1 Contributions

- A programmable method to restrict the values a recursive type can take.
- An algorithm that uses Boltzmann generation and constraint solving to automatically derive uniform generators for recursive constrained types.
- An experimental study of the performances of our technique.

1.2 Outline

This paper is organised as follows: Sec.2 presents use cases of our methods on some examples of realistic code. Sec.3 defines our solving technique which mixes Boltzmann generation and global constraint solving. Sec.4 recalls some elementary notions about Boltzmann sampling and details some specifics about our use-case. Sec.5 presents our prototype and gives some details about its functioning, current capabilities and restrictions. We also give some details about our implementation and measure experimentally the performances of the generators we derive for recursive constrained types. Sec.6 describes some related work. Finally, Sec.7 summarises our work and discusses possible future improvements.

2 A Declarative Programming Approach

We propose a testing framework that allows programmers to specify constraints on recursive data structures. From these constraints, the framework extracts a Constraint Satisfaction Problem (CSP) which is solved in such a way that uniform random instances (i.e., test cases) are generated. These instances are then used for testing functions in order to find defects.

2.1 Preliminaries

A pseudo-random generator g for an algebraic data-type τ is a function g of type $\mathcal{S} \rightarrow \tau$. Here, $\mathcal{S}$ is the random state used by the pseudo random number generator. A constrained type is a pair $\langle \tau, p \rangle$, with τ an algebraic data-type and $p : \tau \rightarrow bool$ a predicate over values of type τ. The set of its inhabitants is defined as $\{t \in \tau \mid p(t) = true\}$. A pseudo-random generator g for a constrained type $\langle \tau, p \rangle$ is a function $g : \mathcal{S} \rightarrow \tau$ s.t $\forall s \in \mathcal{S}, p(g(s)) = true$.

Here, we face two main challenges for automating random testing of recursive data-types. First, we have to equip the developer with convenient means for specifying constraints attached to a given data-type. For example, we want to express that a list of integers is sorted or that a tree is a binary search tree (i.e., the left child node value is always smaller than the right one). Second, building an uniform random value generator for constrained recursive data-types is highly challenging. Recursive types can dynamically grow to an arbitrarily large size and, deriving generators for such types requires the resolution of a complex constraint system. In particular, we have to manage CSPs with an a-priori unknown number of variables and constraints. The grammar of Ocaml types and constraints annotations are given in Figure 7. In the following, we give two illustrative examples.

2.2 Example 1: Inserting an element into a set of integers

Let start with `list`, a recursive data-type associated to lists of integers, for which a possible type declaration is given in Fig.1. Using `list` to specify a

```
type list = Empty | Cons of int * list
```

Fig. 1: OCaml type declaration of lists of integers

Set data structure can easily be done using Testify, by using the annotation `[@@satisfying _]` and the `[alldiff]` constraint as illustrated by Fig. 2.

```
type uniquelist =
  | Empty
  | Cons of int * uniquelist [@@satisfying alldiff]
```

Fig. 2: OCaml type declaration of sets of integers using lists

We can automatically test the functions that manipulate instances of the `uniquelist` type by checking if they break the properties attached to it. For that, we have to define a generator and a specification for the corresponding type. To randomly generate instances, we first draw at random an instance of size n using Boltzmann generation (see Sec.4), then we build a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ containing n finite domain variables and solve it using Path-oriented Random Testing (PRT) (see Sec.3) and eventually we build a random generator g able to produce uniformly distributed sets of size n.

A key aspect of Testify is based on the usage of *global constraints*, which are arithmetic-logic constraints holding over a non-fixed number of variables. In the example of Fig.2, we translate the declaration `[alldiff]` into a `all_-different` global constraint implementation and used it to generate uniformly distributed solutions that can be used to polulate test cases. For other recursive data-types, we use combination of multiple global constraints and arithmetic constraints. Possible recursive data-types that can be implemented and tested in our framework include functions that generate and manipulate (un-)ordered lists and sets, trees, binary search trees, quadtrees, etc.

Fig.3 shows an example of a function which implements the insertion of an element within a set of integers and the code that is automatically generated for the testing of this function[6].

```
let rec add (x:int) (l:uniquelist) : uniquelist =
  match l with
  | Empty -> Cons(x,Empty)
  | Cons(h,tl) -> if x <> h then Cons(h,(add x tl) else l)

(* generated code*)
let add_test () =
  let size = Random.int () in let rand_x = Random.int () in
  let rand_l = unique_list size in
  assert (alldiff_checker (add rand_x rand_l))
```

Fig. 3: Insertion of an element into a set, and the generated corresponding test

Here, testing the function function means verifying that every output produced is indeed sorted (`assert (alldiff_checker (add rand_x rand_l))`. Note that we have used the return type annotation to automatically derive a (partial) specification for the function, but the generator we automatically synthesise can also be used to test any hand-written specification.

2.3 Example 2: Binary Search trees

Binary Search Trees (BST) are binary trees that additionally satisfy the following constraint: the key in each node is greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree. Stated differently, the keys in the tree must be in increasing order in a depth-first search traversal, in infix order. From this observation, we propose, using our framework, a possible OCaml declaration for BSTs illustrated in Fig.4.

```
type bst =
  | Node of bst * (int[@collect]) * bst
  | Leaf [@@satisfying fun x -> increasing x)]
```

Fig. 4: Testify type annotation for binary search trees

Rather than defining global constraint for all user-declared data-types, we break the problem in two parts. On the one hand, we define or reuse known global constraints for lists, and on the other hand we define a way to browse data

[6] The predicate `alldiff_checker` checks that the result list does not contain duplicates. It should not be mixed with the version of `alldiff` used in the type declaration which is used to generate randomly distributed solutions of that constraint

structures, in a certain order, by collecting the components that are subject to a global constraint. This is done using the `(int[@collect])` annotation.

Also, the order in which the structure is explored is crucial as it determines the order in which the variables will be passed to the global constraint. By default, a depth first order is assumed. For constructors with several arguments (*e.g.* `Node`), and for tuples, the order in which the traversal is made is mapped on the declaration order of the tuple component, that is in traversal order. Fig.5 shows the code generated that traverses the tree.

```
let rec collect = function
| Node (a, b, c) ->
   List.flatten [collect a; Collect.int b; collect c]
| Leaf -> []
```

Fig. 5: Generated collector for binary trees

Here, the primitive `Collect.int` is a primitive of our framework that takes an integer and builds the singleton list with this element. This way, we first visit the left sub-tree, the root and the right sub-tree. Using pre-order or post-order would give different results. This means that the constructor `Node` must be declared in the above order and, for example, the following would be invalid: `Node of (int[@collect]) * binary_tree * binary_tree`. However, this restriction can easily be lifted by providing an annotation which would allow the programmer to explicitly specify the traversal order. Similarly, a global annotation `[@bfs]` (resp. `[@dfs]`) could be used to specify that the structure must be traversed using a breadth first search (resp. depth first search). This will be studied in future work.

3 Constrained Type Solving

A *Constraint Satisfaction Problem* (CSP) is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a function associating a finite domain (considered here as a subset of $\mathcal{Z}$ without any loss of generality) to every variable and $\mathcal{C}$ is a set of constraints, each of them being $< var(c), rel(c) >$, where $var(c)$ is a tuple of variables $(X_{i_1}, .., X_{i_r})$ called the scope of c, and $rel(c)$ is a relation between these variables, i.e., $rel(c) \subseteq \prod_{k=1}^{r} D(X_{i_k})$. For each constraint c, the tuples of $rel(c)$ indicate the allowed combinations of value assignments for the variables in $var(c)$. Given a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, an *assignment* is a vector $(d_1, .., d_n)$, which associates to each variable $X_i \in \mathcal{X}$ a corresponding domain value $d_i \in D(X_i)$. An assignment satisfies a constraint c if the projection of $\mathcal{X}$ onto $var(c)$ is a member of $rel(c)$. The set of all satisfying assignments is called the solution set, noted $sol(\mathcal{C})$. A constraint c is said to be *satisfiable* if it contains at least one satisfying assignment, it is inconsistent otherwise. A CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is satisfiable if it contains at least one assignment which satisfies all its constraints (i.e., $sol(\mathcal{C}) \neq \emptyset$). A global constraint is an extension of CSPs with relations concerning a non-fixed number of variables. For instance, the *sort(Xs, Ys)* global constraint

[29] takes as inputs two lists of n finite domain variables Xs, Ys (where n is unknown) and states that for each satisfying assignment $(d_1, .., d_n, d'_1, .., d'_n)$ of the constraint, we have $\forall j, \exists i$ s.t. $d'_j = \sigma(d_i)$ and $d'_1 \leq .. \leq d'_n$, where σ is a permutation of $[1..n]$. Filtering a global constraint $c(X_1, .., X_n)$ with the *bound-consistency* local filtering property means to find D' such that for all i, the extrema values of $D'(X_i)$ are parts of satisfying assignments of c.

3.1 Path-Oriented Random Testing

Path-oriented Random Testing (PRT) is basically a divide-and-conquer algorithm, introduced in [22], which aims to generate a uniformly distributed subset of solutions of a CSP. Starting from an initial filtering step result, the general idea is to fairly divide the resulting search space into boxes of equal volumes and, after having discarded inconsistent boxes using constraint refutation, to draw at random satisfying assignments.

More precisely, applying constraint filtering results in domains that can be over-approximated by a larger box (i.e., an hyper-cuboid) that contains all the filtered domains. Based on an external division parameter k, PRT then fairly divides the box into k subdomains of *equal* volume. When a subdomain cannot be divided according to the division parameter k, then it is simply extended until its area can be divided. The iteration of the process leads to a fair partition of the search space into k^n subdomains where n is the number of variables of the CSP. Then constraint refutation can be used to discard (some) subdomains which are inconsistent with the rest of the CSP. As all subdomains have the same volume, it becomes possible to sample first the remaining subdomains and then, second, to randomly draw values from these subdomains. Note that, when all the subdomains are shown to be inconsistent, then the CSP is shown to be inconsistent. This contrasts with reject-based methods which will trigger assignment candidates and will reject them afterwards, without terminating in a reasonable amount of time.

Input: CSP:$(\mathcal{X}, \mathcal{D}, \mathcal{C}), k, N$: #Sol. - **Output:** $t_1, .., t_N$ or $\emptyset$ (Inconsistent)

```
D' := boxfilter_bc(X, D, C); (H_1, .., H_p) := Fairly_Divide(D', k); T := ∅;
while N > 0 and p ≠ 0 do
    Pick up uniformly H at random from H_1, .., H_p;
    if H is inconsistent w.r.t. C then
        remove H from H_1, .., H_p
    else
        Pick up uniformly t at random from H and remove it;
        if C is satisfied by t then
            add t to T ; N := N − 1;
        end
    end
end
return T;
```

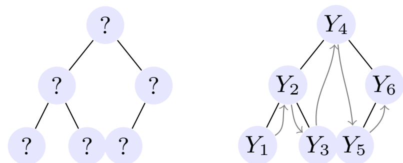
Algorithm 1: Path-Oriented Random Testing adapted from [22] to the uniform random generation of N solutions of a CSP

The PRT algorithm, adapted from [22] to the case of CSP solution sampling, is given in Figure 1. It takes as inputs a CSP, a division parameter k, and N a non-negative integer. Here, we make the hypothesis that, if the CSP is consistent, it contains more than N solutions. The algorithm outputs a sequence of N uniformly distributed random assignments which satisfy the CSP. If the CSP is unsatisfiable, then PRT returns $\emptyset$. After an initial filtering step using bound-consistency, the algorithm partitions the resulting surrounding box in subdomains of equal volume (`Fairly_Divide` function). Then, for each locally consistent subdomain H in the partition, value assignments are randomly selected and checked against the constraints of the CSP. Those which do not satisfy the constraints are simply rejected. As shown in [22], this process ensures the uniform generation of tuples in the solution space.

3.2 Extension with Global Constraints

Handling global constraints is a natural extension of PRT as it allows us to handle recursive constrained data-types. As the shape of the data structure is unknown at constraint generation time, the number of variables to be handled is also unknown in the general case. Thus, using global constraints in this context is particularly useful as it allows us to avoid the decomposition of a global constraint into the conjunction of several simpler constraints. This results in both a stronger and faster pruning. In order to handle recursive constrained data-types, we had to provide a dedicated interface for accessing the deductions from global constraint solving. To facilitate the access to global constraints, we created an API which provides results of PRT over different global constraint combinations. The API provides access to predicates such as `increasing_list(+int LEN,+int GRAIN,-var L)` in which `L` is instantiated to a list of `LEN` uniformly distributed random integers ranked in increasing order, and the random generator is initialised with `GRAIN`. Optionally, the predicate can be called with domain constraints in order to constrain the returned list of values in specific subdomains. Other similar predicates are provided as part of the API, namely `increasing_strict_list/3 (+int LEN,+int GRAIN,-var L)` which returns a list of strictly increasing integers; `decreasing_list/3` (resp. `decreasing_strict_list/3`) which provides a list of integers in (resp. strict) decreasing order or else `alldiff_list/3` which returns a list of uniformly distributed random distinct integers. PRT can also be used in combination with any available global constraint and arithmetico-logic constraint. The following example, given in Fig.6 illustrates how PRT is used in this respect.

In this example, PRT is used with one global constraint, namely $sort(Xs, Ys)$, and some domain and arithmetic constraints to populate a constrained binary search tree (BST) of size 6. In this example, the shape of the BST is unknown and some constraints hold over the keys: the domain of the key-variables is constrained (from an externally specified source), *e.g.*, key $X_1 \in -2..8$, key $X_2 \in -3..5$, etc. and any key of the BST corresponds to the sum of its children (if any), *e.g.*, $Y_{father} = Y_{child_l} + Y_{child_r}$. Note that the keys have to be set in increasing order to correspond to a valid BST. Note also that we ignore in

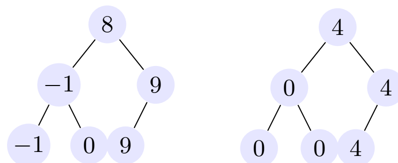


(a) Shape generation

(b) Depth-first walk

$$X_1 \in -2..8, X_2 \in -3..5, X_3 \in -3..10,$$
$$X_4 \in -1..9, X_5 \in 0..7, X_6 \in 0..8,$$
$$\text{sort}((X_1, .., X_6), (Y_1, .., Y_6)),$$
$$Y_2 = Y_1 + Y_3, Y_5 = Y_6, Y_4 = Y_2 + Y_6$$

(c) Generating the corresponding CSP



(d) PRT first sol.

(e) PRT second sol.

(f) PRT third sol.

Fig. 6: Generation of a constrained BST of size 6. Division parameter=2, length of seq.=3, 60 subdomains over 64 have been discarded after the first filtering.

MP: Il est un peu problématique cet exemple je trouve, pour deux raisons.
(1) Il ne correspond pas au type des BST qu'on a donné plus haut où on a que des clefs et pas de valeurs, c'est pour ça que le reviewer 1 râle et ne comprends pas ce que les X_i viennent faire là.
(2) Je comprends qu'avec PRT on peut faire des trucs sophistiqués, mais dans testify on n'a aucun moyen de spécifier ce genre de contraintes non ? Ça ne me paraît pas stratégique de parler de contraintes qu'on se sait pas exprimer.

which order will the keys be positioned in the tree. The first step of our method corresponds to the generation of a uniformly distributed random shape of the BST (Fig.6(a)) using the Boltzmann method, described in Sec.4. Then, a depth-first walk along the tree assigns variable identifiers to the nodes and collects the constraints that must hold over the constrained data structure (Fig.6(b)). The generated CSP (Fig.6(c)) can then be solved by using PRT, which generates, in this example, three uniformly distributed random solutions (Fig.6(d)(e)(f)). It is worth noticing that other uniform random solutions sampling methods such as [21,36] could have been used in this context. PRT was chosen because of its availability and simplicity. However, non-uniform random sampling such as a simple heuristic selecting at random variable and values to be enumerated first would not have been appropriate in this context as the goal was to test the robustness of user-defined functions in functional programming.

4 Boltzmann Sampling

The Boltzmann method was introduced in [17] as an algorithmic method to derive efficient sampler from *combinatorial classes*. Combinatorial classes are just sets of discrete structures with a size (a non-negative integer) and such that the number of structures having the same size is finite. For example, the binary trees whose size is the number of leafs is a combinatorial class, but binary trees whose size is the length of leftmost branch is not because the number of binary trees with a leftmost branch of fixed length k is infinite. We briefly present the method here and refers the reader to [17] for more details.

$\langle decl \rangle$ ::= 'type' $\langle type\ identifier \rangle$ '=' $\langle type \rangle$ — type declaration
    {'[@@satisfying' $\langle constraints \rangle$ ']'} — Testify's annotation

$\langle type \rangle$ ::= $\langle coretype \rangle$
    | $\langle sumtype \rangle$

$\langle coretype \rangle$ ::= 'int' | 'float' | 'char' | ... — basic types
    | $\langle coretype \rangle$ {'*' $\langle coretype \rangle$ } — product
    | $\langle type\ identifier \rangle$

$\langle sumtype \rangle$ ::= $\langle constructor\ identifier \rangle$ {'[@collect]'} 'of' $\langle coretype \rangle$
    | $\langle sumtype \rangle$ {'|' $\langle sumtype \rangle$ }

$\langle constraints \rangle$ ::= 'alldiff' | 'increasing' | 'decreasing' — SICStus global constraints
    | $\langle arith \rangle$ — arithmetic constraints like in [38, Fig. 1]
    | $\langle constraints \rangle$ {'&&' $\langle constraints \rangle$}

Fig. 7: Syntax of OCaml algebraic data-types (ADT) with Testify's annotation

In the context of that paper (similarly to [12]), that method directly translates into an automatic way to derive a uniform random generator of terms for the type language whose syntax is given in Fig.7. In our case, the produced generators only generate a *shape* of tree structure in a first step and the content of this shape is provided in a second step by a constraint solver which makes sure to fill the shape with values that satisfy the specified constraints. For each constrained recursive type declaration, we must therefore generate a glue function between the shapes generated by the Boltzmann sampling method and the solutions returned by the solver used. This function is illustrated in the case of binary search trees in Fig.8

```
let rec fill_binary_tree shape solutions =
  match shape with
  | Label ("Node", [x1; x2; x3]) ->
      let x1 = fill_binary_tree x1 solutions in
      let x2 = Testify_runtime.to_int x2 solutions in
      let x3 = fill_binary_tree x3 solutions in
      Node (x1, x2, x3)
  | Label ("Leaf", []) -> Leaf)
```

Fig. 8: Generated function for filling the shapes for binary search trees.

Here, we consider types as sets of terms (the inhabitants of the type) whose size is the number of `[@collect]` values they contain. For example, using `type binary_tree` of Sec.2, the term `Node(Node(Leaf, 3, Leaf), 25, Leaf)` has size 2.

In the following, we denote $\Gamma\mathcal{A}_x$ a Boltzmann sampler of parameter x for the set $\mathcal{A}$. Such sampler produces an object $\gamma \in \mathcal{A}$ with a probability $\frac{x^{|\gamma|}}{A(x)}$ where $|\gamma|$ is the size of γ and $A(x)$ is a normalizing factor called *generating series*[7]. Note that objects of the same size have the same probability to be drawn.

The second interest of Boltzmann samplers is that they compose well with sum, product and substitution *i.e.* the constructors of ADTs. Fig.9 shows the derivation of such samplers. At the end of the generating process, the object

```
type t = a * b (* a and b are 2 types previously defined *)
let gen_t x = gen_a x, gen_b x

type u = A of a | B of b
let gen_u x =
  if random () < A(x)/(A(x)+B(x))
  then gen_a x else gen_b x

type alst = Nil | Cons of a * alst (* alst(z) = 1 + z · alst(z) *)

let rec gen_alst x : alst =
  if random () < 1/(1 − A(x))
  then Nil else Cons(gen_a x, gen_aList A(x))
```

Fig. 9: Sampler derivation using Boltzmann

drawn has a random size, but we see in the previous code that the choice of the parameter x influences the size. Note that we can precisely and efficiently compute x to target a size (see [6] or [33] for the details).

Still, the size is random. The last ingredient is to choose a parameter ϵ (which does not depend of the targeted size n) and keep only objects of size between $n - \epsilon$ and $n + epsilon$. Thus, the size of the object is kept up to date during the generation and the generation is stopped if that size exceeds the upper bound $n+\epsilon$. At the end, the object may be smaller than $n-\epsilon$ in which case it is rejected too. However, the theory (see [17]) guarantees that the rejections cost remain relatively low, *i.e.* the cumulated size of objects sampled to obtain an object of size in the interval $[n - \epsilon, n + \epsilon]$ is in $\mathcal{O}(n)$. So the complexity of the overall process is linear in the size of the generated object.

An important point to mention is the case of polymorphic types. From a theoretical point of view they fit in the framework. But from a practical point of view it is hard to sample a "polymorphic value". To deal with that limitation, the Boltzmann samplers are instantiated only for concrete types *e.g.* not for `'a list` but for `int list`.

[7] The generating series $A(z)$ of a combinatorial class $\mathcal{A}$ is defined by $A(z) = \sum_{\gamma\in\mathcal{A}} z^{|\gamma|}$

5 Implementation and Experiments

We have implemented the work presented in the previous sections in a tool available at the url `https://github.com/ghilesZ/Testify`. Our implementation relies on several state-of-the-art tools. The derivation of OCaml code from annotated OCaml source files is done using the *ppx* framework, as in [37,5], which is a form of generic programming [24]. Pre-processors using *ppx* are applied to source files before passing them on to the compiler. They can be seen as self-maps over abstract syntax trees. In our case, the source files are traversed to find OCaml type declarations and derive their associated generators. These generators are then used to provide inputs for the functions that must be tested. We have implemented the techniques presented here for the global constraints that we have been able to identify in real data structures (BSTs, Sets, etc) namely *alldiff*, *increasing* and *decreasing* (both strict and large versions). Note that to extend our implementation, *i.e.* add a global constraint,, it is sufficient to add to the constraint solver a propagator for the said global constraint, as both the step of traversing the structure and the random generation procedure presented in section 3 being common to all types.

The work done by Testify is divided into two phases: the first is the preprocessing phase during which our tool collects some information on the types needed to build the generators. The second is the testing phase, where the generated code is executed to produce inputs for the functions under test. Note that the pre-processing phase is performed only once while the testing phase can be triggered multiple times, each time one needs to run the tests.

We distinguish four kinds of types, for which we provide four different synthesis techniques:

- For non recursive unconstrained types (*e.g.* `int`, `float * (int * int)` ...) we determine at pre-processing time the function to be used as a generator. For that, we rely on the qcheck [15] library, which provides the primitives for building and composing generators.
- For non recursive constrained types (*e.g.* `int[@satisfying fun x -> x >=0]`), we extract a single CSP which is solved once, still at pre-processing time. From this resolution is extracted a code that draws uniformly solutions of this CSP and rebuild from them a value of the corresponding type. This is the method described in [38].
- For recursive unconstrained types (*e.g.* lists, binary trees), we build samplers by using the Arbogen [19] tool. This tools implements the Boltzmann method presented in Sec.4. The tuning of the Boltzmann parameter is done at preprocessing time while the shape generation, and the conversion of this shape to a value of the targeted type is done at testing time.
- Finally, for recursive constrained types (*e.g.* sorted lists, binary search trees), the previous techniques are mixed together to produce efficient generators: first, a targeted size n is drawn, then, a shape of size n is sampled. We then browse the generated shape, collecting constrained values to build a CSP as explained in Sec.3. This CSP is then fed to the SICStus Prolog [4]

solver, which builds from it a generator using the *PRT* library [22]. Finally we put together shapes and constrained values. All of these steps are made at testing time, that is every time we have to generate a value we must solve a CSP. This is arguably the bottleneck of our architecture, but experiments still demonstrate the usability of our method.

5.1 Experiments

In this section, we focus on the performance of our automatically derived generators. We measure the generation times (in seconds) obtained with our method for different constrained recursive types and by varying the size of the generated structure. The types we are interested in are: lists sorted in ascending order, association lists with unique keys, lists of pairs in ascending order $((x, y) \leq (x', y) \Leftrightarrow x \leq x' \wedge y \leq y')$, binary search trees (unbalanced), functional maps (key-value stores as binary search trees) and quadtrees. These types are among the most frequents in the literature, and they only involve numerical constraints, which Testify is able to manage.

Types	**Targeted**	**Average**	**$\sharp$Objects**	**time**
increasing_list	10	8.50	2889	0.020
	100	93.95	13691	0.004
	1000	948.57	17763	0.003
	10000	9392.45	79	0.757
assoc_list	10	8.49	2534	0.023
	100	93.96	11949	0.005
	1000	947.87	13660	0.004
	10000	9406.92	76	0.786
bicollect	10	6.99	2492	0.024
	100	93.04	6418	0.009
	1000	947.73	16048	0.003
	10000	9456.85	1596	0.037
binary_tree	10	9.00	238690	0.001
	100	94.35	21214	0.001
	1000	948.00	3416	0.006
	10000	9740.00	1500	0.040
map	10	9.00	238690	0.001
	100	94.37	21208	0.001
	1000	947.08	3423	0.006
	10000	9047.00	1276	0.047
quad_tree	10	8.00	3590507	0.001
	100	93.88	228357	0.001
	1000	947.79	23548	0.002
	10000	9489.19	2191	0.027

Fig. 10: Generation time per object according to the size of the structure

The experience was to sample as much as possible constrained structures during one minute. The results are shown in Fig.10. For each type we report the size of the terms (number of `[@collect]` values) targeted, the average size of the generated terms, the number of terms sampled and the average time to sample one term. The computer running the experiments has an Intel Core i7-6700 CPU cadenced at 3.40GHz with 8 GB of RAM.

As expected, at least for the tree-like types, we observe that the complexity is quite linear in the size of the sampled terms: the Boltzmann method keeps its promises and the use of a constraint solver proves to be fast enough to be used in our context. For most of these structures we manage to generate several hundred values per second, up to a certain structure size. These results prove the relevance of our method in the context of testing, as it can allow the user to fine-tune the generators to decide whether he wants to test his functions on several small structures and/or a few large ones. However, we may note that sampling of lists is much slower than sampling of trees. This is due to the fact that the Boltzmann method is not tailored for regular languages (such as list). It would probably be more efficient to use specialised algorithms for regular languages such as the one of [8].

6 Related Work

In this section we focus on related work dealing with constraint-based generation techniques. Constraint-based generation of test data has been exploited in white-box testing to produce inputs that will follow some execution paths, as well as in functional testing to generate constrained inputs. In [35], Senni applies constraint logic programming to systematically develop generators of structurally complex test data, e.g. red-black trees, in the context of Bounded-Exhaustive Testing.

PBT, as exemplified by Quickcheck for Haskell, has been adapted to many programming languages but also to proof assistants to test conjectures before proving them, e.g. [18,10,32,13]. In [18] restricted classes of indexed families of types are provided with surjective generators. In [13], the authors propose the FocalTest framework for testing - conditional - conjectures about functional programs and for automatically generating constrained values. In this work, CP global constraints are not used and thus FocalTest does not take benefit from the corresponding efficient filtering ad hoc procedures.
In the context of PBT of Erlang programs, De Angelis *et al* propose in [16] an approach to automatically derive generators of values that satisfy a given specification. Generation is performed via symbolic execution of the specification using constraint logic programming. A difference between their approach and ours is that we craft a suitable representation of a given type at static time, which is then compiled into an efficient generator. In [16], generators are built at execution time, while testing, which ultimately leads to a slower generation.
The Coq plugin QuickChick helps to test Coq conjectures as soon as involved properties are executable. It allows the automatic synthesis of random generators for algebraic data-types, recursive or not, and also the definition of simple inductive properties, e.g. a property specifying binary search trees whose elements are between two bounds, to be turned into random generators of constrained values [25]. The approach is narrowing-based, like in [14]. Such a binary

tree is built lazily while solving the constraints found in the inductive property while in Testify, the shape of the data structure is randomly chosen and then its elements are obtained by solving constraints. This tool comes with different primitives or mechanisms allowing for some flexibility in the distribution of the sampled values. For example the user can annotate the constructors of an inductive data-type with weights that are used when automatically deriving generators. Furthermore, it also produces proofs of the generators correctness. In [12], the authors adapt a Boltzmann model for random generation of OCaml algebraic data-types, possibly recursive, but not constrained. Generators are automatically derived from type declarations. In [14], Claessen et al. propose an algorithm that, from a data-type definition, a constraint defined as a Boolean function and a test data size, produces random constrained values with a uniform distribution. However the authors show that this uniformity has a high cost. They combine this perfect generator with a more efficient one based on backtracking. Limiting the class of constraints and combining it with an efficient solving process, Testify can generate constrained values with a uniform distribution in a reasonable time. Some work focus on the enumeration or sampling of combinatorial structures, like lambda-terms, using Boltzmann samplers [27], Prolog mechanisms [9] or both [7]. These approaches are dedicated to objects of recursive algebraic data-types with complex constraints, like typed lambda-terms, closed lambda-terms, linear lambda-terms, etc. This kind of constraints is out of reach of our tool whose objective is not only to generate constrained values but also to provide the programmer with syntactic facilities to specify them.

7 Conclusion

We have proposed in this paper a technique based on declarative programming, to derive generators of random and uniform values for constrained recursive types. We have proposed a small description language for recursive structure traversal which allows us to build a custom CSP for each term to be generated. The code we generate is efficient, and outperforms a naive generation technique based on rejection, and allows us to generate large recursive structures quickly. Starting from the constraints attached to a type, we first sample the shape of the value to generate and then build a CSP that encodes the valid representations of the terms that have this shape. Then, our tool uses the SICStus Prolog constraint solver to filter invalid representations and produce a uniform solution sampler. Our technique is integrated into the Testify framework, which embeds these generators within a fully automatic test system. The generators derived by our framework are fast enough to allow the user to run tests each time he compiles his code. This would allow him to be able to detect bugs very quickly and fix them before they become potentially harmful. However, we still have a lot of work to do to improve Testify. For example, we can extend the constraint language to be able to handle types with shape constraints (*e.g.* balanced trees). This would require adapting the Boltzmann technique to random sampling of tree structures under constraints. Also, when dealing with a functional language, functions as values cannot be avoided: it will be necessary to have techniques for the derivation of generators for functions, and explore what kind of constrained functions (monotonic, bijective functions, etc.) appear in practice in programs. Moreover, in this paper we have only studied *tree-like* recursive data-structures. Some structures do not fit into this framework (*e.g.* graphs, doubly linked lists) and it would be interesting to see to what extent our methods adapt to these structures. Also, our current implementation tests functions by generating any random input, disregarding their body. This is naturally an important point of improvement. For example, one could imagine a static analysis of the body of the function, to conduct the input generation more precisely, and find bugs faster. Finally, our framework targets OCaml but the methods developed in this paper can be adapted to most programming languages and proof assistants.

References

1. Olfa Abdellatif-Kaddour, Pascale Thévenod-Fosse, and Hélène Waeselynck. Property-oriented testing: A strategy for exploring dangerous scenarios. In Gary B. Lamont, Hisham Haddad, George A. Papadopoulos, and Brajendra Panda, editors, *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC), March 9-12, 2003, Melbourne, FL, USA*, pages 1128–1134. ACM, 2003.
2. Thomas E. Allen, Judy Goldsmith, Hayden Elizabeth Justice, Nicholas Mattei, and Kayla Raines. Uniform random generation and dominance testing for cp-nets. *J. Artif. Intell. Res.*, 59:771–813, 2017.
3. Cláudio Amaral, Mário Florido, and Vítor Santos Costa. PrologCheck - property-based testing in Prolog. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2014.
4. Johan Andersson, Stefan Andersson, Kent Boortz, Mats Carlsson, Hans Nilsson, Thomas Sjöland, and Johan Widén. SICStus Prolog user"s manual. 1993.
5. Florent Balestrieri and Michel Mauny. Generic programming in OCaml. *Electronic Proceedings in Theoretical Computer Science*, 285:59–100, 12 2018.
6. Maciej Bendkowski, Olivier Bodini, and Sergey Dovgal. Polynomial tuning of multiparametric combinatorial samplers. In Markus E. Nebel and Stephan G. Wagner, editors, *Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2018, New Orleans, LA, USA, January 8-9, 2018*, pages 92–106. SIAM, 2018.
7. Maciej Bendkowski, Katarzyna Grygiel, and Paul Tarau. Random generation of closed simply typed λ-terms: A synergy between logic programming and Boltzmann samplers. *Theory Pract. Log. Program.*, 18(1):97–119, 2018.
8. Olivier Bernardi and Omer Giménez. A linear algorithm for the random sampling from regular languages. *Algorithmica*, 62(1–2):130–145, feb 2012.
9. Olivier Bodini and Paul Tarau. On uniquely closable and uniquely typable skeletons of lambda terms. In Fabio Fioravanti and John P. Gallagher, editors, *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 252–268. Springer, 2017.

10. Lukas Bulwahn. The new quickcheck for isabelle - random, exhaustive and symbolic testing under one roof. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2012.
11. Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 463–473, New York, NY, USA, 2009. Association for Computing Machinery.
12. Benjamin Canou and Alexis Darrasse. Fast and sound random generation for automated testing and benchmarking in objective caml. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, ML '09, page 61–70, New York, NY, USA, 2009. Association for Computing Machinery.
13. Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. Focaltest: A constraint programming approach for property-based testing. In José Cordeiro, Maria Virvou, and Boris Shishkov, editors, *Software and Data Technologies - 5th International Conference, ICSOFT 2010, Athens, Greece, July 22-24, 2010. Revised Selected Papers*, volume 170 of *Communications in Computer and Information Science*, pages 140–155. Springer, 2010.
14. Koen Claessen, Jonas Duregård, and Michał H Pałka. Generating constrained random data with uniform distribution. *Journal of functional programming*, 25, 2015.
15. Simon Cruanes. *QuickCheck inspired property-based testing for OCaml*. `https://github.com/c-cube/qcheck`.
16. Emanuele De Angelis, Fabio Fioravanti, Adrian Palacios, Alberto Pettorossi, and Maurizio Proietti. *Property-Based Test Case Generators for Free*, pages 186–206. 09 2019.
17. Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13(4-5):577–625, 2004.
18. Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. volume 2758, 06 2003.
19. Frederic Peschanski et al. *Arbogen, a fast uniform random generator of tree structures*. `https://https://github.com/fredokun/arbogen`.
20. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.
21. Vibhav Gogate and Rina Dechter. A new algorithm for sampling CSP solutions uniformly at random. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 711–715. Springer, 2006.
22. Arnaud Gotlieb and Matthieu Petit. A uniform random test data generator for path testing. *J. Syst. Softw.*, 83(12):2618–2626, 2010.
23. John Hughes. Quickcheck testing for fun and profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2007.
24. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. *SIGPLAN Not.*, 38(3):26–37, jan 2003.

25. Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL):45:1–45:30, 2018.
26. Sophie Laplante, Richard Lassaigne, Frédéric Magniez, Sylvain Peyronnet, and Michel de Rougemont. Probabilistic abstraction for model checking: An approach based on property testing. *ACM Trans. Comput. Log.*, 8(4):20, 2007.
27. Pierre Lescanne. On counting untyped lambda terms. *Theor. Comput. Sci.*, 474:80–97, 2013.
28. Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. In *Proc. of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-17)*, pages 46–56, 07 2017.
29. Kurt Mehlhorn and Sven Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In Rina Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 306–319. Springer, 2000.
30. Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI Mag.*, 28(3):13–30, 2007.
31. Michal Palka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. *Proceedings - International Conference on Software Engineering*, 01 2011.
32. Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.
33. Carine Pivoteau, Bruno Salvy, and Michele Soria. Algorithms for combinatorial structures: Well-founded systems and Newton iterations. *Journal of Combinatorial Theory, Series A*, 119(8):1711–1773, November 2012.
34. Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, volume 44, pages 37–48, 01 2008.
35. Valerio Senni and Fabio Fioravanti. Generation of test data structures using constraint logic programming. In Achim D. Brucker and Jacques Julliand, editors, *Tests and Proofs - 6th International Conference, TAP@TOOLS 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, volume 7305 of *Lecture Notes in Computer Science*, pages 115–131. Springer, 2012.
36. Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme. Solution sampling with random table constraints. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 56:1–56:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
37. Jeremy Yallop. Practical generic programming in OCaml. pages 83–94, 01 2007.
38. Ghiles Ziat, Matthieu Dien, and Vincent Botbol. Automated Random Testing of Numerical Constrained Types. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 59:1–59:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

A Revisions

A.1 To Reviewer 1

> *The authors state that they deal with testing. However a crucial aspect of testing seems neglected - checking whether the program produces correct output (when run on the test data). The system, as described in l.324-341, does not perform such checking. (BTW, how could we know which results are to be produced by the randomly generated test data?). The example of Fig.3 is an exception, see below.*

GZ-MD proposed answer: It is true that the main issue addressed in present paper is that of automatically obtaining uniform random generators for constrained type. Our focus is set on these types only, and we provide tools to check programs manipulating them:

- automatic synthesis of uniform samplers;
- automatic verification that the invariants *encoded in the types* are satisfied. We have added an explanation of this point in sec 2

These two contributions makes PBT simpler by alleviating the need to write samplers and automatically checking the invariants that we can guess from the types. Any other property a user whishes to check has to be specified.

AG proposed answer: As part of property-based testing where the test oracle is provided by an user-specified property to check, there is a need to construct uniform samplers of algebraic constrained recursive data types. Our paper is focused on this issue and the underlying methodology can thus be applied in various contexts. First, the case where only program crash/no-crash is checked is useful in robustness testing and it requires the automatic synthesis of uniform test inputs. Second, the case where one has a property to satisfy on the expected output of the program is illustrated in our paper by the automatic verification of type-based invariant. Both cases show that the contribution of the paper is a brick that serve the purpose of property-based testing.

> *The authors neglect one of basic principles of testing - that the test data should include border (or extreme) cases. (This is because errors are likely to manifest on such cases.)*

GZ-MD proposed answer: We agree that test data should include border cases, but we consider this to be a complementary approach. Border cases such as `nan`, `min_int`, `max_int`, *etc.* allow to look in places where it is known that bugs like to hide. This usually allows to find more bugs, but we argue that this is not the only place to search. Uniform generation, on the other hand, look everywhere and test the robustness of the program. We believe that you need to two approaches to achieve a good level of confidence in your program.

Il faut intégrer ce discours dans le corps du papier.

AG proposed answer: We agree that test input generation is not a monolithic process and that it should include several complementary approaches. In particular, corner-based test input generation (a.k.a., boundary testing) is crucial to detect subtle faults. However, in property-based testing, several test input generators can be combined and our paper focusses on random sampling of algebraic constrained recursive data types which is an open problem. We understand that this is not clearly stated in the paper and clarified the contribution in the revised submission.

> *I disagree with the claim (l.43-46) that generating inputs according to an uniform probability distribution is crucial to ensure that all the distinct program behaviours have the same chance to be triggered. For instance, it seems obvious that under uniform distribution of the input data various fragments of the program code are executed with various frequency/probability. (Sorry I cannot provide references.) This drawback is not present e.g. in [21], where test data for a given path in the program are generated.*

GZ-MD proposed answer: This is indeed wrong. What we meant here is that all inputs, even those that are heavily constrained have the same chance to be produced. This differs from the usual way to do property based testing where, in order to test a function with a pre-condition (a.k.a. receiving data from a constrained type), you usually generate unconstrained data and filter out the data that does not satisfy the pre-condition. This has two major drawbacks:

- if the constraint rejects a lot of inputs, either you have less tests or it has a high impact on performance;
- this bias the distribution of the inputs towards less constrained data.

This second point is particularly problematic since constrained data is precisely where you need to check that the program does not mess up with your invariants.

AG proposed answer: It is true that uniform sampling over the program input space leads to a non-uniform sampling of the program paths. By performing random sampling over the input space, we ensure that each possible input test data has exactly the same probability to be triggered. That is the crucial property we want to guarantee. Again, it is the state-of-the-art approach used in property-based testing and a crucial requirement of robustness testing. We clarified this issue in the revised submission.

> *It is stated (Conclusion, l.444...) that "We have proposed a small description language for recursive structure traversal". This language is however not explained. It seems to appear in the examples, but the semantics is far from clear.*

> *What is "[@satisfying]" (l.124) and "[@satisfying a]" (Fig. 2) ? What is "[@@satisfying ...]" and "[@collect]" (Fig.4) ?*

In brief, the `[@@foo bar]` and `[@foo bar]` syntax in OCaml are dedicated to annotate parts of the program, where `foo` is an alphanumeric identifier and `bar` is arbitrary OCaml code. They allow us to attach information to any expression, value or type declaration, etc. These annotations are by default ignored by the compiler, but they can be read by compiler extensions / pre-processors (such as the Testify tool presented in the paper), to implement extra features. AG suggested complementary answer: We have clarified the syntax/semantic of the language in the revised submission. MD : Especially, we have added the grammar of constraint annotations to the Figure 7.

> *It looks like the paper deals with producing _test_input_data_that satisfy some given constraints. The example of Fig.3 seems to be the only place where one considers if the _test_results_satisfy the required constraints. However, this still does not mean that a test produces correct results.*

GZ-MD proposed answer (with revised text from AG): As explained before, we focus on test input generation in the context of robustness testing or property-based testing. Our contribution lies in the generation of algebraic recursive data types. For instance, if the type `nat` of non negative integers is defined using a `[@@satisfying fun x -> x >= 0]` annotation and if a function of type `XXX -> nat list` is coded, then Testify will synthesise the following specification for `nat` lists:

```
let rec nat_list_spec xs =
  match xs with
  | [] -> true
  | x :: xs -> (fun x -> x >= 0) x && nat_list_spec xs
```

This specification will then be used to check the output of such a function producing nat lists.

> *The example of 258-278. It is absolutely unclear what is to be tested here. Are we going to test some procedure that checks whether its input is a BST of the chosen kind? No - such test data should result in both positive and negative outcome (and here we generate only BST's of the chosen kind). Are we going to test some procedure producing such BST's? No, we generate here such results, instead of input data for the procedure.*

> *l.263-264 "any key of the BST corresponds to the sum of its two children". This condition cannot be satisfied by a finite tree (it implies that there are no leaves).*

We meant, “its children (if any)”. This has been fixed.

> *l.258... It is absolutely not clear what is the reason of additional constraints ("Yfather = Ychildl + Ychildr" and the domain restrictions on X_i variables). Also, the X_i variables are not related to the search tree.*

MP: cf ma remarque dans la Figure 6. Je ne comprends pas non plus ce que ces contraintes viennent faire là.

The authors state that the proposed technique is "based on declarative programming" (l.442). However it seems to me that it has nothing to do with declarative programming, except for the constraints of CLP. What is used is the type definitions of the programming language dealt with, and code fragments added to programs in this language.

MP: TODO

A.2 To Reviewer 2

Sect. 3 is more problematic. Algorithm 1, which is the core of the approach, lacks a bit of explanation, e.g.: why p subdomains are returned instead of k? What is boxfilter_bc? (bound-consistency is mentioned but never explained or referenced). A non-expert reader may struggle to understand how t cannot satisfy C if H is consistent w.r.t. C. Here a simple example would be useful (maybe using the problem in Fig. 6). I don't think there are problems of space, e.g., Sect. 1.1 and 1.2 can be merged into Sect. 1.

Thanks for the questions and suggestions to improve the description of Algo.1. In our setting k is a user-specified parameter and in the worst case, there are $p = 2^k$ subdomains returned by the `Fairly_divide` function. Hopefully, some subdomains can often be discarded using constraint refutation and thus $p < 2^k$. As observed by the reviewer, the `boxfilter` is parameterized by the local consistency property used in the solver and a non-expert may have hard time to understand why global consistency cannot always be achieved using a local consistency property for filtering. We have improved the description in the revised submission, according to the suggestion of reviewer 2.

Thus, using global constraints in this context is particularly useful as it allows us to constrain a non-fixed number of variables at once". But I wouldn't call using global constraint an extension, and the fact that the number of variables to be handled is unknown in general does not necessarily imply the use of global constraints.

GZ-MD proposed answer: We have clarified that the interest of global constraints is not related to expressiveness but to efficiency, *i.e.* we want to avoid the decomposition of a global constraint into a potentially large number of basic constraints.

In Sect. 6 the authors say that "CSP solving is not used in this work, the approach is narrowing based". But narrowing domains is actually a significant part of CSP solving. What is not used here is branching on variables to find one or all the solutions.

Catherine?

AG proposed answer: We agree with the reviewer that narrowing is a significant part of CSP solving and have corrected it in the revised submission.

> *Also, this work looks related to: https://link.springer.com/chapter/10.1007/978-3-030-31157-5_12 but that paper is not mentioned.*

The work of De Angelis *et al* has been included in Section 6 of the revised submission.

> *In Sect. 7 the authors claim that "The code we generate is efficient, and outperforms a naive generation technique based on rejection, and allows us to generate large recursive structures quickly". Here comes the main weakness of the paper in my opinion: the evaluation does not compare against any other approach, including the mysterious "naive generation technique" mentioned in the conclusions. In this setting the claim that the proposed approach is efficient does not have any empirical support.*

Matthieu

A.3 To Reviewer 3

> *A weak point of the paper is that the title promises more than what is actually covered by the paper.*

A more accurate title has been given. It states that the focus of the paper is on the generation of numerically constrained **algebraic** Recursive Types (these types being described accurately by the grammar in 7).

MP: Je pense que ça ne suffit pas. Comme je le comprends, le reproche qui nous est fait c'est de dire qu'on fait des types contraints alors qu'on n'a des contraintes que numériques et pas du tout sur la forme des objects. À la place, je mettrais "Numerically Constrained Algebraic Data Types" dans le titre.

CD: J'ai aussi ajouté Random

> *Although the paper contains interesting examples, it lacks a precise specification of the kind of constraints and types covered by this approach.*

We have enumerated in sec 5 the exhaustive list of GC we currently handle, why these, and what would be needed to extend this list.