Declarative Algorithms for Generation, Counting and Random Sampling of Term Algebras

Paul Tarau
Department of Computer Science and Engineering
University of North Texas
Denton, Texas
paul.tarau@unt.edu

ABSTRACT

From a declarative variant of Rémy's algorithm for uniform random generation of binary trees, we derive a generalization to term algebras of an arbitrary signature. With trees seen as sets of edges connecting vertices labeled with logic variables, we use Prolog's multiple-answer generation mechanism to derive a generic algorithm that counts terms of a given size, generates them all, or samples a random term given the signature of a term algebra.

As applications, we implement generators for term algebras defining Motzkin trees, use all-term and random-term generation for mutual cross-testing and describe an extension mechanism that transforms a random sampler for Motzkin trees into one for closed lambda terms.

CCS CONCEPTS

• **Theory of computation → Generating random combinatorial structures**; *Lambda calculus*; • **Software and its engineering → Constraint and logic languages**; *Software testing and debugging*;

KEYWORDS

declarative algorithms, combinatorial generation, random term generation, edge-based tree representations, term algebras, Motzkin trees, generation of random lambda terms

ACM Reference format:
Paul Tarau. 2018. Declarative Algorithms for Generation, Counting and Random Sampling of Term Algebras. In *Proceedings of SAC 2018: Symposium on Applied Computing , Pau, France, April 9–13, 2018 (SAC 2018),* 8 pages.
https://doi.org/10.1145/3167132.3167262

1 INTRODUCTION

Rémy's algorithm [10] elegantly generates random binary trees of a given size. It is a *uniform* random sampling algorithm as any tree of a given size is equally likely to be generated. It is also an *exact* sampler as the random trees are exactly of the specified size. Rémy's original algorithm works by grafting new leaves at internal or leaf nodes of a binary tree. Typical implementations like algorithm **R** in [6] involve an array representation of the tree with destructive assignments used to update and extend the array. While one could easily transliterate such procedural algorithms by using non-backtrackable destructive assignments on arrays represented as compound terms in Prolog, we have chosen to design a declarative algorithm as this usually reveals the intuitive idea behind the construction process - in this case that one can grow a tree by *grafting at the splitting point of and edge a left or right leaning leaf.*

More importantly, the idea of grafting edges ending in leaves to each member of a set of edges can be lifted naturally from binary trees to trees that grow by grafting k such edges corresponding to a function-symbol of arity k. This leads us to a declarative implementation of an algorithm that generates trees representing elements of a term algebra specified by a given signature (i.e., the set of function-symbols of different arities including constant symbols of arity 0), with its most obvious application being generation of random Prolog terms for testing purposes.

Furthermore, one cannot avoid noticing that the *generation of all trees of a given size*, and the *random generation of a tree*, can share exactly the same algorithm, when seen as Prolog code, except that multiple-answer predicates like `member/2` are replaced with counterparts picking a random element of a list.

At the same time, this fortunate *sharing of the declarative description of the two generation mechanisms* suggests means for checking the correctness of each other and observe some of their otherwise intractable properties. For instance, if the all-term generator would miss terms, it would entail that the random generator would also do the same. On the other hand, the distribution obtained by counting the function-symbols and constants on random terms at sizes unreachable by all-term generators is a good estimate of what is likely to happen to the all-term generators asymptotically.

As applications, Motzkin trees (also called binary-unary trees) are of special importance as they are close to lambda terms in de Bruijn notation (and even isomorphic with the very interesting subset of neutral normal forms as shown in [3]). We will add to them an extension algorithm that "completes" a Motzkin tree to a closed lambda term involving very few or most of the time no retries for random terms above size 1000.

The main contributions of the paper are:

- a new, declarative implementation of a variant of Rémy's algorithm
- all-terms and random term generation in term algebras of a given signature, in particular for Prolog terms built from a set of constants and function-symbols of given arities

SAC 2018, April 9–13, 2018, Pau, France
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167262

- mutual correctness checking by sharing the code between all-terms and random generators
- an algorithm to derive closed lambda terms from Motzkin trees

The rest of the paper is organized as follows. Section 2 revisits Rémy's algorithm and proposes a simplified implementation based on extending edges holding vertices represented as logic variables. Section 3 describes generators for term algebras of a given signature. Section 4 overviews applications to generate Motzkin trees and shows mechanisms to cross-validate all-term and random term generators. Section 5 describes an algorithm that extends Motzkin trees to closed lambda terms. Section 6 overviews related work and discusses some properties of our algorithms, including their limitations and possible future generalizations. Section 7 concludes the paper.

The paper is structured as a literate Prolog program to facilitate an easily replicable, concise and declarative expression of our concepts and algorithms. The code extracted from it is at http://www.cse.unt.edu/~tarau/research/2017/ranalg.pro with an extended version at http://www.cse.unt.edu/~tarau/research/2017/lpgen.tar.gz , tested with SWI-Prolog [13] version `7.4.2`.

2 REVISITING RÉMY'S ALGORITHM, DECLARATIVELY

One might wonder why binary trees cannot be generated by randomly adding nodes at their leaves, as a naive algorithm would proceed. As thoroughly explained, for instance in [6], this would not produce a uniform distribution, i.e., not all trees of a given size would have the same chance to be generated.

Rémy's original algorithm [10] grows binary trees by grafting new leaves with equal probability for *each node* in a given tree. An elegant procedural implementation is given in [6] as algorithm **R**, by using destructive assignments in an array representing the tree. While one could emulate it on top of a procedural or declarative emulation of updatable arrays (e.g., with `nb_setarg/3` in SWI-Prolog), we will design here a declarative implementation.

2.1 Trees are connected graphs: let's build them as sets of edges

First, as trees are (connected) graphs, one can represent them as sets of edges. We will use *logic variables* to label their ends representing either internal or leaf nodes. We would also label each edge as `left` or `right` to indicate their position relative to a node in the binary tree. Thus a `left` edge originating in `A` with target `B` will be represented as `e(left,A,B)`. We start with a list of two edges from root node `A` returned by the predicate `remy_init/1`.

```
remy_init([e(left,A,_),e(right,A,_)]).
```

The random choice of the edges (or the non-deterministic one, by replacing `choice_of/2` with its commented out alternative [1]) is generated by the predicate `left_or_right/2` as:

[1] Note that in the extended code covering the paper, we provide specific files to parameterize either random or all-term generation that include a shared generic algorithm, as close to an object-oriented style as possible in a language like Prolog. They contain also, specific signatures supporting the generalization of the algorithm to several term algebras.

```
left_or_right(I,J):-
  choice_of(2,Dice),
  left_or_right(Dice,I,J).

choice_of(N,K):-K is random(N).
% choice_of(N,K):-N>0,N1 is N-1,between(0,N1,K).

left_or_right(0,left,right).
left_or_right(1,right,left).
```

This avoids adding the actual choice predicates as arguments partly to keep the code described here less verbose and partly to avoid meta-calls in the inner loops slowing down the execution of the algorithm.

We can grow a new edge by "splitting an existing edge in two" via the predicate `grow/3`:

```
grow(e(LR,A,B), e(LR,A,C),e(I,C,_),e(J,C,B)):-
  left_or_right(I,J).
```

Note that a single clause defines `grow/3`, independently of the `left` or `right` denoting the relation of the edge to its source node `A`. It adds three new edges corresponding to arguments 2, 3 and 4 and removes one, represented as its first argument. Note also, that contrary to Rémy's original algorithm, our tree grows "downward" as new edges are inserted at the target of existing ones. As the set of edges is in bijection with the set of vertices of a binary tree, except the root, this choice does not change any of the correctness assumptions of Rémy's original algorithm, as proven in [10].

The new node `C`, connected to `A` by inheriting the type `LR` of the edge `e(LR,A,B)` will be made to point to a new leaf "`_`" via the edge `e(I,C,_)` and to the tree below node `B` via the edge `e(J,C,B)`.

The `left / right` choice among `I` and `J`, is provided by the predicate `left_or_right(I,J)`.

This leads us the basic step of the algorithm, where a set of edges `Es` is rewritten as a set of new edges `NewEs` as given by the predicate `remy_step/4`. To avoid computing the size `L` of the set `Es` we maintain it by adding `2=3-1` as one node is removed and `3` are added at a given step. Note that we pick an edge *randomly* among the `L` available by calling `choice_of/2`, operation provided by `remy_step1`, that navigates the list to to where the rewriting step `grow/3` happens.

```
remy_step(Es,NewEs,L,NewL):-
  NewL is L+2,
  choice_of(L,Dice),
  remy_step1(Dice,Es,NewEs).

remy_step1(0,[U|Xs],[X,Y,Z|Xs]):-grow(U, X,Y,Z).
remy_step1(D,[A|Xs],[A|Ys]):-D>0,D1 is D-1,
  remy_step1(D1,Xs,Ys).
```

The predicate `remy_loop` iterates over `remy_step` until the desired $2K$ size is reached, in K steps as we grow by 2 edges at each step. Note also that the initial 2 edges are added when `K=1` by calling `remy_init`.

```
remy_loop(0,[],0).
remy_loop(1,Es,2) :-remy_init(Es).
remy_loop(K,NewEs,N3):-K>1, K1 is K-1,
  remy_loop(K1,Es,N2),
  remy_step(Es,NewEs,N2,N3).
```

Example 1. *illustrates the generation of a random list of edges of size* 4*:*

```
?- remy_loop(2,Edges,N).
Edges =
  [e(left,A,B),e(right,A,C),e(right,C,D),e(left,C,E)],
N = 4.
```

2.2 From sets of edges to trees as Prolog terms

The final step, "unleashing"the power of logic variables, extracts a Prolog term representing the binary tree from the list of edges labeled with unbound variables. The predicate `bind_nodes/2` iterates over edges, and for each internal node it binds it to terms provided by the constructor a/2, left or right, depending on the type of the edge. Note that, given the order-independence of the binding of logical variables, the same term is built independently of the order of the edges.

Next, the predicate `bind_leaf` binds the remaining unbound nodes with the constant `v/0` labeling the leaf nodes. Correctness follows from the fact that a node is a leaf if and only if it remains unlabeled when the source of an edge is marked with the `a/2` constructor, i.e, if it is not the source of an edge.

Note that we use `maplist` to iterate over lists and to apply a predicate to their corresponding elements.

```
bind_nodes([],v).
bind_nodes([X|Xs],Root):-X=e(_,Root,_),
  maplist(bind_internal,[X|Xs]),
  maplist(bind_leaf,[X|Xs]).

bind_internal(e(left,a(A,_),A)).
bind_internal(e(right,a(_,B),B)).

bind_leaf(e(_,_,Leaf)):-Leaf=v->true;true.
```

The predicate `remy_term/2` puts the two main steps together.

```
remy_term(K,B):-
  remy_loop(K,Es,_),
  bind_nodes(Es,B).
```

Example 2. *illustrates the generation of a random term with* 4 *internal nodes as well timings for a larger random tree.*

```
?- remy_term(4,T).
T = a(a(v, v), a(v, a(v, v))) .
?- time(remy_term(1000,_)).
1,025,950 inferences,
0.085 CPU in 0.098 seconds (12113466 Lips)
```

While the algorithm handles fairly large terms in reasonable time, we do not claim that its average performance is linear, like in the case of Knuth's procedural implementation, given that it takes time proportional to the size of the set of edges to pick the one to be expanded. However, that one can improve its expected $O(N^2)$ performance with a tree representation of the set of edges to $O(Nlog(N))$ or even to amortized $O(N)$ with a dynamically growing array representation using arbitrary arity compound terms as containers.

3 A GENERAL ALGORITHM FOR TERM ALGEBRAS OF GIVEN SIGNATURE

Combinatorial algorithms (as shown for instance in [4]) are a natural match to Prolog's synergy between unification, multiple-answer generation and definite clause grammars (DCGs). We will start with a simple generator, that we will use as a reference implementation for an algorithm generating term algebras of a given signature, that can be seen as a generalization of Rémy's algorithm.

3.1 A simple generator, using DCGs

As we want to ensure that terms of an exact size are generated, for a given "size" definition, we spend some "Fuel" at each step, as implemented by the predicate `spend/3`, that decrements the amount of remaining "Fuel".

```
spend(Fuel,From,To):-From>=Fuel,To is From-Fuel.
```

We adopt an empirically justified definition of size, in the sense that when creating a function symbol of arity `N`, we consume `N` units of "Fuel". This will result in *a term having a size proportional to the size that a Prolog term has when represented on the heap.*

We will use Prolog's DCG mechanism to chain the arguments controlling the size consumed at each step. The predicate `gen(Fs,T)` generates a term `T` using the list `Fs` of function-symbol/arity pairs (including constants seen as having arity 0). At each step in the predicate `gen/4`, a function-symbol `F/K` is non-deterministically chosen. Size is implicitly defined as the arity `K` of the function-symbol, thus `0` for constants in the predicate `gen_cont`, responsible to start the predicate `gens/5` which iterates with Prolog's `arg/3` over each argument of a compound term created with Prolog's generic term constructor `functor/3`.

```
gen(Fs,T)-->{member(F/K,Fs)},gen_cont(K,F,Fs,T).

gen_cont(0,F,_,F)-->[].
gen_cont(K,F,Fs,T)-->{K>0},spend(K),
  {functor(T,F,K)},
  gens(Fs,0,K,T).

gens(_,N,N,_)-->[].
gens(Fs,I,N,T)-->{I1 is I+1,arg(I1,T,A)},
  gen(Fs,A),
  gens(Fs,I1,N,T).
```

For the reader unfamiliar with DCGs, we mention that the 2 extra arguments constraining the size of the terms are added when the "`-->`" clause constructor is used. We expose the generator via the predicate gen/3, that given input arguments N=intended size of a term and `Fs`=set of function-symbol/arity pairs, iterates over all terms `T` of size `N` built using `Fs`.

```
gen(N,Fs, T):-gen(Fs,T,N,0).
```

Example 3. *illustrates the generation of all binary trees of size* 6 *seen as defined by the signature* `[v/0,a/2]`.

```
?- gen(6,[v/0,a/2],T).
T = a(v,a(v,a(v,v))) ;
T = a(v,a(a(v,v),v)) ;
T = a(a(v,v),a(v,v)) ;
T = a(a(v,a(v,v)),v) ;
```

```
T = a(a(a(v,v),v),v) .
```

3.2 A unified "choice definition" for all-term and random-term generators

We start by observing that by replacing in our variant of Rémy's algorithm of section 2 the predicate `choice_of/2` by

```
choice_of(N,K):-N>0,N1 is N-1,between(0,N1,K).
```

we obtain a generator for all terms of a given size. Let us note that it is a fortunate feature of Prolog that an one-line code change turns a random term generator into an all-term generator. This brings us to design our choice operator to be oblivious to iterating over all choices or picking a choice randomly. For a random term generator we define the following "customized" choice operators:

```
member_choice(Choice,Choices):-
  length(Choices,L),L>0,
  I is random(L),
  nth0(I,Choices,Choice).
select_choice(Choice,Choices,ChoicesLeft):-
  length(Choices,L),L>0,
  I is random(L),
  nth0(I,Choices,Choice,ChoicesLeft).
between_choice(From,To,Choice):-
  D is To-From+1,
  Choice is From+random(D).
```

The predicate `member_choice/2` pics randomly an element of a list. It could also be defined in terms of `select_choice/3` that picks an element randomly and returns the remaining ones on a list. The predicate `between_choice/3` pics randomly an integer `Dist` between `From` and `To`, endpoints included.

As one can notice, they mimic some well-known Prolog predicates, which are used if one wants to iterate over *all terms* of a given size:

```
member_choice(Choice,Choices):-
  member(Choice,Choices).
select_choice(Choice,Choices1,Choices2):-
  select(Choice,Choices1,Choices2).
between_choice(From,To,I):-
  between(From,To,I).
```

As we will see, except for these alternatives for choice predicates, exactly the same Prolog code can be used to implement our generalization of Rémy's algorithm. Moreover, the number of solutions of the generator provides a counting mechanism, of interest especially when no closed formulas or generator functions exist for it (e.g., the case of simply-typed lambda terms).

Note that we do not expect the random sampler to be *uniform*, given that different function symbol arities introduce a selection bias. On the other hand, the resulting samplers are always *exhaustive*, with every term in the set of terms of a given size having a chance to be selected. In the case of binary trees, as for Rémy's original algorithm, this chance is the same for all terms of a given size, while, for instance, in the case of Motzkin trees, much more intricate algorithms, as shown in [1] are needed for uniformity. While we do not ensure the uniform distribution of random terms of a given size, we can control the probability with which function symbols are picked, for instance, to customize the generators to match their frequency in a segment of code for which we would like to build a random tester.

3.3 Parameterizing with the signature

Like in the case of the generator defined in subsection 3.1, we will parameterize our program with a set of function-symbol/arity pairs. The predicate `classify_funs` separates that list into constants and arity / list of function-symbol pairs.

```
classify_funs(FNs,Cs,SortedFs):-
  findall(N-F, member(F/N,FNs), NFs),sort(NFs,Ordered),
  group_pairs_by_key(Ordered,ByArityFs),
  keysort(ByArityFs,[0-Cs|SortedFs]).
```

Example 4. *illustrates this "optional, but convenient" interface element.*

```
?- classify_funs([g/2,c/0,f/2,d/0,h/3,t/2,s/3],Cs,FXs).
Cs = [c, d], FXs = [2-[f, g, t], 3-[h, s]].
```

3.4 Distilling some essence : generating the arity-skeleton

As multiple function-symbols may share the same arity, we choose to abstract away an ‘*‘arity-skeleton"* of the generated term, that will be fleshed out later with the actual function-symbols. This has the advantage of limiting combinatorial explosion in the case of multiple function symbols having the same arity. We start by extracting the set of non-zero arities with `get_arities/2`.

```
get_arities(NFs,Ns):-maplist(arg(1),NFs,Ns).
```

We then initialize our set of edges by picking an edge (randomly or non-deterministically) depending on `member_choice/2`.

```
init_funs(NFs,Ns,Root,Es):-
  get_arities(NFs,Ns),
  member_choice(N,Ns),
  init_fun(Root,N,Es).
```

The predicate `init_fun/3` *sketches the key idea of the algorithm: adding a new function-symbol of arity* `N` *means connecting a logic variable representing the source (in this case the root) to* `N` *edges representing its arguments represented as (still unbound) logic variables.*

```
init_fun(Root,N,Es):-N>0,
  length(Ns,N),N1 is N-1,
  numlist(0,N1,Is),
  maplist(=(N),Ns),
  maplist(make_edge(Root),Ns,Is,Es).

make_edge(X,N,I, e(N,I,X,_)).
```

Note that we store in an edge `e(N,I,From,To)` the arity `N` of the function-symbol it originates from and the argument position `I`, that it points to, as well as the logic variables `From` and `To` representing the source and the target of the edge.

The predicate `extension_step/3` extends the work of `init_fun/3` to the case where the insertion happens by "splitting an existing edge in two", as in the case of our variant of `Rémy's` algorithm in section 2. We insert a new term `A` by splitting edge `X->Y` into `X->A` and `A->Y`, and then inserting leaves in all positions, except a position `K` to where we insert a new edge from `A`.

Note that we select a new arity among those smaller or equal to D, a parameter which limits how much size we have left. This prunes function-symbols that would bring too many edges, exceeding the prescribed size.

While in the case of the binary trees in section 2 we have extended an edge by adding to it a leaf to the left or the right, here we add N leaves centered on a chosen argument position K with N-1 leaves added around it, and the tree below the edge inserted at position K.

```
extension_step(GoodNs,OldEs,[e(M,I,X, A),
               e(Arity,K,A,Y)|Es],N1,N2):-
  GoodNs=[_|_],
  select_choice(e(M,I,X, Y),OldEs,OtherEs),
  member_choice(Arity,GoodNs),
  Last is Arity-1,
  N2 is N1+Arity,
  between_choice(0,Last,K),
  add_leaves(0,Arity,K,A,Es,OtherEs).
```

The predicate select_choice/3 helps rewriting an edge e(M,I,X, Y) as a set of edges where leaves will be inserted in all positions, except position K where the tree below the end of the edge Y will be attached.

The predicate add_leaves extends the set of edges with leaves seen as unbound variables. It exempts edge K, taken care of by the predicate extension_step. Note that DCGs are used to chain together the states of the lists of edges at each step.

```
add_leaves(N,N,_,_)-->[].
add_leaves(I,N,K,A)-->{I<N,I=:=K,I1 is I+1},
  add_leaves(I1,N,K,A).
add_leaves(I,N,K,A)-->{I<N,I=\=K,I1 is I+1},
  [e(N,I,A,_)],
  add_leaves(I1,N,K,A).
```

Iterating the extension steps is similar to the process described for binary trees. The predicate extend_to(M,NFs,Root,NewEs) starts with a set of function/arity pairs NFs from where it initializes the list of edges Es, extracts the root of the tree and the list of arities Ns that it passes to the predicate extension_loop.

```
extend_to(M,NFs,Root,NewEs):-
  init_funs(NFs,Ns,Root,Es),
  length(Es,N),
  extension_loop(Ns,Es,NewEs,N,M).
```

The predicate extension_loop iterates over extension steps until the prescribed size of the edge set is reached.

```
extension_loop(_,Es,Es,N,N).
extension_loop(Ns,Es,NewEs,N1,N3):-D is N3-N1,D>0,
  filter_smaller(Ns,D,GoodNs),
  extension_step(GoodNs,Es,EsSoFar,N1,N2),
  extension_loop(GoodNs,EsSoFar,NewEs,N2,N3).
```

The predicate filter_smaller/3 ensures that only only arities that will not overflow the size limit are used to extend the set of edges.

```
filter_smaller([],_,[]).
filter_smaller([I|_Is],D,[]):-I>D. % they are sorted!
filter_smaller([I|Is],D,[I|Js]):-I=<D,
  filter_smaller(Is,D,Js).
```

We can test the generation of edges driven by "arity-skeletons" with the predicate ext_test, that, given a desired number of edges M and a set of function-symbol-arity pairs, returns a Root paired with a list of edges.

```
ext_test(M,CFs, Root-Edges):-classify_funs(CFs,_,NFs),
  extend_to(M,NFs,Root,Edges).
```

Example 5. *illustrates its work the predicate* ext_test *that, given the signature* [v/0,l/1,a/2], *generates a random set of edges of size* 5.

```
?- ext_test(5,[v/0,l/1,a/2],Root-Edges).
Root=A,
Edges=[e(2,1,A,B),e(2,1,B,C),e(2,0,B,D),e(2,0,A,E),
                                        e(1,0,E,F)] .
```

3.5 Fleshing-it out: functors first, then constants at leaves

Like in the case of our variant of Rémy's algorithm in section 2, we extract a term by iterating over the edges and binding the logic variables according to their intended semantics, with function-symbols of the appropriate arity for internal nodes and constant symbols for the leaves.

First, the predicate edges2term/3 applies to each edge the predicate edge2fun/2 which covers internal nodes, but leaves edges unbound as they do not point to any other node. The predicate leaf2constant/2 finishes the work by binding the leaves to constants.

```
edges2term(Cs,NFs,Xs):-
  maplist(edge2fun(NFs),Xs),
  maplist(leaf2constant(Cs),Xs).
```

The predicate edge2fun/2 selects among function-symbols using member_choice/2 before building the corresponding terms. To ensure that in the case of random generation two edges originating from the same node do not try to build different terms when multiple function-symbols of the same arity are present, we need to only call this operation once, when the variable T is unbound. Note also that the (unique) arity / set of function-symbols list needs to be selected with member/2 from the set NFs, as otherwise a random choice could induce failure by picking the wrong arity form the set NFs.

```
edge2fun(NFs,e(N,I,T,A)):-I1 is I+1,member(N-Fs,NFs),
  (var(T)->member_choice(F,Fs);true),
  functor(T,F,N),arg(I1,T,A).
```

The predicate leaf2constant binds the unbound target Leaf of an edge to the constant C selected by member_choice(C,Cs) from the set Cs.

```
leaf2constant(Cs,e(_,_,_,Leaf)):-
  member_choice(C,Cs),
  ( Leaf=C->true
  ; true
  ).
```

3.6 Putting it all together

The predicate `gen_terms(M,FCs,T)` takes as input the desired size of a generated term `M` and a set of function-symbol / arity pairs `FCs` with constants represented as having arity `0`. It returns a term `T` of size `M`, based on a size definition that weights each function-symbol as its arity.

```
gen_terms(M,FCs,T):-classify_funs(FCs,Cs,NFs),
  extend_to(M,NFs,T,Es),edges2term(Cs,NFs,Es).
```

The predicate `gen_terms/3` puts together the main steps of our algorithm by first separating the constants `Cs` from the arity / function-symbol set pairs, then extending the set of edges to size `M` and finally extracting the terms from the set of edges. Note that the algorithm generates a *multiset* of terms in the case we define the all-terms choice predicates, that can be trimmed to a *set* of unique terms using SWI-Prolog's `distinct/2` predicate with

```
gen_term(M,FCs,T):-distinct(T,gen_terms(M,FCs,T)).
```

As this does not make any difference when a unique random term is generated, we expose the overall functionality of our algorithm through a simple interface defined by the predicate `gen_term/3`.

Example 6. *illustrates some uses of* `gen_term/3` *to generate all-term terms.*

```
?- gen_term(3,[v/0,l/1,a/2],T).
T = l(l(l(v))) ;
T = l(a(v, v)) ;
T = a(l(v), v) ;
T = a(v, l(v)) .
```

Example 7. *illustrates some uses of* `gen_term/3` *to generate random terms.*

```
?- gen_term(30,[0/0,1/0,(~)/1,(*)/2,(+)/2],T).
T = ~((~(0*0)* ~(~(~(~(~(~(~(~(1))))*
    (1+1)*0))+0*1))+ ~(1))* ~(1)) .

?- time(gen_term(4000,[v/0,a/2],_)).
% 2,192,586 inferences, 0.515 CPU
% in 0.549 seconds (94% CPU, 4259980 Lips)

?- time(gen_term(6000,[v/0,a/2],_)).
% 4,792,151 inferences, 1.104 CPU
% in 1.149 seconds (96% CPU, 4339722 Lips)
```

As one can notice, performance for binary trees is comparable than with the specialized variant of Rémy's algorithm described in section 2.

4 APPLICATIONS

4.1 Motzkin trees

We can specialize our generators to a given set of function symbols. As an example, Motzkin trees (also called binary-unary trees) are described by

```
mot_funs([v/0,l/1,a/2]).
```

Then, as each of our generators is parameterized by the signature of the term algebra, we obtain:

```
mot(N,T):-mot_funs(CFs),gen(N,CFs,T).
```

for generating plain Motzkin trees. In section 5, we will use the Motzkin tree generator as a skeleton to be extended to lambda terms.

4.2 Correctness cross-checks between all-terms generators and random-term generators

One of the fallouts of having the same code work as an all-terms and random term generator is that we can do some cross-checking of properties that we expect to hold in both cases.

4.2.1 Testing the equivalence between all-term generators. First, we can check the empirical soundness of the generator by comparing the terms of a given size it outputs with those of the vanilla generator `gen/2` described in subsection 3.1. While this can be done for a few terms with human eyes, the faster than exponential growth with respect to size is better served by counting the terms generated for successive sizes. We will do that for two term algebras - one for binary trees and the other for Motzkin trees. As expected, for binary trees, we obtain in both cases `1, 0, 1, 0, 2, 0, 5, 0, 14, 0, 42, 0, 132, 0, 429 ...` with terms in even positions corresponding to the *Catalan numbers*, counted by sequence **A000108** in [11]. For Motzkin trees we obtain in both cases `0, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, ...` corresponding to the *Motzkin numbers* that are counted by sequence **A001006** in [11].

As in all-terms generation mode our generic code works significantly slower than the depth-first generator `gen/2` of subsection 3.1, once we trust their equivalence, we can rely on comparing assertions that should hold for the random terms as well as terms provided by our more efficient depth-first term generator `gen/2`.

4.2.2 No term left behind: checking that any term can be the chosen one. As we use the same code for the all-terms and random term generators, equivalence of the former with as the standard `gen/2` generator entails that, in principle, all terms have a chance to be generated when running the generator in random mode. But, as a side note, one should keep in mind that the humungous size of the space of possibilities for a random term of, say, size `1000`, cannot be matched by the period of the random generators we use. Consequently, only as many distinct terms as the period of the concrete random generator have a chance to be generated.

4.3 Function-symbol counts: checking the ingredients of the random recipe

Combinatorial proofs of properties of a Rémy-like generator for a term algebra of a given signature are fairly intricate and require creative techniques even for very simple ones like in the case of Motzkin trees, as shown in [1].

This raises the question if there are simple properties that could indicate that a similar, "close-enough" empirical distribution exists for the random terms of large sizes we can generate.

A good candidate for that is the *relative count of the function-symbols occurring in the output of random term and in all-term generators.* In the case of all-terms of a given size we compute that by summing them up over all the generated terms. Given their large number, even for small sizes, it is reasonable to think that they are an indicator of what should happen when building large random terms of the same signature.

We will not do this for binary trees where for each tree with N internal nodes we always have N+1 leaves, but we can start with Motzkin trees, where the counts for unary nodes are independent of those for binary nodes and leaves. As we can trust the larger counts reachable by our equivalent standard generators we obtain:

```
Total counts for size 14:
  [a/2-4343160,l/1-4969152,v/0-5196627]
Relative percentages:
  [a/2-0.2993,l/1-0.3424,v/0-0.3581]

Counts for random term of size 4000:
  [a/2-1334,l/1-1332,v/0-1335]
Relative percentages:
  [a/2-0.3334,l/1-0.3329,v/0-0.3336]
```

Note that the first two counts indicate a (slow) convergence process toward the asymptotic 1/3 value for each distribution [1, 8]. The last line, closely matching the asymptotic distribution of 1/3 for each constructor, is a good indicator of how close our random generator is to a uniform one.

By using Maciej Bendkowski's ingenious Boltzmann-sampler generator [2] one can compare distributions corresponding to Boltzmann samplers with those of our generators for any concrete function-symbol / arity pairs set.

5 ONE MORE LIFT: DECORATING MOTZKIN TREES TO CLOSED LAMBDA TERMS

By starting from our random generator for Motzkin trees, or, if one prefers a uniform distribution for a given size, by using a Boltzmann sampler as the one automatically generated by [2], one can "decorate" it to lambda terms in de Bruijn notation [5] simply by labeling its leaves with de Bruijn indices, indicating their binder as the number of l/1 constructors encountered on the path to the root of the tree. To mimic (actually in a stronger way) the ideas behind the "natural size" described in [3], that favors variables closer to their binders, one can build for each list of binders from a leaf to the root, a distribution that decays exponentially with each step, as defined by nat2probs/2.

```
nat2probs(0,[]).
nat2probs(N,Ps):-N>0,
  Sum is 1-1/2^N,
  Last is 1-Sum,
  Inc is Last/N,
  make_probs(N,Inc,1,Ps).
```

In this case, at each step, the probability to continue further is reduced to half, work done by make_probs/4.

```
make_probs(0,_,_,[]).
make_probs(K,Inc,P0,[P|Ps]):-K>0,
  K1 is K-1,P1 is P0/2, P is P1+Inc,
  make_probs(K1,Inc,P1,Ps).
```

Once the list of probability thresholds is build, the predicate walk_probs/3 is used to decide how far, on the list of available binders it will point.

```
walk_probs([P|Ps],K1,K3):-random(X),X<P,!,
  K2 is K1+1,
  walk_probs(Ps,K2,K3).
walk_probs(_,K,K).
```

Given a Motzkin tree, we decorate each leaf v/0 by turning it into a natural number representing a de Bruijn index. The value of the de Bruijn index is determined for each leaf independently by walking up on the chain of lambda binders with decaying probabilities.

```
decorate(v,0,X)-->[X]. % free variable
decorate(v,N,K)-->
  {N>0,nat2probs(N,Ps),walk_probs(Ps,0,K)}.
decorate(l(X),N,l(Y))-->{N1 is N+1},
  decorate(X,N1,Y).
decorate(a(X,Y),N,a(A,B))-->
  decorate(X,N,A),
  decorate(Y,N,B).
```

The predicate plain_gen collects the list of free variables left over when called with a size N and generating a lambda term T.

```
plain_gen(N,T,FreeVars):-
  mot_gen(N,B),
  decorate(B,0,T,FreeVars,[]).
```

To ensure that a term is closed, one restarts until the list of free variables is empty as shown by closed_gen/3, also returning the number of retries I.

```
closed_gen(N,T,I):-
  Lim is 100+2^min(N,24),
  try_closed_gen(Lim,0,I,N,T).
```

These restarts are managed by the predicate try_closed_gen/5, which, when the decorated term is not closed, tries generating a new term.

```
try_closed_gen(Lim,I,J,N,T):- I<Lim,
  ( mot_gen(N,B),decorate(B,0,T,[],[])*->J=I
  ; I1 is I+1, try_closed_gen(Lim,I1,J,N,T)
  ).
```

As an interesting Prolog feature, we use a multiple try if-then-else (denoted "*->" in SWI-Prolog), to ensure that backtracking occurs in the *condition part* of the "*->" operation. Should, however, failure occur, typically because a given leaf has no unary nodes to be used as a lambda binder above it, a fresh retry is triggered by calling the same predicate recursively. The predicate also maintains a count I->I+1 of the retries, which, in our experiments, are typically not more than 2 or 3.

Example 8. *illustrates random closed lambda terms obtained by decorating motzkin trees.*

```
?- closed_gen(10,T,I).
T = a(l(1), l(l(l(a(a(2, 1), 1))))), I = 0 .
?- closed_gen(5000,_,I).
I = 3 .
```

6 RELATED WORK

Rémy's algorithm [10], procedurally implemented as algorithm **R** in [6] has generated a significant number of attempts to adapt it to uniformly generate similar data types. Among them we mention [1] where it is also shown how difficult it is to ensure uniformity. The use of de Bruijn indices for the study of combinatorial properties of lambda terms is introduced in [7]. In [9] a "type-directed" mechanism for the generation of random terms is introduced, resulting in more realistic (while not uniformly random) terms, used successfully in discovering some bugs in the Glasgow Haskell Compiler (GHC).

For uniform generation of arbitrary data-types specified by a context-free grammar, the Boltzmann sampler generator of [2] stands out, as it produces efficient Haskell programs generating uniformly terms of an expected size or above. By contrast to [4], that uses data computed by the generator in [2] to build a Boltzmann sampler for simply-typed closed lambda terms, this paper uses a variant of Rémy's algorithm that is also generalized to term algebras of an arbitrary signature, and thus directly useful for generating random Prolog terms for test automation of logic programs.

In [12] an serialization algorithm for Prolog terms using a bijection from a term algebra to to natural numbers is described, that can, in principle, also be used for the generation of random terms, although at a much smaller scale, given the limitations on the size of the integers used as encodings for the terms and the computational effort involved in the decoding of integers to terms.

While we have dropped the uniformity requirement in our generalization to term algebras, we have ensured that the generators are exhaustive - i.e., that every term of a given size has a chance to be chosen. Given that the same generator is used for all-term and random term generation, depending only on the choice of selection predicates, our random samplers are automatically exhaustive. On the other hand, we can uniformly choose function symbols from a given signature, and one can customize this selection to a different distribution if needed. For instance, choosing probabilities to be inverse proportional to arities would increase the frequency of symbols with smaller arities in a random term. Thus, in a random testing application, one can mimic the distribution of the function symbols occurring in the program to be tested.

7 CONCLUSIONS

Our declarative implementations of random and all-term generation algorithms, show that logic programming languages, often seen outside our field as "domain specific", can provide means for implementing simple and naturally generic algorithms, thought to be exclusively in the realm of procedural or object oriented languages.

We have used some essential features of logic programming languages: the ability of logic variables to stand for absent information to be completed later and the ability to configure choice operations as random single-answer or "nondeterministic" multiple answer, without any other change in the code. Data type genericity" has spread these days from functional languages like ML and Haskell to the procedural world, ranging from mechanisms like standard templates in C++ and generic types in Java, Scala or Swift. While also supported indirectly by using libraries of monads and monad transformers in functional languages, the more subtle "algorithm genericity", allowing one to overlap via the same code deterministic execution for random sampling and non-derministic execution for multiple answer generation, happens with no notational clutter or semantic complexity in a logic programming language like Prolog.

Our generator for term algebras can be useful, as a practical application for automating the generation of random tests for logic programming languages. The decoration algorithm lifting random Motzkin trees into closed lambda terms can be further specialized to simply-typed terms as shown in [4] and can be useful for testing correctness and scalability of compilers for functional languages and proof assistants, given the fact that we are able to generate very large such terms. The "edge-splitting" mechanism used for Rémy's algorithm and its generalization is likely to be also usable for generation of random graphs, and in particular for generating cyclic terms relevant when testing compilers, run-time systems and libraries of Prolog implementations.

ACKNOWLEDGEMENT

This research is supported by NSF grant **1423324**.

We thank the anonymous reviewers of SAC'2018 for their suggestions, Maciej Bendkowski for his enlightening comments on a previous draft of this paper and the participants of the *CLA'2017* workshop (https://cla.tcs.uj.edu.pl/programme.html) for illuminating discussions.

REFERENCES

[1] A. Bacher, O. Bodini, and A. Jacquot. 2013. Exact-size Sampling for Motzkin Trees in Linear Time via Boltzmann Samplers and Holonomic Specification. In *2013 Proceedings of the Tenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, Markus E. Nebel and Wojciech Szpankowski (Eds.). SIAM, 52–61.

[2] Maciej Bendkowski. 2017. Boltzmann-brain. (2017). Software (Haskell stack module), published electronically at https://github.com/maciej-bendkowski/boltzmann-brain.

[3] Maciej Bendkowski, Katarzyna Grygiel, Pierre Lescanne, and Marek Zaionc. 2016. A Natural Counting of Lambda Terms. In *SOFSEM 2016: Theory and Practice of Computer Science - 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings (Lecture Notes in Computer Science)*, Rusins Martins Freivalds, Gregor Engels, and Barbara Catania (Eds.), Vol. 9587. Springer, 183–194. https://doi.org/10.1007/978-3-662-49192-8_15

[4] Maciej Bendkowski, Katarzyna Grygiel, and Paul Tarau. 2017. Boltzmann Samplers for Closed Simply-Typed Lambda Terms. In *Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings (Lecture Notes in Computer Science)*, Yuliya Lierler and Walid Taha (Eds.), Vol. 10137. Springer, 120–135. https://doi.org/10.1007/978-3-319-51676-9_8 , Best student paper award.

[5] N. G. de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae* 34 (1972), 381–392.

[6] Donald E. Knuth. 2006. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming).* Addison-Wesley Professional.

[7] Pierre Lescanne. 2013. On counting untyped lambda terms. *Theoretical Computer Science* 474 (2013), 80 – 97. https://doi.org/10.1016/j.tcs.2012.11.019

[8] Pierre Lescanne. 2014. Boltzmann samplers for random generation of lambda terms. *CoRR* abs/1404.3875 (2014). http://arxiv.org/abs/1404.3875

[9] Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST'11).* ACM, New York, NY, USA, 91–97.

[10] Jean-Luc Rémy. 1985. Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications* 19, 2 (1985), 179–195.

[11] N. J. A. Sloane. 2017. The On-Line Encyclopedia of Integer Sequences. (2017). Published electronically at https://oeis.org/.

[12] Paul Tarau. 2013. Compact Serialization of Prolog Terms (with Catalan Skeletons, Cantor Tupling and Gödel Numberings) . *Theory and Practice of Logic Programming* 13, 4-5 (2013), 847–861.

[13] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjorn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12 (1 2012), 67–96. Issue Special Issue 1-2. https://doi.org/10.1017/S1471068411000494