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

Paul Tarau

University of North Texas

ACM SAC'2018

Research supported by NSF grant **1423324**

Overview

- 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 application, we show generators for term algebras defining Motzkin trees, SK-combinator trees, Categorial grammar expressions.
- We use all-term and random-term generation for mutual cross-testing.
- We show extension mechanism that transforms a random sampler for Motzkin trees into one for closed lambda terms.

Outline

the paper is organized as a literate Prolog program - our code is available at:
`http://www.cse.unt.edu/~tarau/research/2017/ranalg.pro`

Revisiting Rémy’s algorithm, declaratively

Rémy’s algorithm, with a twist

- Rémy’s original algorithm grows binary trees by grafting new leaves with equal probability for each node in a given tree
- an elegant procedural implementation is described by Knuth as algorithm **R**, by using destructive assignments in an array representing the tree
- we will work with edges instead of nodes and graft new edges at each step
- a two stage algorithm: first sets of edges, then trees represented as Prolog terms
- some “magic with logic variables” and unification
- while the algorithm handles terms with thousands of nodes its average performance is slightly above linear as it takes time proportional to the size of the set of edges to pick the one to be expanded

Same code for all-terms and random term generation

- initial set of two edges

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

- open-ended edges, marked with logic variables
- same code for all-term or random-term generation
- except for defining a random choice of the edges or a backtracking one, by replacing `choice_of/2` with its commented out alternative

```
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).
```

The grafting step

- we can grow a new edge by “splitting an existing edge in two”

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

- we add three new edges corresponding to arguments `2`, `3` and `4`
- we remove one, represented as the first argument
- contrary to Rémy’s original algorithm, our tree grows “downward” as new edges are inserted at the target of existing ones

Finding the spot where we graft

- chose the grafting point's position

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

- perform the graft

```
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 iteration to desired size

- The predicate `remy_loop` iterates over `remy_step` until the desired $2K$ size is reached, in K steps
- we grow by `2` edges at each step, 2K edges total

```
remy_loop(0,[],N,N).
remy_loop(1,Es,N1,N2):-N2 is N1+2,remy_init(Es).
remy_loop(K,NewEs,N1,N3):-K>1,K1 is K-1,
  remy_loop(K1,Es,N1,N2),
  remy_step(Es,NewEs,N2,N3).
```

- an example of output

```
?- remy_loop(2,Edges,0,N).
Edges = [e(left, A, B), e(right, A,C),
         e(right,C,D), e(left,C,E)],
N = 4.
```

From sets of edges to trees as Prolog terms

- the predicate `bind_nodes/2` iterates over edges, and for each internal node it binds it with terms provided by the constructor c/2, left or right, depending on the type of the edge

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

- bind an internal node with constructor `c/2`

```
bind_internal(e(left,c(A,_),A)).
bind_internal(e(right,c(_,B),B)).
```

- bind a leaf node with the constant `v/0`

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

Running the algorithm

- the predicate `remy_term/2` puts the two main steps together

```
remy_term(K,B):-remy_loop(K,Es,0,_),bind_nodes(Es,B).
```

- uniformly random generation of a random term with 4 internal nodes and timings for a large tree:

```
?- remy_term(4,T).
T = c(c(e, e), c(c(e, e), e)) .
?- time(remy_term(1000,_)).
526,895 inferences, 0.066 CPU in 0.078s (7,995,978 Lips)
```

A simple "vanilla" generator for term algebras

A simple generator, with definite clause grammars (DCGs)

- burning some fuel at each step to control size of the trees

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

- size controlled depths-first

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

gen_cont(0,F,_,F)-->[]. % an atomic term
gen_cont(K,F,Fs,T)-->{K>0},spend(K),  <== implicit size control
  {functor(T,F,K)}, % apply a constructor
  gens(Fs,0,K,T). % iterate over the arguments

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

- DCGs: the 2 extra arguments constraining the size of the terms are added when the "-->" constructor expands clauses

The simple generator at work

- 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) .
```

- possible to transform into a Boltzmann sampler when signature is known (e.g., see our previous work at PADL'17)
- while distribution is uniform, with Boltzmann samplers, the generated size is only predictable within a range

From Rémy’s algorithm to the general case: term algebras of given signature

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

- in the case of Rémy's algorithm, a one-line code change turns `choice_of/2` form *random term generator* into an *all-term generator*

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

- likewise for our choice operator for the case of arbitrary term algebras
- we do not expect the random sampler to be *uniform*, (except for binary trees), as different function symbol arities introduce a selection bias
- but the resulting samplers are always *exhaustive*, with every term in the set of terms of a given size having a chance to be selected
- we can control the probability with which function symbols are picked
- we can customize the generators to match their frequency in code for which we would like to build a random tester

Parameterizing with the signature

- we parameterize our program with a set of function-symbol/arity pairs
- we classify them into constants and function symbols:

```
?- 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]].
```

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.

The edge splitting step

- we represent the tree as a list of edges
- logic variables mark their open ends
- we store in an edge 4 fields `e(N,I,From,To)`:
  - the arity `N` of the parent function-symbol it originates from
  - the argument position `I` in the parent
  - the logic variables `From` and `To` representing the source and the target of the edge
- in the case of the binary trees 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*
- *we also add the tree below the edge inserted at position* `K`

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

- Like in the case of our variant of Rémy’s algorithm we extract a term by iterating over the edges and binding the logic variables with function-symbols of the appropriate arity for internal nodes and constant symbols for the leaves.
- First we add function symbols to all places in the tree skeleton where their arity matches.
- We fill the remaining free variables, corresponding to leaves, with constants.

Putting it all together

- the predicate `gen_terms(M,FCs,T)`:
- takes as input:
  - the desired size of a generated term `M`
  - 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

Using the generator

- 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)) .
```

- 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],_)).
% in 0.549 seconds
?- time(gen_term(6000,[v/0,a/2],_)).
% in 1.149 seconds
```

Applications

Motzkin trees,SK-combinator trees,Categorial grammars

- Motzkin trees: v/0, l/1, a/2
- SK combinator trees: s/0, k/0, a/2
- categorial grammar expressions: t/0, e/0, left_arrow/2, right_arrow/2

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

- having the same code work as an all-terms and random term generator $\Rightarrow$ cross-checking of properties that we expect to hold in both cases.
- we can check the empirical soundness of the generator by comparing the terms of a given size it outputs with those of the (faster) vanilla generator `gen/2`
- binary trees: 1, 0, 1, 0, 2, 0, 5, 0, 14, 0, 42, 0, 132, 0, 429 ...
- Motzkin trees: 0, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, ...
- $\Rightarrow$ correctness test for for random code generators, as they share the same code
- 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 all terms have a chance to be generated when running the generator in random mode

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

- we compute relative count of the function-symbols occurring in the output of random term and in all-term generators
  - for random terms we sample on a small set, given their very large sizes
  - for all-terms of a given size we compute that by summing them up over all the generated terms of a given size

```
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]
```

Decorating Motzkin trees to closed lambda terms: just add de Bruijn indices

- one can “decorate” it to lambda terms in de Bruijn notation 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
- we can customize it with alternative size definitions, giving 0,1,n weight to variables to closely mimic empirical counts in lambda terms used as intermediate code in functional programs

Conclusions and future directions

Future directions

- 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 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.

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.