Boltzmann Samplers for Closed Simply-Typed Lambda Terms

Maciej Bendkowski[1], Katarzyna Grygiel[1], and Paul Tarau[2(✉)]

[1] Theoretical Computer Science Department, Faculty of Mathematics and Computer Science Jagiellonian University, ul. Prof. Łojasiewicza 6, 30-348 Kraków, Poland
{bendkowski,grygiel}@tcs.uj.edu.pl
[2] Department of Computer Science and Engineering, University of North Texas, Denton, TX, USA
paul.tarau@unt.edu

Abstract. Simply-typed lambda terms are often used in the internal language of compilers and proof assistants, for which generation of large, uniformly distributed random terms is instrumental for testing correctness and scalability. Recently, Boltzmann samplers have enabled uniform random generation of large terms belonging to several families of combinatorial objects that have a regular structure, amenable to methods from analytic combinatorics. Unfortunately, no closed formula or generating function facilitating such methods is known for closed simply-typed lambda terms. Moreover, given their asymptotic sparsity in the family of closed lambda terms, filtering simply-typed terms in the much larger set of terms generated by a Boltzmann sampler becomes quickly intractable. By taking advantage of the synergy between logic variables, unification with occurs check and efficient backtracking in today's Prolog systems we advance this technique to term sizes interesting not only for correctness but also for scalability tests, by deriving Boltzmann samplers returning in a few seconds simply-typed random lambda terms of size 120 and above. We also apply our techniques to the generation of uniformly random closed simply-typed normal forms and give some hints on pushing them further via parallel execution algorithms.

Keywords: Boltzmann samplers · Random generation of simply-typed lambda terms · Type inference · Combinatorics of lambda terms · Random generation of simply-typed normal forms

1 Introduction

Simply-typed lambda terms [1,2] enjoy a number of nice properties, such as strong normalization, i.e., termination for all evaluation-orders, a Cartesian closed category mapping and a set-theoretical semantics. More importantly, via

The first two authors have been partially supported by the Polish National Science Center grant 2013/11/B/ST6/00975. The third author has been supported by NSF grant 1423324.

© Springer International Publishing AG 2017
Y. Lierler and W. Taha (Eds.): PADL 2017, LNCS 10137, pp. 120–135, 2017.
DOI: 10.1007/978-3-319-51676-9_8

the Curry-Howard isomorphism, closed lambda terms that are *inhabitants* of simple types can be seen as proofs for tautologies in the implicational fragment of *minimal logic* which, in turn, correspond to the simple types. Extended with a fix-point operator, simply-typed lambda terms can be used as the intermediate language for compiling Turing-complete functional languages. Recent work on the combinatorics of lambda terms [3–5], relying on generating functions and techniques from analytic combinatorics [6], has provided counts for several families of lambda terms and clarified important quantitative properties of interesting subclasses of lambda terms. With the techniques provided by generating functions [6], it was possible to separate the *counting* of the terms of a given size for several families of lambda terms from their more computation intensive *generation*, resulting in several additions (e.g., **A220894**, **A224345**, **A114851**) to the On-Line Encyclopedia of Integer Sequences [7].

On the other hand, the combinatorics of simply-typed lambda terms, given the absence of closed formulas or context-free grammar-based generators, due to the intricate interaction between type inference and the applicative structure of lambda terms, has left important problems open, including the very basic one of counting the number of closed simply-typed lambda terms of a given size. At this point, obtaining counts for simply-typed lambda terms requires going through the more computation-intensive generation process.

Fortunately, by taking advantage of the synergy between logic variables, unification with occurs check and efficient backtracking it is possible to significantly accelerate the generation of simply-typed lambda terms [8] by interleaving it with type inference steps.

While the generators described in the afore-mentioned paper can push the size of the simply-typed lambda terms by a few steps higher, one may want to obtain uniformly sampled random terms of significantly larger size, especially if one is concerned not only about correctness but also about scalability of compilers and program transformation tools used in the implementation of functional programming languages and proof assistants.

This brings us to the main contribution of this paper. We will first build efficient generators for simply-typed lambda terms that work by interleaving term building and type inference steps. From them, we will derive Boltzmann samplers returning random simply-typed lambda terms [9] of sizes between 120 and 150, assuming a slight variation of the "natural size" introduced in [10], assigning to each constructor a size given by its arity. We will also extend this technique to the random generation of simply-typed closed normal forms, based on the same definition of size.

The paper is organized as follows. Section 2 describes generators for plain, closed and simply-typed terms of a given size. Section 3 derives Boltzmann samplers for random generation of simply-typed closed lambda terms. Section 4 describes generators for lambda terms in normal form as well as their closed and simply-typed subsets. Section 5 derives Boltzmann samplers for random generation of simply-typed closed lambda terms in normal form. Section 6 discusses techniques for possibly pushing higher the sizes of generated random terms. Section 7 overviews related work and Sect. 8 concludes the paper.

The paper is structured as a literate Prolog program. The code has been tested with SWI-Prolog 7.3.8 and YAP 6.3.4. It is also available as a separate file at http://www.cse.unt.edu/~tarau/research/2016/ngen.pro.

2 Generators for Lambda Terms of a Given Natural Size

We start by generating all lambda terms of a given size, in the de Bruijn notation.

2.1 De Bruijn notation

De Bruijn indices [11] provide a robust *name-free* representation of lambda term variables. Closed terms[1] that are identical up to renaming of variables, i.e., are α-convertible, share a unique representation. This allows each variable occurrence to be replaced by a non-negative integer marking the number of lambda abstractions between the variable and its binder. Following [10] we assume a unary notation of integers using the constant `0` and the constructor `s/1` for the successor. Lambda abstraction and application constructors are represented using `l/1` and `a/2`, respectively. And so, the set $\mathcal{L}$ of *plain lambda terms* is given by the following grammar:

$$\mathcal{L} = \mathcal{L}\,\mathcal{L} \;\mid\; \lambda\,\mathcal{L} \;\mid\; \mathcal{D},$$

where $\mathcal{D}$ denotes the set {`0`, `s(0)`, `s(s(0))`, ...} of de Bruijn indices.

Throughout the paper we assume that each constructor is of *weight* equal to its arity and the *size* of a lambda term is the sum of the weights of its building constructors.

2.2 Generating Plain Lambda Terms

Generation of plain lambda terms of a given size proceeds by consuming at each step a size unit, represented by the constructor `s/1`. This ensures that, for a size definition allocating a number of size units to each of the constructors of a term, generation is constrained to terms of a given size. As there are $n+1$ leaves (labeled `0`) in a tree with n `a/2` constructors, we implement our generator to consume as many size-units as the arity of each constructor, in particular 0 for `0` and 2 for the constructor `a/2`. This means that we will obtain the counts for terms of natural size $n+1$ when consuming n size-units.

```
genLambda(s(S),X):-genLambda(X,S,0).

genLambda(X,N1,N2):-nth_elem(X,N1,N2).
genLambda(l(A),s(N1),N2):-genLambda(A,N1,N2).
genLambda(a(A,B),s(s(N1)),N3):-
  genLambda(A,N1,N2),
  genLambda(B,N2,N3).
```

[1] A lambda term is called *closed* if it has no free variables and *open* otherwise. A term is called *plain* if it is either closed or open.

Note that `nth_elem/3` consumes progressively larger size-units for variables of a higher de Bruijn index, a property that conveniently mimics the fact that, in practical programs, variables located farther from their binders are likely to occur less frequently than those closer to their binders.

```
nth_elem(0,N,N).
nth_elem(s(X),s(N1),N2):-nth_elem(X,N1,N2).
```

Example 1. *Plain lambda terms of size 2 (with size of each constructor given by its arity).*

```
?- genLambda(s(s(s(0))),X).
X = s(s(0)) ; X = l(s(0)) ; X = l(l(0)) ; X = a(0, 0) .
```

Counts for plain lambda terms are given by the sequence **A105633** in [7].

2.3 Generating Closed Lambda Terms

We derive a generator for closed lambda terms by counting with help of a list of logic variables. At each lambda binder `l/1` step, a new variable is added to the list associated with a path from the root. For now, we simply use the length of the list as a counter for `l/1` nodes on the path.

The predicate `genClosed/2` builds this list of logic variables as it generates binders. When generating a leaf variable, it picks "non-deterministically" one of the variables among the list of variables corresponding to binders encountered on a given path from the root `Vs`. In fact, this list of variables will be ready to be used later to store the types inferred for a given binder.

```
genClosed(s(S),X):-genClosed(X,[],S,0).

genClosed(X,Vs,N1,N2):-nth_elem_on(X,Vs,N1,N2).
genClosed(l(A),Vs,s(N1),N2):-genClosed(A,[_|Vs],N1,N2).
genClosed(a(A,B),Vs,s(s(N1)),N3):-
  genClosed(A,Vs,N1,N2),
  genClosed(B,Vs,N2,N3).
```

Like `nth_elem` in the case of plain lambda terms, the predicate `nth_elem_on` assigns larger and larger `s/1` weights as the de Bruijn indices, computed in successor arithmetic.

```
nth_elem_on(0,[_|_],N,N).
nth_elem_on(s(X),[_|Vs],s(N1),N2):-nth_elem_on(X,Vs,N1,N2).
```

Example 2. *Closed lambda terms of natural size 5.*

```
?- genClosed(s(s(s(s(0)))),X).
X = l(l(l(s(0)))) ; X = l(l(l(l(0)))) ; X = l(l(a(0, 0))) ;
X = l(a(0, l(0))) ; X = l(a(l(0), 0)) ; X = a(l(0), l(0)) .
```

Counts for closed lambda terms are given by the sequence **A275057** in [7].

2.4 Generating Simply-Typed Lambda Terms

We will derive a generator for simply-typed lambda terms with help from the logic variables used simply as counters in the case of closed terms, to contain the types on which de Bruijn indices pointing to the same binder should agree.

```
genTypable(X,V,Vs,N1,N2):-genIndex(X,Vs,V,N1,N2).
genTypable(l(A),(X->Xs),Vs,s(N1),N2):-genTypable(A,Xs,[X|Vs],N1,N2).
genTypable(a(A,B),Xs,Vs,s(s(N1)),N3):-
  genTypable(A,(X->Xs),Vs,N1,N2),
  genTypable(B,X,Vs,N2,N3).

genIndex(0,[V|_],V0,N,N):-unify_with_occurs_check(V0,V).
genIndex(s(X),[_|Vs],V,s(N1),N2):-genIndex(X,Vs,V,N1,N2).
```

We expose this algorithm via two interfaces: one for plain terms and one for closed terms.

```
genPlainTypable(S,X,T):-genTypable(S,_,X,T).

genClosedTypable(S,X,T):-genTypable(S,[],X,T).

genTypable(s(S),Vs,X,T):-genTypable(X,T,Vs,S,0).
```

For convenience, we shift the sequence by one to match the size definition where both application nodes and `0` leaves have size 1 as originally given in [10]. As there are $n+1$ leaf nodes for n application nodes, consuming two units for an application rather than one for an application and one for a leaf as done in [10], speeds up the generation process as we are able to apply the size constraints at application nodes, earlier in the recursive descent.

Example 3. *Plain simply-typed lambda terms of natural size 3.*

```
?- genPlainTypable(s(s(s(s(0)))),X,T).
X = s(s(s(0))),T = A ;
X = l(s(s(0))),T =  (A->B) ;
X = l(l(s(0))),T =  (A->B->A) ;
X = l(l(l(0))),T =  (A->B->C->C) ;
X = a(0, s(0)),T = A ;
X = a(0, l(0)),T = A ;
X = a(s(0), 0),T = A ;
X = a(l(0), 0),T = A .
```

Counts for plain simply-typed lambda terms, up to size 16, are given by the sequence:

$$0, 1, 2, 3, 8, 17, 42, 106, 287, 747, 2069, 5732, 16012, 45283, 129232, 370761, 1069972.$$

Counts for closed simply-typed lambda terms are given by the sequence **A272794** in [7]. The first 16 entries are:

$$0, 0, 1, 1, 2, 5, 13, 27, 74, 198, 508, 1371, 3809, 10477, 29116, 82419, 233748.$$

3 A Boltzmann Sampler for Simply-Typed Terms

A naive way of sampling uniformly random lambda terms is to generate all terms of a given size and extract a random one out of them. Unfortunately, given the fact that the number of lambda terms grows exponentially with n, this technique quickly becomes intractable.

3.1 Designing Boltzmann Samplers

In their breakthrough paper [12], Duchon et al. introduced a powerful framework of *Boltzmann samplers* meant for random generation of combinatorial objects. Exploiting the analytic nature of the formal power series (see, e.g. [6]) related to the counts of objects in question, as well as their intrinsic recursive structure, it is possible to develop an efficient sampling algorithm.

The key idea behind Boltzmann samplers consists in setting a proper probability space defined on the set of combinatorial objects in such a way that any two objects of the same size are equally likely to be sampled. The price we pay for the efficiency and uniformity is the lack of control over the exact outcome size.

The process of sampling lambda terms follows their top-down recursive structure. At each step, the algorithm decides which constructor to use next, according to pre-computed *branching probabilities*. Depending on the type of the chosen constructor, the sampler either terminates, if `0` was chosen, or proceeds to construct the arguments recursively.

Although the size of the outcome is not deterministic, it is possible to control its *expected* size by adjusting the branching probabilities used in the sampling process. As in [9], the desired branching probabilities can be calibrated to set the expected size to a given finite value.

Such an approach allows us to rapidly sample random plain lambda terms of sizes of order 500,000. Given the asymptotic sparsity of closed simply-typed lambda terms in the set of plain ones [10], the sampling process has to be interleaved with a *rejection* phase where undesired terms are discarded as soon as possible and the whole process is restarted. Due to the immense number of expected retrials, the power of Boltzmann samplers is therefore significantly constrained. Following our empirical experiments, we calibrated the branching probabilities so to set the expected outcome size to 120 – the currently biggest practical size achievable.

3.2 Deriving a Boltzmann Sampler from an Exhaustive Generator

When generating all terms of a given size, the Prolog system explores all possibilities via backtracking. For a random generator, deterministic steps will be used instead, guided by the probabilities determined by the Boltzmann sampler.

Our code is parameterized by the size interval for the generated random terms as well as the maximum number of steps until the *being closed* and *being simply-typed* constraints are both met. Moreover, the code relies on precomputed

constants corresponding to branching probabilities. Their values are obtained according to the recursive combinatorial specification of lambda terms by determining the appropriate complex function and evaluating it in the vicinity of its dominant singularity. The detailed process of computing the desired values is described in [9]. In our case, it turns out that in order to construct a plain term of expected size 120 the probabilities in question are as follows:

– the probability of constructing a de Bruijn index is 0.35700035696434995
– the probability of a lambda abstraction is 0.29558095907
– the probability of an application is 0.34741868396.

Furthermore, whenever we decide to create a de Bruijn index the probability of constructing zero is equal to 0.7044190409261122, while a successor is chosen with probability 0.29558095907. Hence, at each step of the construction process we draw uniformly at random a real from the interval $[0, 1]$ and on its basis we decide which constructor to add.

```
min_size(120).
max_size(150).
max_steps(10000000).
boltzmann_index(R):-R<0.35700035696434995.
boltzmann_lambda(R):-R<0.6525813160382378.
boltzmann_leaf(R):-R<0.7044190409261122.
```

The very high value of retries, `max_steps`, is coming from the discussed sparsity of simply-typed terms among all plain terms. The Boltzmann sampler can be fine-tuned via `min_size` and `max_size` to search for terms in an interval for which the probabilities of the sampler have been calibrated.

The predicate `ranTypable` returns a term `X`, its type `T` as well as the size of the term and the number of trial steps it took to find the term.

```
ranTypable(X,T,Size,Steps):-
  max_size(Max),
  min_size(Min),
  max_steps(MaxSteps),
  between(1,MaxSteps,Steps),
    random(R),
    ranTypable(Max,R,X,T,[],0,Size0),
  Size0>=Min,
  !,
  Size is Size0+1.
```

Note that it calls the predicate `random/1`, returning a random value between 0 and 1, with the convention that each predicate provides such a value for the next one(s) it calls, convention that will be consistently followed in the code.

The predicate `ranTypable/7` follows the outline of the corresponding nondeterministic generator, except that it is driven by deterministic choices provided by the Boltzmann branching probabilities that decide which branch is taken.

Note that the parameter `Max` preempts growing a term above the specified size interval as early as that happens. Like in the generator, on which it is based,

type inference is interleaved with term building. As a result, we prevent building terms with subterms that are not simply-typed, as soon as such a subterm is found.

```
ranTypable(Max,R,X,V,Vs,N1,N2):-boltzmann_index(R),!,
  random(NewR),
  pickIndex(Max,NewR,X,Vs,V,N1,N2).
ranTypable(Max,R,l(A),(X->Xs),Vs,N1,N3):-boltzmann_lambda(R),!,
  next(Max,NewR,N1,N2),
  ranTypable(Max,NewR,A,Xs,[X|Vs],N2,N3).
ranTypable(Max,_R,a(A,B),Xs,Vs,N1,N5):-
  next(Max,R1,N1,N2),
  ranTypable(Max,R1,A,(X->Xs),Vs,N2,N3),
  next(Max,R2,N3,N4),
  ranTypable(Max,R2,B,X,Vs,N4,N5).
```

Besides ensuring that types assigned to a leaf are consistent with the type acquired so far by their binder, the predicate `pickIndex/7` also enforces the property of being a closed term by picking variables from the list of possible binders above it, on the path to the root.

```
pickIndex(_,R,0,[V|_],V0,N,N):-boltzmann_leaf(R),!,
  unify_with_occurs_check(V0,V).
pickIndex(Max,_,s(X),[_|Vs],V,N1,N3):-
  next(Max,NewR,N1,N2),
  pickIndex(Max,NewR,X,Vs,V,N2,N3).
```

Finally, the helper predicate `next/4` ensures that the size count accumulated so far is not above the required interval, while providing a random value to be used by the next call.

```
next(Max,R,N1,N2):-N1<Max,N2 is N1+1,random(R).
```

Example 4. *A uniformly random simply-typed lambda term of size* 137 *and its type, obtained after* 1070126 *trial steps in* 4.388 *s.*

```
l(a(l(l(l(l(l(a(s(s(0)),a(l(a(l(l(l(0))),l(a(0,a(0,a(s(s(0)),
  a(l(a(l(0),a(a(l(l(l(l(s(s(s(0)))))),s(s(0))),a(0,a(0,a(l(l(0)),
  l(a(l(l(l(s(s(s(0))))),s(0))))))))),l(0))))))),a(0,a(s(s(0)),
  a(a(s(0),0),0))))))))),l(a(l(a(0,a(l(l(s(0))),l(l(l(0)))))),
  l(a(l(a(0,a(l(a(l(l(l(l(s(0))))),l(s(s(0))))),l(s(0))))),a(l(l(a(l(0),
  l(a(l(l(l(a(0,a(0,l(l(0)))))),l(s(0)))))),s(s(0)))))))))

(A->B->((C->D->D)->E->F->G)->(((E->F->G)->G)->
  ((E->F->G)->G)->C->D->D)->((E->F->G)->G)->E->F->G)
```

4 Generating Simply-Typed Normal Forms

Normal forms are lambda terms that cannot be further β-reduced. In other words, they avoid *redexes* as subterms, i.e., applications with lambda abstractions on their left branches.

4.1 Generating Normal Forms of Given Size

To generate normal forms we simply add to `genLambda` the constraint `notLambda/1` ensuring that the left branch of an application node is anything except an `l/1` lambda node.

```
genNF(s(S),X):-genNF(X,S,0).

genNF(X,N1,N2):-nth_elem(X,N1,N2).
genNF(l(A),s(N1),N2):-genNF(A,N1,N2).
genNF(a(A,B),s(s(N1)),N3):-notLambda(A),genNF(A,N1,N2),genNF(B,N2,N3).

notLambda(0).
notLambda(s(_)).
notLambda(a(_,_)).
```

Example 5. *Plain normal forms of natural size 5.*

```
?- genNF(s(s(s(s(0)))),X).
X = s(s(s(0))) ;
X = l(s(s(0))) ;
X = l(l(s(0))) ;
X = l(l(l(0))) ;
X = l(a(0, 0)) ;
X = a(0, s(0)) ;
X = a(0, l(0)) ;
X = a(s(0), 0) .
```

Counts for plain (untyped) normal forms, up to size 16, are given by the sequence:

$$0, 1, 2, 4, 8, 17, 38, 89, 216, 539, 1374, 3562, 9360, 24871, 66706, 180340, 490912.$$

4.2 Interleaving Generation and Type Inference

Like in the case of the set of simply-typed lambda terms, we can define the more efficient combined generator and type inferrer predicate `genTypableNF/5`.

```
genPlainTypableNF(S,X,T):-genTypableNF(S,_,X,T).

genClosedTypableNF(S,X,T):-genTypableNF(S,[],X,T).

genTypableNF(s(S),Vs,X,T):-genTypableNF(X,T,Vs,S,0).

genTypableNF(X,V,Vs,N1,N2):-genIndex(X,Vs,V,N1,N2).
genTypableNF(l(A),(X->Xs),Vs,s(N1),N2):-genTypableNF(A,Xs,[X|Vs],N1,N2).
genTypableNF(a(A,B),Xs,Vs,s(s(N1)),N3):-notLambda(A),
  genTypableNF(A,(X->Xs),Vs,N1,N2),
  genTypableNF(B,X,Vs,N2,N3).
```

Example 6. *Simply-typed normal forms of size 6 and their types.*

```
?- genClosedTypableNF(s(s(s(s(s(0))))),X,T).
X = l(l(l(s(0)))),T =  (A->B->C->B) ;
X = l(l(l(l(0)))),T =  (A->B->C->D->D) ;
X = l(a(0, l(0))),T =  (((A->A)->B)->B) ;
```

We are now able to efficiently generate counts for simply-typed normal forms of a given size.

Example 7. *Counts for closed simply-typed normal forms up to size 18.*

$$0, 0, 1, 1, 2, 3, 7, 11, 25, 52, 110, 241, 537, 1219, 2767, 6439, 14945, 35253, 83214.$$

5 Boltzmann Sampler for Simply-Typed Normal Forms

When restricted to normal forms, the Boltzmann sampler is derived in a similar way from the corresponding exhaustive generator. In order to find the appropriate branching probabilities, we exploit the following combinatorial system defining the set $\mathcal{N}$ of *normal forms* using the set $\mathcal{M}$ of so called *neutral forms*.

$$\mathcal{N} = \mathcal{M} \mid \lambda\mathcal{N}$$
$$\mathcal{M} = \mathcal{M}\mathcal{N} \mid \mathcal{D}$$

A normal form is either a neutral term, or an abstraction followed with a normal form. A neutral term, in turn, is either an application of a neutral term to a normal form, or a de Bruijn index.

With this description of normal forms, we are ready to recompute the branching probabilities (see [12] for details) for a Boltzmann sampler generating normal forms. Similarly as in the case of plain terms, we calibrated the branching probabilities so to set the expected outcome size to 120.

The resulting probabilities are given by the following predicates:

```
boltzmann_nf_lambda(R):-R<0.3333158264186935. % an l/1, otherwise neutral
boltzmann_nf_index(R):-R<0.5062759837493023.  % neutral: index, not a/2
boltzmann_nf_leaf(R):-R<0.6666841735813065.   % neutral: 0, otherwise s/1
```

The predicate `ranTypableNF` generates a simply-typed term `X` in normal form and its type `T`, while computing the size of the term and the number of trial steps used to find it. Note the use of Prolog's CUT `!` operation to stop the search once the right size is reached.

```
ranTypableNF(X,T,Size,Steps):-
  max_nf_size(Max),
  min_nf_size(Min),
  max_nf_steps(MaxSteps),
  between(1,MaxSteps,Steps),
    random(R),
    ranTypableNF(Max,R,X,T,[],0,Size0),
  Size0>=Min,
  !,
  Size is Size0+1.
```

First, a probabilistic choice is made between a normal form wrapped up by a lambda binder and a *neutral term*.

```
ranTypableNF(Max,R,l(A),(X->Xs),Vs,N1,N3):-
  boltzmann_nf_lambda(R),!, %lambda
  next(Max,NewR,N1,N2),
  ranTypableNF(Max,NewR,A,Xs,[X|Vs],N2,N3).
```

The choice between the next two clauses is decided by the guard `boltzmann_nf_index`. If satisfied, the recursive path towards a de Bruijn index is chosen. Otherwise, an application is generated. Note the use of the CUT operation (`!` to commit to the first clause when its guard succeeds.

```
ranTypableNF(Max,R,X,V,Vs,N1,N2):-boltzmann_nf_index(R),!,
  random(NewR),
  pickIndexNF(Max,NewR,X,Vs,V,N1,N2). % an index
ranTypableNF(Max,_R,a(A,B),Xs,Vs,N1,N5):- % an application
  next(Max,R1,N1,N2),
  ranTypableNF(Max,R1,A,(X->Xs),Vs,N2,N3),
  next(Max,R2,N3,N4),
  ranTypableNF(Max,R2,B,X,Vs,N4,N5).
```

Finally, the choice is made between the two alternatives deciding how many successor steps are taken until a `0` leaf is reached.

```
pickIndexNF(_,R,0,[V|_],V0,N,N):-boltzmann_nf_leaf(R),!, % zero
  unify_with_occurs_check(V0,V).
pickIndexNF(Max,_,s(X),[_|Vs],V,N1,N3):- % successor
  next(Max,NewR,N1,N2),
  pickIndexNF(Max,NewR,X,Vs,V,N2,N3).
```

Example 8. *A random simply-typed term of size* `63` *in normal form and its type, generated after* `1312485` *trial steps in less than a second.*

```
l(l(l(l(a(a(s(s(0)),l(a(0,a(l(l(s(0))),l(l(l(l(l(a(s(0),l(l(a(s(0),
l(s(0))))))))))))))),l(a(a(l(l(a(l(s(0)),a(a(a(l(s(0)),a(l(0),0)),
l(s(s(0)))),l(l(l(0)))))))),0),l(0)))))))))

(A->((((B->C->D->E->((((F->G)->H)->G->H)->I)->J->I)->K)->K)->
(L->((M->N->O->O)->L)->(M->N->O->O)->L)->P)->Q->R->P)
```

As there are fewer lambda terms of a given size in normal form, one may wonder why we are not reaching comparable or larger sizes to plain lambda terms, where our sampler was able to generate terms over size `120`. An investigation of the relative densities of simply-typed terms in the two sets provides the explanation.

The table in Fig. 1 compares the changes in density for simply-typed terms and simply-typed normal forms. The first column lists the sizes of the terms. Column **A** lists the number of closed simply-typed terms of a given size. Column **B** lists the ratio between plain terms and simply-typed terms. Column **C** lists counts for closed simply-typed normal forms. Column **D** lists the ratio between

Size	A: simpl.-typed	B: plain/simpl.-typed	C: TNF	D: NF/TNF	E: Dens. ratios
5	5	4.400	3	5.666	0.776
10	508	6.988	110	12.490	0.559
15	82,419	10.568	6,439	28.007	0.377
20	16,019,330	15.800	473,628	60.040	0.263

Fig. 1. Comparison of the ratios of simply-typed terms and simply-typed normal forms

terms in normal form and closed simply-typed terms in normal form. Finally, column **E** computes the ratio of the two densities given in columns **B** and **D**.

The plot in Fig. 2 shows the much faster growing sparsity of simply-typed normal forms, measured as the ratio between plain terms and their simply-typed subset and respectively the ratio between normal forms and their simply-typed subset, i.e., the results shown in columns **B** and **D**, for sizes up to `20`.

Finally, the plot in Fig. 3 shows the ratio between these two quantities, i.e., those listed in column **E**, for sizes up to `20`. In both charts the horizontal axis stands for the size, while the vertical one for the number of terms.

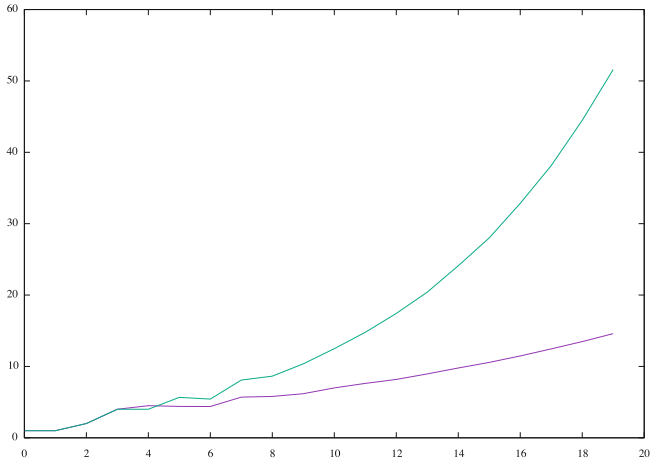

Fig. 2. Sparsity of simply-typed terms (lower curve) vs. simply-typed normal forms (upper curve)

Therefore, we see that closed simply-typed normal forms are becoming very sparse much earlier than their plain counterparts. While, e.g., for size `20` there are around `1/16` closed simply-typed terms for each term, at the same size, for each term in normal form there are around `1/60` simply-typed closed terms in normal form. As at sizes above `50` the total number of terms is intractably high, the increased sparsity of the simply-typed terms in normal form becomes the critical element limiting the chances of successful search.

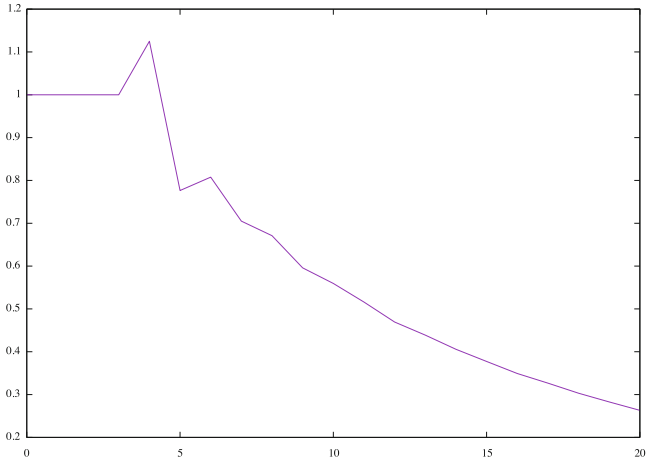

Fig. 3. Ratio between the density of simply-typed closed normal forms and that of simply-typed closed lambda terms

We leave as an open problem the study of the asymptotic behavior of the ratio between the density of simply-typed closed normal forms in the set of all normal forms and the density of simply-typed closed lambda terms in the set of lambda terms. While our empirical data hints to the possibility that it is asymptotically 0 for $n \to \infty$, it is still possible to converge to a small finite limit. Also, this behavior could be dependent on the size definition we are using.

6 Discussion

An interesting open problem is if our method can be pushed significantly farther. We have looked into deep hashing based indexing (`term_hash` in SWI Prolog) and tabling-based dynamic programming algorithms, using de Bruijn terms. Unfortunately as subterms of closed terms are not necessarily closed, even if de Bruijn terms can be used as ground keys, their associated types are incomplete and dependent on the context in which they are inferred.

While it only offers a constant factor speed-up, parallel execution is a more promising possibility. For exhaustive generation, given the small granularity of the generation and type inference process, the most useful parallel execution mechanism would simply split the task of combined generation and inference process into a number of disjoint sets. For instance, assuming size n, and $k \leq n$ `l/1` constructors, one would launch a thread exploring all possible choices, with the remaining $n - k$ size-units to be shared by the applications `a/2` and the weights of indices `s/1`.

For the generation of random terms via Boltzmann sampling, one would simply launch as many threads as the number of processors, with each thread exploring independently the search space.

7 Related Work

The problem of counting and generating uniformly random lambda terms is extensively studied in the literature.

In [5] authors considered a canonical representation of closed lambda terms in which variables do not contribute to the overall term size. The same model was investigated in [3], where a sampling method based on a *ranking-unranking* approach was developed. A binary variant of lambda calculus was considered in [9], leading to a generation method employing Boltzmann samplers. The natural size notion was introduced in [10]. The presented results included quantitative investigations of certain semantic properties, such as strong normalization or typability.

Other, non-uniform generation, approaches are also studied in the context of automated software verification. Prominent examples include Quickcheck [13] and GAST [14] – two frameworks offering facilities for random (yet not necessarily uniform) and exhaustive test generation, used in the verification of user-defined function properties and invariants.

In [15] a "type-directed" mechanism for generation of random terms was introduced, resulting in more realistic (from the particular use case point of view) terms, employed successfully in discovering optimization bugs in the Glasgow Haskell Compiler (GHC).

Function synthesis, given a finite set of input-output examples, was considered in [16]. In this approach, the set of candidate functions is restricted to a subset of primitive recursive functions with abstract syntax trees defined by some context-free grammar, yielding an effective method of finding "natural" functions matching the given example set.

A statistical exploration of the structure of the simple types of lambda terms of a given size in [17] gives indications that some types frequent in human-written programs are among the most frequently inferred ones for terms of a given size.

8 Conclusion

We have derived from logic programs for exhaustive generation of lambda terms programs that generated uniformly distributed simply-typed lambda terms via Boltzmann samplers.

This has put at test a simple but effective program transformation technique naturally available in logic programming languages: interleaving generators and constraints by integrating them in the same predicate.

For the exhaustive generation, we have also managed to work within the minimalist framework of Horn clauses with sound unification, showing that non-trivial combinatorial problems can be handled without any of Prolog's impure features.

Our empirical study of Boltzmann samplers has revealed an intriguing discrepancy between the case of simply-typed terms and simply-typed normal

forms. While these two classes of terms are both known to asymptotically vanish, the significantly faster growth of the sparsity of the later has limited our Boltzmann sampler to sizes below 60.

Our techniques, combining unification of logic variables with Prolog's backtracking mechanism, recommend logic programming as a convenient metalanguage for the manipulation of various families of lambda terms and the study of their combinatorial and computational properties.

The ability to generate uniformly random simply-typed closed lambda terms of sizes above 120 opens the doors for applications to testing compiler components for functional languages and proof assistants, not only for correctness but also for scalability. We hope that simply-typed lambda terms above 120 can be also useful to spot out performance and memory management issues for several algorithms used in these tools, including β-reduction, lambda lifting and type inference.

References

1. Hindley, J.R., Seldin, J.P.: Lambda-Calculus and Combinators: An Introduction, vol. 13. Cambridge University Press, Cambridge (2008)
2. Barendregt, H.P.: Lambda calculi with types. In: Handbook of Logic in Computer Science, vol. 2. Oxford University Press (1991)
3. Grygiel, K., Lescanne, P.: Counting and generating lambda terms. J. Funct. Program. **23**(5), 594–628 (2013)
4. Bodini, O., Gardy, D., Gittenberger, B.: Lambda terms of bounded unary height. In: 2011 Proceedings of the Eighth Workshop on Analytic Algorithmics and Combinatorics (ANALCO), pp. 23–32 (2011)
5. David, R., Grygiel, K., Kozik, J., Raffalli, C., Theyssier, G., Zaionc, M.: Asymptotically almost all λ-terms are strongly normalizing. Logical Meth. Comput. Sci. **9**(1:02), 1–30 (2013)
6. Flajolet, P., Sedgewick, R.: Analytic Combinatorics, 1st edn. Cambridge University Press, New York (2009)
7. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences (2014). https://oeis.org/
8. Tarau, P.: On logic programming representations of lambda terms: de Bruijn indices, compression, type inference, combinatorial generation, normalization. In: Pontelli, E., Son, T.C. (eds.) PADL 2015. LNCS, vol. 9131, pp. 115–131. Springer, Cham (2015). doi:10.1007/978-3-319-19686-2_9
9. Grygiel, K., Lescanne, P.: Counting and generating terms in the binary lambda calculus. J. Funct. Program. **25**, e24 (2015)
10. Bendkowski, M., Grygiel, K., Lescanne, P., Zaionc, M.: A natural counting of lambda terms. In: Freivalds, R.M., Engels, G., Catania, B. (eds.) SOFSEM 2016. LNCS, vol. 9587, pp. 183–194. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49192-8_15
11. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Math. **34**, 381–392 (1972)
12. Duchon, P., Flajolet, P., Louchard, G., Schaeffer, G.: Boltzmann samplers for the random generation of combinatorial structures. Comb. Probab. Comput. **13**(4–5), 577–625 (2004)

13. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp. 268–279. ACM, New York (2000)
14. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: generic automated software testing. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, pp. 84–100. Springer, Berlin (2003). doi:10.1007/3-540-44854-3_6
15. Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, pp. 91–97. ACM, New York (2011)
16. Koopman, P., Plasmeijer, R.: Systematic synthesis of functions, pp. 68–83. The University of Nottingham (2006)
17. Tarau, P.: On Type-directed Generation of Lambda Terms. In: De Vos, M., Eiter, T., Lierler, Y., Toni, F. (eds.) 31st International Conference on Logic Programming (ICLP 2015), Technical Communications, Cork, Ireland, CEUR (2015). http://ceur-ws.org/Vol-1433/