arXiv:1612.07682v1 [cs.LO] 22 Dec 2016

RANDOM GENERATION OF CLOSED SIMPLY-TYPED λ-TERMS

A SYNERGY BETWEEN LOGIC PROGRAMMING AND BOLTZMANN SAMPLERS

MACIEJ BENDKOWSKI, KATARZYNA GRYGIEL, AND PAUL TARAU

Abstract. A natural approach to software quality assurance consists in writing unit tests securing programmer-declared code invariants. Throughout the literature a great body of work has been devoted to tools and techniques automating this labour-intensive process. A prominent example is the successful use of randomness, in particular random typeable λ-terms, in testing functional programming compilers such as the Glasgow Haskell Compiler. Unfortunately, due to the intrinsically difficult combinatorial structure of typeable λ-terms no effective uniform sampling method is known, setting it as a fundamental open problem in the random software testing approach. In this paper we combine the framework of Boltzmann samplers, a powerful technique of random combinatorial structure generation, with today's Prolog systems offering a synergy between logic variables, unification with occurs check and efficient backtracking. This allows us to develop a novel sampling mechanism able to construct uniformly random closed simply-typed λ-terms of up size 120. We apply our techniques to the generation of uniformly random closed simply-typed normal forms and design a parallel execution mechanism pushing forward the achievable term size to 140.

1. Introduction

Simply-typed λ-terms [11, 1] constitute the theoretical foundations of modern functional programming languages, such as Haskell or OCaml. Types in the language provide an additional safety layer as typeable λ-terms are necessarily strongly normalizing, i.e. terminate in all evaluation orders, hindering the programmer from introducing some easy to avoid software bugs. Moreover, via the famous Curry-Howard isomorphism, closed λ-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 itself.

In [14] the authors used random typeable λ-terms as a tool for testing the prominent Glasgow Haskell Compiler (GHC). Though successful for the purpose of finding optimisation bugs in GHC, their random terms were not *uniformly* random with respect to size. In other words, some kinds of typeable λ-terms were favoured over other kinds of equal size terms. Uniform generation, on the other hand, assigns equal probability to terms of equal size and hence produces 'typical' typeable λ-terms, without introducing an unintended nor explicit bias in the sampling process.

Recent work on the combinatorics of λ-terms [9, 3, 5], relying on generating functions and techniques from analytic combinatorics [8], has provided counts for several families of λ-terms and clarified important quantitative properties of interesting subclasses of λ-terms. With the techniques provided by generating functions [8], it was possible to separate the *counting* of the terms of a given size for several families of λ-terms from their more computation intensive *generation*, resulting in several additions (e.g., **A220894**, **A224345**, **A114851**) to the On-Line Encyclopedia of Integer Sequences [15].

On the other hand, the combinatorics of simply-typed λ-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 λ-terms, has left important problems open, including the very basic one of counting the number of closed simply-typed λ-terms of a given size. At this

Date: December 23, 2016.

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.

point, obtaining counts for simply-typed λ-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 [16] by interleaving it with type inference steps. While the generators described in the aforementioned paper can push the size of the simply-typed λ-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 λ-terms that work by interleaving term building and type inference steps. From them, we will derive Boltzmann samplers returning random simply-typed λ-terms [10] of sizes between 120 and 140, assuming a slight variation of the 'natural size' introduced in [2], 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 revisits key notions from analytic combinatorics and the general design of Boltzmann samplers. Section 4 derives Boltzmann samplers for random generation of simply-typed closed λ-terms. Section 5 describes generators for λ-terms in normal form as well as their closed and simply-typed subsets. Section 6 derives Boltzmann samplers for random generation of simply-typed closed λ-terms in normal form. Section 7 describes a simple parallel execution model using multiple independent threads. Section 8 discusses techniques for possibly pushing higher the sizes of generated random terms. Finally, section 9 overviews related work and section 10 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/bol.pro`.

2. Generators for λ-terms of a given natural size

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

2.1. **De Bruijn notation.** De Bruijn indices [6] 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 [2] 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 λ-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 $\{\texttt{0}, \texttt{s(0)}, \texttt{s(s(0))}, \ldots\}$ of de Bruijn indices.

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

2.2. **Generating plain λ-terms.** Generation of plain λ-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

[1] A λ-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.

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).
```

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 λ-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 λ-terms are given by the sequence **A105633** in [15].

2.3. **Generating closed λ-terms.** We derive a generator for closed λ-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 λ-terms, the predicate `nth_elem_on` consumes progressively larger size-units for variables of a higher de Bruijn index. At the same time, its second (list) argument ensures that on each branch leading from a leaf to the root of the tree, there is a variable introduced by a lambda binder above it. This gets our code ready for the next refinement, where we will use these variables to store our inferred types.

```
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 λ-terms of natural size 5.*

```
?- genClosed(s(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 λ-terms are given by the sequence **A275057** in [15].

2.4. **Generating simply-typed λ-terms.** We will derive a generator for simply-typed λ-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).
```

The predicate `genIndex/5` ensures, via *unification with occurs-check*, that the same non-cyclic type is assigned to each leaf corresponding to an occurrence of a variable introduced by a given lambda binder.

```
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. Their only difference is the constraint that the list of available variables for closed terms is initially empty.

```
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 [2]. 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 [2], 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 λ-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 λ-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 λ-terms are given by the sequence **A272794** in [15]. The first 17 entries are:

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

3. Analytic combinatorics

Our approach to random λ-term generation relies on the powerful theory of *analytic combinatorics* and, in particular, the design of Boltzmann samplers. In this section we excerpt the main ideas and notions used throughout the remainder of the paper. We refer the curious reader to [8, 18] for a detailed exposition on generating functions, the singularity analysis process and its applications, as well as [7] for a reference on Boltzmann samplers in general.

3.1. **Symbolic method.** Let $\mathcal{A}$ be a denumerable set of combinatorial objects, e.g. inhabitants of an algebraic data type. Suppose that $\mathcal{A}$ is additionally equipped with a *size notion* $|\cdot|\colon \mathcal{A} \to \mathbb{N}$, assigning each $\alpha \in \mathcal{A}$ its size $|\alpha|$ in such a way that for each $n \in \mathbb{N}$ there are only finitely many objects in $\mathcal{A}$ of size n. Then, $\mathcal{A}$ together with $|\cdot|$ form a *combinatorial class* – the central object in the theory of *analytic combinatorics* [8]. Through analytic combinatorics, we obtain systematic methods of creating, manipulating and studying the behaviour of combinatorial structures, in particular, the properties of large typical objects, as well as their effective random generation.

Let a_n denote the number of objects in $\mathcal{A}$ of size n. Then, $(a_n)_{n\in\mathbb{N}}$ becomes the *counting sequence* of $\mathcal{A}$. Suppose that we assign a formal power series $A(z)$ to $\mathcal{A}$'s counting sequence in such a way that a_n becomes $A(z)$'s nth coefficient (denoted $[z^n]A(z)$), i.e.

$$A(z) = \sum_{n\geq 0} a_n z^n.$$

The series $A(z)$ is called then the *generating function* of $\mathcal{A}$ (see, e.g. [18]).

Such a compact representation of $\mathcal{A}$'s counting sequence enjoys a number of elegant manipulation properties. Suppose we have two disjoint combinatorial classes $\mathcal{A}$ and $\mathcal{B}$ and wish to construct a combinatorial class $\mathcal{C} = \mathcal{A} + \mathcal{B}$ consisting of the union of $\mathcal{A}$ and $\mathcal{B}$ with unmodified size functions. Then, the generating function $C(z)$ of $\mathcal{C}$ is given by

$$C(z) = \sum_{n\geq 0} (a_n + b_n) z^n.$$

Hence, using the formal series addition,

$$C(z) = A(z) + B(z).$$

Now, suppose that we wish to construct a combinatorial class $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ consisting of *pairs* in the form of (α, β) where $\alpha \in \mathcal{A}$, $\beta \in \mathcal{B}$ and the size of (α, β) is the sum of α's and β's sizes. In other words, on the level of generating functions

$$C(z) = \sum_{n\geq 0} \sum_{k=0}^{n} (a_{n-k} b_k) z^n.$$

Note that this is precisely the Cauchy product formula, therefore

$$C(z) = A(z)B(z).$$

The above, so called *admissible constructions*, allow us to find generating functions for a broad class of algebraic data types, including λ-terms in the de Bruijn notation, by means of the following *symbolic method* [8].

Let us start with de Bruijn indices. Recall that de Bruijn indices are defined by the following grammar

$$\mathcal{D} = 0 \mid S(\mathcal{D}).$$

Note that on the right-hand side we have two disjoint sets of indices – a singleton class consisting of 0 and a class of successors. Since zero is of size 0 (as it is a constant), its corresponding generating function is simply equal to $z^0 = 1$. The class of successors, on the other hand, is a bit more involved. Suppose that we construct an auxiliary class $\mathcal{S}$ consisting of a single object $\mathtt{s}$ of size 1. Then, we can bijectively assign to each successor in $\mathcal{D}$ a pair from $\mathcal{S} \times \mathcal{D}$. Since $S(z) = z$, we obtain the following functional equation on $\mathcal{D}$'s generating function $D(z)$

$$\begin{aligned} D(z) &= 1 + zD(z) \\ &= \frac{1}{1-z}. \end{aligned}$$

Now, let us consider the class $\mathcal{L}$ of λ-terms, given by the following grammar

$$\mathcal{L} = \mathcal{D} \mid \lambda\mathcal{L} \mid \mathcal{L}\mathcal{L}.$$

Again, on the right-hand side we have three disjoint combinatorial classes – the set of de Bruijn indices, the set of λ-terms starting with an abstraction, and finally the class of term applications.

Using the same symbolic method as for $\mathcal{D}$, we obtain the following functional equation defining $L(z)$ (recall that the abstraction is of size 1 whereas the application is of size 2):

$$\begin{aligned} L(z) &= D(z) + zL(z) + z^2 L(z)^2 \\ &= \frac{1}{1-z} + zL(z) + z^2 L(z)^2. \end{aligned}$$

Solving the above quadratic equation in $L(z)$ we obtain two possible solutions:

$$L(z) = \frac{1 - z + \frac{\sqrt{1-3z-z^2-z^3}}{\sqrt{1-z}}}{2z^2} \quad \text{or} \quad L(z) = \frac{1 - z - \frac{\sqrt{1-3z-z^2-z^3}}{\sqrt{1-z}}}{2z^2}.$$

Note that since there exists just a single λ-term of size equal to 0 (the de Bruijn index 0), we expect that $\lim_{z\to 0} L(z) = [z^0]L(z) = 1$. This condition holds only for the latter equation, hence finally

$$L(z) = \frac{1 - z - \frac{\sqrt{1-3z-z^2-z^3}}{\sqrt{1-z}}}{2z^2}. \tag{1}$$

3.2. **Singularity analysis.** Although generating functions, as discussed previously, are formal series, *analytic combinatorics* [8] links their analytic properties, when viewed as complex functions in one variable z, with the properties of the underlying counting sequences. Surprisingly profound questions regarding the asymptotic behaviour and statistical properties of the underlying counting sequences might be addressed by carefully examining the *dominant singularities* of the corresponding generating functions (so called *singularity analysis*).

And so, for a broad class of combinatorial classes, including λ-terms in the de Bruijn notation, it is possible to give accurate approximations on the number of objects of size n, or investigate the properties of large random structures. In [2] the authors gave the following asymptotic approximation of $[z^{n+1}]L(z)$ (note that $[z^{n+1}]L(z)$ in their size notion is equal to $[z^n]L(z)$ as given by Equation (1)):

$$[z^{n+1}]L(z) \sim \left(\frac{1}{\rho}\right)^n \frac{C}{n^{3/2}},$$

where $\rho \approx 0.29560$ (hence $1/\rho \approx 3.38298$) and $C \approx 0.60676$. Here, ρ is the *dominant singularity* of $L(z)$, i.e. the radius of convergence of $L(z)$.

Note that in the case of $L(z)$, the location of ρ dictates the exponential rate of growth of $[z^{n+1}]L(z)$. The precise nature and neighbourhood of ρ determine the sub-exponential factor $\frac{C}{n^{3/2}}$. For the purpose of this paper, we are interested in the approximation of ρ and the evaluation of $L(z)$ in arbitrary parameters from $(0, \rho)$, as we use them in the construction of a rejection Boltzmann sampler for λ-terms.

3.3. **Boltzmann samplers.** In their breakthrough paper [7], Duchon et al. introduced a powerful framework of *Boltzmann samplers* meant for random generation of combinatorial structures. Suppose we have a generating function

$$A(z) = \sum_{n \geq 0} a_n z^n.$$

We wish to design an efficient algorithm, which returns a random structure $\alpha \in \mathcal{A}$ in such a way that any two structures of equal size have the same probability of being chosen. In other words, we want the probability $\mathbb{P}(\alpha)$ that $\alpha \in \mathcal{A}$ of size n is the sampler's outcome to be equal to

$$\mathbb{P}(\alpha) = \frac{1}{a_n}.$$

Duchon et al. proposed the following approach. Suppose we relax our restriction that the sampler's outcome size is deterministic and parametrize the sampler with an additional real parameter $x \in (0, \rho)$ where ρ is the dominating singularity of $A(z)$. Let us set a probability space on $\mathcal{A}$ such that $\mathbb{P}_x(\alpha)$, the probability that $\alpha \in A$ is the sampler's outcome, is equal to

$$\mathbb{P}_x(\alpha) = \frac{x^{|\alpha|}}{A(x)}.$$

Let N be the random variable marking the size of the sampler's outcome. Then, the probability $\mathbb{P}_x(N = n)$ that the sampler returns an object of size n is equal to

$$\mathbb{P}_x(N = n) = \frac{a_n x^n}{A(x)}.$$

Note that this is indeed a probability since

$$\sum_{n \geq 0} \mathbb{P}_x(N = n) = \frac{1}{A(x)} \sum_{n \geq 0} a_n x^n = 1.$$

We can therefore consider the *expected* outcome size and all its higher moments. In particular, it is easy to verify that the expected size $\mathbb{E}_x(N)$ and the standard deviation $\sigma_x(N)$ are given by

$$\mathbb{E}_x(N) = x\frac{A'(x)}{A(x)} \quad \text{and} \quad \sigma_x(N) = \sqrt{\frac{x^2 A''(x) + xA'(x)}{A(x)} - \left(x\frac{A'(x)}{A(x)}\right)^2}. \tag{2}$$

Hence, in this model we do not control the exact size of the sample, although we can *calibrate* its expected size and standard deviation by choosing a suitable parameter x.

3.4. **Constructing Boltzmann samplers.** Let $\mathcal{A}$ be a combinatorial class for which we want to design a Boltzmann sampler $\Gamma_x(\mathcal{A})$. The process of constructing $\Gamma_x(\mathcal{A})$ described by Duchon et al. [7] follows the recursive structure of $\mathcal{A}$.

Suppose that $\mathcal{A} = \mathcal{B} + \mathcal{C}$. Let $\alpha \in \mathcal{A}$. Since both $\mathcal{B}$ and $\mathcal{C}$ are disjoint, the probabilities $\mathbb{P}_{\Gamma,x}(\alpha \in \mathcal{B})$ that $\alpha \in \mathcal{B}$ and $\mathbb{P}_{\Gamma,x}(\alpha \in \mathcal{C})$ that $\alpha \in \mathcal{C}$ are equal to

$$\mathbb{P}_{\Gamma,x}(\alpha \in \mathcal{B}) = \frac{B(x)}{A(x)} \quad \text{and} \quad \mathbb{P}_{\Gamma,x}(\alpha \in \mathcal{C}) = \frac{C(x)}{A(x)}.$$

It means therefore that in order to sample an object from $\mathcal{A}$ we have to make a probabilistic decision which branch, i.e. $\mathcal{B}$ or $\mathcal{C}$, to choose. We draw uniformly at random a real $r \in [0, 1]$ and compare it with the *branching probabilities* $\frac{B(x)}{A(x)}$ and $\frac{C(x)}{A(x)}$. Then, we call recursively one of the corresponding samplers $\Gamma_x(\mathcal{B})$ or $\Gamma_x(\mathcal{C})$, continuing the sampling process.

Now, suppose that $\mathcal{A} = \mathcal{B} \times \mathcal{C}$. Let $\alpha = (\beta, \gamma) \in \mathcal{A}$. Note that

$$\mathbb{P}_{\Gamma,x}(\alpha \in \mathcal{A}) = \frac{x^{|\alpha|}}{A(x)} = \frac{x^{|\beta|+|\gamma|}}{B(x)C(x)} = \mathbb{P}_{\Gamma,x}(\beta \in \mathcal{B}) \cdot \mathbb{P}_{\Gamma,x}(\gamma \in \mathcal{C}).$$

In other words, in order to sample an object from $\mathcal{A}$ we have to sample two objects – independently one from $\mathcal{B}$ and one from $\mathcal{C}$ – and make a pair out of them.

The recursion stops at the level of singleton classes. In such a case, we simply return the single object in our class, since

$$\mathbb{P}_{\Gamma,x}(\alpha \in \mathcal{A}) = \mathbb{P}_x(\alpha) = \frac{x^{|\alpha|}}{A(x)} = \frac{x^{|\alpha|}}{x^{|\alpha|}} = 1.$$

Using the above design patterns, we can now easily construct a Boltzmann sampler for plain λ-terms, exploiting the generating function $L(z)$ (see Equation (1)) and its dominant singularity ρ.

4. A Boltzmann sampler for simply-typed terms

The Boltzmann sampler approach allows us to rapidly generate random plain λ-terms of sizes of order 500,000. Unfortunately, given the asymptotic sparsity of closed simply-typed λ-terms in the set of plain ones [2], 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. In order to find the suitable x, we solve numerically Equation (2) for x with $\mathbb{E}_x(N) = 120$. The numerical approximation of x is then

$$x \approx 0.29558095907.$$

Following the construction of Boltzmann samplers in the case of plain λ-terms, we have to compute three branching probabilities deciding whether the sampler generates a random de Bruijn index, an abstraction or an application. And so

- the probability of constructing a de Bruijn index becomes 0.35700035696434995,
- the probability of a lambda abstraction becomes 0.29558095907, and finally
- the probability of an application becomes 0.34741868396.

Furthermore, whenever we decide to create a de Bruijn index, the probability of constructing `0` is equal to 0.7044190409261122, while a successor is chosen with probability 0.29558095907.

4.1. 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 parametrized 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 branching probabilities. 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 non-deterministic 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 λ-term of size* `137` *and its type, obtained after* `1070126` *trial steps in* `4.388` *seconds.*

```
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)
```

5. Generating simply-typed normal forms

Normal forms are λ-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.

5.1. **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:

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

5.2. **Interleaving generation and type inference.** Like in the case of the set of simply-typed λ-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.*

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

6. 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*.

$$\begin{aligned} \mathcal{N} &= \mathcal{M} \mid \lambda\mathcal{N} \\ \mathcal{M} &= \mathcal{M}\mathcal{N} \mid \mathcal{D} \end{aligned}$$

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 [7] 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 and limits 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
min_nf_size(60).
max_nf_size(80).
max_nf_steps(10000000).
```

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(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 λ-terms of a given size in normal form, one may wonder why we are not reaching comparable or larger sizes to plain λ-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 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**.

Size	A: typed	B: plain/typed	C: TNF	D: NF/TNF	E: Density 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

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

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.

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.

We leave as an open problem the study of the asymptotic behaviour of the ratio between the density of simply-typed closed normal forms in the set of all normal forms and the density

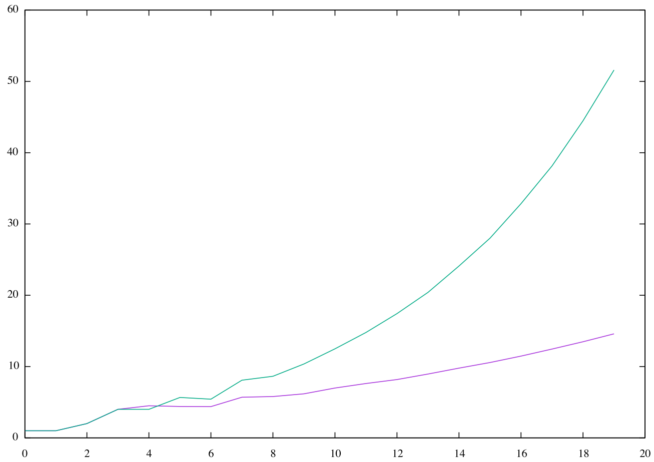

FIGURE 2. Sparsity of simply-typed terms (lower curve) vs. simply-typed normal forms (upper curve).

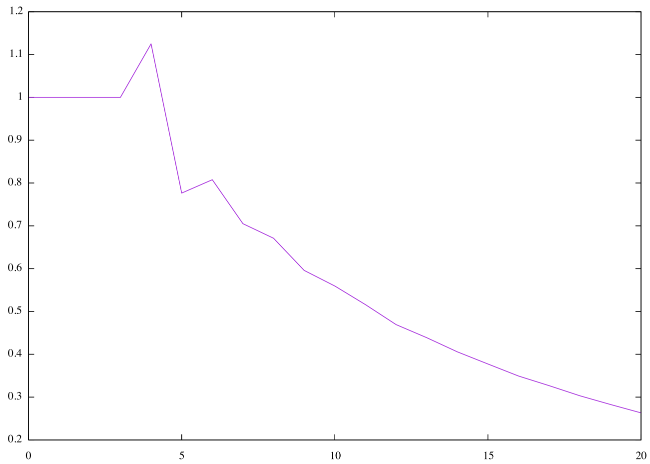

FIGURE 3. Ratio between the density of simply-typed closed normal forms and that of simply-typed closed λ-terms.

of simply-typed closed λ-terms in the set of λ-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 behaviour could be dependent on the size definition we are using.

7. Parallelizing the search

As multiple independent fresh restarts are used in the search for a simply-typed term or normal form, it makes sense to run them in parallel. For multiple threads to work as efficiently as possibly on this task, their number needs to be close to the number of actual processing elements (cores and/or 'hyper-threads', depending on the actual computer running the program). Then, each thread can run exactly the same code until a simply-typed term of the desired size is found, at which point all the other threads should be terminated and the answer returned.

Implementing this model is unusually simple in SWI-Prolog, by using the built-in predicate `first_solution`, that does exactly what we just described, for a list of goals that match the detected number of processing elements. The predicate `multi_gen/1` returns a simply-typed closed `X`, its type `T` as well as its size and the number of steps it took to find it. First it generates its list of goals by replicating the `ranTypable/4` query based on the number of available `cpu_count` flag. Then, the predicate `first_solution` is called to spawn as many threads as the number of goals. The `on_fail(continue)` property hint ensures that one thread failing will allow the others to keep searching.

```
multi_gen(Res):-
  Res=[X,T,Size,Steps],
  prolog_flag(cpu_count,MaxThreads),
  G=ranTypable(X,T,Size,Steps),
  length(Goals,MaxThreads),
  maplist(=(G),Goals),
  first_solution(Res,Goals,[on_fail(continue)]).
```

Our experiments on a 8-threads Ubuntu Linux Machine, with an Intel i-7 processor have consistently returned simply-typed terms of size 140 and larger in less than a minute.

Similarly, the predicate `multi_gen_nf/1` returns a simply-typed normal form `X`, its type `T` as well as its size and the number of steps it took to find it.

```
multi_gen_nf(Res):-
  Res=[X,T,Size,Steps],
  prolog_flag(cpu_count,MaxThreads),
  G=ranTypableNF(X,T,Size,Steps),
  length(Goals,MaxThreads),
  maplist(=(G),Goals),
  first_solution(Res,Goals,[on_fail(continue)]).
```

With 8 threads running, our experiments have consistently returned simply-typed normal forms of size 70 and larger in less than a minute.

8. 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, as shown in section 7, is quite effective in generating closed simply-typed terms of size 140 and larger and simply-typed normal forms of size 70 and higher. This mechanism is based on independent threads, running identical programs. Experiments with more fine-grained execution models (e.g. allowing some sharing of generated subterms) might need more sophisticated inter-thread communication mechanisms, with possible changes to the underlying runtime system.

Note also that 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 `l/1` constructors for $k \leq n$, 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`.

9. Related work

The problem of counting and generating uniformly random λ-terms is extensively studied in the literature. In [5] authors considered a canonical representation of closed λ-terms in which variables do not contribute to the overall term size. The same model was investigated in [9], where a sampling method based on a *ranking-unranking* approach was developed. A binary variant of lambda calculus was considered in [10], leading to a generation method employing Boltzmann samplers. The natural size notion was introduced in [2]. 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 [4] and GAST [12] – 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 [14] 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 [13]. 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 λ-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.

10. Conclusion

We have derived from logic programs for exhaustive generation of λ-terms programs that generated uniformly distributed simply-typed λ-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 sparsity growth of the latter has limited our Boltzmann sampler to sizes roughly equal to 70.

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 λ-terms and the study of their combinatorial and computational properties. The ability to generate uniformly random simply-typed closed λ-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 λ-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] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991.

[2] Maciej Bendkowski, Katarzyna Grygiel, Pierre Lescanne, and Marek Zaionc. A Natural Counting of Lambda Terms. In Rusins Martins Freivalds, Gregor Engels, and Barbara Catania, editors, *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*, volume 9587 of *Lecture Notes in Computer Science*, pages 183–194. Springer, 2016.

[3] Olivier Bodini, Danièle Gardy, and Bernhard Gittenberger. Lambda terms of bounded unary height. In *2011 Proceedings of the Eighth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 23–32, 2011.

[4] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[5] René David, Katarzyna Grygiel, Jakub Kozik, Christophe Raffalli, Guillaume Theyssier, and Marek Zaionc. Asymptotically almost all λ-terms are strongly normalizing. *Logical Methods in Computer Science*, 9(1:02):1–30, 2013.

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

[7] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13(4-5):577–625, 2004.

[8] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 1 edition, 2009.

[9] Katarzyna Grygiel and Pierre Lescanne. Counting and generating lambda terms. *Journal of Functional Programming*, 23(5):594–628, 2013.

[10] Katarzyna Grygiel and Pierre Lescanne. Counting and generating terms in the binary lambda calculus. *Journal of Functional Programming*, 25, 2015.

[11] J Roger Hindley and Jonathan P Seldin. *Lambda-calculus and combinators: an introduction*, volume 13. Cambridge University Press, 2008.

[12] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In *Implementation of Functional Languages: 14th International Workshop, IFL 2002 Madrid, Spain, September 16–18*, pages 84–100, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[13] Pieter Koopman and Rinus Plasmeijer. Systematic synthesis of functions. In *The University of Nottingham*, pages 68–83, 2006.

[14] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST'11, pages 91–97, New York, NY, USA, 2011. ACM.

[15] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. 2014. Published electronically at https://oeis.org/.

[16] Paul Tarau. On Logic Programming Representations of Lambda Terms: de Bruijn Indices, Compression, Type Inference, Combinatorial Generation, Normalization. In E. Pontelli and T. C. Son, editors, *Proceedings of the Seventeenth International Symposium on Practical Aspects of Declarative Languages PADL'15*, pages 115–131, Portland, Oregon, USA, June 2015. Springer, LNCS 8131.

[17] Paul Tarau. On Type-directed Generation of Lambda Terms. In Marina De Vos, Thomas Eiter, Yuliya Lierler, and Francesca Toni, editors, *31st International Conference on Logic Programming (ICLP 2015), Technical Communications*, Cork, Ireland, September 2015. CEUR. available online at http://ceur-ws.org/Vol-1433/.

[18] Herbert S. Wilf. *Generatingfunctionology*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

Theoretical Computer Science Department, Faculty of Mathematics and Computer Science, Jagiellonian University, ul. Prof. Łojasiewicza 6, 30-348 Kraków, Poland

E-mail address: `{bendkowski,grygiel}@tcs.uj.edu.pl`

Department of Computer Science and Engineering, University of North Texas, Denton, TX, USA

E-mail address: `paul.tarau@unt.edu`