HOW TO GENERATE RANDOM LAMBDA TERMS?

MACIEJ BENDKOWSKI[1]

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

Abstract. We survey several methods of generating large random λ-terms, focusing on their closed and simply-typed variants. We discuss methods of exact- and approximate-size generation, as well as methods of achieving size-uniform and non-uniform outcome distributions.

Contents

1 Introduction 1
1.1 Why do we want random lambda terms? 2
1.2 What lambda terms do we want to sample? 3
2 Combinatorial generation methods 4
2.1 Exact-size sampling and the recursive method 5
2.2 Generating functions and the analytic toolbox 7
2.3 Combinatorial bijections and Rémy's algorithm 8
2.4 Boltzmann models 10
2.5 Rejection samplers 12
2.6 Multiparametric samplers 14
3 Conclusion 16
References 16

1. Introduction

Generating examples of various combinatorial structures is an essential part of devising counterexamples to working hypotheses as well as building up confidence in their genuineness. It is a standard mathematical practice used to sieve out and test promising ideas or conjectures before more rigorous and labour-intensive treatment is called for. The more examples pass our tests, the more evidence supports our hypothesis. To illustrate this point, consider the famous $3n+1$ Collatz conjecture.

Conjecture 1 (Collatz, 1937)**.** Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function defined as

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2 \end{cases} \tag{1}$$

E-mail address: `maciej.bendkowski@{tcs.uj.edu.pl,gmail.com}`.
Date: 19th May 2020.

Then, for all $n \geq 1$ there exists a sufficiently large k such that $f^{(k)}(n) = 1$.

The Collatz conjecture remains one of the most prominent open problems in mathematics. It states that given *any* natural number $n \geq 1$, the infinite sequence

$$f(n),\ f(f(n)),\ f(f(f(n))), \ldots \tag{2}$$

must eventually reach the value 1 or, equivalently, fall into the cycle $4 \to 2 \to 1 \to 4 \to \cdots$.

A myriad of computer-generated empirical evidence shows that it holds for all positive integers up to $20 \times 2^{58} \approx 5.7646 \times 10^{18}$ [34]. Although no *proof* of Collatz's conjecture is known, the sheer bulk of empirical data provides high confidence in its genuineness which, in turn, inspires new attempts at solving its mystery.

Nevertheless, even the highest levels of confidence cannot replace rigorous mathematical reasoning. After all, we cannot test a hypothesis for all, infinitely many different inputs. By a twist of perspective, however, all it takes to *disprove* a conjecture is a *single* counterexample. That, of course, can be looked for! A famous example of a conjecture which was refuted by a single counterexample is Euler's sum of powers conjecture.

Conjecture 2 (Euler, 1769)**.** Let $n, k \geq 2$. If $a_1^k + a_2^k + \cdots + a_n^k = a_{n+1}^k$, then $n \geq k$.

This long-standing conjecture was finally settled in the negative by Lander and Parkin [35]. Using a systematic, computer-assisted search procedure they found the following neat counterexample to Euler's claim:

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5. \tag{3}$$

Without the use of computers, such a counterexample might not have been found.

1.1. **Why do we want random lambda terms?** The *generate-and-check* principle of testing working hypotheses is mirrored in software development by *software verification*. Since *proving* properties about large, industrial-strength programs is tremendously difficult and time-consuming, in fact almost infeasible for most types of applications, the modern practice is to *test* them using a large body of carefully crafted tests cases *checking* the intended behaviour of the program. Absolute, mathematical assurances are often given up in favour of feasible, yet reasonable levels of *confidence* in the actual program's *properties*.

Remarkably, test cases need not to be created manually by programmers but, can (at least to some level) be generated *automatically* by a computer. The perhaps most notable examples of such a general testing process in software development include model checking [21] and property-based software testing such as QuickCheck [20]. In testing environments like QuickCheck, the programmer is spared the burden of writing single test cases. Instead, she is given the task of defining *properties* (*invariants*) of tested software and providing a recipe for generating random instances of input data. The testing environment generates hundreds of random test cases and attempts to *disprove* programmer-specified properties. If a counterexample is found, the programmer is given *constructive evidence* that the indented property does not hold. Otherwise, with hundreds or thousands of successful test cases, she can become quite confident in the software's correctness.

One prominent application of random λ-terms comes from testing the correctness of certain program transformations in GHC — Haskell's most famous optimising compiler [39]. Lambda terms are a simple (if not *the* simplest) example of a non-trivial combinatorial structure with variables and corresponding name scopes. Consequently, λ-terms provide the scaffold of simple functional programs and can be used to generate random inputs for compilers to consume. If, for some specific input program P, the tested compiler generates two *observably different* executables (depending on the level of enabled optimisations) P forms a constructive *counterexample* to the working assumption that performed optimisations do not change the semantics of the input program. Remarkably, such a strikingly simple testing method lead to the discovery of several important bugs in GHC's strictness optimiser [44] (see also [40]). In particular, its misuse of the

```
seq :: a -> b -> b
```

function of the following semantics:

```
⊥ `seq` b = ⊥
a `seq` b = b
```

In the current paper we discuss several techniques of generating large, random λ-terms. We do not touch on the conceptually simpler *exhaustive generation* problem in which we are interested in the construction of *all* terms of a fixed size n, like in frameworks such as SmallCheck [48]. Instead, we discuss the various methods of *sampling* (uniformly) random λ-terms of size n, including the two important classes of simply-typed and closed terms.

1.2. **What lambda terms do we want to sample?** Before we begin to discuss *how* to generate random λ-terms, let us pause for a moment and discuss *what* terms we want to sample. In order to sample (uniformly) random terms of some size n, we have to ensure that there exists only a *finite* number of terms of that size. It means that we have to decide what to do with α-equivalent terms and how to *represent* λ-terms. Let us consider these issues in order.

Let N and M be two distinct α-equivalent λ-terms, differing only in names of their bound variables, for instance $\lambda xyz.xy(xz)$ and $\lambda abc.ab(ac)$. Although syntactically different, both represent the same, famous **S** combinator. It can be therefore argued that under reasonable[1] size models, both should be assigned the same *size*. Since we have an infinite supply of variable names, we are consequently inclined to consider α-equivalent terms *indistinguishable*. In other words, focus on sampling α-equivalence classes of λ-terms, instead of their inhabitants.

Given these concerns, we have to decide on a representation in which we do not differentiate bound variable names. We can therefore *pick* one *canonical* representative for each α-equivalence class of terms, in effect sampling terms *up to* α-equivalence, see [54, 55, 12], or use a representation in which there exists just one such representative, for instance the De Bruijn notation [18], see [37, 14, 8]. Recall that in the De Bruijn notation there are no named variables. Instead, we use *dummy* indices $\underline{0}, \underline{1}, \underline{2}, \ldots$ to denote variable occurrences. The index $\underline{n}$ represents a variable which is bound by its $(n+1)$st proceeding abstraction. Should there be fewer abstractions than $n+1$, then $\underline{n}$ represents a free variable occurrence. And so, in the De Bruijn notation both $\lambda xyz.xy(xz)$ and $\lambda abc.ab(ac)$ admit the same representation $\lambda\lambda\lambda.\underline{21}(\underline{20})$, see Figure 1.

Neither representation is better than the other one. What makes them quite different, however, are the *details* of related size models, in particular what *weights* we assign to abstractions, applications, and variables. For instance, how should measure the size of a variable? Should it be proportional to the distance between the variable occurrence and its binding abstraction, or should it be independent of that distance? What toll should be impose on the binder distance? Should it be a linear function or perhaps a more sophisticated one, say logarithmic? If we use De Bruijn indices, how should we

[1]Size models in which the terms size does not depend on the particular names of bound variables.

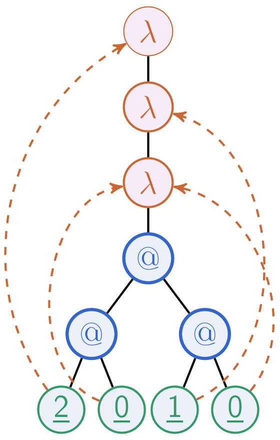


FIGURE 1. Tree-like representation of the term $\lambda\lambda\lambda.\underline{2}\underline{1}(\underline{2}\underline{0})$. Note that the traditionally implicit term application is presented in an explicit form.

represent them? Should we use the more traditional unary base where $\underline{\mathsf{n}}$ is just an n-fold application of the successor function to zero, or should we represent indices in binary format?

Naturally, these representation details prompt *different* size models and, consequently, alter the landscape of random λ-terms we are going to sample. What details should we therefore agree on? Depending on the specifics of the representation and size model, large random λ-terms change their *statistical characteristics*. Knowing these traits, we can choose a specific size model (and representation) which best suits our needs. For instance, large random λ-terms in representations in which variables have constant *weight* tend to be *strongly-normalising* [23]. On the other hand, in De Bruijn models in which variables have size proportional to the distance between them and their binding abstractions, large random terms are almost never *strongly-normalising* [10].

Finding properties of large, random λ-terms is a fascinating and surprisingly challenging endeavour. It often involves sophisticated counting arguments using generating functions and advanced methods of analytic combinatorics [27]. For some size models, however, even these powerful techniques are not so easily applicable. In the current survey, we do not dive into theoretical details of these methods. Instead, we focus more on their *application* to the random generation of λ-terms. We invite the curious reader to follow the referenced literature for more details.

Remark 3. We are interested in generating large, uniformly random λ-terms. With large, *unbiased* terms it is possible to test not only the *correctness* of a compiler or abstract machine implementing the target language, but also test its *performance* and *scalability*, otherwise impossible to test through small input programs.

Our decision has a few notable consequences. We are *not* interested in type-oriented generation methods similar to the Ben-Yelles algorithm [4] or ad-hoc techniques which do not give the user direct and rigorous control over the outcome term distribution. While we are primarily interested in *uniform* distributions, we also address the issue of *non-uniform generation* in Section 2.6.

2. Combinatorial generation methods

Generating random combinatorial structures is a well-established topic in theoretical computer science. Let us examine how standard generation methods can be tailored to construct uniformly random λ-terms.

2.1. **Exact-size sampling and the recursive method.** Nijenhuis and Wilf's recursive method [43], later systematised by Flajolet, Zimmerman and Van Cutsem [28], is a simple, general-purpose recursive sampling template for a wide range of combinatorial objects, including context-free languages and various tree-like structures.

Given a formal specification of so-called *decomposable* structures, the recursive method provides an inductive scheme of constructing recursive samplers meant to generate objects of some fixed size n. In what follows we adopt the recursive method and design a simple sampler for closed λ-terms in the De Bruijn notation:

$$N, M := \underline{\mathsf{n}} \mid (NM) \mid \lambda N. \tag{4}$$

Following the recursive method, terms will be built in a top-down fashion, according to their inductive specification (4). In fact, we are going to devise a slightly more general recursive sampler $\textsc{Gen}_\Gamma(n)$ building terms of size n, with free indices in the set Γ. The desired closed λ-term sampler $\textsc{Gen}_\varnothing(n)$ will be recovered afterwards.

During its execution, $\textsc{Gen}_\Gamma(n)$ will make a series of random choices. Each of these random decisions will we consulted with an external oracle $\mathcal{O}$ providing suitable *branching probabilities* ensuring a uniform outcome distribution. We explain later how to construct such an oracle given the formal λ-term specification (4).

When invoked, $\textsc{Gen}_\Gamma(n)$ has to decide whether to output one of the available indices $\underline{\mathsf{n}} \in \Gamma$, an application of two terms (NM), or an abstraction λN. The sampler queries the oracle $\mathcal{O}$ for respective branching probabilities, and chooses a constructor according to the obtained distribution. Building an index $\underline{\mathsf{n}} \in \Gamma$ is trivial. Assuming that we have enough *fuel* n to pay for the index $\underline{\mathsf{n}}$, we just need to look it up in Γ. Applications and abstractions are a bit more involved.

Note that if λN is a term with free indices in Γ, then the free indices of its subterm N are elements of $\Gamma' = \Gamma_\uparrow \cup \{\underline{\mathsf{0}}\}$ where $\Gamma_\uparrow = \{\underline{\mathsf{n+1}} \colon \underline{\mathsf{n}} \in \Gamma\}$ is the *lifted* variant of Γ. Therefore, if $\textsc{Gen}_\Gamma(n)$ decides to construct an abstraction λN, it can build N by invoking $\textsc{Gen}_{\Gamma'}(n-a)$, accounting a size units for the head abstraction (depending, of course, on the assumed size model).

Let us consider what happens when $\textsc{Gen}_\Gamma(n)$ decides to generate an application (NM). Denote the size contribution of a single application as b. Now, in order to build (NM) we first query the oracle $\mathcal{O}$ for a random $i = 1, \ldots, n-1$. Next, we construct two terms N, M of sizes i and $n-i-b$, respectively. Note that variable contexts of both N and M do not change. We can therefore readily invoke $\textsc{Gen}_\Gamma(i)$ and $\textsc{Gen}_\Gamma(n-i-b)$, and apply N to M.

Oracle construction. In order to make its random decisions $\textsc{Gen}_\Gamma(n)$ repeatedly queries an external oracle $\mathcal{O}$. Suppose that we invoke $\textsc{Gen}_\Gamma(n)$. How should $\mathcal{O}$ compute the necessary branching probabilities? The idea is quite simple.

Let $L_\Gamma(n)$ denote the set of λ-terms of size n with free indices in the set Γ. In order to assign each term in $L_\Gamma(n)$ a uniform probability

$$p = \frac{1}{|L_\Gamma(n)|} \tag{5}$$

we first determine $|L_\Gamma(n)|$. Then, the branching probabilities p_1, p_2 and p_3 corresponding to choosing and index $\underline{\mathsf{n}}$, an abstraction λN, and an application (NM), respectively, are given by

$$
\begin{aligned}
p_1 &= p \cdot |\{\underline{\mathsf{n}} \colon \underline{\mathsf{n}} \in L_\Gamma(n)\}| \\
p_2 &= p \cdot |\{\lambda N \colon \lambda N \in L_\Gamma(n)\}| \\
p_3 &= p \cdot |\{(NM) \colon (NM) \in L_\Gamma(n)\}|.
\end{aligned}
\tag{6}
$$

Likewise, the probability of choosing an application (NM) in which N is of size i and M is of size $n-i-b$ is equal to the proportion of such applications among all applications of size n, all in the common context Γ. It means that in order to compute appropriate branching probabilities our oracle needs to calculate cardinalities of involved sets.

Computing the cardinality of $L_\Gamma(n)$, as well as specific subsets can be achieved using dynamic programming, much like $\textsc{Gen}_\Gamma(n)$ itself. The oracle $\mathcal{O}$ can be precomputed once, memorised, and reused in between subsequent sampler invocations.

Complexity and limitations. The recursive method, albeit simple, admits considerable practical limitations. Both the oracle construction and the following sampling procedure require $\Theta(n^2)$ arithmetic operations on integers exponential in the target size n, turning the effective bit complexity of the recursive method to $O(n^{3+o(1)})$. It is therefore possible to sample λ-terms of moderate sizes in the thousands.

Let us remark that Denise and Zimmerman combined the recursive method with certified floating-point arithmetic, reducing the expected time and space complexity of sampling down to $O(n^{1+o(1)})$ and $O(n^{2+o(1)})$ preprocessing time [24]. Notably, for context-free languages the preprocessing time can be further reduced to $O(n^{1+o(1)})$.

Remark 4. Grygiel and Lescanne [31] proposed a somewhat similar sampler for closed λ-terms based on the following *ranking-unranking* principle.

Given a set S of n elements we wish to sample from, we construct a pair of mutually inverse bijections $f \colon S \to \{0, 1, \ldots, n-1\}$ and $f^{-1} \colon \{0, 1, \ldots, n-1\} \to S$. The former function defines the *rank* $f(s)$ of an element $s \in S$, whereas the latter *unranking* function maps a given rank $f(s)$ to the unique element $s \in S$ attaining its value. Both f and its inverse f^{-1} are constructed in a recursive manner. With these two functions, sampling elements of S boils down to sampling a uniformly random number $i = 0, \ldots, n-1$. Once we obtain a random rank i, we can use the unranking function f^{-1} to construct the corresponding element $f^{-1}(i)$.

Based on the ranking-unranking sampler for closed λ-terms (in the size model where variables do not contribute to the term size) Grygiel and Lescanne derived a rejection sampler for simply-typed λ-terms. Such a sampler generates closed (untyped) terms until a typeable one is sampled. Intermediate terms are simply discarded.

The efficiency of such a rejection scheme depends on two factors — the efficiency of the underlying sampler for untyped terms and, even more importantly, the *proportion* of typeable terms among closed (untyped) terms. Unfortunately, already for $n = 50$ the ratio between typeable terms and untyped ones is less than 10^{-5}, see [31, Section 9.3]. It means that the average number of trials required to find a typeable term of size 50 is already of order 10^5. Such a sampler is not likely to output terms larger than $n = 50$.

Let us remark that Lescanne devised a more direct sampler for *linear* and *affine* λ-terms based on the same ranking-unranking principle [38]. Tracking the evolution of respective (simpler) contexts, Lescanne proposed samplers for a few existing size models.

Remark 5. Wang proposed a recursive sampler for closed λ-terms in a size model where all constructors (*i.e.* variables, applications, and abstractions) contribute one to the overall term size [54]. She then adopted her samplers to simply-typed β-normal forms [55] using a finite truncation of Takahashi, Akama, and Hirokawa's context-free grammars $G(\Gamma, \alpha)$ generating β-normal forms of type α in the context Γ [51].

Since her sampler is based on a dedicated grammar for β-normal forms, it cleverly avoids the slowdown imposed by rejection sampling from the larger set of untyped terms. Unfortunately, such a method comes at a significant price. Grammar-based samplers do not scale onto larger types. Involved grammars $G(\Gamma, \alpha)$ quickly become intractable and too expensive to be effectively constructed.

Let us remark that using the idea of truncated type grammars Asada, Kobayashi, Sin'ya and Tsukada showed recently that, asymptotically almost all typeable λ-terms of order k have a $(k-1)$-fold exponential reduction sequence [2].

2.2. **Generating functions and the analytic toolbox.** Counting and generating random objects are intimately connected activities. In order to construct random λ-terms we usually need to compute a series of branching probabilities. These, in turn, depend on the specific cardinalities of many different term families for various size values.

In order to conveniently operate on (infinite) *counting sequences* enumerating respective cardinalities, we resort to *generating functions*, *i.e.* formal power series $\sum_{n \geq 0} f_n z^n$ whose coefficients $(f_n)_n$ encode the counting sequence we are interested in. To illustrate the convenience of generating functions, consider the famous example of *Catalan numbers* $(c_n)_n$ enumerating, *inter alia*, the number of expressions containing n pairs of matched parentheses, or plane binary trees with n internal nodes. These numbers satisfy the following *recursive* relation:

$$c_{n+1} = \sum_{i=0}^{n} c_i \cdot c_{n-i} \qquad \text{where} \qquad c_0 = 1. \tag{7}$$

With the help of generating functions, its possible to represent the entire infinite counting sequence of Catalan numbers as a *finite* expression. Lifting (7) onto the level of generating functions involving $C(z) = \sum_{n \geq 0} c_n z^n$ we find that

$$C(z) = 1 + zC(z)^2 = \frac{1 - \sqrt{1-4z}}{2z}. \tag{8}$$

In this form, we can recover the nth Catalan number by finding the nth coefficient $[z^n]C(z) = C^{(n)}(0)/n!$ of $C(z)$ using the Taylor series expansion of $C(z)$ around $z = 0$.

Typically, when solving recursive equations like (7) we do not concern ourself with the convergence of associated generating functions. If, however, the generating functions happen to be convergent, they correspond to complex *analytic functions*. We can then draw from the rich fountain of analytic combinatorics [27] — a theory devoted to giving precise quantitative predictions of large combinatorial structures, exploiting for that purpose the analytic properties of related generating functions.

Holonomic functions and P-recursive sequences. We say that a formal power series $f(z) = \sum_{n \geq 0} f_n z^n$ is *holonomic (D-finite)* if it satisfies a linear differential equation

$$P_r(z)\frac{d^r}{dz^r}f(z) + P_{r-1}(z)\frac{d^{r-1}}{dz^{r-1}}f(z) + \cdots + P_1(z)\frac{d}{dz}f(z) + P_0(z)f(z) = 0 \tag{9}$$

for some polynomials $P_i(X) \in \mathbb{C}[X]$, *cf.* [27, Appendix B.4]. By extension, a function $f\colon \mathbb{C} \to \mathbb{C}$ analytic at 0 is said to be *holonomic* if its power series representation is holonomic. Likewise, the coefficient sequence $(f_n)_n$ of a holonomic power series $f(z)$ is called *holonomic* (*P-recursive*).

Holonomic functions are a rich subclass of analytic functions emerging naturally in the context of various enumeration problems. Notably, D-finite functions enjoy a series of pleasing properties. For instance, holonomic generating functions subsume rational and algebraic functions — two classes of generating functions linked to counting problems involving rational and context-free languages. Given an algebraic function $f(z)$ represented as a branch of a polynomial equation $P(z, f(z)) = 0$ it is possible to compute its corresponding D-finite representation [22]. In fact, modern computer algebra systems provide dedicated packages specifically for that purpose, *cf.* [50].

D-finite functions have found numerous applications in combinatorial enumeration and symbolic computations [45]. Remarkably, using respective linear differential equations (9) it is possible to find linear recurrences defining their coefficient sequences $(f_n)_n$ and, consequently, compute the nth coefficient f_n using just $O(n)$ arithmetic operations.

Consider our running example of Catalan numbers. The related holonomic equation for $C(z)$ takes the form

$$1 + (2z-1)\,C(z) + \left(4z^2 - z\right)\frac{d}{dz}C(z) = 0. \tag{10}$$

In the special case of generating functions, the differential operator $\frac{d}{dz}$ admits a natural *combinatorial* interpretation. Note that the defining relation

$$z\frac{d}{dz}C(z) = z\frac{d}{dz}\sum_{n\geq 0} c_n z^n = \sum_{n \geq 1} n \cdot c_n z^n \tag{11}$$

implies that the coefficient $[z^n]z\frac{d}{dz}C(z)$ is equal to $n \cdot [z^n]C(z)$. We can therefore interpret the generating function $z\frac{d}{dz}C(z)$ as encoding the counting sequence of plane binary trees with *pointed* internal nodes. Each tree of size n has n internal nodes and therefore gives rise to n different variants of itself, each with a different node being pointed.

Using (11) we can translate the holonomic equation defining $C(z)$ into a linear recurrence involving its coefficients:

$$(n+2)\,c_{n+1} = 2\,(2\,n+1)\,c_n \quad \text{with} \quad c_0 = 1. \tag{12}$$

It is clear that with the help of (12) we can compute the nth Catalan number c_n using only a linear number of arithmetic operations. On the other hand, using the naïve method derived from the defining identity $C(z) = 1 + zC(z)^2$, we have to compute a sum of products of all $c_0, c_1, \ldots, c_n$ in order to compute the next Catalan number c_{n+1}. Such a process requires $\Theta(n^2)$ arithmetic operations.

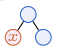
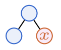
Following similar steps, each holonomic (in particular algebraic) function admits a linear recurrence governing its coefficients. Note that by exploiting this fact we can, for instance, speed up the oracle computations involved in the recursive method.

2.3. **Combinatorial bijections and Rémy's algorithm.**

The simple recurrence (12) defining c_n provides the basis for an elegant sampling algorithm for plane binary trees due to Rémy [47]. The main idea is to interpret it *combinatorially*, using it as an indication on how to map the set of binary trees with n internal nodes to the larger set of trees with $n+1$ internal nodes.

We start with noticing that each binary tree with n internal nodes has $n+1$ leaves and so the total of $2n+1$ nodes. It means that we can interpret (12) as a bijection between two sets of trees — on the left-hand side, the set of binary trees with $n+1$ internal nodes, each with a single *pointed* leave; and on the right-hand side, the set of binary trees with n internal nodes, each with a single pointed node coloured with one of two distinct colours, say red 🔴 or blue 🔵. Let T be a binary tree with n internal nodes. Consider the following *grafting* operation:

- Select a random node (be it internal or external) in T and call it x.

- Flip a coin. Depending on the outcome, colour x either red ● or blue ●.
- If x was coloured red ● replace it with  in T.
- Symmetrically, if x was coloured blue ● replace it with  in T.
- Point to the newly created leave (sibling of x).

Suppose that we started the grafting operation with a (uniformly) random tree with n internal nodes. Note that once we select a random node x and assign it a colour, we end up with a random tree T enumerated by the right-hand side of (12). At this point we readily notice that after the above grafting operation we obtain a random tree enumerated by the left-hand side of (12). We can therefore forget the pointer created in the final grafting step and obtain a random binary tree with $n + 1$ internal nodes.

Clearly, this process can be iterated as many times as required. If we want to sample a random binary tree of size $2n + 1$, we can start with a single node ○ and repeat the grafting operation $n - 1$ times. In his celebrated *Art of Computer Programming* [33] Knuth provides an elegant implementation of Rémy's algorithm which he calls algorithm **R**. With its help, we obtain an efficient, linear time, exact-size sampling algorithm for plane binary trees.

Remark 6. Finding constructive interpretations for combinatorial identities is, in general, a non-trivial and creative task. Rémy's algorithm is somewhat *ad-hoc* and therefore cannot be easily generalised onto other tree structures. Interestingly enough, let us remark that Bacher, Bodini, and Jacquot were able to use ideas of Boltzmann sampling and exploit the holonomic specification for so-called Motzkin trees (*i.e.* unary-binary trees), developing a linear time sampler [3].

Remark 7. Quite often appropriate bijections are derived only after combinatorial identities involving respective generating function are established. For instance, let us mention the non-trivial bijections between combinatorial maps and certain classes of enriched trees leading to efficient samplers for linear and affine λ-terms in the *canonical* representation of λ-terms constructed up to α-equivalence [12], or the bijection between neutral λ-terms and Motzkin trees in the so-called *natural* size model under the De Bruijn notation [10].

Random combinators. Rémy's algorithm is an important sampling algorithm allowing us to generate combinators of any finite bases of primitive combinators. To illustrate this point, consider the example of **SK**-combinators specified as

$$C := \mathbf{S} \mid \mathbf{K} \mid (CC). \tag{13}$$

Assume that the size of a combinator is equal to the number of its internal applications[2]. Suppose that we want to sample a combinator of size n. Note that we can *decompose* each combinator of size n into a plane binary scaffold with n internal nodes, and a sequence of $n+1$ primitive combinators **S** and **K**. The scaffold determines the application structure of the combinator whereas the sequence governs the placement of **S** and **K**s in the term. Moreover, such a decomposition is clearly *invertible* — simply traverse the scaffold in-order and assign elements of the sequence to successive leaves.

The unambiguous decomposition of combinators allows us to use Rémy's algorithm to sample a random scaffold with n internal nodes and, independently, generate a sequence of $n+1$ random primitive combinators. Afterwards, we compose the two into a uniformly random combinator.

[2]If **S** and **K** contribute to the size, the corresponding counting sequence becomes periodic and introduces some technical (though manageable) difficulties, similarly to the case of leaves in plane binary trees.

Remark 8. The above sampling scheme is quite efficient. It allows us to easily sample random combinators even of sizes in the hundreds of millions. Unfortunately, even with the Motzkin tree sampler of Bacher, Bodini, and Jacquot [3] it is not so easy to provide an efficient, analogous sampler for closed λ-terms.

Unlike binary trees, Motzkin trees of some fixed size n might have a varying number of leaves. Moreover, in order to interpret a leave as a variable x, it is important to know the number of its potential binders, *i.e.* unary nodes between x and the term root. We refer the curious reader interested in empirical evaluation of such scaffold decompositions for λ-terms to [52, 16].

2.4. **Boltzmann models.** For many years, the exact-size sampling paradigm was the *de facto* standard in combinatorial generation. Its long-lasting dominance was brought to an end with the seminal paper of Duchon, Flajolet, Louchard, and Schaeffer [25] who introduced the framework of *Boltzmann models.* Instead of generating random structures of fixed size n, Boltzmann models provide a more relaxed, *approximate-size* sampling environment which uses the analytic properties of associated generating functions.

Fix a class $\mathcal{A}$ of combinatorial structures we want to sample. Under the *Boltzmann model* we assign to each object $\alpha \in \mathcal{A}$ a probability measure

$$\mathbb{P}_x(\alpha) = \frac{x^{|\alpha|}}{A(x)} \tag{14}$$

where x is some aptly chosen, positive real-valued *control parameter*, and $|\alpha|$ denotes the size of object α. If we sample in accordance with the above Boltzmann distribution, outcome structures of equal size invariably occur with *the same* probability. Boltzmann models retain therefore the usual requirement of *uniformity* however now, the size of the outcome object is no longer fixed, but instead varies. Indeed, let N be a random variable denoting the outcome size of a *Boltzmann sampler* generating objects according to (14). Summing over all $\alpha \in \mathcal{A}$ of size n we find that

$$\mathbb{P}_x(N = n) = \frac{a_n x^n}{A(x)}. \tag{15}$$

We can control the expected size or generated objects by choosing suitable values of the control parameter x. The expected average and standard deviation of N satisfy, respectively,

$$\mathbb{E}_x(N) = x\frac{A'(x)}{A(x)} \quad \text{and} \quad \sigma_x(N) = \sqrt{x\mathbb{E}'(N)}. \tag{16}$$

Suppose, for instance, that we want to design a Boltzmann model for plane binary trees. Recall that the associated generating function $C(z)$ satisfies the relation $C(z) = 1 + zC(z)^2$. If we want to centre the mean output tree size around some fixed value n, we need to find a suitable control parameter x such that $\mathbb{E}_x(N) = n$. A direct computations reveals that we should use

$$x = \frac{n(n+1)}{(2n+1)^2}. \tag{17}$$

So, if we want to obtain a random tree in some admissible size window $[(1-\varepsilon)n, (1+\varepsilon)n]$ with *tolerance* ε, we calibrate x according to (17) and reject trees falling outside of the prescribed size window.

Boltzmann samplers. For many interesting instances of decomposable combinatorial classes, such as rational languages or algebraic data types, it is possible to *automatically* design appropriate Boltzmann models and corresponding samplers generating random objects in accordance with the underlying Boltzmann distribution, see [25, 19, 6, 46].

Much like exact-size recursive samplers, the Boltzmann sampler layout follows the inductive structure of the target specification. Suppose that we want to sample an object from a (disjoint) union class $\mathcal{A} = \mathcal{B} + \mathcal{C}$. According to the Boltzmann distribution (14) the probability of sampling an object from $\mathcal{B}$ is equal to $B(x)/A(x)$. Likewise, the probability of sampling an object from $\mathcal{C}$ is equal to $C(x)/A(x)$. The corresponding Boltzmann sampler $\Gamma(\mathcal{A})$ for $\mathcal{A}$ performs therefore a single Bernoulli trial with parameter $B(x)/A(x)$. If successful, the sampler invokes $\Gamma(\mathcal{B})$. Otherwise, it calls the remaining sampler $\Gamma(\mathcal{C})$. Clearly, such a sampler conforms with the Boltzmann distribution.

Now, suppose that we want to sample an object from a product class $\mathcal{A} = \mathcal{B} \times \mathcal{C}$. According to the Boltzmann distribution, the probability measure of a *product object* $\alpha = (\beta, \gamma) \in \mathcal{A}$ satisfies

$$\mathbb{P}_x(\alpha) = \frac{x^{|\alpha|}}{A(x)} = \frac{x^{|\beta|+|\gamma|}}{A(x)} = \frac{x^{|\beta|}x^{|\gamma|}}{A(x)} = \mathbb{P}_x(\beta) \cdot \mathbb{P}_x(\gamma). \tag{18}$$

Therefore, in order to sample an object from $\mathcal{A}$ we can independently invoke samplers $\Gamma(\mathcal{B})$ and $\Gamma(\mathcal{C})$, collect their outcomes, and construct a pair out of them. The result is a random object from $\mathcal{A}$.

Remark 9. The disjoint union $+$ and Cartesian product $\times$ suffice to design Boltzmann samplers for algebraic specifications, in particular unambiguous context-free grammars. Let us remark, however, that Boltzmann sampler are not just limited to these two operations. Over the years, efficient Boltzmann samplers for both labelled and unlabelled structures were developed. Let us just mention Boltzmann samplers for various Pólya structures [26], first-order differential specifications [15], labelled planar graphs [29], or plane partitions [11].

In most cases, especially when optimised using *anticipated rejection* [13], Boltzmann samplers are remarkably efficient, frequently generating random structures of sizes in the millions. For typical algebraic specifications, Boltzmann samplers calibrated to the target size window $[(1-\varepsilon)n, (1+\varepsilon)n]$ admit, on average, a linear $O(n)$ time complexity under the real arithmetic computation model [25, 13]. For exact-size sampling where the tolerance $\varepsilon = 0$, the complexity becomes $O(n^2)$.

Remark 10. Boltzmann samplers for λ-terms were first developed by Lescanne [36, 32] for Tromp's binary λ-calculus [53]. Tromp's special encoding of (plain) lambda terms as binary strings, provided an unambiguous context-free specification fitting the framework of Boltzmann samplers. With their help, it became possible to generate λ-terms a couple of orders of magnitude larger than using recursive method.

Boltzmann models, especially if used with rejection methods, provide a rich source of random λ-terms. Although quite general, Boltzmann samplers cannot be used for all term representations and size notions. Recall that in order to derive a corresponding Boltzmann model, we have to, *inter alia*, evaluate the associated generating functions at the calibration parameter x. For some representations, such as for instance Wang's closed λ-term model [54], the associated generating function is not analytic around the complex plane origin[3]. Moreover, without a finite specification we cannot derive a finite set of Boltzmann samplers. In such cases we have to resort to alternative sampling methods including, *e.g.* rejection-based samplers.

[3]In such a case the corresponding counting sequence grows super-exponentially fast, *i.e.* asymptotically faster than any exponential function C^n.

2.5. **Rejection samplers.** Most interesting subclasses of λ-terms, such as closed or (simply) typeable terms, have no known finite, *admissible*[4] specification which would allow us to derive effective Boltzmann samplers. In order to sample from such classes, we *approximate* them using finite, admissible specifications, using rejection methods if needed.

Closed λ-terms. Most λ-term representations, such as [17, 23, 10] admit a common, general symbolic specification template, differing only in weights assigned to specific constructors (*i.e.* variables, applications, and abstractions). Let $\mathcal{L}$, $\mathcal{F}$, $\mathcal{B}$, $\mathcal{A}$, and $\mathcal{U}$ denote the class of (open or closed) λ-terms, free and bound variables, a single application node, and a single abstraction node, respectively. Then, $\mathcal{L}$ admits the following specification:

$$\mathcal{L} = \mathcal{F} + (\mathcal{A} \times \mathcal{L}^2) + \mathcal{U} \times \operatorname{subs}(\mathcal{F} \mapsto \mathcal{F} + \mathcal{D}, \mathcal{L}). \tag{19}$$

In words, a λ-term is either a free variable in $\mathcal{F}$, an application of two terms $(\mathcal{A} \times \mathcal{L}^2)$ joined by an application node, or an abstraction followed by a lambda term in which some free variables become bound by the head abstraction (*i.e.* are substituted by some bound variables in $\mathcal{D}$). Since not all free variables have to be bound by the topmost abstraction, free variables can also be substituted for variables in $\mathcal{F}$.

Let $FV(T)$ denote the set of free variables in T. Consider the following *bivariate* generating function $L(z, f)$ defined as

$$L(z, f) = \sum_{T \in \mathcal{L}} z^{|T|} f^{|FV(T)|}. \tag{20}$$

$L(z, f)$ sums over all terms T using variable z to *account* for Ts size, and variable f to account for the number of its free variables. For instance, consider the combinator $\mathbf{S} = \lambda xyz.xz(yz)$. If we assume that $|\mathbf{S}| = 10$ (one size unit for each abstraction, applications, and variable), then its respective monomial in $L(z, f)$ becomes $z^{10} f^0$.

If we group matching monomials we note the coefficient $[z^n f^k]L(z, f)$ stands for the number of terms of size n and exactly k free variables. In particular, the coefficient $[z^n f^0]L(z, f)$ corresponds to the number of closed λ-terms of size n. We can also look at whole series associated with specific powers. For instance, $[f^0]L(z, f) = \sum_{n \geq 0} z^n f^0$ corresponds to the univariate generating function enumerating closed λ-terms. In general, $[f^k]L(z, f)$ is the generating function corresponding to λ-terms with exactly k free variables.

Unfortunately, by (19) the series $[f^0]L(z, f)$ depends on $[f^1]L(z, f)$ which, in turn, depends on $[f^2]L(z, f)$, *etc.* Note that each new abstraction can bind occurrences of a variable. Consequently, the abstraction body can have one more free variable than the whole term. Due to this apparent infinite inductive nesting, finding the generating function $[f^0]L(z, f)$ for closed terms, as well as generating respective terms using Boltzmann models, is as difficult as the general case $[f^k]L(z, f)$.

Remark 11. Despite the inadmissible specification (19) it is still possible to use recursive samplers for closed λ-terms. The number of free variables is bounded by the overall term size and, in particular, related to the number of term applications. Hence, for target term size n we essentially need to memorise all of the values $[z^{\leq k} f^0]L(z, f), \ldots, [z^{\leq k}, f^n]L(z, f)$ for $k = 0, \ldots, n$.

Still, such a method allows us to sample terms of relatively small sizes, especially if the involved numbers $[z^{\leq k} f^m]L(z, f)$ grow too fast, cf. [17, 54]. Consequently, the standard

[4]Purely in terms of basic constructions like the disjoint sum or Cartesian product, cf. [27, Part A. Symbolic Methods].

practice is to *restrict* the class of generated terms, using practically justifiable criteria, such as limiting their unary height, number of abstractions, or the maximal allowed De Bruijn index, see *e.g.* [14, 17, 5]. In all these cases, the infinite inductive nesting of parameters becomes finite, and so it is possible to use more efficient sampling techniques, such as Boltzmann sampling.

Under certain term representations, it is possible to sample large, unrestricted closed λ-terms, despite the infinite nesting problem. Let us consider the class of λ-terms in the De Bruijn notation. For convenience, assume the natural size notion in which each constructor (including the successor and zero) weights one [10]. Let $L(z)$ denote the corresponding generating function for plain (open or closed) terms and $L_m(z)$ denote the generating function corresponding to so-called m-open λ-terms, *i.e.* terms which when prepended with m head abstractions become closed. Note that if a term is m-open, then it is also $(m+1)$-open.

With the help of $L_m(z)$ we can rewrite (19) into a more verbose, infinite specification exhibiting the precise relations among various openness levels:

$$\begin{aligned}
L_0(z) &= zL_0(z)^2 + zL_1(z)\\
L_1(z) &= zL_1(z)^2 + zL_2(z) + z\\
L_2(z) &= zL_2(z)^2 + zL_3(z) + z + z^2\\
&\dots\\
L_m(z) &= zL_m(z)^2 + zL_{m+1}(z) + z\frac{1-z^m}{1-z}\\
&\dots
\end{aligned} \tag{21}$$

In words, if a term is m-open, then it is either an application of two m-open terms, accounted for in the expression $zL_m(z)^2$, an abstraction followed by an $(m+1)$-term, accounted for in the expression $zL_{m+1}(z)$ or, finally, one of m available indices $\underline{\mathsf{0}}, \ldots, \ldots, \underline{\mathsf{m\text{-}1}}$. Note that $\underline{\mathsf{n}} = S^{(n)}0$ and so $|\underline{\mathsf{n}}| = n+1$. Hence the final expression $z\frac{1-z^m}{1-z}$.

Although infinite, such a specification can be effectively analysed using recently developed analytic techniques [14, 7]. As a by-product of this analysis, it is possible to devise a simple and highly effective rejection-based sampling scheme for closed λ-terms. Consider the following *truncated* variant of (21):

$$\begin{aligned}
L_{0,N}(z) &= zL_{0,N}(z)^2 + zL_{1,N}(z)\\
L_{1,N}(z) &= zL_{1,N}(z)^2 + zL_{2,N}(z) + z\\
L_{2,N}(z) &= zL_{2,N}(z)^2 + zL_{3,N}(z) + z + z^2\\
&\dots\\
L_{N,N}(z) &= zL_{N,N}(z)^2 + zL(z) + z\frac{1-z^N}{1-z}\\
L(z) &= zL(z)^2 + zL(z) + \frac{z}{1-z}
\end{aligned} \tag{22}$$

Note that instead of unfolding the definition of $L_{m+1}(z)$ indefinitely, we stop at level N and use the generating function $L(z)$ corresponding to *all* λ-terms at the final level N. Such a finite specification defines a subclass of plain λ-terms excluding open terms without sufficiently large index values. In particular, this class of terms approximates quite well the set of closed terms.

Following [14] we use (22) to sample plain λ-terms and reject those which are not closed. The asymptotic number $[z^n]L_{0,N}(z)$ of such terms approaches the asymptotic number $[z^n]L_0(z)$ of closed terms exponentially fast as N tends to infinity. In consequence, the imposed rejection does not influence the linear $O(n)$ time complexity of the sampler drawing objects from some interval $[(1-\varepsilon)n, (1+\varepsilon)n]$. In fact, with $N = 20$ the probability of rejection is already of order 10^{-8}, cf. [14, Section 5.4]. Consequently, such a sampler rarely needs to dismiss generated terms and is able to generate large random closed λ-terms of sizes in the millions.

Remark 12. The discussed sampling method works for a broad class of size notions within the De Bruijn term representation [30]. It is, however, specific to the unary encoding of De Bruijn indices.

The intuitive reason behind the success of the above rejection sampler lies is the fact that random plain terms strongly resemble closed ones. Typically, large random terms are *almost* closed. For instance, on average, a random term contains a constant number of free variables, and needs just a few additional head abstractions to become closed [7].

Typeable λ-terms. Much like closed λ-terms, simply-typed terms do not admit a finite, admissible specification from which we could derive efficient samplers. Simple recursive samplers work, however cease to be effective already for small term sizes, cf. Remark 4. Sampling simply-typed terms seems notoriously more challenging than sampling closed ones. Even rejection sampling, whenever applicable, admits serious limitations due to the imminent *asymptotic sparsity problem* — asymptotically almost no term, be it either plain or closed, is at the same time (simply) typeable. This problem is not only intrinsic to De Bruijn models, see [9], but also seems to hold in models in which variables contribute constant weight to the term size, see *e.g.* [23, 17].

Asymptotic sparsity of simply-typed λ-terms is an impenetrable barrier to rejection sampling techniques. As the term size tends to infinity, so does the induced rejection overhead. In order to postpone this inevitable obstacle, it is possible to use dedicated mechanisms interrupting the sampler as soon as it is clear that the partially generated term cannot be extended to a typeable one. The current state-of-the-art samplers take this approach, combining Boltzmann models with modern logic programming execution engines backed by highly-optimised unification algorithms [8]. Nonetheless, even with these sophisticated optimisations, such samplers are not likely to generate terms of sizes larger than one hundred.

Remark 13. For some restricted classes of typeable terms there exist effective sampling methods, such as the already mentioned samplers for linear and affine terms [12]. Since Boltzmann models support all kinds of unambiguous context-free languages, it is also possible to sample other fragments of minimal intuitionistic logic, for instance, using finite truncations *à la* Takahashi *et al.* [51].

2.6. **Multiparametric samplers.**

For some practical applications, especially in industrial property-based software testing [1], the uniform distribution of outcome structures might not be the most useful choice. In fact, it can be argued that most software bugs are miniscule, neglected corner cases, which will not be caught using large, typical instances of random data, see [44, 49].

Remarkably, combinatorial samplers based on either Boltzmann models or the recursive method, can be effectively *tuned* so to distort the intrinsic outcome distribution [6]. For instance, consider again the class $\mathcal{L}$ of λ-terms in the De Bruijn notation with natural size. The corresponding class $\mathcal{D}$ of De Bruijn indices satisfies

$$\mathcal{D} = \mathcal{Z} \times \textsc{Seq}\,(\mathcal{Z}). \tag{23}$$

In words, a De Bruijn index is a (possibly empty) sequence of successors applied to zero. Both each successors and zero contribute weight one to the overall term size.

Such a specification inevitably dictates a geometric distribution of index values and, consequently, an average constant distance between encoded variables and their binders [7]. If such a distribution is undesired, we can, for instance, force the sampler to output terms according to a slightly more *uniform distribution*. Suppose that we change the De Bruijn index specification to the more verbose

$$\mathcal{D} = \mathcal{U}_0\mathcal{Z} + \mathcal{U}_1\mathcal{Z}^2 + \cdots + \mathcal{U}_k\mathcal{Z}^{k+1} + \mathcal{Z}^{k+2}\,\textsc{Seq}\,(\mathcal{Z}) \tag{24}$$

assigning *marking* variables $\mathcal{U}_0, \ldots, \mathcal{U}_k$ to the initial $k+1$ distinct indices $\underline{0}, \ldots, \underline{k}$, leaving the remaining indices unmarked. In doing so, we end up with a multiparametric specification for λ-terms where $\mathcal{Z}$, as usual, marks the term size, and $\mathcal{U}_0, \ldots, \mathcal{U}_k$ mark some selected indices.

This new multivariate specification can be effectively *tuned* in such a way, that branching probabilities governing the sampler's decisions impose a different, non-geometric distribution of index values. For that purpose the problem is effectively expressed as a convex optimisation problem. With the seminal advancements in interior-point methods for convex programming [42] the time required to compute the branching probabilities is proportional to a polynomial involving the optimisation problem size and the number of variables.

To illustrate the distortion effect, suppose that we mark the first nine indices and impose a uniform distribution of 8% of the total term size among them. Remaining indices, are left unaltered. Table 1, taken from [7], provides some empirical frequencies obtained using a tuned sampler and its regular, undistorted counterpart. Specific values were obtained for terms of size around 10,000.

Table 1. Empirical frequencies of index distribution in relation to the term size.

Index	$\underline{0}$	$\underline{1}$	$\underline{2}$	$\underline{3}$	$\underline{4}$	$\underline{5}$	$\underline{6}$	$\underline{7}$	$\underline{8}$
Tuned frequency	8.00%	8.00%	8.00%	8.00%	8.00%	8.00%	8.00%	8.00%	8.00%
Observed frequency	7.50%	7.77%	8.00%	8.23%	8.04%	7.61%	8.53%	7.43%	9.08%
Default frequency	21.91%	12.51%	5.68%	2.31%	0.74%	0.17%	0.20%	0.07%	- - -

Remark 14. Multiparametric tuning requires a few mild technical assumptions, such as the *feasibility* of requested tuning (*e.g.* we cannot ask for the impossible). It can also be used for exact-size sampling or even more involved admissible constructions, including Pólya structures, labelled sets, *etc.* The tuning procedure can be carried out automatically and therefore used to rigorously control the sample distribution without resorting to heuristics, or in-depth expertise of the user.

Remark 15. Boltzmann samplers provide an incredibly versatile and efficient framework for combinatorial generation. Despite the underlying use of heavy theories such as analytic combinatorics and complex analysis, their *design* and *usage* follow simple, recursive rules resembling a small, dedicated *domain-specific language*. With effective tools of their design and tuning, Boltzmann samplers can be automatically compiled without additional expertise in analytic combinatorics or complex analysis [25, 19, 6, 46]. We refer the more *code-minded* reader to concrete implementations[5].

Let us note that analytic methods of (multivariate) Boltzmann sampling are not the only rigorous methods of generating random algebraic data types or λ-terms in particular. While other methods exists, for instance branching processes of Mista *et al.* [41], analytic methods seem to support the generation of far larger structures.

3. Conclusion

Random generation of λ-terms is a young and active research topic on the border of software testing, theoretical computer science and combinatorics. It combines utmost practical experimentation and theoretical advances in analytic combinatorics. Still, many important problems remain open. For instance, how to deal with representations in which corresponding generating functions cease to be analytic, or how to generate large random simply-typed terms?

Acknowledgements. We would like to thank Pierre Lescanne and Sergey Dovgal for encouraging us to write this survey and their valuable remarks during its formation.

References

[1] Thomas Arts, John Hughes, Ulf Norell and Hans Svensson. “Testing AUTOSAR software with QuickCheck”. In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2015, pp. 1–4.

[2] Kazuyuki Asada, Naoki Kobayashi, Ryoma Sin’ya and Takeshi Tsukada. “Almost Every Simply Typed Lambda-Term Has a Long Beta-Reduction Sequence”. In: *Logical Methods in Computer Science* Volume 15, Issue 1 (Feb. 2019).

[3] Axel Bacher, Olivier Bodini and Alice Jacquot. “Exact-size Sampling for Motzkin Trees in Linear Time via Boltzmann Samplers and Holonomic Specification”. In: *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*. SIAM, 2013, pp. 52–61.

[4] Choukri-Bey Ben-Yelles. “Type-assignment in the Lambda-calculus”. PhD thesis. Swansea University, 1979.

[5] Maciej Bendkowski. “Quantitative aspects and generation of random lambda and combinatory logic terms”. PhD thesis. Kraków, Poland: Jagiellonian University, May 2017.

[6] Maciej Bendkowski, Olivier Bodini and Sergey Dovgal. “Polynomial tuning of multiparametric combinatorial samplers”. In: *2018 Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*. 2018, pp. 92–106. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975062.9.

[7] Maciej Bendkowski, Olivier Bodini and Sergey Dovgal. “Statistical properties of lambda terms”. In: *CoRR* abs/1805.09419 (2018). arXiv: 1805.09419.

[8] Maciej Bendkowski, Katarzyna Grygiel and Paul Tarau. “Random generation of closed simply typed λ-terms: A synergy between logic programming and Boltzmann samplers”. In: *Theory and Practice of Logic Programming* 18.1 (2018), 97–119.

[9] Maciej Bendkowski, Katarzyna Grygiel, Pierre Lescanne and Marek Zaionc. “A Natural Counting of Lambda Terms”. In: *SOFSEM 2016: Theory and Practice of Computer Science*. Ed. by Rūsiņš Mārtiņš Freivalds, Gregor Engels and Barbara Catania. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 183–194. ISBN: 978-3-662-49192-8.

[5] https://github.com/Lysxia/boltzmann-samplers,
https://github.com/maciej-bendkowski/paganini,
https://github.com/maciej-bendkowski/boltzmann-brain,
https://github.com/maciej-bendkowski/lambda-sampler.

[10] Maciej Bendkowski, Katarzyna Grygiel, Pierre Lescanne and Marek Zaionc. “Combinatorics of *lambda*-terms: a natural approach”. In: *Journal of Logic and Computation* 27.8 (June 2017), pp. 2611–2630. ISSN: 0955-792X.

[11] Olivier Bodini, Éric Fusy and Carine Pivoteau. “Random sampling of plane partitions”. In: *Combinatorics, Probability and Computing* 19.2 (2010), pp. 201–226.

[12] Olivier Bodini, Danièle Gardy and Alice Jacquot. “Asymptotics and random sampling for BCI and BCK lambda terms”. In: *Theoretical Computer Science* 502 (2013). Generation of Combinatorial Structures, pp. 227 –238. ISSN: 0304-3975.

[13] Olivier Bodini, Antoine Genitrini and Nicolas Rolin. “Pointed versus singular Boltzmann samplers: a comparative analysis”. In: *Pure Mathematics and Application* 25.2 (2015), pp. 115–131.

[14] Olivier Bodini, Bernhard Gittenberger and Zbigniew Gołębiewski. “Enumerating lambda terms by weighted length of their De Bruijn representation”. In: *Discrete Applied Mathematics* 239 (2018), pp. 45 –61. ISSN: 0166-218X.

[15] Olivier Bodini, Olivier Roussel and Michèle Soria. “Boltzmann samplers for first-order differential specifications”. In: *Discrete Applied Mathematics* 160.18 (2012). V Latin American Algorithms, Graphs, and Optimization Symposium — Gramado, Brazil, 2009, pp. 2563 – 2572. ISSN: 0166-218X.

[16] Olivier Bodini and Paul Tarau. “On Uniquely Closable and Uniquely Typable Skeletons of Lambda Terms”. In: *Logic-Based Program Synthesis and Transformation.* Ed. by Fabio Fioravanti and John P. Gallagher. Cham: Springer International Publishing, 2018, pp. 252–268. ISBN: 978-3-319-94460-9.

[17] Olivier Bodini, Danièle Gardy, Bernhard Gittenberger and Zbigniew Gołębiewski. “On the Number of Unary-Binary Tree-Like Structures with Restrictions on the Unary Height”. In: *Annals of Combinatorics* 22.1 (2018), pp. 45–91. ISSN: 0219-3094.

[18] Nicolaas G. de Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392.

[19] Benjamin Canou and Alexis Darrasse. “Fast and Sound Random Generation for Automated Testing and Benchmarking in Objective Caml”. In: *Proceedings of the 2009 ACM SIGPLAN Workshop on ML.* ML ’09. Edinburgh, Scotland: ACM, 2009, pp. 61–70. ISBN: 978-1-60558-509-3.

[20] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming.* ICFP ’00. New York, NY, USA: ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6.

[21] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu and Helmut Veith. “Counterexample-Guided Abstraction Refinement”. In: *Computer Aided Verification.* Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4.

[22] Louis Comtet. “Calcul pratique des coefficients de Taylor d’une fonction algebrique”. In: *Enseignement Mathématiques* 10 (1964), pp. 267 –270.

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

[24] Alain Denise and Paul Zimmermann. “Uniform random generation of decomposable structures using floating-point arithmetic”. In: *Theoretical Computer Science* 218.2 (1999), pp. 233–248.

[25] Philippe Duchon, Philippe Flajolet, Guy Louchard and Gilles Schaeffer. “Boltzmann Samplers for the Random Generation of Combinatorial Structures”. In: *Combinatorics, Probability and Computing* 13.4-5 (2004), pp. 577–625. ISSN: 0963-5483.

[26] Philippe Flajolet, Éric Fusy and Carine Pivoteau. “Boltzmann Sampling of Unlabelled Structures”. In: *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics.*

ANALCO '07. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 201–211.

[27] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009. ISBN: 978-0-521-89806-5.

[28] Philippe Flajolet, Paul Zimmermann and Bernard Van Cutsem. "A calculus for the random generation of labelled combinatorial structures". In: *Theoretical Computer Science* 132.1 (1994), pp. 1–35.

[29] Éric Fusy. "Quadratic exact size and linear approximate size random generation of planar graphs". In: *International Conference on Analysis of Algorithms*. Discrete Mathematics & Theoretical Computer Science, 2005, pp. 125–138.

[30] Bernhard Gittenberger and Zbigniew Gołębiewski. "On the Number of Lambda Terms With Prescribed Size of Their De Bruijn Representation". In: *33rd Symposium on Theoretical Aspects of Computer Science, STACS*. 2016, 40:1–40:13.

[31] Katarzyna Grygiel and Pierre Lescanne. "Counting and generating lambda terms". In: *Journal of Functional Programming* 23.5 (2013), 594–628.

[32] Katarzyna Grygiel and Pierre Lescanne. "Counting and generating terms in the binary lambda calculus". In: *Journal of Functional Programming* 25 (2015), e24.

[33] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. Addison-Wesley Professional, 2006. ISBN: 0321335708.

[34] Jeffrey C. Lagarias, ed. *The Ultimate Challenge: The* $3x + 1$ *Problem*. American Mathematical Society, 2011.

[35] L. J. Lander and T. R. Parkin. "Counterexample to Euler's conjecture on sums of like powers". In: *Bull. Amer. Math. Soc.* 72.6 (Nov. 1966), p. 1079.

[36] Pierre Lescanne. "Boltzmann samplers for random generation of lambda terms". In: *CoRR* abs/1404.3875 (2014). arXiv: 1404.3875.

[37] Pierre Lescanne. "On Counting Untyped Lambda Terms". In: *Theor. Comput. Sci.* 474 (Feb. 2013), 80–97. ISSN: 0304-3975.

[38] Pierre Lescanne. "Quantitative Aspects of Linear and Affine Closed Lambda Terms". In: *ACM Trans. Comput. Logic* 19.2 (June 2018), 9:1–9:18. ISSN: 1529-3785.

[39] Simon Marlow and Simon Peyton Jones. "The Glasgow Haskell Compiler". In: *The Architecture of Open Source Applications, Volume 2*. The Architecture of Open Source Applications, Volume 2. Lulu, 2012.

[40] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson and Hanne Riis Nielson. "Effect-Driven QuickChecking of Compilers". In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017).

[41] Agustín Mista, Alejandro Russo and John Hughes. "Branching Processes for QuickCheck Generators". In: *SIGPLAN Not.* 53.7 (2018), 1–13. ISSN: 0362-1340.

[42] Yurii Nesterov and Arkadii Nemirovskii. *Interior-point polynomial algorithms in convex programming*. SIAM, 1994.

[43] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithms*. 2nd ed. Academic Press, 1978.

[44] 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. Waikiki, Honolulu, USA: ACM, 2011, pp. 91–97. ISBN: 978-1-4503-0592-1.

[45] Marko Petkovšek, Herbert S. Wilf and Doron Zeilberger. $A = B$. CRC Press, 1996. ISBN: 1439864500.

[46] Carine Pivoteau, Bruno Salvy and Michèle Soria. "Algorithms for combinatorial structures: Well-founded systems and Newton iterations". In: *Journal of Combinatorial Theory, Series A* 119.8 (2012), pp. 1711 –1773. ISSN: 0097-3165.

[47] Jean-Luc Rémy. "Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire". In: *ITA* 19.2 (1985), pp. 179–195.

[48] Colin Runciman, Matthew Naylor and Fredrik Lindblad. "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values." In: *Haskell.* Ed. by Andy Gill. ACM, 2008, pp. 37–48. ISBN: 978-1-60558-064-7.

[49] Colin Runciman, Matthew Naylor and Fredrik Lindblad. "Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values". In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell.* Haskell '08. Victoria, BC, Canada: ACM, 2008, pp. 37–48. ISBN: 978-1-60558-064-7.

[50] Bruno Salvy and Paul Zimmermann. "Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable". In: *ACM Transactions on Mathematical Software* 20.2 (1994), pp. 163–177.

[51] Masako Takahashi, Yohji Akama and Sachio Hirokawa. "Normal Proofs and Their Grammar". In: *Information and Computation* 125.2 (1996), pp. 144 –153. ISSN: 0890-5401.

[52] Paul Tarau. "On k-colored Lambda Terms and Their Skeletons". In: *Practical Aspects of Declarative Languages.* Ed. by Francesco Calimeri, Kevin Hamlen and Nicola Leone. Cham: Springer International Publishing, 2018, pp. 116–131. ISBN: 978-3-319-73305-0.

[53] John Tromp. "Binary Lambda Calculus and Combinatory Logic". In: *Kolmogorov Complexity and Applications.* Ed. by Marcus Hutter, Wolfgang Merkle and Paul M.B. Vitanyi. Dagstuhl Seminar Proceedings 06051. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[54] Jue Wang. *Generating random lambda calculus terms.* Tech. rep. Boston University, 2005.

[55] Jue Wang. *Generating Random Terms in Beta Normal Form of the Simply-Typed Lambda Calculus.* Tech. rep. Boston University, 2005.