Counting and generating lambda terms

Katarzyna Grygiel, Pierre Lescanne

Counting and generating lambda terms

Katarzyna Grygiel*
Theoretical Computer Science Department,
Faculty of Mathematics and Computer Science,
Jagiellonian University,
ul. Prof. Łojasiewicza 6, 30-348 Kraków, Poland
email: grygiel@tcs.uj.edu.pl

Pierre Lescanne
ENS de Lyon,
LIP (UMR 5668 CNRS ENS Lyon UCBL INRIA)
University of Lyon,
46 allée d'Italie, 69364 Lyon, France
email: pierre.lescanne@ens-lyon.fr

April 11, 2013

Abstract

Lambda calculus is the basis of functional programming and higher order proof assistants. However, little is known about combinatorial properties of lambda terms, in particular, about their asymptotic distribution and random generation. This paper tries to answer questions like: How many terms of a given size are there? What is a "typical" structure of a simply typable term? Despite their ostensible simplicity, these questions still remain unanswered, whereas solutions to such problems are essential for testing compilers and optimizing programs whose expected efficiency depends on the size of terms. Our approach toward the afore-mentioned problems may be later extended to any language with bound variables, i.e., with scopes and declarations.

This paper presents two complementary approaches: one, theoretical, uses complex analysis and generating functions, the other, experimental, is based on a generator of λ-terms. Thanks to de Bruijn indices, we provide three families of formulas for the number of closed lambda terms of a given size and we give four relations between these numbers which have interesting combinatorial interpretations. As a by-product of the counting formulas, we design an algorithm for generating lambda terms. Performed tests provide us with experimental data, like the average depth

*Supported by the National Science Center of Poland, grant number 2011/01/B/HS1/00944.

of bound variables and the average number of head lambdas. We also create random generators for various sorts of terms. Thereafter, we conduct experiments that answer questions like: What is the ratio of simply typable terms among all terms? *(Very small!)* How are simply typable lambda terms distributed among all lambda terms? *(A typable term almost always starts with an abstraction.)*

In this paper, abstractions and applications have size 1 and variables have size 0.

Keywords: lambda calculus, combinatorics, functional programming, test, random generator

1 Introduction

Let us start with a few questions relevant to the problems we address.

- How many closed λ-terms are of size 50 (up to α-conversion)?

  $$99665778334452328341705500204014807522670099639155869526994685 2267.$$

- How many closed terms of size n are there?
  We will give a recursive formula for this number in Section 2*.*

- What does the following sequence enumerate:

  $$0, 1, 3, 14, 82, 579, 4741, 43977, 454283, 5159441, 63782411?$$

  This sequence enumerates closed terms of size 0*,* 1*,* 2*,* 3*,* 4*,* 5*,* 6*,* 7*,* 8*,* 9*,* 10*. It is the sequence A220894 of the Online Encyclopedia of Integer Sequences (*`https://oeis.org/A220894`*). We will provide three ways to compute it (Section* 4*).*

- Is it possible to generate simply typable terms randomly?
  Yes, according to the process which consists in generating random lambda terms with uniform probability and sieving those that are simply typable. Thus, we can generate random simply typable terms of size up to 50*.*

- Is a term starting with an abstraction more likely to be typable than a term starting with an application?
  The answer is positive as shown in Figure 9*, which gives the distribution of simply typable lambda terms among all lambda terms.*

- Do these results have practical consequences?
  Yes, they enable random generation of simply typable terms in an efficient way in the case of terms of size up to 50 *in order to debug compilers or other programs, manipulating terms, e.g., type checkers or pretty printers.*

The above questions seem rather classical, but amazingly very little is known about combinatorial aspects of lambda terms, probably because of the intrinsic difficulty of the combinatorial structure of lambda calculus due to the presence of bound variables. However, the answers to these questions are extremely important not only for a better

understanding of the structure of lambda terms, but also for people who build test samples for debugging compilers. Perhaps the reason of this ignorance lies in the surprising form of the recurrences. Indeed, due to the presence of bound variables, the recurrence does not work in the way mathematicians expect and are used to. The induction lies on two variables, one decreasing (the size), the other increasing (the number of free variables). Thus none of the methods used in the reference book of Flajolet and Sedgewick [7] applies. Why is that? In what follows we compute the number of lambda terms (and of normal forms) of size n with at most m distinct free variables. Denoting the number of such terms by $T_{n,m}$, the formula for $T_{n,m}$ contains $T_{n-1,m+1}$ and this growth of m makes the formula averse to treatments by generating functions and classical analytic combinatorics. We notice that for a given n the expression for $T_{n,m}$ is a polynomial in m. These polynomials can be described inductively and their coefficients are given by recurrence formulas. These formulas are still complex, but can be used to compute the constant coefficients, which correspond to the numbers of closed lambda terms. For instance, the leading coefficients of the polynomials are the well known Catalan numbers which count binary trees.

In order to find the recurrence formula for the number of λ-terms of a given size, we make use of the representation of variables in λ-terms by de Bruijn indices. Recall that a de Bruijn index is a natural number which replaces a term variable and enumerates the number of λ's encountered on the way between the variable and the λ which binds the latter. In this paper, we assume the combinatorial model in which the size of each occurrence of abstraction or application is counted as 1, while the size of variables (de Bruijn indices) as 0. This method is a realistic model of the complexity of λ-terms and allows us to derive the recurrences very naturally.

From the formula for counting λ-terms we derive one-to-one assignments of terms of size n with at most m distinct free indices to the numbers in the interval $[1..T_{n,m}]$. From this correspondence, we develop a program for generating λ-terms, more precisely for building λ-terms associated with numbers in the interval $[1..T_{n,m}]$. If we pick a random number in the interval, then we get a random term of size n with at most m distinct free variables. Most of the time we consider closed λ-terms, which means $m = 0$. Beside the interest in such a random generation for applications like testing, this allows us to compute practical values of parameters by Monte-Carlo methods. Overall, we are able to build a random generator for simply typable terms. Unlike the method used so far [17], which consists in unfolding the typing tree, we generate random λ-terms and test their typability, until we find a simply typable term. This method allows us to generate on a laptop simply typable λ-terms up to size 50. We also use this method to describe the distribution of typable terms among all terms and typable normal forms among all normal forms.

Structure of the paper

According to its title, the paper is divided into two parts, one focuses on counting terms and its mathematical treatment, the other on term generation and its applications. The first part (Sections 2 and 5) is devoted to the formulas counting λ-terms. In Section 2 we study polynomials giving the numbers of terms of size n with at most m distinct free variables. In Section 3, we show that the numbers of i-contexts give a combinatorial interpretation of the coefficients of the polynomials and yield a new formula for counting

the closed terms of size n. If we add formulas for counting lambda terms of size n with exactly m distinct free variables, we have three formulas of three different origins for counting closed terms which we describe and compare in Section 4. In Section 5 we derive generating functions and asymptotic values for these coefficients. In Section 6 we give a formula for counting normal forms. In the second part of the paper, i.e., in Section 7 and Section 8, we propose programs to generate untyped and typable terms and normal forms. Section 9 is devoted to experimental results. Section 10 presents related works.

2 Counting terms with at most m distinct free variables

We represent terms using de Bruijn indices [5], which means that variables are represented by numbers $\underline{1}, \underline{2}, \ldots, \underline{\mathsf{m}}, \ldots$, where an index, for instance $\underline{\mathsf{k}}$, is the number of λ's above the location of the index and below the λ that binds the variable, in a representation of λ-terms by trees. For instance, the term with variables $\lambda x.\lambda y.x\,y$ is represented by the term with de Bruijn indices $\lambda\lambda\underline{21}$. The variable x is bound by the top λ. Above the occurrence of x there are two λ's, therefore x is represented by $\underline{2}$, and from the occurrence of y we count just the λ that binds y, so y is represented by $\underline{1}$. Notice that unlike [12] and like [5, 1] we start indices at 1, since it fits better with our aim of counting terms.

In what follows, by *terms* we mean untyped terms with de Bruijn indices and we often speak indistinctively of variables and (de Bruijn) indices. Assume that in a term t not all occurrences of indices need to be bound, i.e., there may occur indices that do not correspond to surrounding λ's. Such indices are called "free" in t. Now, we introduce the notational convention for "free" indices occurring in terms. An *interval of free indices* for a term t is a set $\{\underline{1}, \underline{2}, \ldots, \underline{\mathsf{m}}\}$ of indices, written $[\underline{1}..\underline{\mathsf{m}}]$, such that

(i) if t is an index $\underline{\mathsf{i}}$, then any interval $[\underline{1} \ldots \underline{\mathsf{m}}]$ with $1 \leq i \leq m$ is an interval for t,

(ii) if t is an abstraction λs and an interval of free indices for s is $[\underline{1} \ldots \underline{\mathsf{m}+1}]$, then the interval of free indices for t is $[\underline{1} \ldots \underline{\mathsf{m}}]$ (since the index $\underline{1}$ is now bound and the others are assumed to decrease by one),

(iii) if t is an application $t_1 t_2$ and an interval of indices for t_1 and t_2 is $[\underline{1} \ldots \underline{\mathsf{m}}]$, then an interval of indices for t is $[\underline{1} \ldots \underline{\mathsf{m}}]$.

We measure the size of a term in the following way:

$$\begin{aligned} |\underline{\mathsf{m}}| &= 0, \text{ for every index } \underline{\mathsf{m}}, \\ |\lambda t| &= |t| + 1, \\ |ts| &= |t| + |s| + 1. \end{aligned}$$

Since lambda terms can be represented as unary-binary trees with labels or pointers, the notion of size of a term t corresponds to the number of unary and binary vertices in the tree representing t. This also means that adding a new variable (in other words, adding a new leaf to a tree) or a new operator (a unary or a binary vertex) increases always the size of a term by 1.

n\m	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	3	7	13	21	31	43
2	3	13	41	99	199	353	573
3	14	76	312	962	2386	5064	9596
4	82	542	2784	10732	32510	82122	181132
5	579	4493	27917	131715	482015	1440929	3687513
6	4741	42131	307943	1741813	7612097	26763551	79193491
7	43977	439031	3690055	24537945	126536933	519788827	1771730211
8	454283	5020105	47635777	365779679	2198772055	10477986133	40973739725
9	5159441	62382279	658405747	5744911157	39769404045	218213327131	974668783199
10	63782411	835980065	9695617821	94786034723	746744227319	4681133293821	23769847893305
11	851368766	12004984120	151488900012	1639198623818	14531624611594	103244315616876	593009444765240
12	12188927818	183754242626	2502346785164	29658034018852	292747054367966	2338363467319958	15112319033576416
13	186132043831	2984264710781	43560247035581	560484305049943	6100545513799835	54347237563601321	393031286917940401
14	3017325884473	51220227153987	796828655891895	11046637024014049	131425939696979805	1295642289776992983	10425601907159190187

Figure 1: Values of $T_{n,m}$ for n and m up to 14 and 6, respectively

Let us denote by $\mathcal{T}_{n,m}$ the set of terms of size n with at most m distinct free de Bruijn indices. This notation for combinatorial structures is borrowed from [7].

$$\begin{aligned}\mathcal{T}_{0,m} &= [\underline{1}\dots\underline{m}]\\ \mathcal{T}_{n+1,m} &= \lambda\mathcal{T}_{n,m+1} \uplus \biguplus_{k=0}^{n} \mathcal{T}_{n-k,m}@\mathcal{T}_{k,m}.\end{aligned}$$

For all $n, m \in \mathbb{N}$, let $T_{n,m}$ denote the cardinality of the set $\mathcal{T}_{n,m}$ (again a notation of [7]). According to the definition of size, operators λ and @ have size 1 and de Bruijn indices have size 0. Therefore, we get the following two equations specifying $T_{n,m}$:

$$\begin{aligned}T_{0,m} &= m\\ T_{n+1,m} &= T_{n,m+1} + \sum_{i=0}^{n} T_{i,m}T_{n-i,m}.\end{aligned}$$

This means that there are m terms of size 0 with at most m distinct free de Bruijn indices, which are terms that are just these indices. Terms of size $n+1$ with at most m distinct free de Bruijn indices are either abstractions with at most $m+1$ distinct free indices on a term of size n or applications of terms with at most m distinct free indices to make a term of size $n+1$. As we said in the introduction, the 11 first values of $T_{n,0}$ are:

$$0, 1, 3, 14, 82, 579, 4741, 43977, 454283, 5159441, 63782411.$$

$T_{n,0}$ is sequence **A220894** in the *On-line Encyclopedia of Integer Sequences*.

Figure 1 gives all the values of $T_{n,m}$ for n up to 14 and m up to 6. For instance, there is 1 closed term of size 1, namely $\lambda\underline{1}$, there are 3 closed terms of size 2, namely $\lambda\lambda\underline{1}, \lambda\lambda\underline{2}, \lambda\underline{1}\,\underline{1}$, and there are 14 closed terms of size 3, namely

$$\begin{array}{ccccccc}\lambda\lambda\lambda\underline{1} & \lambda\lambda\lambda\underline{2} & \lambda\lambda\lambda\underline{3} & \lambda\lambda\underline{1}\,\underline{1} & \lambda\lambda\underline{1}\,\underline{2} & \lambda\lambda\underline{2}\,\underline{1} & \lambda\lambda\underline{2}\,\underline{2}\\ \lambda(\underline{1}\,\lambda\underline{1}) & \lambda(\underline{1}\,\lambda\underline{2}) & \lambda\underline{1}(\underline{1}\underline{1}) & \lambda((\lambda\underline{1})\,\underline{1}) & \lambda((\lambda\underline{2})\,\underline{1}) & \lambda((\underline{1}\,\underline{1})\,\underline{1}) & (\lambda\underline{1})\,\lambda\underline{1}.\end{array}$$

Notice that in Section 7 we describe how to assign a term to a number and therefore how to list terms with increasing numbers. The above terms are listed in that order.

2.1 Computing the $T_{n,m}$'s

The recursive definition of T yields an easy naive program in a functional programming language (here Haskell):

```
naiveT :: Int -> Int -> Integer
naiveT 0 m = (fromIntegral m)
naiveT n m = naiveT (n-1) (m+1) +
             foldl (+) 0 (let tt = [naiveT i m | i<- [0..(n-1)]] in
                          (map (uncurry (*)) (zip tt (reverse tt))))
```

This program is inefficient since it recomputes the values of T at each recursive call. For actual computations a program with memoization is required. In Sage this is obtained by requiring the function to be "cashed". In Haskell we use the laziness of streams:

```
ttab :: [[Integer]]
ttab = [0..]:[[ttab !! (n-1) !! (m+1) + s n m
                | m <- [0..]] | n <- [1..]]
  where s n m  = let ti = [ttab !! i !! m | i <- [0..(n-1)]] in
                 foldl (+) 0 (map (uncurry (*)) (zip ti (reverse ti)))

t :: Int -> Int -> Integer
t n m = ttab !! n !! m
```

The first computation of `ttab !! n !! 0` requires $O(n^2)$ additions and $O(n^2)$ multiplications. Therefore, the complexity of the first computation of `t n 0` depends on the complexity of the addition and of the multiplication of big numbers which we may assume (intuitively) to be $O(log(T_{n,0}))$. Although the question of the asymptotic size of the number $T_{n,0}$ is open, we know that it is superexponential in n and, on the other hand, it is asymptotically smaller than n^n. Therefore, `t n 0` runs in $O(n^2 \times log(T_{n,0}))$ which is at least of order n^3 and at most of order $n^3 \log(n)$. Such estimations seem to be in accordance with our experiments. Once the table `ttab` is constructed, the runtime of `t` is constant.

2.2 The polynomials P_n

In the end of this section and in the four coming sections we present results of mainly combinatorial flavor. We focus there on the quantitative approach to lambda calculus, with special emphasis on the challenging problem of counting lambda terms and approximating its asymptotic behavior. Therefore, a reader interested mostly in term generation and experimental results can skip this material and go directly to Section 7.

The problem of determining the asymptotic estimation of the number of closed terms of a given size turns out to be a non-trivial task. Due to the unusual combinatorial structure of lambda terms, such objects seem to resist methods developed in combinatorics so far. There are a few papers devoted to this challenging problem [2, 4, 13], however, none of the methods used by now could provide the final solution. Our approach can be considered as the development of the previous research carried out by the second author in [13].

n	P_n
0	m
1	$m^2 + m + 1$
2	$2m^3 + 3m^2 + 5m + 3$
3	$5m^4 + 10m^3 + 22m^2 + 25m + 14$
4	$14m^5 + 35m^4 + 94m^3 + 154m^2 + 163m + 82$
5	$42m^6 + 126m^5 + 396m^4 + 838m^3 + 1277m^2 + 1235m + 579$
6	$132m^7 + 462m^6 + 1654m^5 + 4260m^4 + 8384m^3 + 11791m^2 + 10707m + 4741$
7	$429m^8 + 1716m^7 + 6868m^6 + 20742m^5 + 49720m^4 + 90896m^3 + 120628m^2 + 104055m + 43977$
8	$1430m^9 + 6435m^8 + 28396m^7 + 98028m^6 + 275886m^5 + 617096m^4 + 1068328m^3 + 1352268m^2 + 1117955m + 454283$

Figure 2: The first eight polynomials P_n

For every $n \geq 0$, we associate with $T_{n,m}$ a polynomial $P_n(m)$ in m. First, let us define polynomials P_n in the following recursive way:

$$\begin{aligned} P_0(m) &= m, \\ P_{n+1}(m) &= P_n(m+1) + \sum_{i=0}^{n} P_i(m) P_{n-i}(m). \end{aligned}$$

The sequence $(P_n(0))_{n\geq 0}$ corresponds to the sequence $(T_{n,0})_{n\geq 0}$ enumerating closed lambda terms. The first nine polynomials are given in Figure 2.

This means that the constant coefficient of a polynomial $P_n(m)$ is exactly the number of closed lambda terms of size n.

Lemma 1 *For every n, the degree of the polynomial P_n is equal to $n+1$.*

Proof: The result follows immediately by induction on n from the definition of P_n. □

For $i > 0$ and $n \geq 0$, let us denote by $p_n^{(i)}$ the i^{th} leading coefficient of the polynomial P_n, i.e., we have

$$P_n(m) = p_n^{(1)} m^{n+1} + p_n^{(2)} m^n + \ldots + p_n^{(i)} m^{n+2-i} + \ldots + p_n^{(n+1)} m + p_n^{(n+2)}.$$

Lemma 2 *For every $n \geq 0$ and $i > 0$,*

$$\begin{aligned} p_0^{(1)} &= 1, \quad p_0^{(i)} = 0 \quad \textit{for} \quad i > 1, \\ p_{n+1}^{(i)} &= \sum_{j=0}^{i-2} \binom{n+1-j}{i-2-j} p_n^{(j+1)} + \sum_{k=1}^{i} \sum_{j=0}^{n} p_j^{(k)} p_{n-j}^{(i+1-k)}. \end{aligned}$$

Proof: Since $P_0(m) = m$, equations from the first line in the above lemma are trivial.

The i^{th} leading coefficient in the polynomial $P_{n+1}(m)$ is equal to the sum of coefficients standing at m^{n+3-i} in polynomials $P_n(m+1)$ and $\sum_{j=0}^{n} P_j(m) P_{n-j}(m)$.

The first of these polynomials, $P_n(m+1)$, is as follows:

$$p_n^{(1)}(m+1)^{n+1} + \ldots + p_n^{(i-1)}(m+1)^{n+3-i} + \ldots + p_n^{(n+2)},$$

therefore the coefficient of m^{n+3-i} in $P_n(m+1)$ is equal to

$$\binom{n+1}{i-2}p_n^{(1)}+\binom{n}{i-3}p_n^{(2)}+\ldots+\binom{n+3-i}{0}p_n^{(i-1)} \;=\; \sum_{j=0}^{i-2}\binom{n+1-j}{i-2-j}p_n^{(j+1)}.$$

In the case of the second polynomial, $\sum_{j=0}^{n} P_j(m)P_{n-j}(m)$, we have

$$\begin{aligned}&\left(p_j^{(1)}m^{j+1}+\ldots+p_j^{(k)}m^{j+2-k}+\ldots+p_j^{(j+2)}\right)\\ \cdot&\left(p_{n-j}^{(1)}m^{n-j+1}+\ldots+p_{n-j}^{(i+1-k)}m^{n-j+1+k-i}+\ldots+p_{n-j}^{(n-j+2)}\right),\end{aligned}$$

therefore the coefficient of m^{n+3-i} in $\sum_{j=0}^{n} P_j(m)P_{n-j}(m)$ is equal to

$$\sum_{k=1}^{i}\sum_{j=0}^{n} p_j^{(k)}p_{n-j}^{(i+1-k)}.$$

□

The next section proposes a combinatorial interpretation of the coefficients $p_j^{(i)}$.

3 Counting contexts

In λ-calculus, an i-context is a closed term with i holes. We consider that each hole has size 0 and we assume that the holes are numbered $1,\ldots,i$ as they appear in the term from left to right. For instance, if we denote every hole by $[\,]$, then $(\lambda\underline{1}[\,])\lambda\lambda[\,]\underline{2}$ is a 2-context of size 6 and its holes are numbered as follows $(\lambda\underline{1}[\,]_1)\lambda\lambda[\,]_2\underline{2}$. 0-contexts correspond to closed terms. There is only one 1-context of size 0 and there are no i-contexts of size 0 for $i \neq 0$. Let us write $c_{n,i}$ for the number of i-contexts of size n. One notices that

$$\left.\begin{array}{rcl} c_{0,1} &=& 1 \\ c_{0,i} &=& 0 \text{ for } i \neq 1. \end{array}\right\} (\dagger)$$

Now, let us see how we construct an i-context of size $n+1$ from smaller ones.

By abstraction: let us take a j-context (for $j \in [i..n+1]$) of size n and add a new lambda above it. Then we choose a set of $j-i$ holes among the j holes which we substitute by variables (or indices) abstracted by the new lambda. For a fixed j there are $\binom{j}{i}\,c_{n,j}$ such i-contexts. Finally, we sum these quantities over every j from i to $n+1$ to get the numbers of i-contexts constructed this way.

By application: let us apply a j-context of size k to an $(i-j)$-context of size $n-k$ (for $j \in [0..i]$ and $k \in [0..n]$). This gives us an i-context of size $n+1$ since the application operator has size 1. For fixed j and k there are $c_{k,j}c_{n-k,i-j}$ such i-contexts. Finally, we sum these numbers from $j=0$ to $j=i$ and from $k=0$ to $k=n$.

Hence, we get the following formula:

$$c_{n+1,i} \quad = \quad \sum_{j=i}^{n+1} \binom{j}{i} c_{n,j} + \sum_{j=0}^{i} \sum_{k=0}^{n} c_{k,j} c_{n-k,i-j}. \qquad (\star)$$

From the contexts we can see how we can build terms. More precisely, from an i-context of size n and a map f from $[1..i]$ to $[1..m]$, we can insert the index $f(j)$ in the j^{th} hole to build a term of size n with i occurrences of free variables taken among m ones. There are $c_{n,i}m^i$ such terms. Therefore

$$T_{n,m} \quad = \quad c_{n,n+1}m^{n+1} + \ldots + c_{n,i}m^i + \ldots + c_{n,0}$$

is the number of λ-terms of size n with at most m distinct free variables, which is the polynomial P_n. In particular, $c_{n,n+2-i} = p_n^{(i)}$. This can be written as follows:

$$P_n(m) = \sum_{i=0}^{n+1} c_{n,i} m^i.$$

The coefficients $c_{n,i}$ of the polynomials P_n's count the i-contexts of size n. We see that $c_{n,i} = 0$ when $i > n+1$.

The case $i = n+2$. In the case when $i = n+2$, using the fact that $c_{n,i} = 0$ for $i > n+1$, the equations (†) and ($\star$) boil down to:

$$\begin{aligned} c_{0,1} &= 1 \\ c_{n+1,n+2} &= \sum_{k=0}^{n} c_{k,k+1} c_{n-k,n-k+1}, \end{aligned}$$

which is characteristic of the Catalan numbers. Indeed, $(n+1)$-contexts of size n have only applications and no abstractions and are therefore binary trees.

3.1 The generating function for $(c_{n,i})_{n,i\in\mathbb{N}}$

Proposition 3 *Consider the bivariate generating function* $L(z,u) = \sum_{n,i\geq 0} c_{n,i} z^n u^i$. *Then*

$$L(z,u) = u + zL(z,u+1) + zL(z,u)^2.$$

Proof: Notice that

$$\begin{aligned} L(z,u) &= \sum_{n=0}^{\infty} \left(\sum_{i=0}^{\infty} c_{n,i} u^i \right) z^n \\ &= \sum_{n=0}^{\infty} P_n(u) z^n \\ &= u + z \sum_{n=0}^{\infty} P_{n+1}(u) z^n \\ &= u + z \sum_{n=0}^{\infty} P_n(u+1) z^n + z \sum_{n=0}^{\infty} \sum_{k=0}^{n} P_k(u) P_{n-k}(u) z^n \\ &= u + zL(z,u+1) + zL(z,u)^2. \end{aligned}$$

□

A similar equation was known from Bodini, Gardy and Gittenberger [2] (for variables size 1). However, notice that what they call $\mathcal{L}$ is not the class of open λ-terms, but the class of i-contexts. Notice that the function $L(z,0)$ is the generating function for the number of closed terms of size n.

The equation

$$zL(z,u)^2 - L(z,u) + u + zL(z,u+1) = 0$$

has the following solution

$$L(z,u) \;=\; \frac{1-\sqrt{1-4z(u+zL(z,u+1))}}{2z}.$$

Let us state

$$M(z,u) = 2z\,L(z,u).$$

Then

$$M(z,u) = 1 - \sqrt{1-4zu-2zM(z,u+1)}$$

and hence

$$M(z,0) = 1 - \sqrt{1-2z(1-\sqrt{1-4z-2z(1-\sqrt{1-8z-2z(1-\sqrt{1-12z-2z(1-\sqrt{1-16z-\ldots})})})})}.$$

3.2 The asymptotic behavior of $T_{n,0}$

The function $M(z,u)$ has a singularity z_u for $1-4z_u u-2z_u M(z_u,u+1)=0$, in addition to the singularities of $M(z,u+1)$. Notice that since $z_u M(z_u,u+1)>0$, we get $z_u < \frac{1}{4u}$. Therefore $L(z,0)$ has a sequence of singularities $(z_u)_{u\in\mathbb{N}}$ which tends to 0. Thus the radius of convergence of $L(z,0)$ is 0.

4 Three formulas for counting closed terms

We have found three formulas to compute the number of closed terms of size n. Let us summarize them. In what follows the bracketed notation $[k=j]$ is the function which is 1 if $k=j$ and 0 if $k\neq j$.

Case $m=0$ for terms with at most m distinct free variables

$T_{n,0}$ where

$$\begin{aligned} T_{0,m} &= m \\ T_{n+1,m} &= T_{n,m+1} + \sum_{i=0}^{n} T_{i,m}T_{n-i,m}. \end{aligned}$$

This formula is clearly the simplest. Its simplicity, one sum and no binomial, allows to unfold it and on this basis it enables building a program for term generation (see Section 7).

Case $m = 0$ for terms with exactly m distinct free variables

$f_{n,0}$ where

$$\begin{aligned} f_{0,m} &= [m = 1] \\ f_{n,m} &= 0 \text{ if } m > n+1 \\ f_{n+1,m} &= f_{n,m} + f_{n,m+1} + \\ &\quad \sum_{p=0}^{n} \sum_{c=0}^{m} \sum_{k=0}^{m-c} \binom{m}{c} \binom{m-c}{k} f_{p,k+c} f_{n-p,m-k}. \end{aligned}$$

This formula is the most complex. Appendix A.1 shows how it is constructed.

0-contexts

$c_{n,0}$ where

$$\begin{aligned} c_{0,i} &= [i = 1] \\ c_{n+1,i} &= \sum_{j=i}^{n+1} \binom{j}{i} c_{n,j} + \sum_{j=0}^{i} \sum_{k=0}^{n} c_{k,j} c_{n-k,i-j}. \end{aligned}$$

Four relations. Let us use the notation $R_i^{(m)}$ (see Flajolet and Sedgewick's book [7]) for the number of surjections from $[1..i]$ to $[1..m]$. Recall that

$$R_i^{(m)} = \sum_{j=0}^{i} \binom{i}{j} (-1)^j (i-j)^m.$$

The numbers $T_{n,m}$, $f_{n,m}$ and $c_{n,i}$ are related as follows (see Appendix A.2):

$$\begin{aligned} T_{n,m} &= \sum_{i=0}^{m} \binom{m}{i} f_{n,i} &&= \sum_{i=1}^{n+1} c_{n,i} m^i \\ f_{n,m} &= \sum_{i=0}^{m} (-1)^{m+i} \binom{m}{i} T_{n,i} &&= \sum_{i=1}^{n+1} c_{n,i} R_i^{(m)}. \end{aligned}$$

5 More generating functions

For every positive integer i, let us denote by a_i the generating function for the sequence $\left(p_n^{(i)}\right)_{n \geq 0}$, i.e.,

$$a_i(z) = \sum_{n=0}^{\infty} p_n^{(i)} z^n = \sum_{n=0}^{\infty} c_{n,n+2-i} z^n.$$

The $p_n^{(i)}$'s count the number of contexts of size n having $n + 2 - i$ holes. For the sake of clarity, instead of writing $a_i(z)$ sometimes we simply write a_i.

In order to compute the functions a_i, we apply the following basic fact about generating functions.

Fact 4 *Let f and g be generating functions for sequences $(f_n)_{n\geq 0}$ and $(g_n)_{n\geq 0}$, respectively. Then*

(i) the generating function for the sequence $\left(\binom{n}{k} f_n\right)_{n\geq 0}$, where k is a fixed positive integer, is given by $\frac{z^k f^{(k)}}{k!}$,

(ii) the generating function for the sequence $\left(\sum_{i=0}^{n} f_i g_{n-i}\right)_{n\geq 0}$ is given by $f \cdot g$,

(iii) the generating function for the sequence $\left(\binom{n-j}{i} f_n\right)_{n\geq 0}$, where $i \geq 0$ and $j > 0$, is given by $\sum_{k=0}^{i} (-1)^k \binom{k+j-1}{j-1} z^{i-k} \frac{f^{(i-k)}}{(i-k)!}$.

Proof: Items (i) and (ii) can be found, e.g., in Chapter 7 of [8].

The third part follows from (i) and the following equality:

$$\binom{n-j}{i} = \sum_{k=0}^{i} (-1)^k \binom{n}{i-k} \binom{k+j-1}{j-1},$$

which holds for every $n, i \geq 0$ and $j > 0$. This equality can be easily derived from two equalities known as "upper negation" and "Vandermond convolution", which can be found in Table 174 of [8]. □

Now we are ready to provide a recurrence for functions a_i.

Theorem 5 *The following equations are valid:*

$$\begin{aligned}
a_1 &= z a_1^2 + 1, \quad a_1(0) = 1 \\
a_2 &= z a_1 + 2 z a_1 a_2 \\
a_i &= z^{i-1} \frac{a_1^{(i-2)}}{(i-2)!} + z^{i-2} \frac{a_1^{(i-3)}}{(i-3)!} + z^{i-2} \frac{a_2^{(i-3)}}{(i-3)!} \\
&\quad + z \cdot \sum_{j=1}^{i-3} \sum_{k=0}^{i-3-j} (-1)^k \binom{k+j-1}{j-1} z^{i-3-j-k} \frac{a_{j+2}^{(i-3-j-k)}}{(i-3-j-k)!} \\
&\quad + z \cdot \sum_{j=1}^{i} a_j a_{i-j+1}, \qquad \text{for } i > 2.
\end{aligned}$$

Proof: All these equations follow from Lemma 2 and Fact 4. □

Notice that the a_i's can be computed by induction. Indeed, a_i occurs twice in the lefthand side of the last equation and we have:

$$\begin{aligned}
a_i(1 - 2a_1) &= z^{i-1} \frac{a_1^{(i-2)}}{(i-2)!} + z^{i-2} \frac{a_1^{(i-3)}}{(i-3)!} + z^{i-2} \frac{a_2^{(i-3)}}{(i-3)!} \\
&\quad + z \cdot \sum_{j=1}^{i-3} \sum_{k=0}^{i-3-j} (-1)^k \binom{k+j-1}{j-1} z^{i-3-j-k} \frac{a_{j+2}^{(i-3-j-k)}}{(i-3-j-k)!} \\
&\quad + z \cdot \sum_{j=2}^{i-1} a_j a_{i-j+1}.
\end{aligned}$$

$$
\begin{aligned}
a_1(z) &= \left(\frac{1}{2} - \frac{(1-4z)^{1/2}}{2}\right) z^{-1} \\
a_2(z) &= -\frac{1}{2} + \frac{1}{2\,(1-4z)^{1/2}} \\
a_3(z) &= \left(\frac{1}{1-4z} + \frac{z}{(1-4z)^{3/2}}\right) z \\
a_4(z) &= \left(\frac{3}{(1-4z)^2} + \frac{z}{(1-4z)^{5/2}}\right) z^2 \\
a_5(z) &= \left(\frac{4z+9}{(1-4z)^3} + \frac{z^2-19z+5}{(1-4z)^{7/2}}\right) z^3 \\
a_6(z) &= \left(\frac{24z+31}{(1-4z)^4} + \frac{3z^2-203z+51}{(1-4z)^{9/2}}\right) z^4 \\
a_7(z) &= \left(\frac{16z^2-128z+181}{(1-4z)^5} + \frac{2z^3-194z^2-1541z+398}{(1-4z)^{11/2}}\right) z^5
\end{aligned}
$$

Figure 3: The generating functions for the coefficients of the polynomials $P_n(m)$

Since $1-2a_1 = \sqrt{1-4z}$, we get:

$$
a_i = \left(\begin{array}{l} z^{i-1}\frac{a_1^{(i-2)}}{(i-2)!} + z^{i-2}\frac{a_1^{(i-3)}}{(i-3)!} + z^{i-2}\frac{a_2^{(i-3)}}{(i-3)!} \\ +z\cdot\sum_{j=1}^{i-3}\sum_{k=0}^{i-3-j}(-1)^k\binom{k+j-1}{j-1} z^{i-3-j-k}\frac{a_{j+2}^{(i-3-j-k)}}{(i-3-j-k)!} \\ +z\cdot\sum_{j=2}^{i-1} a_j a_{i-j+1} \end{array} \right) / \sqrt{1-4z}. \qquad (\ddagger)
$$

Corollary 6 *Exact formulas for the functions a_1–a_7 are given in Figure 3.*

Proof: Let us first compute the function a_1 which, according to Theorem 5, is given by

$$a_1 = z a_1^2 + 1, \quad a_1(0) = 1.$$

By solving this equation, we obtain $a_1(z) = \frac{1-\sqrt{1-4z}}{2z}$, which is exactly the generating function for Catalan numbers—see, e.g., Chapter I.1 of [7].

Now, let us notice that on the basis of Theorem 5 all the other functions can be immediately obtained by tedious, however elementary, computations. In order to get exact values we applied Sage software [19]. □

Let $[z^n]f(z)$ denote the n^{th} coefficient of z^n in the formal power series $f(z) = \sum_{n=0}^{\infty} f_n z^n$. As usual, we use the symbol $\sim$ to denote the asymptotic equivalence of two sequences, i.e., we write $f_n \sim g_n$ iff the limit of the sequence $(f_n/g_n)_{n\geq 0}$ is 1. Similarly, by $f(z) \underset{z\to z_0}{\sim} g(z)$ we mean that the limit of $f(z)/g(z)$ is 1 when $z \to z_0$. We say that a function $f(z)$ is of order $g(z)$ for $z \to z_0$ iff there exists a positive constant A such that $f(z) \underset{z\to z_0}{\sim} A \cdot g(z)$.

The theorem below (Theorem VI.1 of [7]) serves as a powerful tool that allows us to estimate coefficients of certain functions that frequently appear in combinatorial considerations.

Fact 7 *Let α be an arbitrary complex number in $\mathbb{C} \setminus \mathbb{Z}_{\leq 0}$. The coefficient of z^n in*

$$f(z) = (1-z)^\alpha$$

admits the following asymptotic expansion:

$$[z^n]f(z) \sim \frac{n^{\alpha-1}}{\Gamma(\alpha)}\left(1 + \frac{\alpha(\alpha-1)}{2n} + \frac{\alpha(\alpha-1)(\alpha-2)(3\alpha-1)}{24n^2} + \frac{\alpha^2(\alpha-1)^2(\alpha-2)(\alpha-3)}{48n^3} + O\left(\frac{1}{n^4}\right)\right),$$

where Γ is the Euler Gamma function defined for $\Re(\alpha) > 0$ as

$$\Gamma(\alpha) := \int_0^\infty e^{-t}t^{\alpha-1}dt.$$

Now we are ready to prove the following approximation.

Proposition 8 *The exact order of functions a_i for $z \to 1/4$ is given by*

$$a_i(z) \underset{z\to\frac{1}{4}}{\sim} \frac{C_{i-2}}{2^{3i-5}(1-4z)^{(2i-3)/2}},$$

where C_i is the i^{th} Catalan number.

Proof: We prove the result by induction using Theorem 5. For the sake of simplicity, we write $\sim$ and "is of order" to denote $\underset{z\to\frac{1}{4}}{\sim}$ and "is of order for $z \to 1/4$".

The result is true for $i = 1$. For $i > 1$ and $j \leq i$, assume that $a_j(z)$ is of order $\frac{1}{(1-4z)^{(2j-3)/2}}$ and look at equation (‡) to prove that $a_{i+1}(z)$ is of order $\frac{1}{(1-4z)^{(2i-1)/2}}$.

Notice that the i^{th} derivative of a_1 is of order $\frac{1}{(1-4z)^{2i-1/2}}$, hence its $(i-2)^{th}$ derivative is of order $\frac{1}{(1-4z)^{(2i-5)/2}}$ and its $(i-3)^{th}$ derivative is of order $\frac{1}{(1-4z)^{(2i-7)/2}}$. Similarly, the i^{th} derivative of a_2 is of order $\frac{1}{(1-4z)^{2i+1/2}}$, hence its $(i-3)^{th}$ derivative is of order $\frac{1}{(1-4z)^{(2i-5)/2}}$.

By induction for $j+2 \leq i-3$, a_{j+2} is of order $\frac{1}{(1-4z)^{(2j+1)/2}}$. Among its successive derivatives we derive at most $i-3-j$ times, hence the items in the sum are of order at most $\frac{1}{(1-4z)^{(2i-5)/2}}$.

Now, every product $a_j a_{i-j+1}$ is of order $\frac{1}{(1-4z)^{i-2}}$, therefore the first four terms in (‡) do not contribute to the asymptotic value of $a_{i+1}(z)$. Hence the contribution to the asymptotic value is given only by products $a_j a_{i-j+1}$'s. Multiplying their order by $\frac{1}{\sqrt{1-4z}}$ we obtain that the last sum is of order $\frac{1}{(1-4z)^{(2i-1)/2}}$.

Let us denote by K_i the multiplicative coefficient $C_{i-2}/2^{3i-5}$ of $\frac{1}{(1-4z)^{(2j-3)/2}}$. One notices that $K_2 = \frac{1}{2} = \frac{C_0}{2^{3.2-5}}$. The sum $z \sum_{j=2}^{i-1} a_j a_{i-j+1}$ shows the

inductive part. Indeed, when $z = \frac{1}{4}$:

$$\begin{aligned} z \sum_{j=2}^{i-1} K_j K_{i-j+1} &= \frac{1}{4} \sum_{j=2}^{i-1} \frac{C_{j-2}}{2^{3j-5}} \frac{C_{i-j+1-2}}{2^{3(i-j+1)-5}} \\ &= \frac{1}{2^{3i-5}} \sum_{j=0}^{i-3} C_j C_{i-j-3} \\ &= \frac{C_{i-2}}{2^{3i-5}} = K_i. \end{aligned}$$

□

Finally, we are able to provide asymptotic values of coefficients of functions a_i.

Theorem 9 *The coefficient of z^n in the function $a_k(z)$ admits the following asymptotic expansion:*

$$[z^n]a_k(z) = \frac{1}{2^{k-1}(k-1)!\sqrt{\pi}} \, 4^n n^{(2k-5)/2} \cdot \Psi(n,k)$$

where

$$\begin{aligned} \Psi(n,k) = \quad & 1 + \frac{(2k-3)(2k-5)}{8n} + \frac{(2k-3)(2k-5)(2k-7)(3k-11)}{384n^2} + \\ & \frac{(2k-3)^2(2k-5)^2(2k-7)(2k-9)}{3672n^3} + O(\frac{1}{n^4}). \end{aligned}$$

Proof: First recall that

$$\Gamma((2k-3)/2) = \Gamma((k-2)+\frac{1}{2}) = \frac{(2(k-2))!\sqrt{\pi}}{2^{2(k-2)}(k-2)!}.$$

Now using Fact 7, we can compute the principal part:

$$\begin{aligned} [z^n]a_k(z) &= \frac{C_{k-2}}{2^{3k-5}} \, 4^n \, [z^n](1-z)^{(2k-3)/2} \\ &\sim \frac{C_{k-2}}{2^{3k-5}} \, 4^n \, \frac{n^{(2k-5)/2}}{\Gamma((2k-3)/2)} \\ &= \frac{C_{k-2}}{2^{3k-5}} \frac{(k-2)!2^{2(k-2)}}{(2(k-2))!\sqrt{\pi}} \, 4^n n^{(2k-5)/2} \\ &= \frac{C_{k-2}(k-2)!}{2^{k-1}(2(k-2))!\sqrt{\pi}} \, 4^n n^{(2k-5)/2} \\ &= \frac{1}{2^{k-1}(k-1)!\sqrt{\pi}} \, 4^n n^{(2k-5)/2}. \end{aligned}$$

For $\Psi(n,k)$ we use Fact 7, with $\alpha = \frac{2k-3}{2}$. □

By looking at Figure 3, we can easily notice a recurring pattern concerning the structure of functions a_i. Therefore, we state the following proposition.

Proposition 10 *For every $i > 2$ we have*

$$a_i(z) = z^{i-2}\left(\frac{Q_i(z)}{(1-4z)^{i-2}} + \frac{S_i(z)}{(1-4z)^{i-\frac{3}{2}}}\right),$$

where Q_i and S_i are polynomials over $\mathbb{Z}$ in z and $\deg Q_i = \left\lfloor \frac{i-3}{2} \right\rfloor$ and $\deg S_i = \left\lfloor \frac{i-1}{2} \right\rfloor$.

Proof: By induction using formula (‡), in the same vein as the proof of Proposition 8. In particular, the two first members of (‡) are derivatives of the generating function of Catalan numbers studied in [11]. □

As we have already mentioned, the number of closed terms of size n is given by $P_n(0)$, which corresponds to the n^{th} term of the Taylor expansion of the function a_{n+2}. Hence, the sequence of the numbers of closed lambda terms is equal to the sequence $([z^n]a_{n+2}(z))_{n\geq 0}$. From Proposition 10, the number of closed terms of size n is equal to $Q_{n+2}(0) + S_{n+2}(0)$. Currently, we have no recursive formula for the Q_n's and the S_n's. However, by Proposition 8, we know that

$$S_{n+2}\left(\frac{1}{4}\right) = \frac{C_n}{2^{n+1}}.$$

6 Counting normal forms

Beside counting terms, it is also interesting to count normal forms. To this end, we describe the set of normal forms as follows

$$\begin{aligned} \mathcal{G}_m &= [\underline{1}\ldots\underline{\mathsf{m}}] \uplus \mathcal{G}_m @ \mathcal{F}_m \\ \mathcal{F}_m &= \lambda\, \mathcal{F}_{m+1} \uplus \mathcal{G}_m \end{aligned}$$

Recall that a normal form is made by a (possibly empty) sequence of abstractions on a term which is made as follows: a de Bruijn index applied to a sequence of normal forms. $\mathcal{F}_m$ represents the normal forms and $\mathcal{G}_m$ represents the terms starting with an index. From this we derive the formulas for counting:

$$\begin{aligned} G_{0,m} &= m \\ G_{n+1,m} &= \sum_{k=0}^{n} G_{n-k,m} F_{k,m}, \end{aligned}$$

$$\begin{aligned} F_{0,m} &= m \\ F_{n+1,m} &= F_{n,m+1} + G_{n+1,m}. \end{aligned}$$

The values of $F_{n,0}$ up to $n = 10$ are:

$$0, 1, 3, 11, 53, 323, 2359, 19877, 188591, 1981963, 22795849.$$

This sequence is **A224345** in the *On-line Encyclopedia of Integer Sequences*. A Haskell program for computing the values of $F_{n,m}$ and $G_{n,m}$ efficiently is as follows:

```
ftab :: [[Integer]]
ftab = [0..]:[[ftab !! (n-1) !! (m+1) + gtab !! n !! m
              | m<-[0..]]
             | n<-[1..]]

gtab :: [[Integer]]
gtab = [0..] : [[s n m |  m <- [0..]] | n <- [1..]]
  where s n m  = let fi =  [ftab !! i !! m | i <- [0..(n-1)]]
                     gi =  [gtab !! i !! m | i <- [0..(n-1)]]
                 in foldl (+) 0 (map (uncurry (*))
                                     (zip fi (reverse gi)))

f :: Int -> Int -> Integer
f n m = ftab !! n !! m
```

Like for terms we derive polynomials:

$$\begin{aligned} {}^{\mathsf{NF}}P_0(m) &= m \\ {}^{\mathsf{NF}}P_{n+1}(m) &= {}^{\mathsf{NF}}P_n(m+1) + {}^{\mathsf{NF}}Q_{n+1}(m) \end{aligned}$$

$$\begin{aligned} {}^{\mathsf{NF}}Q_0(m) &= m \\ {}^{\mathsf{NF}}Q_{n+1}(m) &= \sum_{k=0}^{n} {}^{\mathsf{NF}}P_k(m){}^{\mathsf{NF}}Q_{n-k}(m). \end{aligned}$$

Lemma 11 *For every n, the degree of the polynomials ${}^{\mathsf{NF}}P_n$ and ${}^{\mathsf{NF}}Q$ is equal to $n+1$.*

Proof: Like the proof of Lemma 1, by induction on n from the definition of ${}^{\mathsf{NF}}P_n$ and ${}^{\mathsf{NF}}Q$. □

We have not derived the formulas for the coefficients yet. But these formulas are useful to derive generators of normal forms used in the rest of the paper.

6.1 Coefficients of the polynomials ${}^{\mathsf{NF}}P_n$ and ${}^{\mathsf{NF}}Q_n$

Let us count i-nf-contexts. They are closed normal forms with i holes. The i-nf-contexts of size n are counted by $d_{n,i}$. They are abstractions of i-contexts of the form $[\,]\,N_1 \dots N_p$, which we call i-pre-nf-contexts, where each N_j is a i_j-nf-context (with $i_1{+}\dots{+}i_j{+}\dots{+}i_p = i-1$) and which are counted by $g_{n,i}$. There is one 1-nf-context and one 1-nf-pre-context of size 0, whereas there are 0 i-nf-contexts and 0 i-nf-pre-contexts for $i \neq 1$ of size 0. Thus we get

$$\begin{aligned} d_{0,i} &= [i=1], \\ g_{0,i} &= [i=1]. \end{aligned}$$

By reasoning similarly as in Section 3 and by using the description of normal forms given above, we get:

$$\begin{aligned} d_{n+1,i} &= \sum_{j=i}^{n+1} \binom{j}{i} d_{n,j} + g_{n+1,i}, \\ g_{n+1,i} &= \sum_{j=0}^{i} \sum_{k=0}^{n} g_{k,j} d_{n-k,i-j}. \end{aligned}$$

Therefore

$$\begin{aligned} {}^{\mathsf{NF}}P_n(m) &= \sum_{i=0}^{n} d_{n,i} m^i, \\ {}^{\mathsf{NF}}Q_n(m) &= \sum_{i=0}^{n} g_{n,i} m^i. \end{aligned}$$

6.2 Generating functions

Consider the two generating functions:

$$\begin{aligned} D(z,u) &= \sum_{n,i\geq 0} d_{n,i} z^n u^i, \\ G(z,u) &= \sum_{n,i\geq 0} g_{n,i} z^n u^i. \end{aligned}$$

Then we have

$$\begin{aligned} D(z,u) &= \sum_{n=0}^{\infty} {}^{\mathsf{NF}}P_n(u), \\ G(z,u) &= \sum_{n=0}^{\infty} {}^{\mathsf{NF}}Q_n(u). \end{aligned}$$

Therefore

$$\begin{aligned} D(z,u) &= u + z \sum_{n=0}^{\infty} {}^{\mathsf{NF}}P_n(u+1) + \sum_{n=1}^{\infty} {}^{\mathsf{NF}}Q_n(u) \\ &= zD(z,u+1) + G(z,u) \end{aligned}$$

and

$$\begin{aligned} G(z,u) &= u + z \sum_{n=0}^{\infty} {}^{\mathsf{NF}}Q_n(u) {}^{\mathsf{NF}}P_n(u) z \\ &= u + z \sum_{n=0}^{\infty} \sum_{k=0}^{n} g_{k,j} z^k u^i d_{n-k,i-j} z^{n-k} n u^{i-j} \\ &= u + z D(z,u)\, G(z,u). \end{aligned}$$

Consequently the two functions D and G satisfy

$$\begin{aligned} D(z,u) &= zD(z,u+1) + G(z,u), \\ G(z,u) &= u + zD(z,u)\,G(z,u). \end{aligned}$$

$D(z,0)$ is the generating function for the numbers of closed normal forms of size n.

6.3 An expression for $D(z,u)$

By solving the above system of equations, we get:

$$zD(z,u) - (1 + z^2 D(z,u+1))D(z,u) + u + zD(z,u+1) = 0,$$

which yields

$$D(z,u) = \frac{1 + z^2 D(z,u+1) - \sqrt{(1 + z^2 D(z,u+1))^2 - 4z(u + zD(z,u+1))}}{2z}.$$

7 Lambda term generation

From the simple equation defining the number $T_{n,m}$ of terms, we define the function generating them. More precisely, we define a function $term(k,n,m)$ which returns the k^{th} term of size n with at most m distinct free variables (see the Haskell program in Figure 4). The variable k is an Integer (i.e., a big number) which belongs to the interval $[1..T_{n,m}]$. The program mimics counting terms. If n is 0, then the program returns the de Bruijn index k. Otherwise, if k is less than $T_{n-1,m+1}$, the rank k lies in the part of the interval $[1..T_{n,m}]$ with terms that are abstractions. Therefore, for $k \leq T_{n-1,m+1}$ $term(k,n,m)$ returns $\lambda\, term(k,n-1,m+1)$. If the rank k is larger than $T_{n-1,m+1}$, it lies in the part of the interval $[1..T_{n,m}]$ with applications. Therefore we call a function $appTerm$ which tries to identify which sub-interval contains a pair of terms with indices k' and k'' such that $k' + k''$ is at the right place. The product of these values correspond to one of the products $T_{j,m}T_{n-j,m}$ in the sum. When the number j is found, two recursive calls of $term$, with appropriate k' and k'', build the subterms of the application. One may notice $(h-1)$ and $+1$ which take into account the fact that k lies in an interval $[1..T_{\_,\_}]$ and div and mod work in an interval $[0..(T_{\_,\_} - 1)]$.

The function $term$ relies on the function t presented in Section 2.1 and called here $O(n)$ times. Assuming that t has been called already once and therefore runs in constant time, $term$ performs $O(n)$ recursive calls and its complexity depends linearly on the operations div, mod, $(-)$ and $(*)$ performed on big numbers.

For a given n, this program can be used to enumerate all the closed λ-terms of size n and, more generally, all the λ-terms of size n with at most m distinct free variables. This is appropriate only for small values of n, since the number of λ-terms gets superexponentially large with n. But overall, in order to generate a random term of size n with at most m distinct free variables, it suffices to feed T with a random value k in the interval $[1..T_{n,m}]$. Similarly, one defines from the recursive formula for the number of normal forms a program for their generation (Figure 5).

```
data Term = Index Integer
          | Abs Term
          | App Term Term

term :: Integer -> Int -> Int -> Term
term k 0 m = Index k
term k n m
    | k <= (t (n-1) (m+1)) = Abs (term k (n-1) (m+1))
    | (t (n-1) (m+1)) < k = appTerm (k - t (n-1) (m+1)) (n-1) 0
    where appTerm h n j
            | h <=  tjmtnjm  = App (term ((h-1) `div` tnjm +1) j m)
                                  (term ((h-1) `mod` tnjm +1) (n-j) m)
            | otherwise = appTerm (h -tjmtnjm) n (j + 1)
            where tnjm = t (n-j) m
                  tjmtnjm = (t j m) * tnjm
```

Figure 4: Haskell program for term generation

```
nf :: Integer -> Int -> Int -> Term
nf k 0 m = Index k
nf k n m
  | k <= ftab !! (n-1) !! (m+1) = Abs (nf k (n-1) (m+1))
  | ftab !! (n-1) !! (m+1) < k = ng (k - ftab !! (n-1) !! (m+1)) n m

ng :: Integer -> Int -> Int -> Term
ng k 0 m = Index k
ng k n m = appNF k (n-1) 0 m

appNF :: Integer -> Int -> Int -> Int -> Term
appNF h n j m
    |  h <=  gjmfnjm  = App (ng ((h-1) `div` fnjm +1) j m)
                            (nf ((h-1) `mod` fnjm +1) (n-j) m)
    | otherwise = appNF (h -gjmfnjm) n (j + 1) m
    where fnjm = ftab !! (n-j) !! m
          gjmfnjm = gtab !! j !! m * fnjm
```

Figure 5: Haskell program for normal form generation

8 Simply typable terms

Once we have a random generator for untyped terms, it is easy to build a random generator for simply typable terms. It suffices to sieve all terms by a predicate, which we call *is_typable*. This predicate is a classical principal type algorithm [15, 3, 9]. In Appendix B, we give a Haskell program. For instance, applying the random generator with parameter

10 (for the size of the term), we got:

$$\lambda(\lambda(((\underline{1}\ \lambda(\underline{1}))\ \lambda((\underline{3}\ \lambda(((\underline{1}\ \underline{2})\ \underline{3})))))))).$$

This is a "typical" simply typable random closed lambda term of size 10 written with de Bruijn indices. Its type is

$$((\alpha \to (((\beta \to \beta) \to (\alpha \to \gamma) \to \delta) \to \zeta)) \to \zeta) \to \gamma \to \\ ((\beta \to \beta) \to (\alpha \to \gamma) \to \delta) \to \delta.$$

We were able to generate terms of size 50. For such terms, the generating process is slow, since it requires 50 000 generations of terms, with (unsuccessful) tests of their typability before getting a typable one. But for size 40, the number of attempts falls to 3 for 10 000, which is acceptable.

This kind of random generator is useful for testing functional programs. Michał Pałka [16, 17] proposed a tool to debug Haskell compilers based on a lambda term generator. His generator is designed on the development of a typing tree, with choices made when a new rule is created. Such a method needs to cut branches in developing the tree to avoid loops. This way his generator is not random, which may be a drawback in some cases. As a matter of fact, a method for generating simply typed terms based on developing a typing tree does not produce terms on a uniform random distribution since it requires to cut the tree at arbitrary locations to avoid loops, "arbitrary" in the sense of randomness preservation. In other words, there is no simple recursive definition of simply typed terms, as well as of simply typable terms, that would allow an easy uniform random generation. This is also what makes the combinatorial study of typed terms difficult. A term is typable because it satisfies some constraints, not because it is generated in a specific way.

9 Experimental data

Given a random term generator, we are able to write programs to make statistics on some features of terms. There are plenty of such experiments that can be lead. We have kept here only two that look interesting and which are somewhat paradigmatic. Experiments recorded here have been performed on a laptop.

9.1 Average variable depth in terms and normal forms

Let us define the *variable depth* as the number of symbols (abstractions and applications) between a variable and the top of the term. For instance, given the term $\lambda x.(\lambda yz.x)(\lambda u.u)$, the first occurrence of variable x has depth 1 and the second occurrence of variable x has depth 3, while the depth of u is 2. This gives the average depth 2 for this term. Viewed on de Bruijn indices of the brother term $\lambda\underline{1}\,(\lambda\lambda\underline{3}\,\lambda\underline{1})$, we say that the first index $\underline{1}$ has depth 1, the second index $\underline{3}$ has depth 3 and the third index $\underline{1}$ has depth 2, with the same average 2 as previously. In Figure 6, we draw the average variable depth for 300 random terms of size 15 up to size 175 (above scatter plot) and the average variable depth for 300 random normal forms of size 15 up to size 175 (below scatter plot) squeezed between the curves $\frac{2n}{\ln(n)^a}$ for $a = 1$ and $a = 1.1$ (plain lines). In Figure 7 we see the same four

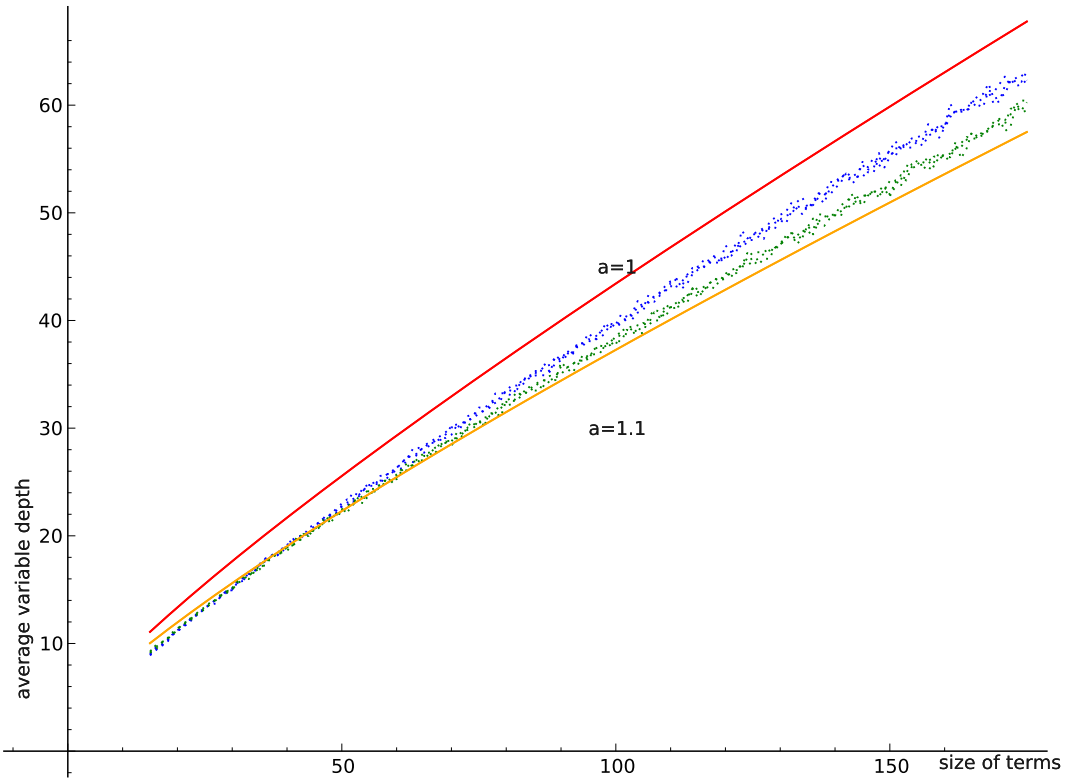

Figure 6: From above to below: Curve $\frac{2n}{\ln(n)}$, average variable depth for terms, average variable depth for normal forms and curve $\frac{2n}{\ln(n)^{1.1}}$.

drawings enlarged in the interval [170..175]. This shows clearly that the average variable depth of terms and normal forms are different. On this basis, we conjecture that the average depth of variables in terms has an asymptotic upper bound $\frac{2n}{\ln(n)}$ and that the average variable depth is slightly smaller for normal forms than for terms.

9.2 Average number of head λ's in terms

We say that λx is a *head lambda* in a term t if the latter is of the form $\lambda x_1 \ldots \lambda x_n \lambda x.s$ for some positive integer n and a certain term s. In order to know the structure of an average term, we are interested in the average number of head λ's occurring in terms. In Figure 8, we compare the average number of head λ's in 1000 random terms of size 15 to 150 with some functions $\sqrt{\frac{n}{\ln(n)^a}}$ and we see that, in the case of all terms, this is in accordance with Theorem 35 in [4].

9.3 Ratio of simply typable terms among terms

It is interesting to investigate the ratio of simply typable terms among all terms. Actually, there are 851 368 766 lambda terms of size 11, whereas there are 63 782 411 lambda terms of size 10. Therefore, we performed computations for terms of size less than 11. In fact, one cannot go much further due to the superexponential growth of the sequence

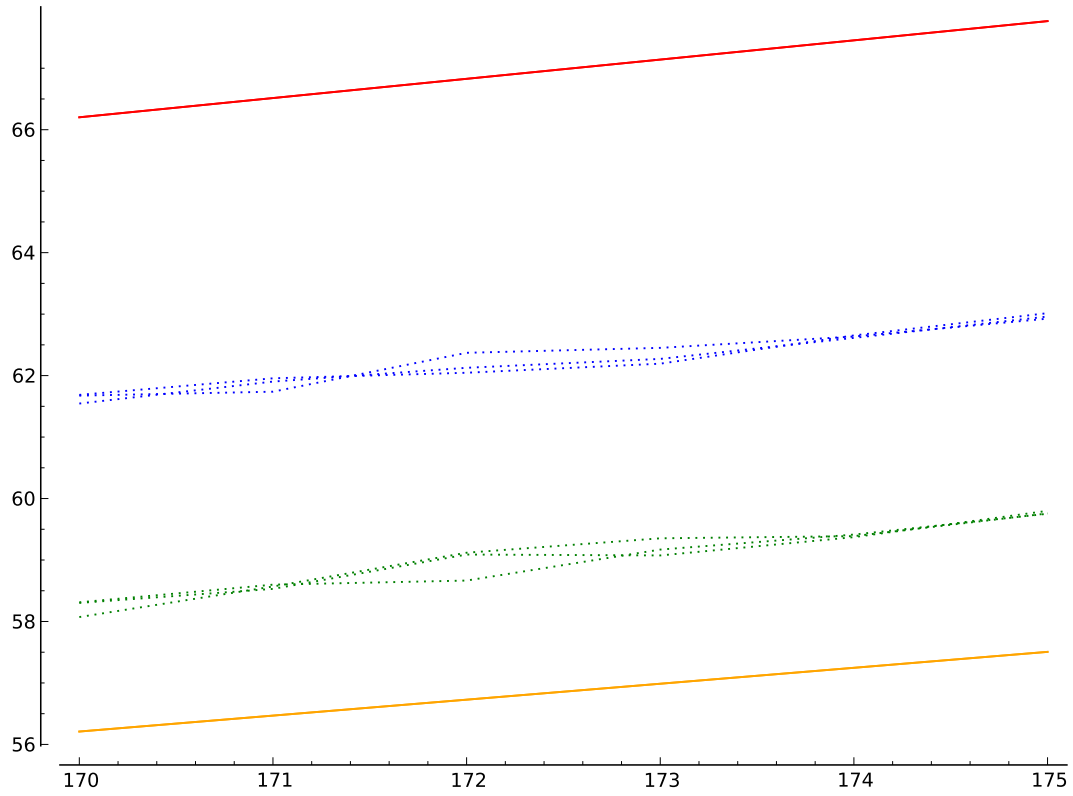

Figure 7: Magnification of Figure 6 between $n = 170$ and $n = 175$.

enumerating closed terms. Table 1 gives the ratio of simply typable terms over all terms by an exhaustive examination of the terms up to 10. For terms of size 8 or larger, we

size	4	5	6	7	8	9	10
nb of terms	82	579	4 741	43 977	454 283	5 159 441	63 782 411
nb of typables	40	238	1 564	11 807	98 529	904 318	9 006 364
ratio	0.4878	0.4110	0.3299	0.2684	0.2168	0.1752	0.1412

Table 1: Numbers and ratios of simply typable terms up to size 10

computed the ratio by the Monte Carlo method. The results are given in Table 2. The sequence of the numbers of simply typable terms of a given size can be found in the *On-line Encyclopedia of Integer Sequences* under the number **A220471**.

size	8	9	10	11	12	13	14	15	16	20	30	40	45	50
ratio	.216	.175	.141	.111	.089	.073	.056	.047	.039	.0014	.0012	.0003	.00005	$<10^{-5}$

Table 2: Ratios of simply typable terms (of size at least 8)

We conclude that simply typable terms become very scarce as the size of the terms grows, falling to less than one over 100 000 when the size gets larger than 50. Like before,

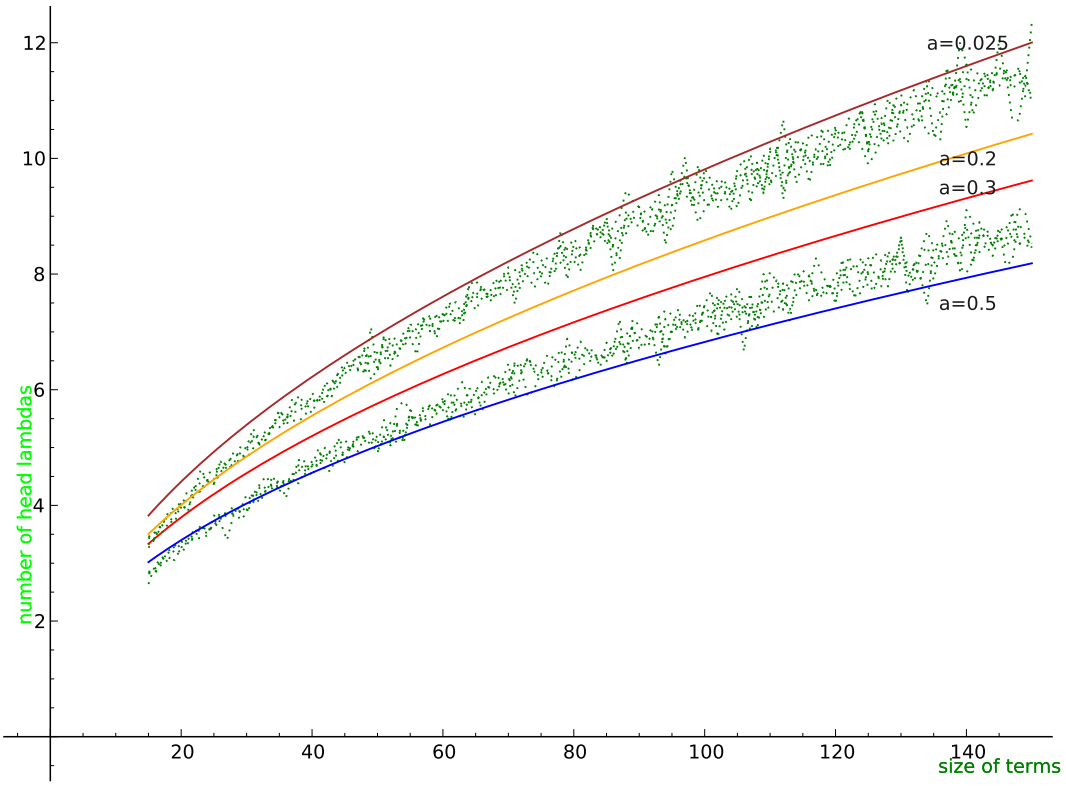

Figure 8: Below: average number of head λ's in terms. Above: average number of head λ's in normal forms. In between: curves $\sqrt{\frac{n}{\ln(n)^a}}$ for $a = 0.025, 0.2, 0.3, 0.5$.

we have done the same task for normal forms. We got the ratio by an exhaustive examination of normal forms up to 10 in Table 3 and by the Monte Carlo method thereafter in Table 4.

size	4	5	6	7	8	9	10
nb of NF	53	323	2 359	19 877	188 591	1 981 963	22 795 849
nb of typable NF	23	108	618	4 092	30 413	252 590	2 297 954
ratio	0.4339	0.3343	0.2619	0.2058	0.1612	0.1274	0.1008

Table 3: Numbers and ratios of simply typable normal forms up to size 10

size	8	9	10	11	12	13	14	15	16	20	30	40	45
ratio	.159	.128	.102	.079	.063	.049	.040	.031	.024	.010	.0006	2.10^{-5}	$<10^{-5}$

Table 4: Ratios of simply typable normal forms

9.4 Distribution of simply typable lambda terms among terms

We said that simply typable terms are scarce, but we may wonder what scarce exactly means. More precisely, we may wonder how terms are distributed. To provide an answer

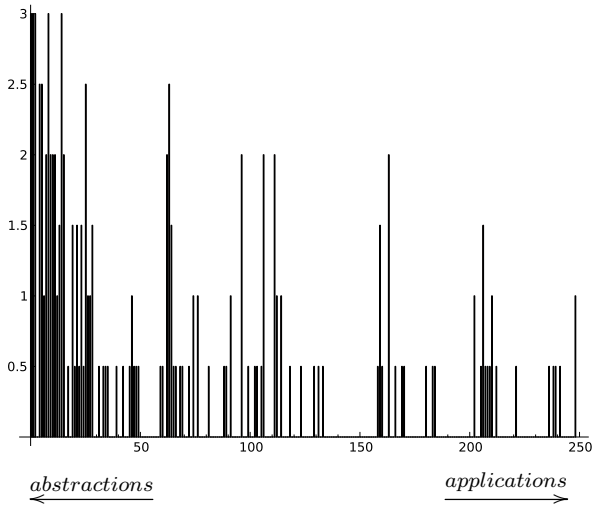

Figure 9: Distribution of simply typable lambda terms of size 25. 250 segments on the horizontal axis, percentage (0% – 3%) of typable terms in segments on the vertical axis.

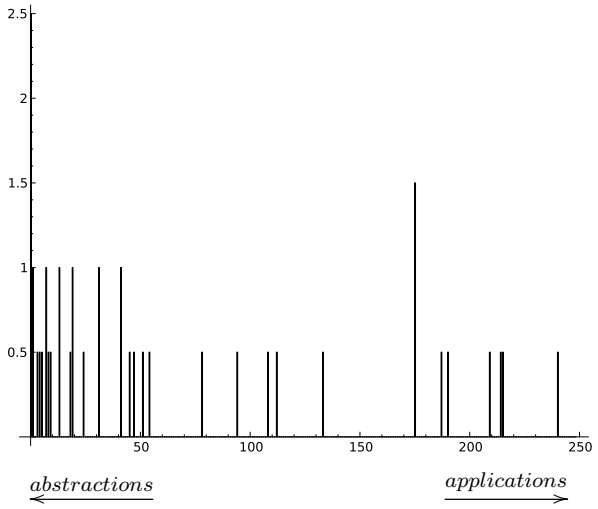

Figure 10: Distribution of simply typable lambda terms of size 30. 250 segments on the horizontal axis, percentage (0% – 2.5%) of typable terms in segments on the vertical axis.

to this question, we realized experiments for approaching the distribution of the frequency of typable lambda terms in segments of the interval $[1..T_{m,n}]$. For that we divided the interval $[1..T_{m,n}]$ in segments and we computed on samples of randomly generated terms, the ratio of simply typable terms over general terms we may expect in each segment. Figure 9 is typical of the results we got. This corresponds to an experiment on terms of size 25 on 250 segments with tests for simple typability on 200 random terms in each segment. For each segment the height of the vertical bar represents the ratio of typable terms in the corresponding segment. It shows that the simply typable terms are not evenly distributed. They are more concentrated on the left of the interval corresponding to terms with low numbers. Those terms correspond to terms starting more often with abstractions than with applications and this is recursively so for subterms giving this impressions of rolling waves. For instance, there are 2% to 3% of typable terms (of size

25) starting with many abstractions, whereas for terms starting with many applications, there are large subintervals with almost no typable terms. Figure 10, which gives the same statistics for terms of size 30, shows that typable terms get more scarce as the size of the terms grows.

The normal forms are even more scarcely distributed. As a comparison, we drew the same graphs for normal forms (size of the normal forms: 25 and 30, number of segments 250, tests on 200 terms) in Figure 11. The typable normal forms aggregate more on the left of the interval where terms start mostly with abstractions, with peaks of 4% to 6% by segments. Figure 12 shows that scarcity of typable normal forms increases as the size of terms grow.

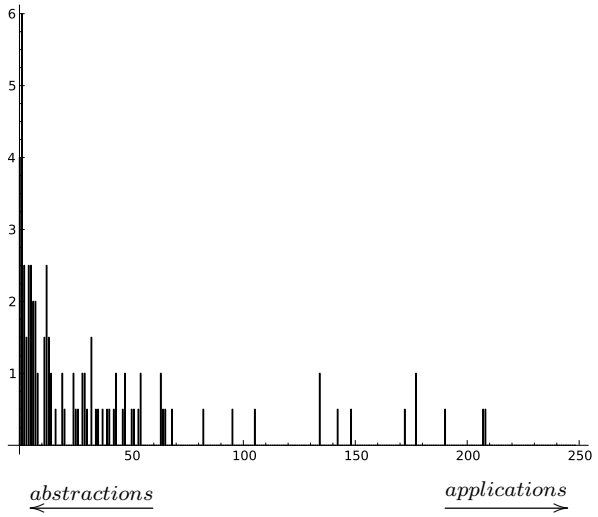

Figure 11: Distribution of simply typable normal forms of size 25. 250 segments on the horizontal axis, percentage (0% – 6%) of typable normal forms in segments on the vertical axis.

10 Related works

There are very few papers on counting lambda terms, whereas counting first order terms is a classical domain of combinatorics. Apparently, the first traces of counting expressions with (unbound) variables can be attributed to Hipparchus of Rhodes (c. 190–120 BC) (see [7] p. 68). Flajolet and Sedgewick's book [7] is the reference on this subject. Concerning counting λ-terms, we can cite only five works. [4] and [2] study asymptotic behavior of formulas on counting lambda-terms. Strictly speaking, they do not exhibit a recurrence formula for counting. In particular, David et al. [4] provide only upper and lower bounds for the numbers of λ-terms in order to get information about the distribution of families of terms. For instance, they prove that "asymptotically almost all λ-terms are strongly normalizing". In [13] the second author of the present paper proposes formulas for counting λ-terms in the case of variables of size 1, with more complex formulas and less results. On another hand, Christophe Raffalli proposed a formula for counting closed λ-terms, which he derives from the formula for counting λ-terms with exactly m distinct free variables. His formula appears in the *On-line Encyclopedia of Integer Sequences* under the number

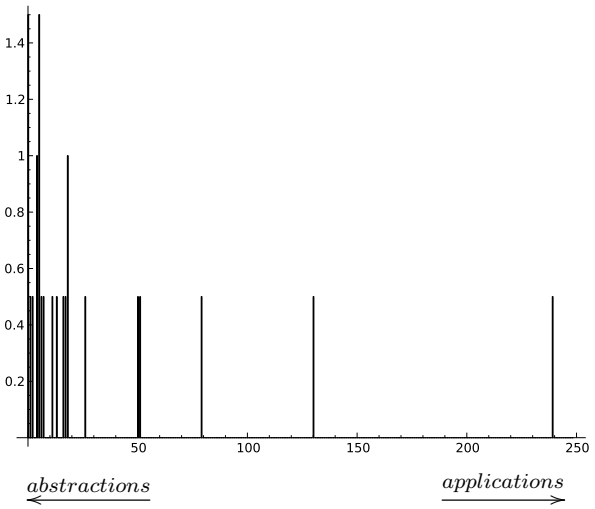

Figure 12: Distribution of simply typable normal forms of size 30. 250 segments on the horizontal axis, percentage (0% – 1.45%) of typable normal forms in segments on the vertical axis.

A135501. He considers size 1 for the variables. Beside those works, John Tromp [20] proposes a rather different way of counting λ-terms which deserves to be investigated further from the viewpoint of combinatorics. His size function works on terms with de Bruijn indices like ours and is (in our convention of starting at 1) as follows:

$$\begin{aligned} |\underline{\mathsf{n}}| &= n+1 \\ |\lambda M| &= |M|+2 \\ |M\,N| &= |M|+|N|+2 \end{aligned}$$

producing sequence **A114852** (and sequence **A195691** for closed normal forms). This work is connected to program size complexity and *Algorithmic Information Theory* [14].

As concerns random generation, Wang in [21] proposed algorithms for randomly generating untyped λ-terms in the spirit of the counting formula of Raffalli. On term generation, we can also mention the work of [6], in which the authors enumerate and generate many structures other than λ-terms. Pałka [16, 17] uses generation of typabke λ-terms to test Haskell compilers. He acknowledges that, due to his method, he cannot guarantee the randomness of his generator (see discussion in [16] p. 21 and p. 45). Nonetheless, he found eight failures and four bugs in the *Glasgow Haskell Compiler* showing the interest in the method. [18] studies the feasibility of generic programming for the enumeration of typed terms. The given examples are of size 4 or 5, no realistic examples are provided, randomness is not addressed and the authors confess that their algorithm is not efficient. Knowing that there are 63 782 411 simply typable closed terms of size 10, one wonders if there is the actual use for such enumeration and it seems unrealistic to utilize enumeration for larger numbers. The "related work" section of [18] covers similar approaches, which all consist in cutting branches. They all fail to generate random terms. A presentation of tree-like structure generation and a history of combinatorial generation is given in [10].

11 Acknowledgments

We would like to thank Marek Zaionc for interesting discussions and for setting the problem of counting lambda terms and Bruno Salvy for his help in the proof of Proposition 8. We also thank the referees of this paper for their fruitful suggestions.

12 Conclusion

This paper opens tracks of research in two directions, which are intrinsically complementary, namely counting and generating. On counting terms, some hard problems remain to be solved. Probably the hardest and the most informative one is to give an asymptotic estimation for the numbers of closed terms of size n. It seems that big obstacles remain to be hurdled before getting a solution, since combinatorial structures with binders have not been studied so far by combinatorists. On generation of terms, implementations have to be improved to go further in the production of uniformly distributed terms, in particular, of uniformly generated typed terms. This work has to be done in interaction with recent advances on generation of functional algebraic structures of several kinds.

References

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

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

[3] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212. ACM Press, 1982. Richard A. DeMillo ed.

[4] René David, Katarzyna Grygiel, Jakub Kozik, Christophe Raffalli, Guillaume Theyssier, and Marek Zaionc. Asymptotically almost all λ-terms are strongly normalizing. *CoRR*, abs/0903.5505v3, 2009.

[5] Nicolaas Govert de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen*, 75(5):381–392, 1972.

[6] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: functional enumeration of algebraic types. In Janis Voigtländer, editor, *Haskell*, pages 61–72. ACM, 2012.

[7] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2008.

[8] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics*. Addison-Wesley, Reading, MA, 1989.

[9] J. Roger Hindley. M. H. Newman's typability algorithm for lambda-calculus. *J. Log. Comput.*, 18(2):229–238, 2008.

[10] Donald E. Knuth. *The art of Computer Programming, Generating All Trees-History of Combinatorial Generation*, volume 4 (fascicle 4). Addison-Wesley Publishing Company, 2006.

[11] Wolfdieter Lang. On polynomials related to derivatives of the generative functions of the Catalan numbers. *The Fibonacci Quarterly*, 40(4):299–313, 2002.

[12] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Proceedings of the 21st Annual ACM Symposium on Principles Of Programming Languages, Portland (Or., USA)*, pages 60–69. ACM, 1994.

[13] Pierre Lescanne. On counting untyped lambda terms. *Theor. Comput. Sci.*, 474:80–97, 2013.

[14] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[15] Max H. A. Newman. Stratified systems of logic. *Proceedings of the Cambridge Philosophical Society*, 39:69–83, 1943.

[16] Michał Pałka. Testing an optimising compiler by generating random lambda terms. Licentiatavhandling, Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, May 2012.

[17] 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.

[18] Alexey Rodriguez Yakushev and Johan Jeuring. Enumerating well-typed terms generically. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *AAIP*, volume 5812 of *Lecture Notes in Computer Science*, pages 93–116. Springer, 2009.

[19] William Stein et al. *Sage Mathematics Software (Version 4.8)*. The Sage Development Team, 2012. `http://www.sagemath.org`.

[20] John Tromp. Binary lambda calculus and combinatory logic. In Marcus Hutter, Wolfgang Merkle, and Paul M. B. Vitányi, editors, *Kolmogorov Complexity and Applications*, volume 06051 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[21] Jue Wang. The efficient generation of random programs and their applications. Master's thesis, Honors Thesis, Wellesley College, Wellesley, MA, May 2004.

A Terms with exactly m distinct free variables

Here we study the numbers of terms with exactly m distinct free variables, the formulas for counting those numbers and their relations with quantities we considered.

A.1 A formula

Let us show how to derive the formula for counting λ-terms with exactly m distinct free variables. This formula is adapted from a similar one when variables have size 1 due to Raffalli (*On-line Encyclopedia of Integer Sequences* under the number **A135501**). We assume that terms are built with usual variables (not de Bruijn indices) and that they are equivalent up to a renaming of bound variables and up to α-conversion. Let us denote the number of λ-terms of size n with exactly m distinct free variables by $f_{n,m}$.

Notice first that there is no term of size 0 with no free variable, hence $f_{0,0} = 0$. There is one term of size 0 with one free variable, hence $f_{0,1} = 1$. The maximum number of variables for a λ-term of size n is when the only operators are applications and all the variables are different. One has then a binary tree with n internal nodes and $n+1$ leaves holding $n+1$ variables. This means that for m beyond $n+1$ variables there is no term of size n with exactly m distinct free variables. Hence

$$f_{n,m} = 0 \qquad \text{when } m > n+1.$$

In the general case, a term of size $n+1$ with m free variables starts either with an abstraction or with an application. Terms starting with an abstraction, say λx, on a term M contribute in two ways: either M does not contain x as a free variables or M contains x as a free variable. There are $f_{n,m}$ such M's in the first case and $f_{n,m+1}$ in the second. This gives the two first summands $f_{n,m} + f_{n,m+1}$ of the formula. Now, let us see how terms starting with an application look like. Assume they are of the form $P\,Q$ and of size $n+1$. For some $p \leq n$, the term P is of size p and Q is of size $n-p$. These terms share c common variables ($0 \leq c \leq m$), while $P\,Q$ has m distinct free variables altogether. The term P has k distinct free variables, which do not occur in Q, hence P has $k+c$ distinct free variables altogether. The term Q has $m-k$ distinct free variables. Therefore, given a set of private variables for P, a set of common variables, and a set of private variables for Q, there are $f_{p,k+c}f_{n-p,m-k}$ possible pairs (P,Q). There are $\binom{m}{c}$ ways to choose the c common variables among m and there are $\binom{m-c}{k}$ ways to split the remaining variables into P and Q, namely k for P and $m-c-k$ for Q, hence the third summand of the formula:

$$\sum_{p=0}^{n}\sum_{c=0}^{m}\sum_{k=0}^{m-c}\binom{m}{c}\binom{m-c}{k}f_{p,k+c}f_{n-p,m-k}.$$

Now, we obtain the whole formula:

$$f_{n+1,m} \;=\; f_{n,m} + f_{n,m+1} + \sum_{p=0}^{n}\sum_{c=0}^{m}\sum_{k=0}^{m-c}\binom{m}{c}\binom{m-c}{k}f_{p,k+c}f_{n-p,m-k}.$$

A.2 Relations between $T_{n,m}$ and $f_{n,m}$

The number of terms of size n with exactly i indices in $[\underline{1}..\underline{\mathsf{m}}]$ is $\binom{m}{i} f_{n,i}$. Therefore the number of terms with indices in $[\underline{1}..\underline{\mathsf{m}}]$ is:

$$T_{n,m} = \sum_{i=0}^{m} \binom{m}{i} f_{n,i}.$$

By the inversion formula ([8] p. 192), we get:

$$f_{n,m} = \sum_{i=0}^{m} (-1)^{m+i} \binom{m}{i} T_{n,i}.$$

This shows with no surprise that $f_{n,m}$ and $T_{n,m}$ are simply connected. Knowing that the $T_{n,m}$'s can be easily computed, this provides a formula simpler than Raffalli's to compute the $f_{n,m}$'s.

A.3 A relation between $f_{n,m}$ and $c_{n,i}$

We write $R_i^{(m)}$ the number of surjections from $[\underline{1}..\underline{\mathsf{i}}]$ to $[\underline{1}..\underline{\mathsf{m}}]$. To get a relation between $f_{n,m}$ and $c_{n,i}$, we can reproduce the process with which we associated $T_{n,m}$ and $c_{n,i}$ (Section 3), but instead of applications from $[\underline{1}..\underline{\mathsf{i}}]$ to $[\underline{1}..\underline{\mathsf{m}}]$, we have surjections from $[\underline{1}..\underline{\mathsf{i}}]$ to $[\underline{1}..\underline{\mathsf{m}}]$, since this time we count terms with exactly m variables and all the de Bruijn indices must be reached by the applications. Therefore

$$f_{n,m} = \sum_{i=0}^{n} c_{n,i} R_i^{(m)}.$$

Recall that

$$R_i^{(m)} = \sum_{j=0}^{m} \binom{m}{j} (-1)^j (i-j)^m.$$

We can now go further in the expression of $f_{n,m}$.

$$\begin{aligned} f_{n,m} &= \sum_{i=0}^{n} c_{n,i} \sum_{j=0}^{m} \binom{m}{j} (-1)^j (m-j)^i \\ &= \sum_{i=0}^{n} c_{n,i} \sum_{k=0}^{m} \binom{m}{k} (-1)^{m-k} k^i \\ &= \sum_{k=0}^{m} \binom{m}{k} (-1)^{m-k} \sum_{i=0}^{n} c_{n,i} k^i \\ &= \sum_{k=0}^{m} (-1)^{m+k} \binom{m}{k} T_{n,k}. \end{aligned}$$

which is another proof of the formula of Section A.2.

B A program for testing simple typability

In this section we give a simple Haskell program for *testing simple typability* of a term also called *type reconstruction*. The program which works on the type Type:

```
data Type = Var Int
          | Arrow Type Type
```

has three parts. First, the function builds the set of typability constraints of a closed term. This function called build_constraint takes a term and returns its potential principal type, which will be made explicit after solving the constraints, and a list of constraints. It requires a function build which will be called through the terms. Along its traversal of the term, the function build has to know the depth d (the number of λ's it crossed). Moreover, build creates type variables. Actually, a constraint builder creates type variables into two situations, when it creates a context for the first time, that is when it deals with a de Bruijn index, and when it creates the type to be returned by an application. Since type variables are objects of the form Var i, where i is an Int, build takes an Int which is increased whenever a new type variable is created. We call the latter a *cursor* and denote it by cu. build returns a 4-uple, namely the potential principal type of the term, a context (a list of types associated with de Bruijn indices), a set of constraints and the updated cursor.

```
build_constraint :: Term -> (Type, [(Type,Type)])
build_constraint t =
  let (ty,[],constraint,_) = build t 0 0
  in (ty, constraint)
  where
    build :: Term -> Int -> Int -> (Type, [Type], [(Type,Type)],Int)
    build (Index i) d cu =
      let ii = fromIntegral i
      in (Var (cu+ii-1), [Var j | j<-[cu..cu+d-1]],[],cu+d)
    build (Abs t) d cu =
      let (ty,(a:cntxt),constraint,cu') = build t (d+1) cu
      in ((Arrow a ty),cntxt,constraint,cu')
    build (App t1 t2) d cu =
      let (ty1, cntxt1, constraint1, cu1) = build t1 d cu in
      let (ty2, cntxt2, constraint2, cu2) = build t2 d cu1 in
      let ty = (Var cu2) in (ty,
                             cntxt1,
                             (ty1,(Arrow ty2 ty)):(zip cntxt1 cntxt2)
                                ++ constraint1 ++ constraint2,
                             cu2+1)
```

To solve constraints we use a method based on transformation rules. For that, we use a function decompose which splits an equation when both sides are arrow types. Moreover, when decompose meets an equation $\sigma_1 \to \sigma_2 = \alpha$, the latter is transformed into $\alpha = \sigma_1 \to \sigma_2$.

```
decompose :: (Type,Type) -> [(Type,Type)]
decompose ((Arrow ty1 ty2), (Arrow ty1' ty2')) =
  decompose (ty1,ty1') ++ decompose (ty2,ty2')
decompose ((Arrow ty1 ty2),(Var i)) = [(Var i,(Arrow ty1 ty2))]
decompose (ty1,ty2) = [(ty1,ty2)]
```

A predicate non_trivial_eq is necessary to filter out the trivial equations, i.e., of the form $\alpha = \alpha$.

```
non_trivial_eq :: (Type,Type) -> Bool
non_trivial_eq (Var i, Var j) = i /= j
non_trivial_eq (ty1, ty2) = True
```

A predicate checks whether a composed type contains a given variable. This is necessary to detect cycles.

```
occurs_in_type :: Type -> Type -> Bool
occurs_in_type (Var i) (Var j) = False
occurs_in_type (Var i) (Arrow ty1 ty2) = in_type (Var i) ty1 ||
                                         in_type (Var i) ty2
    where in_type :: Type -> Type -> Bool
          in_type (Var i) (Var j) = i == j
          in_type (Var i) (Arrow ty1 ty2) = in_type (Var i) ty1 ||
                                            in_type (Var i) ty2
```

Once this test is done, one can replace a variable α occurring in an equation of the form $\alpha = \sigma$ by σ everywhere in the set of constraints before putting the equation $\alpha = \sigma$ in the solved part.

```
replace_in_type :: (Type,Type) -> Type -> Type
replace_in_type (Var i, ty) (Var j) = if i == j then ty else (Var j)
replace_in_type (Var i, ty) (Arrow ty1 ty2) =
  Arrow (replace_in_type (Var i, ty) ty1)
        (replace_in_type (Var i, ty) ty2)

replace_in_eq ::  (Type,Type) -> (Type,Type) -> (Type,Type)
replace_in_eq (Var i,ty) (ty1,ty2) =
  (replace_in_type (Var i,ty) ty1, replace_in_type (Var i,ty) ty2)
```

The function solve solves the set of constraints. It returns True if the set of constraints has a solution, that is when the set of constraints is empty. It returns False when it detects a cycle. Otherwise it tries to apply the transformations whenever it is possible, that is when the set of constraints is not empty. Indeed if there is no cycle and if the set of constraints is not empty, a transformation is always applicable.

```
solve :: [(Type,Type)] -> [(Type,Type)]
                       -> ([(Type,Type)],[(Type,Type)],Bool)
solve ((Var i,ty):l) sol =
  if occurs_in_type (Var i) ty
```

```
  then ((Var i,ty):l,sol,False) -- cycle detected
  else solve (map (replace_in_equation (Var i,ty)) l)
                                        ((Var i,ty):sol)
solve (eq:l) sol =
  solve (filter non_trivial_eq (decompose eq) ++ l) sol
solve [] sol = ([],sol,True)
```

Since we have all the ingredients, the test of typability consists in building the constraint and trying to solve it.

```
is_typable :: Term -> Bool
is_typable t = let (_,c) = build_constraint t
               in let (_,_,b) = solve c [] in b
```

Notice that we have everything to build the principal type of the term if it is typable.