Generating Random Terms in Beta Normal Form of the Simply-Typed Lambda Calculus

Jue Wang
Boston University
juewang@cs.bu.edu

July 10, 2005

1 Introduction

We can use randomly generated lambda calculus terms to perform statistical sampling experiments or to serve as inputs for term manipulation algorithms. In [6], we investigated the problem of generating lambda calculus terms uniformly at random, assuming a uniform distribution over all terms of a given size. However, size is not the only useful criterion that should be considered and often, size alone is too broad to limit the sample space effectively. Instead, we would like to consider additional criteria and the negations of criteria such as context, typability, type, and signature. In this report, we use simple types as a guide and focus on generating closed terms in β-normal form of a given size and a given type.

2 Simply Typed Lambda Calculus

In this report, we present an algorithm to generate closed, pure lambda calculus terms in normal form of a given size and a given simple type uniformly at random. We follow conventions and notations of Barendregt's book [2]. A lambda term can be either a variable, an application, or an abstraction. The set of lambda terms is defined by the following grammar:

$$M ::= x \mid (NP) \mid (\lambda x.N)$$

where x ranges over an infinite set of variables and M, N, and P range over lambda terms. A term is in β-normal form if it does not have any β redexes, i.e., it does not have any subterms of the form $(\lambda x.M)N$.

We use types in the simple type system [1] to limit the sample space of terms. The simple types can either be a type variable or an arrow type. The set of simple types is defined by the following grammar:

$$A ::= a \mid B \to C$$

where lower-case letters at the beginning of the Roman alphabet$(a, b, c, \ldots)$ ranges over type variables and their upper-case counterparts$(A, B, C, \ldots)$ range over types. Lambda calculus terms can be assigned simple types using the following typing rules:

$$(var)\ \Gamma \vdash x : A,\ \ \Gamma(x) = A \qquad (\to E)\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \qquad (\to I)\ \frac{\Gamma \cup \{x : A\} \vdash M : B}{\Gamma \vdash (\lambda x.M) : (A \to B)}$$

In the above typing rules, Γ denotes a type environment containing a set of bindings between distinct lambda calculus variables and types. Every binding within a type environment has the

form $x : A$, where x is a variable and A is the type of x. In addition, we assume the following two functions for a type environment. The function $dom(\Gamma)$ returns the set of distinct variables that are bound in Γ. In addition, $\Gamma(x)$ returns the type of x in Γ if the binding exists.

In our algorithm, when given a type A, we consider all β-normal terms that can be assigned the type A using the given typing rules. For example, when given the type $(a \to a) \to a \to a$, every term inhabiting this type has the form $(\lambda x.x)$ or $\lambda x.\ \lambda y.\ x(x(\ldots(xy)\ldots))$ where x occurs 0 or more times. From this example, we note the following two things. First, we are not only limited to generating principal inhabitants of the given type. A term M is a principal inhabitant of a type A if A is the principal type of M. Here, we include the term $(\lambda x.x)$ as an inhabitant of the type $(a \to a) \to a \to a$ even though $(\lambda x.x)$ has the principal type $a \to a$. Second, there are an infinite number of terms inhabiting $(a \to a) \to a \to a$ since x can occur an infinite number of times in the case $\lambda x.\ \lambda y.\ x(x(\ldots(xy)\ldots))$. Thus, given that some types are inhabited by an infinite number of terms, it is not enough to restrict the sample space by type alone and we use the following definition of size to limit the sample space.

$$size(M) = \begin{cases} 1 & \text{if } M \equiv x, \\ 1 + size(N) + size(P) & \text{if } M \equiv (NP), \\ 1 + size(N) & \text{if } M \equiv (\lambda x.N) \end{cases}$$

In Fig. 1, we show every β-normal term of the type $((a \to b) \to a \to b) \to (a \to b) \to a \to b$ for sizes 2 through 11. For our algorithm, we assume a uniform distribution over all terms of a given size. We note that we do not explicitly annotate the variables with the type.

We generate terms satisfying the given restrictions modulo α-equivalence. To avoid generating α-equivalent terms, we adopt a convention of systematic variable naming for the term variables we generate using the type as an index. Every binding occurrence of a variable with some type A starts with the symbol x and ends with two indices separated by ".". The first index corresponds to the index of the type A in the environment and the second index represents the number of variables of type A that already exist along the path from the root to this new binding occurrence in the tree representation. For example, in Fig. 1, the second and third term of size 11 differ by the bound variable occurrence to which the variable $x_{1.0}$ is applied. Both variables have 0 as the first index, indicating that they both have type a and the second index distinguish whether this refers to the first or second binding occurrence of a variable with type a.

3 A Grammar to Describe Normal Form Terms of a Given Type

The first step to generating β-normal terms satisfying the specified criteria of size and type is to have a way of concisely describing all the terms of interest. Here, we first ignore the criteria of size and focus on β-normal terms of a given type. In [5], Takahashi et al. formulated the grammar presented in Fig. 2 to generate the set $B(\Gamma, A) = \{M \mid \Gamma \vdash M : A \text{ is in } \beta\text{-normal form}\}$. In this grammar, the set of terminals T are the syntactic building blocks of a term such as term variables and lambdas. Each non-terminal has the form $\langle \Gamma, A \rangle$. The set of production rules derived from a given non-terminal $\langle \Gamma, A \rangle$ describes the set of β-normal terms with type A using environment Γ where each rule describes one particular case. A proof of the correctness of this proof can be found in [5].

In Fig. 3, we see a step-by-step construction of the grammar describing the set $B(\Gamma, A)$ where $\Gamma = \emptyset, A = A_1 \to A_2 \to A_3 \to b$, $A_1 = (a \to b) \to a \to b$, $A_2 = a \to b$, and $A_3 = a$. In this example, each horizontal block of the table denotes the new members of the set N, T, R generated from a particular non-terminal. For example, the first block in Fig. 3 lists the terminals, non-terminals, and production rules generated by the starting non-terminal $\langle \Gamma, A \rangle$. Once we are done with the starting non-terminal $\langle \Gamma, A \rangle$, we need to generate rules for the newly created non-terminals. In the second

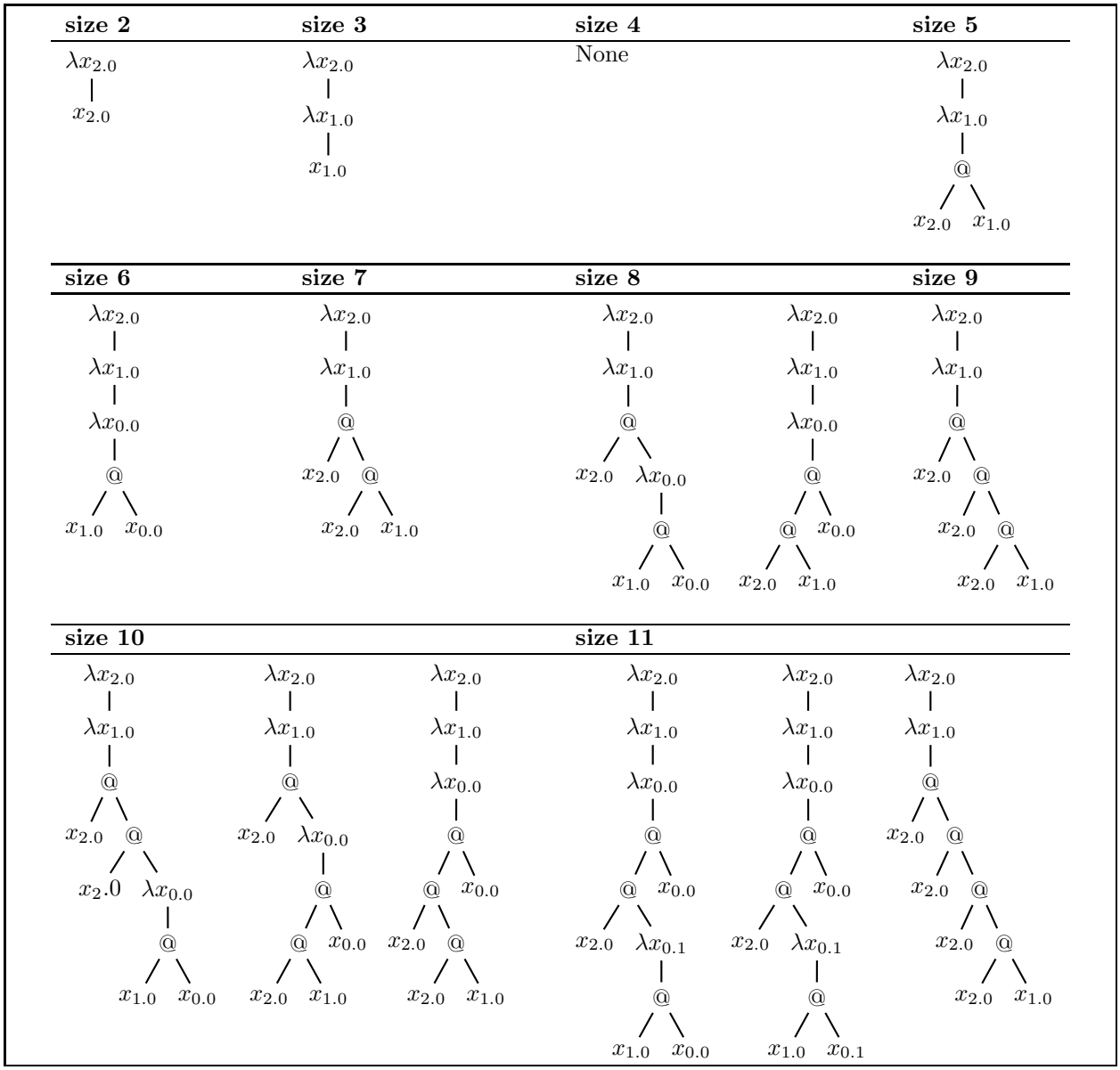

Figure 1: Normal form terms of type $((a \to b) \to a \to b) \to (a \to b) \to a \to b$ for size 2 through 11.

Given an environment Γ and a type A, the grammar $G(\Gamma, A) = (T, N, R, \langle \Gamma, A \rangle)$ generates the set $B(\Gamma, A) = \{M \mid \Gamma \vdash M : A \text{ is in } \beta\text{-normal form}\}$, where T(the set of terminal symbols), N(the set of non-terminal symbols) and R (the set of production rules) are defined as follows:

- $dom(\Gamma) \cup \{\lambda, ., (,)\} \subseteq T$
- $\langle \Gamma, A \rangle \in N$
- For every $\langle \Delta, C \rangle \in N$:
  if $C \equiv C_1 \to C_2 \to \cdots \to C_m \to c$ then,
  for every $p = 1, \ldots, m$:
  if $(x : B_1 \to \cdots \to B_q \to C_{p+1} \to \cdots \to C_m \to c) \in \Delta'$,
  where $\Delta' = \Delta \cup \{x_i^{\langle \Delta, C \rangle} : C_i \mid i = 1, \ldots, p\}$ then,
  - $x_i^{\langle \Delta, C \rangle} \in T \;\; (i = 1, \ldots, m)$,
  - $\langle \Delta', B_j \rangle \in N \;\; (j = 1, 2, \ldots, q)$, and
  - $\langle \Delta, C \rangle \rightsquigarrow (\lambda x_1^{\langle \Delta, C \rangle} x_2^{\langle \Delta, C \rangle} \ldots x_p^{\langle \Delta, C \rangle} . x \langle \Delta', B_1 \rangle \langle \Delta', B_2 \rangle \ldots \langle \Delta', B_q \rangle) \in R$.

Figure 2: A grammar to generate the set of β-normal terms of type A with an environment Γ.

block, we list the rules generated for the non-terminal $\langle \Gamma', A_2 \rangle$ and the terminals and non-terminals resulting from the newly generated rules. From this example, it is easy to see that the grammar show in Fig. 2 is an infinite context-free grammar since the set N could be an infinite set. More specifically, the grammar $G(\Gamma, A)$ shown in Fig. 3 is infinite because each non-terminal $\langle \Gamma, A_2 \rangle$ where Γ denotes some environment will generate a new non-terminal $\langle \Gamma', A_2 \rangle$ where $\Gamma' = \Gamma \cup \{x_1^{\langle \Gamma, A_2 \rangle} : a\}$. Since we also use the restriction of size in our generation, the resulting grammar in our case would be finite.

The current grammar is not concise enough for the purpose of generating terms. Instead, we modify the grammar to better suit the purpose of generating terms based on the following observation. We note that the infinite nature of the grammar arises from the changing environment that generates an infinite set of non-terminals. However, the productions deriving from the two distinct non-terminals $\langle \Gamma', A_2 \rangle$ and $\langle \Gamma'', A_2 \rangle$ are similar in structure and differ only in the environments. Furthermore, when given a type, we can argue that even though the number of distinct variables in the environment increments, the distinct types for the variables in the environment is a constant set.

To show this, we first define the following function which returns all possible argument types embedded in a given type.

$$args(A) = \begin{cases} A_1 \cup A_2 \cup \ldots \cup A_n \cup args(A_1) \cup \ldots \cup args(A_n) & if\ A = A_1 \to A_2 \ldots \to A_n \to a \\ \{\} & if\ A = a \end{cases}$$

Now, we make the following claim.

Claim 1. *Given a closed, β-normal term M with type $A = A_1 \to A_2 \ldots \to A_n \to a$, every variable occurrences of x in M has the type A' such that $A' \in args(A)$.*

Proof. First, we associate the notion of depth with each variable occurrence. Intuitively, the depth of a variable corresponds to the number of application nodes that appear above the binding occurrence for the given variable in the parse tree. We can also use the function defined below to collect the depth of all variables occurring in a term M in a table. We assume the renaming of bound variables so that entries will not be overwritten. For a given variable x, we define $depth(x, M) = lookup(x, getVarDepth(M, 0, emptyTbl))$.

To generate every β-normal term of the type $A_1 \to A_2 \to A_3 \to b$,
where $A_1 = (a \to b) \to a \to b$, $A_2 = a \to b$, and $A_3 = a$.

T	N	R
(,),λ,.	$\langle \Gamma, A \rangle$	$\langle \Gamma, A \rangle \rightsquigarrow \lambda x_1^{\langle \Gamma, A \rangle}.x_1^{\langle \Gamma, A \rangle}$
$x_1^{\langle \Gamma, A \rangle}$	$\langle \Gamma', A_2 \rangle$	$\langle \Gamma, A \rangle \rightsquigarrow \lambda x_1^{\langle \Gamma, A \rangle} \lambda x_2^{\langle \Gamma, A \rangle}.x_1^{\langle \Gamma, A_2 \rangle} \langle \Gamma', A_2 \rangle$
$x_2^{\langle \Gamma, A \rangle}$	$\langle \Gamma'', A_2 \rangle$	$\langle \Gamma, A \rangle \rightsquigarrow \lambda x_1^{\langle \Gamma, A \rangle} \lambda x_2^{\langle \Gamma, A \rangle} \lambda x_3^{\langle \Gamma, A \rangle}.x_1^{\langle \Gamma, A \rangle} \langle \Gamma'', A_2 \rangle \langle \Gamma'', A_3 \rangle$
$x_3^{\langle \Gamma, A \rangle}$	$\langle \Gamma'', A_3 \rangle$	$\langle \Gamma, A \rangle \rightsquigarrow \lambda x_1^{\langle \Gamma, A \rangle} \lambda x_2^{\langle \Gamma, A \rangle} \lambda x_3^{\langle \Gamma, A \rangle}.x_2^{\langle \Gamma, A_2 \rangle} \langle \Gamma'', A_3 \rangle$
$x_1^{\langle \Gamma', A_2 \rangle}$	$\langle \Gamma''', A_2 \rangle$	$\langle \Gamma', A_2 \rangle \rightsquigarrow x_2^{\langle \Gamma, A \rangle}$
	$\langle \Gamma''', A_3 \rangle$	$\langle \Gamma', A_2 \rangle \rightsquigarrow \lambda x_1^{\langle \Gamma', A_2 \rangle}.x_1^{\langle \Gamma, A \rangle} \langle \Gamma''', A_2 \rangle \langle \Gamma''', A_3 \rangle$
		$\langle \Gamma', A_2 \rangle \rightsquigarrow \lambda x_1^{\langle \Gamma', A_2 \rangle}.x_2^{\langle \Gamma, A_2 \rangle} \langle \Gamma''', A_3 \rangle$
$x_1^{\langle \Gamma'', A_2 \rangle}$	$\langle \Gamma'''', A_2 \rangle$	$\langle \Gamma'', A_2 \rangle \rightsquigarrow x_2^{\langle \Gamma, A \rangle}$
	$\langle \Gamma'''', A_3 \rangle$	$\langle \Gamma'', A_2 \rangle \rightsquigarrow \lambda x_1^{\langle \Gamma'', A_2 \rangle}.x_1^{\langle \Gamma, A \rangle} \langle \Gamma'''', A_2 \rangle \langle \Gamma'''', A_3 \rangle$
		$\langle \Gamma'', A_2 \rangle \rightsquigarrow \lambda x_1^{\langle \Gamma'', A_2 \rangle}.x_2^{\langle \Gamma, A_2 \rangle} \langle \Gamma'''', A_3 \rangle$
		$\langle \Gamma'', A_3 \rangle \rightsquigarrow x_3^{\langle \Gamma, A \rangle}$
		$\langle \Gamma''', A_3 \rangle \rightsquigarrow x_3^{\langle \Gamma, A \rangle}$
		$\langle \Gamma''', A_3 \rangle \rightsquigarrow x_1^{\langle \Gamma', A_2 \rangle}$
$\vdots$	$\vdots$	$\vdots$

Figure 3: The grammar for the set $B(\Gamma, ((a \to b) \to a \to b) \to (a \to b))$.

$$getVarDepth(M, d, tbl) = \begin{cases} getVarDepth(N, d, tbl.add(x, d)) & if\ M \equiv (\lambda x.N) \\ getVarDepth(N, d+1, getVarDepth(P, d+1, tbl))) & if\ M \equiv (NP) \\ tbl & if\ M \equiv x \end{cases}$$

We prove the claim by induction on the depth of a variable.

Base case: We consider variables x such that $depth(x, M) = 0$. Then x is defined by a top-level lambda and $type(x) \in args(A)$ by definition.

Induction Hypothesis: We assume that for all variables x such that $depth(x, M) \leq n$, $type(x) \in args(A)$. We show that for all variables y such that $depth(y, M) = n + 1$, $type(y) = A' \in args(A)$.

Let the binding occurrence of y be part of a subterm P with type B of the form $P \equiv \lambda z_1 : B_1.\ \lambda z_2 : B_2 . \ldots \lambda y : A' . \ldots \lambda z_n : B_n.N$. Because we only consider terms in normal form, all lambda bindings can only occur in the operand position. Then, we must supply P as an argument to some variable $x : C_1 \to \cdots \to B \to \cdots \to C_n$ where $depth(x) \leq n$. Since $type(x) \in args(A)$ by the induction hypothesis, then $A' \in args(A)$ by definition. □

Based on the above observations, we can modify the grammar developed in [5] to obtain the grammar presented in Fig. 4 where we treat the environment as an argument which is abstracted over. In Fig. 5, we see an example of the generated grammar that describes the set $B(\emptyset, A)$. In this grammar, we added non-terminal symbols of the form $[\Gamma, A]$ which derives all variables in the environment with the given type A. Using Claim 1, we know that there is a limited number of types available in the environment, so we also have a set number of productions to derive a term of a given type that starts with a head variable.

Given an environment Γ and a type A, the grammar $G(\Gamma, A) = (T, N, R, \langle \Gamma, A \rangle)$ generates the set $B(\Gamma, A) = \{M \mid \Gamma \vdash M : A \text{ is in } \beta\text{-normal form}\}$, where T(the set of terminal symbols), N(the set of non-terminal symbols) and R (the set of production rules) are defined as follows:

- $\{\lambda, ., (,)\} \subseteq T$
- $\langle \Gamma, A \rangle \in N$, $[\Gamma, A] \in N$
- For each non-terminal $\langle \Gamma, A \rangle$,
  - The set of production rules R:

$$\begin{aligned}
\langle \Gamma, A \rangle &\rightsquigarrow [\Gamma, A] \\
&\rightsquigarrow x_i^B \langle \Gamma, B_1 \rangle \ldots \langle \Gamma, B_m \rangle, \\
&\quad \text{for } i = 0, \ldots, (\mathit{typeCnt}(\Gamma, B) - 1), \forall B \equiv B_1 \to \cdots \to B_m \to A \in \mathit{typeSet}(\Gamma)) \\
&\rightsquigarrow \lambda x^{A_1}_{\mathit{typeCnt}(\Gamma, A_1)} : A_1 . \langle \mathit{typeCntInc}(\Gamma, A_1), A_2 \to \cdots \to A_n \to a \rangle, \\
&\quad \text{if } A = A_1 \to A_2 \to \cdots \to A_n \to a, n \geq 1 \\
[\Gamma, A] &\rightsquigarrow x_j^A, \text{ for } j = 0 \ldots (\mathit{numType}(\Gamma, A) - 1)
\end{aligned}$$

  - if $A = A_1 \to A_2 \to \cdots \to A_n \to a$, then $x^{A_i}_{\mathit{typeCnt}(\Gamma, A_i)} \in T$, for $i = 0, \ldots, n$
  - $\langle \Gamma', A_2 \to \cdots \to A_n \to a \rangle \in N$,
    $\langle \Gamma, B_i, \rangle \in N$, for $i = 0, \ldots, m, \forall B \equiv B_1 \to \cdots \to B_m \to A \in \mathit{typeSet}(\Gamma))$

Figure 4: A grammar to generate the set of β-normal terms of type A with an environment Γ.

For the convenience of introducing variable bindings, we define an environment to be a set of bindings between a distinct type A and an integer indicating the number of variables in the environment with the give type A. This definition differs from the traditional sense of an environment which is a mapping between a set of distinct variables and their types. With our definition, the example environment $\Gamma_0 = \{a : 1, a \to b : 3, a \to a : 0\}$ denotes an environment where there is 1 variable of type a, 3 variables of type $a \to b$, and no variable of type $a \to a$. Additionally, we assume the following three functions operating on environments.

- $\mathit{typeSet}(\Gamma)$ is a function that returns a distinct set of types that are bound in the given environment Γ. For example, $\mathit{typeSet}(\Gamma_0) = \{a, a \to b, a \to a\}$.
- $\mathit{typeCnt}(\Gamma, A)$ is a function which returns the number of variables in Γ with type A. Using the example environment given above, we have $tc(\Gamma_0, a) = 1$.
- $\mathit{typeCntInc}(\Gamma, A)$ is a function that increments the number of variables of type A in Γ by 1. For example, $\mathit{typeCntInc}(\Gamma_0, a)$ yields a new environment $\Gamma_1 = \{a : 2, a \to b : 3, a \to a : 0\}$.

Such an environment is sufficient in our grammar since by Claim 1, we know that the environment contains a restricted set of types based on the final desired type we want to generate. Certainly, we can use an environment in the traditional sense as described in Sect.1, but this new notion is a more efficient way of keeping track of available variables. It also helps in the systematic introduction of identifiers in variables bindings described in Sect. 1 to avoid generating α-equivalent terms.

We now briefly argue the correctness of the grammar given in Fig. 4. For a given type A and environment Γ, we prove that a term M has type A by constructing a proof derivation tree using the typing rules presented in Sect. 1. We show that every term M generated by the grammar is of type A and is in β-normal form via a case analysis on the productions of the grammar for the non-terminal $\langle \Gamma, A \rangle$. If the first production of $[\Gamma, A]$ is used, then we generate all variables of type A in Γ and variables are obviously in β-normal form. If the second production of $x_i^B \langle \Gamma, B_1 \rangle \ldots \langle \Gamma, B_m \rangle$

is used, we can assume that $\langle \Gamma, B_1 \rangle \ldots \langle \Gamma, B_m \rangle$ generates m β-normal terms $y_1, \ldots, y_m$ of type $B_1, \ldots, B_M$. The final term $M = (\ldots (x_i^B \ y_1) \ldots y_m)$ can be proved to have the type A by a series of $(\rightarrow E)$ rules. Also, M is in β normal form since $y_1, \ldots, y_m$ are in β-normal form and the subterm $x_i^B \ y_1$ does not have a β-redex. Finally, if $A = A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow a$, then the third production can be used to generate terms. Assuming that $\langle typeCntInc(\Gamma, A_1), A_2 \rightarrow \cdots \rightarrow A_n \rightarrow a \rangle$ generates the set $B(\Gamma, A_2 \rightarrow \cdots \rightarrow A_n \rightarrow a)$, we can prove that the terms $\lambda x^{A_1}_{typeCnt(\Gamma, A_1)} : A_1 . \langle typeCntInc(\Gamma, A_1), A_2 \rightarrow \cdots \rightarrow A_n \rightarrow a \rangle$ has type A by using the $(\rightarrow I)$ rule. It is easy to prove that all β-normal terms of type A can be generated via a case analysis on the last typing rule of the typing derivation used to derive the type A for the given term. This analysis is omitted here.

To generate every β-normal term of the type $A = A_1 \rightarrow A_2 \rightarrow a \rightarrow b$,
where $A_1 = (a \rightarrow b) \rightarrow a \rightarrow b$, $A_2 = a \rightarrow b$.
The initial environment $\Gamma_0 = \{A_1 : 0, A_2 : 0, a : 0\}$.

The set of terminals T:
$\lambda, ., (,)$
$x_j^{A_2 \rightarrow a \rightarrow b}$, for $j = 0 \ldots (typeCnt(\Gamma, A_2 \rightarrow a \rightarrow b) - 1)$, $x_j^{A_2}$, for $j = 0 \ldots (typeCnt(\Gamma, A_2) - 1)$, and
x_j^a, for $j = 0 \ldots (typeCnt(\Gamma, a) - 1)$

The set of non-terminals N:
$\langle \Gamma, A \rangle, \langle \Gamma, A_2 \rightarrow a \rightarrow b \rangle, \langle \Gamma, a \rightarrow b \rangle, \langle \Gamma, a \rightarrow b \rangle, \langle \Gamma, b \rangle, \langle \Gamma, a \rangle$, $[\Gamma, A_2 \rightarrow a \rightarrow b]$, $[\Gamma, A_2]$,$[\Gamma, a]$

The set of rules R:

$\langle \Gamma, A \rangle$	$\rightsquigarrow$	$\lambda x^{A_1}_{typeCnt(\Gamma, A_1)} : A_1 . \langle typeCntInc(\Gamma, A_1), A_2 \rightarrow a \rightarrow b \rangle$
$\langle \Gamma, A_2 \rightarrow a \rightarrow b \rangle$	$\rightsquigarrow$	$[\Gamma, A_2 \rightarrow a \rightarrow b]$ $\lambda x^{A_2}_{typeCnt(\Gamma, A_2)} : A_2 . \langle typeCntInc(\Gamma, A_2), a \rightarrow b \rangle$
$\langle \Gamma, a \rightarrow b \rangle$	$\rightsquigarrow$	$[\Gamma, a \rightarrow b]$ $\lambda x^{a}_{typeCnt(\Gamma, a)} : a . \langle typeCntInc(\Gamma, a), b \rangle$ $[\Gamma, A_1] \langle \Gamma, a \rightarrow b \rangle$
$\langle \Gamma, b \rangle$	$\rightsquigarrow$	$[\Gamma, b]$ $x_i^{A_1} \langle \Gamma, A_2 \rangle \langle \Gamma, a \rangle$, for $i = 0 \ldots (typeCnt(\Gamma, A_1) - 1)$ $x_i^{A_2} \langle \Gamma, a \rangle$, for $i = 0 \ldots (typeCnt(\Gamma, A_2) - 1)$
$\langle \Gamma, a \rangle$	$\rightsquigarrow$	$[\Gamma, a]$
$[\Gamma, A_2 \rightarrow a \rightarrow b]$	$\rightsquigarrow$	$x_j^{A_2 \rightarrow a \rightarrow b}$, for $j = 0 \ldots (typeCnt(\Gamma, A_2 \rightarrow a \rightarrow b) - 1)$
$[\Gamma, A_2]$	$\rightsquigarrow$	$x_j^{A_2}$, for $j = 0 \ldots (typeCnt(\Gamma, A_2) - 1)$
$[\Gamma, a]$	$\rightsquigarrow$	x_j^{a}, for $j = 0 \ldots (typeCnt(\Gamma, a) - 1)$
$[\Gamma, b]$	$\rightsquigarrow$	None

Figure 5: The grammar for the set $B(\Gamma, ((a \rightarrow b) \rightarrow a \rightarrow b) \rightarrow (a \rightarrow b))$.

4 Generating Terms of a Given Type

In this section, we describe an algorithm to generate closed terms in β-normal form of a given size and type uniformly at random, assuming a uniform distribution over all terms of the given size and type. We generate a term in a top-down manner. At each stage, we choose to generate a term guided both by its type and size. When the type and size of the term do not uniquely determine the shape of the generated term, we use the counting functions to determine the probability of each shape to guide the generation. Following the structure of the description given in Fig. 4, it is easy to translate the description into functions that count and generate terms. In Fig. 6, Fig. 7, and Fig. 8, we present the pseudo-code for the functions to count and generate a term of the given specification. We have an implementation of the algorithm in OCaml.

To count the number of terms of a given type A and size s, we define the functions $countTerm$, $countHeadVarTerm$, and $countHeadVarArgTerms$ in a recursive manner. Since we are interested in closed terms, the function *count* initializes $countTerm$ with an empty environment, and the desired type and size. The function $countTerm$ returns the number of β-normal terms satisfying the given specification by performing case analysis based on the size. If the size is 0, then the number of available terms is obviously 0. If the size is 1, then the only terms with the given type are term variables, and we query the given environment for the number of variables in the environment that have the given type. If the size is greater than 1, we consider separately the case where the desired type is a type variable or an arrow type. If the desired type is a type variable a, then we invoke $countHeadVarTerm$ with a, Γ, and s. The function $countHeadVarTerm$, when invoked with a type A, an environment Γ, and a size s, counts the number of application terms of the form $((\ldots(xM_1)\ldots M_{k-1})M_k$ where x is a term variable with type $B_1 \to \cdots \to B_k \to A$ and M_i is some term of type B_i for $1 \leq i \leq k$. $countHeadVarTerm$ works by considering all variables x_j^B in Γ that have A as the return type, and for each of the variables x_j^B, we calculate all the possible ways of generating terms with x_j^B as the head variable using the function $countHeadVarArgTerms$. If A is an arrow type of the form $A_1 \to A_2$, then we can either generate a top-level λ term or a top-level application term and we add up the number of terms in both cases. The number of top-level λ terms with type $A_1 \to A_2$ can be found by invoking $countTerm$ with the type A_2, the size decremented by 1, and a new environment where the number variables with type A_1 is incremented by 1. In the application case, the application must have a term variable as the operator and we again invoke the function $countHeadVarTerm$ to count these terms.

To generate a term, we use the function $genTerm$, $genVarTerm$, $genLamTerm$, and $genAppTerm$ in a recursive manner. We use *gen* to generate a closed β-normal form term of a given size s and type A by invoking $genTerm$ with an empty environment along with the desired size and type. Again, we perform case analysis based on the size. If the size is 0, then no terms exist. If the size is 1, then we return a term variable of the desired type generated by the function $genVarTerm$ which chooses one the term variables of type T in Γ uniformly at random. Otherwise, if the given type is a type variable, then the only typing rule we could have used to derive A is $(\to E)$ and we must use the function $genAppTerm$ to generate a term with top-level application. In generating an application term, we first choose a head variable using the function $chooseHeadVar$ to generate the type of the chosen head variable and the size breakdown of the arguments. With this information, we can now use $genVarTerm$ to generate the head variable, $genTerm$ to generate the argument terms, and construct the term from these pieces. Finally, given that the desired type is $T = A_1 \to A_2$, we can derive it using either $(\to I)$ or $(\to E)$, so we can generate either a lambda term or application term. Using $countTerm$, we count the total number of terms of the given type and size and also the total number of lambda terms with the given type and size. We choose to generate either a top-level lambda term or an application term with a probability corresponding the the percentage of terms with the given shape. If we choose a top-level lambda, then we can generate a variable of type A_1, and generate a body by invoking $genTerm$ on the type A_2, the size

Input: type $A = A_1 \to A_2 \to \ldots \to A_n \to a$, size s
Output: the number of β-normal terms of size s and type A.

$count(A, s) = countTerm(A, \emptyset, s)$

$countTerm(A, \Gamma, s) =$
  IF $s = 0$ THEN
    RETURN 0
  ELSE IF $s = 1$ THEN
    RETURN $typeCnt(\Gamma, A)$
  ELSE
    IF $varType?(A)$ THEN
      RETURN $countHeadVarTerm(A, \Gamma, s)$
    ELSE
      RETURN $countTerm(resType(A), typeCntInc(\Gamma, argType(A)), s-1)$
        $+ countHeadVarTerm(A, \Gamma, s)$

$countHeadVarTerm(A, \Gamma, s) =$
  $numTerms \leftarrow 0$
  FOR each $B = B_1 \to \ldots \to B_m \to A \in validHeadVarTypeSet(A, \Gamma)$
    $numTerms \leftarrow numTerms + countHeadVarArgTerms(B, \Gamma, s)$
  RETURN $numTerms$

$countHeadVarArgTerms(B, \Gamma, s) =$
  $numTerms \leftarrow 0$
  $numVarWithTypeInEnv \leftarrow typeCnt(\Gamma, B)$
  IF $numVarWithTypeInEnv > 0$ THEN
    FOR each $(s_1, \ldots, s_m) \in ndk(s-1-m, m)$
      $numTerms \leftarrow numTerms + \prod_{i=1}^{m} \text{countTerm}(B_i, \Gamma, s_i)$
  RETURN $numVarWithTypeInEnv \times numTerms$

Figure 6: The algorithm to count the number of normal form terms of a given type and size.

$$varType?(A) = \begin{cases} \text{true} & \text{if } A = a \\ \text{false} & \text{if } A = A_1 \to A_2 \end{cases} \qquad arrowType?(A) = \begin{cases} \text{false} & \text{if } A = a \\ \text{true} & \text{if } A = A_1 \to A_2 \end{cases}$$

$$argType(A) = \begin{cases} Error & \text{if } A = a \\ A_1 & \text{if } A = A_1 \to A_2 \end{cases} \qquad resType(A) = \begin{cases} Error & \text{if } A = a \\ A_2 & \text{if } A = A_1 \to A_2 \end{cases}$$

$validHeadVarTypeSet(A, \Gamma) = \{B \mid B = B_1 \to \ldots \to B_m \to A, m \geq 0, B \in dom(\Gamma)\}$

$ndk(n, k) = \{(s_1, \ldots, s_k) \mid s_i \geq 1, \sum_{i=1}^{k} s_i = n\}$

$random(bound)$ =random number generator that returns a random number between 0(inclusive) and $bound$(exclusive).

Figure 7: Auxiliary functions shared between count and gen functions.

decremented by 1 and a new environment where the count of A_1 is incremented. If we choose a top-level application, then we again use *genAppTerm* to generate the desired term.

5 Related Work

There is a lot of previous work done on the problem of counting and enumerating β-normal terms of a given simple type([1],[5], [7]). The different approaches exploit the same idea of using the fixed structure of the terms derived from the type and β-normal shape restrictions to guide the counting and enumeration. In this report, we adapted the grammar presented in [5]. In [1], Hindley presents a search algorithm due to Ben-Yelles which finds the number of terms of a given type in β normal form. Ben-Yelles' method depends on a notion of meta-variables to denote the set of terms of a given type and proceeds by expanding these meta-variables. These meta-variables can in fact be thought of as the non-terminals in the grammar presentation. In [7], Zaionc proposed a fixpoint technique for counting the number of closed terms in long β-normal form. Zaionc translated the set of equations derived to solve the number of terms of a given type into a set of polynomials. Then, the problem of counting terms is reduced to the problem of finding a fixpoint to this set of polynomials. In all of these cases, the focus has been on the counting and enumeration of terms aimed at using the solution to solve the type inhabitation problem and counting proofs in the implicational fragment of the propositional intuitionistic logic using the Curry-Howard isomorphism. In our case, we deal with the random generation of β-normal terms of a given type for the purpose of generating viable test cases.

Harry Mairson first considered a similar problem in the context of unambiguous context free grammar. In his paper [3], he presented two algorithms that generated random words derived from a context free grammar. Both algorithms used a preprocessing stage where a table is built to store information for later use in generating words uniformly at random. There was a time versus space trade-off between the two algorithm presented. The first algorithm generates a word in $\mathrm{O}(n^2)$ time using a data structure of size $\mathrm{O}(n)$, and the second algorithm uses data structure of $\mathrm{O}(n^2)$ and runs in $\mathrm{O}(n)$ time. Unlike the problem considered here, Mairson's algorithm did not consider issues of binding forms and free variables found in generating random lambda calculus terms.

6 Future Work

6.1 Beyond Current Restrictions

The current formulation of the problem imposes many restrictions that simplify the problem, but also make the solution too simplistic to be of real practical use. We would like to go beyond the current restrictions and in the following subsections, we describe some of the additional criteria we would like to investigate.

6.1.1 Terms in Non β-Normal Form

In this report, we only generated terms already in β-normal form. From the standpoint of type inhabitance, it is enough to be only concerned with normal forms since if a type is inhabited, it is inhabited by a term in normal form. However, in the context of using these generated terms for experimentation and testing purposes, it would be nice to go beyond normal forms and generate terms that contain β-redexes. One possible approach could be to first generate all terms in normal form, and then consider all terms that are some k number of β steps away from normal form by generating λ-I β redexes.

Input: type $A = A_1 \to A_2 \to \ldots \to A_n \to a$, size s
Output: a β-normal term of size s and type A, generated uniformly at random.

$gen(A, s) = genTerm(A, \emptyset, s)$

$genTerm(A, \Gamma, s) =$
  IF $s < 1$ THEN
    RETURN $None$
  ELSE IF $s = 1$ THEN
    IF $typeCnt(\Gamma, A) > 0$ THEN
      RETURN $genVarTerm(\Gamma, A)$
    ELSE
      RETURN $None$
  ELSE IF $arrowType?(A)$ THEN
    $totalNumTerm \leftarrow countTerm(A, \Gamma, s)$
    $numLamTerm \leftarrow countTerm(resType(A), typeCntInc(\Gamma, argType(A)), s-1)$
    IF $(random\ totalNumTerm) < numLamTerm$ THEN
      RETURN $genLamTerm(argType(A), resType(A), \Gamma, s)$
    ELSE
      RETURN $genAppTerm(A, \Gamma, s, totalNumTerms - numLamTerm)$
  ELSE
    RETURN $genAppTerm(A, \Gamma, s, countTerm(A, \Gamma, s)$

$genVarTerm(A, \Gamma) =$
  IF $typeCnt(\Gamma, A) = 0$ THEN
    RETURN $None$
  ELSE
    RETURN $x^{index(\Gamma,A)}_{random\ typeCnt(\Gamma,A)}$

$genLamTerm(argType, resType, \Gamma, s) =$
  $var \leftarrow x^{index(\Gamma,argType)}_{typeCnt(\Gamma,argType)}$
  $body \leftarrow genTerm(resType, typeCntInc(\Gamma, argType), s-1)$
  RETURN $\lambda var.\ body$

$genAppTerm(A, \Gamma, s, numAppTerms) =$
  $(B, (s_1, s_2, \ldots, s_m)) \leftarrow chooseHeadVar(A, \Gamma, s, numAppTerms)$
  $headVar \leftarrow genVarTerm(B, \Gamma)$
  FOR $i = 1$ TO m
    $term_i \leftarrow genTerm(B_i, \Gamma, s_i)$
  RETURN $(\ldots((headVar\ term_1)\ldots term_{m-1})term_m)$

$chooseHeadVar(A, \Gamma, s, numAppTerms) =$
  $randNum \leftarrow random\ numAppTerms$
  $numTerms \leftarrow 0$
  FOR each $B = B_1 \to \ldots \to B_m \to A \in validHeadVarTypeSet(A, \Gamma)$
    $numTerms \leftarrow numTerms + countHeadVarArgTerms(B, \Gamma, s)$
    IF $randNum < numTerms$ THEN
      RETURN $(B, chooseArgSize(B, \Gamma, s, (countHeadVarArgTerms(B, \Gamma, s))/typeCnt(\Gamma, B)))$

$choooseArgSize(B, \Gamma, S, numArgTerms) =$
  $randNum \leftarrow random\ numArgTerms$
  $numTerms \leftarrow 0$
  FOR each $(s_1, \ldots, s_m) \in ndk(s-1-m, m)$
    $numTerms \leftarrow numTerms + \prod_{i=1}^{m} \mathrm{countTerm}(B_i, \Gamma, s_i)$
    IF $randNum < numTerms$ THEN
      RETURN $(s_1, \ldots, s_m)$

Figure 8: The algorithm to generate a term of a given type and size.

6.1.2 Different Type Systems

The current algorithm revolves around the simple type system. For the goals of using this algorithm as a tool for generating test cases, it is not so useful to be restricted to the relatively weak simple type system where most interesting questions regarding this type system have already been addressed. Instead, it will be more interesting to adapt this approach to more expressive type systems such as intersection types so that we can use this tool to generate test cases to perform average-case analysis of algorithms involving intersection types.

6.1.3 All Typable Terms

In [6], we showed through statistical sampling experiments that the percentage of well typed terms is extremely low. For the purpose of generating test cases for performing average-case analysis on type inference algorithms, the untyped generator is not ideal due to the low percentage of well typed terms. In this context, we are not interested in a specific type, but instead, we would like to have just well-typed terms. We can certainly start with an enumeration of types and generate terms of these enumerated types in some order. Another possible approach is to investigate syntactic restrictions of well-typed terms and come up with some ad-hoc heuristics to increase the percentage of well-typed terms we generate.

6.1.4 Different Notion of Size

In this report, the size is defined as the number of nodes in the parse tree. This notion is problematic for two different reasons. First, this definition is very restrictive. For some types, there does not exist terms of a given size even though the type itself is inhabited for other sizes. For example, given the type $(a \to a) \to a \to a$, the size of the terms that inhabit the type is $2, 3, 5, 7, 9, 11, \ldots$. To solve this problem, we can relax the restriction of an exact size to be around the given size. In addition, the current size definition results in an explosion of computational complexity that can not be easily avoided. If we want to find the number of terms with size 100 that is an application of 10 arguments to a term variable, then we have to consider all the ways of distributing the remaining size between the required arguments. This is the *n composition k part* problem[4], and the answer in this case is $\binom{89-1}{89-10}$ different ways. Moreover, it not so clear that this notion of size is the most useful and obvious definition. In [7], Zaionc introduced a complexity measure π for closed terms in long β normal form as defined below.

$$\pi(T) = \begin{cases} 1 & \text{if } A = \lambda x_1 \ldots x_n.x_i \\ max_{j=1\ldots k}(\pi(\lambda x_1 \ldots x_n.x_i A_k)) + 1 & \text{if } A = \lambda x_1 \ldots x_n.x_i A_1 \ldots A_k \end{cases}$$

This notion of complexity counts the work done in counting and generating more closely. Thus, it will be worthwhile to investigate the trade-offs of some other notions of size.

6.2 A More Generic Tool

The work done so far is only concerned with the pure lambda calculus, we can enrich the grammar by including additional program constructs so that the generated programs will resemble more realistic programs. In addition, it will be nice to develop a generic tool that can take a grammar as input and output a random generator based on the grammar.

6.3 Engineering the algorithm

The current implementation of the algorithm is implemented as a proof of concept. It is impractical in that it uses the brute force approach and is limited in the size of the terms that can be generated.

There are some easy optimization techniques we can add such as the engineering techniques we used in [6]. Additionally, we need to improve the user-interface to produce a user-friendly tool.

7 Acknowledgment

I would like to thank Assaf Kfoury for providing valuable guidance and feeback on this project.

References

[1] J.Roger Hindley. *Basic Simple Type Theory.* Cambridge University Press, United Kingdom, 1997.

[2] H.P.Barnendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984.

[3] H. G. Mairson. Generating words in a context free language uniformly at random. *Information Processing Letters*, 49:95–99, 1994.

[4] Albert Nijenhuis and Herbert Wilf. *Combinatorial Algorithms.* Academic Press, United States, 1978.

[5] Masako Takahashi, Yohji Akama, and Sachio Hirokawa. Normal proofs and their grammar. *Inf. Comput.*, 125(2):144–153, 1996.

[6] Jue Wang. Generating random lambda calculus terms. Unpublished manuscript.

[7] Marek Zaionc. Fixpoint technique for counting terms in typed lambda calculus. Technical Report 95-20, Department of Computer Science at State University of New York at Buffalo, 14, 1995.