A Hiking Trip Through the Orders of Magnitude: Deriving Efficient Generators for Closed Simply-Typed Lambda Terms and Normal Forms

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cse.unt.edu

Abstract. Contrary to several other families of lambda terms, no closed formula or generating function is known and none of the sophisticated techniques devised in analytic combinatorics can help with counting or generating the set of *simply-typed closed lambda terms* of a given size.
Moreover, their asymptotic scarcity among the set of closed lambda terms makes counting them via brute force generation and type inference quickly intractable, with previous work showing counts for them only up to size 10.
By taking advantage of the synergy between logic variables, unification with occurs check and efficient backtracking in today's Prolog systems, we climb 4 orders of magnitude above previously known values by deriving progressively faster Horn Clause programs that generate and/or count the set of closed simply-typed lambda terms of sizes up to 14. A similar count for *closed simply-typed normal forms* is also derived up to size 14.
Keywords: *logic programming transformations, type inference, combinatorics of lambda terms, simply-typed lambda calculus, simply-typed normal forms.*

1 Introduction

Generation of lambda terms [1] has practical applications to testing compilers that rely on lambda calculus as an intermediate language, as well as in generation of random tests for user-level programs and data types. At the same time, several instances of lambda calculus are of significant theoretical interest given their correspondence with logic and proofs.

Simply-typed lambda terms [2, 3] enjoy a number of nice properties, among which strong normalization (termination for all evaluation-orders), a cartesian closed category mapping and a set-theoretical semantics. More importantly, via the Curry-Howard correspondence lambda terms that are *inhabitants* of simple types can be seen as proofs for tautologies in *minimal logic* which, in turn, correspond to the types. Extended with a fix-point operator, simply-typed lambda

terms can be used as the intermediate language for compiling Turing-complete functional languages. Random generation of simply-typed lambda terms can also help with automation of debugging compilers for functional programming languages [4].

Recent work on the combinatorics of lambda terms [5–8], relying on recursion equations, generating functions and techniques from analytic combinatorics [9] has provided counts for several families of lambda terms and clarified important properties like their asymptotic density. With the techniques provided by generating functions [9], it was possible to separate the *counting* of the terms of a given size for several families of lambda terms from their more computation intensive *generation*, resulting in several additions (e.g., A220894, A224345, A114851) to The On-Line Encyclopedia of Integer Sequences, [10].

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

As a fortunate synergy, Prolog's sound unification of logic variables, backtracking and and definite clause grammars has been shown to provide compact combinatorial generation algorithms for various families of lambda terms [11–14].

For the case of simply-typed lambda terms, we have pushed (in the unpublished draft [15]) the counts in sequence A220471 of [10] to cover sizes `11` and `12`, each requiring each about one magnitude of extra computation effort, simply by writing the generators in Prolog. In this paper we focus on going two more magnitudes higher, while also integrating the results described in [15]. Using similar techniques, we achieve the same, for the special case of simply-typed normal forms.

The paper is organized as follows. Section 2 describes our representation of lambda terms and derives a generator for closed lambda terms. Section 3 defines generators for well-formed type formulas. Section 4 introduces a type inference algorithms and then derives, step by step, efficient generators for simply-typed lambda terms and simple types inhabited by terms of a given size. Section 5 defines generators for closed lambda terms in normal form and then replicates the derivation of efficient generator for simply-typed closed normal forms. Section 6 aggregates our experimental performance data and hints to future improvements. Section 7 discusses related work and section 8 concludes the paper.

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

2 Deriving a generator for lambda terms

Lambda terms can be seen as Motzkin trees [16], also called unary-binary trees, labeled with lambda binders at their unary nodes and corresponding variables at the leaves. We will thus derive a generator for them from a generator for Motzkin trees.

2.1 A canonical representation with logic variables

We can represent lambda terms [1] in Prolog using the constructors `a/2` for applications, `l/2` for lambda abstractions and `v/1` for variable occurrences. Variables bound by the lambdas and their occurrences are represented as *logic variables*. As an example, the lambda term $\lambda a.(\lambda b.(a\ (b\ b))\ \lambda c.(a\ (c\ c)))$ will be represented as `l(A,a(l(B,a(v(A),a(v(B),v(B)))),l(C,a(v(A),a(v(C),v(C))))))`. As variables share a unique scope (the clause containing them), this representation assumes that *distinct variables are used for distinct scopes induced by the lambda binders* in terms occurring in a given Prolog clause.

Lambda terms might contain *free variables* not associated to any binders. Such terms a called *open*. A *closed* term is such that each variable occurrence is associated to a binder.

2.2 Generating Motzkin trees

Motzkin-trees (also called binary-unary trees) have internal nodes of arities 1 or 2. Thus they can be seen as a skeleton of lambda terms that ignores binders and variables and their leaves.

The predicate `motzkin/2` generates Motzkin trees with `S` internal and leaf nodes.

```
motzkin(S,X):-motzkin(X,S,0).

motzkin(v)-->[].
motzkin(l(X))-->down,motzkin(X).
motzkin(a(X,Y))-->down,motzkin(X),motzkin(Y).

down(s(X),X).
```

Motzkin-trees are counted by the sequence `A001006` in [10]. If we replace the first clause of `motzkin/2` with `motzkinTree(u)-->[]`, we obtain binary-unary trees with `L` internal nodes, counted by the entry `A006318` (Large Schröder Numbers) of [10].

Note the use of the predicate `down/2`, that assumes natural numbers in *unary notation*, with n `s/1` symbols wrapped around `0` to denote $n \in \mathbb{N}$. As our combinatorial generation algorithms will usually be tractable for values of n below `15`, the use of unary notation is comparable (an often slightly faster) than the call to arithmetic built-ins. Note also that this leads, after the DCG translation, to "pure" Prolog programs made exclusively of Horn Clauses.

To more conveniently call these generators with the usual natural numbers we define the converter `n2s` as follows.

```
n2s(0,0).
n2s(N,s(X)):-N>0,N1 is N-1,n2s(N1,X).
```

Example 1 *Motzkin tress with 2 internal nodes.*

```
?- n2s(1,S),motzkin(S,T).
S = s(0), T = l(v) ;
S = s(0), T = a(v, v) .
```

2.3 Generating closed lambda terms

We derive a generator for closed lambda terms by adding logic variables as labels to their binder and variable nodes, while ensuring that the terms are closed, i.e., that the function mapping variables to their binders is total.

The predicate `lambda/2` builds a list of logic variables as it generates binders. When generating a leaf variable, it picks "nondeterministically" one of the binders among the list of binders available, `Vs`. As in the case of Motzkin trees, the predicate `down/2` controls the number of internal nodes.

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

lambda(v(V),Vs)-->{member(V,Vs)}.
lambda(l(V,X),Vs)-->down,lambda(X,[V|Vs]).
lambda(a(X,Y),Vs)-->down,lambda(X,Vs),lambda(Y,Vs).
```

The sequence `A220471` in [10] contains counts for lambda terms of increasing sizes, with *size defined as the number of internal nodes.*

Example 2 *Closed lambda terms with 2 internal nodes.*

```
?- lambda(s(s(0)),Term).
Term = l(A, l(B, v(B))) ;
Term = l(A, l(B, v(A))) ;
Term = l(A, a(v(A), v(A))) .
```

3 A visit to the other side: the language of types

As a result of the Curry-Howard correspondence, the language of types is isomorphic with that of *minimal logic*, with binary trees having variables at leaf positions and the implication operator ("`->`") at internal nodes. We will rely on the right associativity of this operator in Prolog, that matches the standard notation in type theory.

The predicate `type_skel/3` generates all binary trees with given number of internal nodes and labels their leaves with unique logic variables. It also collects the variables to a list returned as its third argument.

```
type_skel(S,T,Vs):-type_skel(T,Vs,[],S,0).

type_skel(V,[V|Vs],Vs)-->[].
type_skel((X->Y),Vs1,Vs3)-->down,
  type_skel(X,Vs1,Vs2),
  type_skel(Y,Vs2,Vs3).
```

Type skeletons are counted by the Catalan numbers (sequence `A000108` in [10]).

Example 3 *All type skeletons for N=3.*

```
?- type_skel(s(s(s(0))),T,_).
T =  (A->B->C->D) ;
T =  (A-> (B->C)->D) ;
T =  ((A->B)->C->D) ;
T =  ((A->B->C)->D) ;
T =  (((A->B)->C)->D) .
```

The next step toward generating the set of all type formulas is observing that logic variables define equivalence classes that can be used to generate partitions of the set of variables, simply by selectively unifying them.

The predicate `mpart_of/2` takes a list of distinct logic variables and generates partitions-as-equivalence-relations by unifying them "nondeterministically". It also collects the unique variables, defining the equivalence classes as a list given by its second argument.

```
mpart_of([],[]).
mpart_of([U|Xs],[U|Us]):-
  mcomplement_of(U,Xs,Rs),
  mpart_of(Rs,Us).
```

To implement a set-partition generator, we will split a set repeatedly in subset+complement pairs with help from the predicate `mcomplement_of/2`.

```
mcomplement_of(_,[],[]).
mcomplement_of(U,[X|Xs],NewZs):-
  mcomplement_of(U,Xs,Zs),
  mplace_element(U,X,Zs,NewZs).

mplace_element(U,U,Zs,Zs).
mplace_element(_,X,Zs,[X|Zs]).
```

To generate set partitions of a set of variables of a given size, we build a list of fresh variables with the equivalent of Prolog's `length` predicate working in unary notation `len/2`.

```
partitions(S,Ps):-len(Ps,S),mpart_of(Ps,_).

len([],0).
len([_|Vs],s(L)):-len(Vs,L).
```

The count of the resulting set-partitions (Bell numbers) corresponds to the entry `A000110` in [10].

Example 4 *Set partitions of size 3 expressed as variable equalities.*

```
?- partitions(s(s(s(0))),P).
P = [A, A, A] ;
P = [A, B, A] ;
P = [A, A, B] ;
P = [A, B, B] ;
P = [A, B, C].
```

We can then define the language of formulas in minimal logic, among which tautologies will correspond to simple types, as being generated by the predicate `maybe_type/2`.

```
maybe_type(L,T,Us):-type_skel(L,T,Vs),mpart_of(Vs,Us).
```

Example 5 *Well-formed formulas of minimal logic (possibly types) of size 2.*

```
?- maybe_type(s(s(0)),T,_).
T =  (A->A->A) ;
T =  (A->B->A) ;
T =  (A->A->B) ;
T =  (A->B->B) ;
T =  (A->B->C) ;
T =  ((A->A)->A) ;
T =  ((A->B)->A) ;
T =  ((A->A)->B) ;
T =  ((A->B)->B) ;
T =  ((A->B)->C) .
```

The sequence `2,10,75,728,8526,115764,1776060,30240210` counting these formulas corresponds to the the product of Catalan and Bell numbers.

4 Merging the two worlds: generating simply-typable lambda terms

One can observe that per-size counts of both the sets of lambda terms and their potential types are very fast growing. There's an important difference, though, between computing the type of a given lambda term (if it exists) and computing an inhabitant of a type (if it exists). The first operation, called *type inference* is an efficient operation (linear in practice) while the second operation, called *the inhabitation problem* is `P-space` complete [17].

This brings us to design a type inference algorithm that takes advantage of operations on logic variables.

4.1 A type inference algorithm

While in a functional language inferring types requires implementing unification with occurs-check, as shown for instance in [5], this operation is available in

Prolog as a built-in, predicate, optimized, for instance, in SWI-Prolog [18], to proceed incrementally, only checking that no new cycles are introduced during the unification step as such.

The predicate `infer_type/3` works by using logic variables as dictionaries associating terms to their types. Each logic variable is then bound to a term of the form `X:T` where `X` will be a component of a fresh copy of the term and `T` will be its type. Note that we create this new term as the original term's variables end up loaded with chunks of the partial types created during the type inference process.

As logic variable bindings propagate between binders and occurrences, this ensures that types are consistently inferred.

```
infer_type((v(XT)),v(X),T):-unify_with_occurs_check(XT,X:T).
infer_type(l((X:TX),A),l(X,NewA),(TX->TA)):-infer_type(A,NewA,TA).
infer_type(a(A,B),a(X,Y),TY):-infer_type(A,X,(TX->TY)),infer_type(B,Y,TX).
```

Example 6 *illustrates typability of the term corresponding to the* `S` *combinator* $\lambda x_0.\lambda x_1.\lambda x_2.((x_0\ x_2)\ (x_1\ x_2))$
and untypabilty of the term corresponding to the `Y` *combinator* $\lambda x_0.(\lambda x_1.(x_0\ (x_1\ x_1))\ \ \lambda x_2.(x_0\ (x_2\ x_2)))$.

```
?- infer_type(l(A,l(B,l(C,a(a(v(A),v(C)),a(v(B),v(C)))))),X,T),
   portray_clause((T:-X)),fail.
(A->B->C)-> (A->B)->A->C :-
   l(D,l(F,l(E, a(a(v(D), v(E)), a(v(F), v(E)))))).

?- infer_type(
     l(A,a(l(B,a(v(A),a(v(B),v(B)))),l(C,a(v(A),a(v(C),v(C)))))), X, T).
false.
```

By combining generation of lambda terms with type inference we have our first cut to an already surprisingly fast generator for simply-typable lambda terms, able to generate in a few hours counts for sizes `11` and `12` for sequence `A220471` in [10].

```
lamb_with_type(S,X,T):-lambda(S,XT),infer_type(XT,X,T).
```

Example 7 *Lambda terms of size up to 3 and their types.*

```
?- lamb_with_type(s(s(s(0))),Term,Type).
Term = l(A, l(B, l(C, v(C)))), Type =  (D->E->F->F) ;
Term = l(A, l(B, l(C, v(B)))), Type =  (D->E->F->E) ;
Term = l(A, l(B, l(C, v(A)))), Type =  (D->E->F->D) ;
Term = l(A, l(B, a(v(B), v(A)))), Type =  (C-> (C->D)->D) ;
Term = l(A, l(B, a(v(A), v(B)))), Type =  ((C->D)->C->D) ;
Term = l(A, a(v(A), l(B, v(B)))), Type =  (((C->C)->D)->D) ;
Term = l(A, a(l(B, v(B)), v(A))), Type =  (C->C) ;
Term = l(A, a(l(B, v(A)), v(A))), Type =  (C->C) ;
Term = a(l(A, v(A)), l(B, v(B))), Type =  (C->C).
```

Note that, for instance, when one wants to select only terms having a given type, this is quite inefficient. Next, we will show how to combine size-bound term generation, testing for closed terms and type inference into a single predicate. This will enable more efficient querying for terms inhabiting a given type, as one would expect from Prolog's multi-directional execution model, and more importantly for our purposes, to climb two orders of magnitude higher for counting simply-typed terms of size `13` and `14`.

4.2 Combining term generation and type inference

We need two changes to `infer_type` to turn it into an efficient generator for simply-typed lambda terms. First, we need to add an argument to control the size of the terms and ensure termination, by calling `down/2` for internal nodes. Second, we need to generate the mapping between binders and variables. We ensure this my borrowing the `member/2`-based mechanism used in the predicate `lambda/4` generating closed lambda terms in subsection 2.3.

The predicate `typed_lambda/3` does just that, with helper from DCG-expanded `typed_lambda/5`.

```
typed_lambda(S,X,T):-typed_lambda(_XT,X,T,[],S,0).

typed_lambda(v(V:T),v(V),T,Vs)--> {
   member(V:T0,Vs),
   unify_with_occurs_check(T0,T)
  }.
typed_lambda(l(X:TX,A),l(X,NewA),(TX->TY),Vs)-->down,
  typed_lambda(A,NewA,TY,[X:TX|Vs]).
typed_lambda(a(A,B),a(NewA,NewB),TY,Vs)-->down,
  typed_lambda(A,NewA,(TX->TY),Vs),
  typed_lambda(B,NewB,TX,Vs).
```

Like `lambda/4`, the predicate `typed_lambda/5` relies on Prolog's DCG notation to thread together the steps controlled by the predicate `down`. Note also the nondeterministic use of the built-in `member/2` that enumerates values for `variable:type` pairs ranging over the list of available pairs `Vs`, as well as the use of `unify_with_occurs_check` to ensure that unification of candidate types does not create cycles.

We will discuss exact performance data later, but let's note here that this operation brings down by an order of magnitude the computational effort to generate simply-typed terms. As expected, the number of solutions is computed as the sequence `A220471` in [10]. Interestingly, by *interleaving* generation of closed terms and type inference in the predicate `typed_lambda`, the time to generate all the well-typed terms is actually shorter than the time to generate all closed terms of the same size, e.g., 17.123 vs. 31.442 seconds for size 10 with SWI-Prolog. As, via the Curry-Howard isomorphism, closed simply typed terms correspond to proofs of tautologies in minimal logic, co-generation of terms and types corresponds to co-generation of tautologies and their proofs for proofs of given length.

Example 8 *A term of size 15 and its type.*

```
l(A,l(B,l(C,l(D,l(E,l(F,l(G,l(H,l(I,l(J,l(K,
           a(v(I),l(L,a(a(v(E),v(J)),v(J))))))))))))))
M->N->O->P-> (Q->Q->R)->S->T->U-> ((V->R)->W)->Q->X->W
```

4.3 One more trim: generating inhabited types

Let's first observe that the actual lambda term does not need to be built, provided that we mimic exactly the type inference operations that one would need to perform to ensure it is simply-typed. It is thus safe to remove the first argument of `typed_lambda/5` as well as the building of the fresh copy performed in the second argument. To further simplify the code, we can also make the DCG-processing of the size computations explicit, in the last two arguments.

This gives the predicate `inhabited_type/4` and then `inhabited_type/2`, that generates *all types having inhabitants of a given size*, but omits the inhabitants as such.

```
inhabited_type(X,Vs,N,N):-
  member(V,Vs),
  unify_with_occurs_check(X,V).
inhabited_type((X->Xs),Vs,s(N1),N2):-
  inhabited_type(Xs,[X|Vs],N1,N2).
inhabited_type(Xs,Vs,s(N1),N3):-
  inhabited_type((X->Xs),Vs,N1,N2),
  inhabited_type(X,Vs,N2,N3).
```

Clearly the multiset of generated types has the same count as the set of their inhabitants and this brings us a nice additional 1.5x speed-up.

```
inhabited_type(S,T):-inhabited_type(T,[],S,0).
```

One more step makes reaching counts for sizes `13` and `14` achievable: using a faster Prolog, with a similar `unify_with_occurs_check` built-in, like YAP [19], with the last value computed on a MacAir in less than a day.

Example 9 *The sequence* **A220471** *completed up to N=14*

```
first 10: 1,2,9,40,238,1564,11807,98529,904318,9006364

11:      96,709,332
12:   1,110,858,977

13:  13,581,942,434
14: 175,844,515,544
```

5 Doing it once more: generating closed simply-typed normal forms

We will devise similar methods for an important subclass of simply-typed lambda terms.

5.1 Generating normal forms

Normal forms are lambda terms that cannot be further reduced. A normal form should not be an application with a lambda as its left branch and, recursively, its subterms should also be normal forms. The predicate `normal_form/2` uses `normal_form/4` to define them inductively and generates all normal forms with `S` internal nodes.

```
normal_form(S,T):-normal_form(T,[],S,0).

normal_form(v(X),Vs)-->{member(X,Vs)}.
normal_form(l(X,A),Vs)-->down,normal_form(A,[X|Vs]).
normal_form(a(v(X),B),Vs)-->down,normal_form(v(X),Vs),normal_form(B,Vs).
normal_form(a(a(X,Y),B),Vs)-->down,normal_form(a(X,Y),Vs),normal_form(B,Vs).
```

Example 10 *illustrates closed normal forms with* `2` *internal nodes.*

```
?- normal_form(s(s(0)),NF).
NF = l(A, l(B, v(B))) ;
NF = l(A, l(B, v(A))) ;
NF = l(A, a(v(A), v(A))) .
```

The number of solutions of our generator replicates entry `A224345` in [10] that counts closed normal forms of various sizes.

The predicate `nf_with_type` applies the type inference algorithm to the generated normal forms of size `S`.

```
nf_with_type(S,X,T):-normal_form(S,XT),infer_type(XT,X,T).
```

5.2 Merging in type inference

Like in the case of the set of simply-typed lambda terms, we can define the more efficient combined generator and type inferrer predicate `typed_nf/2`.

```
typed_nf(S,X,T):-typed_nf(_XT,X,T,[],S,0).
```

It works by calling the DCG-expended `typed_nf/4` predicate, with the last two arguments enforcing the size constraints.

```
typed_nf(v(V:T),v(V),T,Vs)--> {
   member(V:T0,Vs),
   unify_with_occurs_check(T0,T)
  }.
typed_nf(l(X:TX,A),l(X,NewA),(TX->TY),Vs)-->down,
  typed_nf(A,NewA,TY,[X:TX|Vs]).
typed_nf(a(v(A),B),a(NewA,NewB),TY,Vs)-->down,
  typed_nf(v(A),NewA,(TX->TY),Vs),
  typed_nf(B,NewB,TX,Vs).
typed_nf(a(a(A1,A2),B),a(NewA,NewB),TY,Vs)-->down,
  typed_nf(a(A1,A2),NewA,(TX->TY),Vs),
  typed_nf(B,NewB,TX,Vs).
```

Example 11 *Simply-typed normal forms up to size 3.*

```
?- typed_nf(s(s(s(0))),Term,Type).
Term = l(A, l(B, l(C, v(C)))),
Type =  (D->E->F->F) ;
...
Term = l(A, a(v(A), l(B, v(B)))),
Type =  (((C->C)->D)->D) .
```

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

Example 12 *Counts for closed simply-typed normal forms up to* `N=14`.

```
first 10: 1,2,6,23,108,618,4092,30413,252590,2297954

11:     22,640,259
12:    240,084,189
13:  2,721,455,329
14: 32,783,910,297
```

6 Experimental data and discussion

Size	closed λ-terms	gen, then infer	gen + infer	inhabitants	typed normal form
1	15	19	16	9	19
2	44	59	50	28	47
3	166	261	188	113	127
4	810	1,517	864	553	429
5	4,905	10,930	4,652	3,112	1,814
6	35,372	92,661	28,878	19,955	9,247
7	294,697	895,154	202,526	143,431	55,219
8	2,776,174	9,647,495	1,586,880	1,146,116	377,745
9	29,103,799	114,273,833	13,722,618	10,073,400	2,896,982
10	335,379,436	1,471,373,474	129,817,948	96,626,916	24,556,921

Fig. 1. Number of logical inferences used by our generators, as counted by SWI-Prolog

Figure **1** gives the number of logical inferences as counted by SWI-Prolog. This is a good measure of computational effort except for counting operations like `unify_with_occurs_check` as a single step, while it's actual complexity depends on the size of the terms involved. Therefore, figure **2** gives actual timings for the same operations, starting at `N=5` where they start to be meaningful.

The "`closed λ-terms`" column gives logical inferences and timing for generating all closed lambda terms of size given in column 1. The column "`gen,`

`then infer`" gives data for the algorithm that first generates lambda terms and then infers their types. The column "`gen + infer`" gives performance data for the significantly faster for the algorithm that merges generation and type inference in the same predicate. The column "`inhabitants`" gives data for the case when actual inhabitants are omitted in the merged generation and type inference process. The column "`typed normal form`" shows result for the fast, merged generation and type inference for terms in normal form.

Size	closed λ-terms	gen, then infer	gen + infer	inhabitants	typed normal form
5	0.001	0.001	0.001	0.000	0.001
6	0.005	0.011	0.004	0.002	0.004
7	0.028	0.114	0.029	0.018	0.011
8	0.257	1.253	0.242	0.149	0.050
9	2.763	15.256	2.080	1.298	0.379
10	32.239	199.188	19.888	12.664	3.329

Fig. 2. Timings (in seconds) for our generators up to size 10 on a 2015 MacBook

As moving from a size to the next typically adds one order of magnitude of computational effort, computing values for `N=15` and `N=16` is reachable with our best algorithms for both simply typed terms and their normal form subset.

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

While it only offers a constant factor speed-up, parallel execution is a more promising possibility. However, given the small granularity of the generation and type inference process, the most useful parallel execution mechanism would simply split the task of combined generation and inference process into a number of disjoint sets, corresponding to the number of available processors. A way to do this, is by using unranking functions (bijections originating in $\mathbb{N}$) to the sets of combinatorial objects involved, and then, for k processors, assign work on successive numbers belonging to the same equivalence class modulo k.

We have not seen any obvious way to improve these results using constraint programing systems, partly because the key "inner loop" computation is unification with occurs check with computations ranging over Prolog terms rather than being objects of a constraint domain. On the other hand, for a given size, exploring grounding to propositional formulas or answer-set programming seems worth exploring as a way to take advantage of today's fast SAT-solvers.

7 Related work

The classic reference for lambda calculus is [1]. Various instances of typed lambda calculi are overviewed in [3].

The combinatorics and asymptotic behavior of various classes of lambda terms are extensively studied in [5, 8]. Distribution and density properties of random lambda terms are described in [6].

Generation of random simply-typed lambda terms and its applications to generating functional programs from type definitions is covered in [20].

Several concepts of size have been used in the literature, partly to facilitate convergence of formal series in analytic combinatorics [21, 22].

Asymptotic density properties of simple types (corresponding to tautologies in minimal logic) have been studied in [23] with the surprising result that most classical tautologies are also intuitionistic ones.

In [4] a "type-directed" mechanism for the generation of random terms is introduced, resulting in more realistic (while not uniformly random) terms, used successfully in discovering some bugs in the Glasgow Haskell Compiler (GHC).

Generators for closed simply-typed lambda terms, as well as their normal forms, expressed as functional programming algorithms, are given in [5], derived from combinatorial recurrences. However, they are significantly more complex than the ones described here in Prolog, and limited to terms up to size 10.

In the unpublished draft [15] we have collected several lambda term generation algorithms written in Prolog and covering mostly de Bruijn terms and a compressed de Bruijn representation. Among them, linear, affine linear terms as well as terms of bounded unary height and in binary lambda calculus encoding. In [15] type inference algorithms are also given for SK and X-combinator expressions. A similar but about an order of magnitude slower program for type inference using de Bruijn notation is also given in [15], without however describing the step-by-step derivation steps leading to it, as done in this paper.

8 Conclusion

We have derived several logic programs that have helped solve the fairly hard combinatorial counting and generation problem for simply-typed lambda terms 4 orders of magnitude farther than previously published results.

This has put at test two simple but effective program transformation techniques naturally available in logic programming languages: 1) interleaving generators and testers by integrating them in the same predicate and 2) dropping arguments used in generators when used simply as counters of solutions, when their role can be kept implicit in the recursive structure of the program. Both have turned out to be effective for speeding up computations without changing the semantics of their intended application. We have also managed (after a simple DCG translation) to work within in the minimalist framework of Horn Clauses with sound unification, showing that non-trivial combinatorics problems can be handled without any of Prolog's impure features.

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

Acknowledgement

This research has been supported by NSF grant 1423324.

We thank the participants of the 9th Workshop Computational Logic and Applications (https://cla.tcs.uj.edu.pl/) for enlightening discussions and comments and for sharing various techniques clarifying the challenges one faces when having a fresh look at the emerging, interdisciplinary field of the combinatorics of lambda terms and their applications.

References

1. Barendregt, H.P.: The Lambda Calculus Its Syntax and Semantics. Revised edn. Volume 103. North Holland (1984)
2. Hindley, J.R., Seldin, J.P.: Lambda-calculus and combinators: an introduction. Volume 13. Cambridge University Press Cambridge (2008)
3. Barendregt, H.P.: Lambda calculi with types. In: Handbook of Logic in Computer Science. Volume 2. Oxford University Press (1991)
4. Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the 6th International Workshop on Automation of Software Test. AST'11, New York, NY, USA, ACM (2011) 91–97
5. Grygiel, K., Lescanne, P.: Counting and generating lambda terms. J. Funct. Program. **23**(5) (2013) 594–628
6. David, R., Raffalli, C., Theyssier, G., Grygiel, K., Kozik, J., Zaionc, M.: Some properties of random lambda terms. Logical Methods in Computer Science **9**(1) (2009)
7. Bodini, O., Gardy, D., Gittenberger, B.: Lambda-terms of bounded unary height. In: ANALCO, SIAM (2011) 23–32
8. David, R., Grygiel, K., Kozik, J., Raffalli, C., Theyssier, G., Zaionc, M.: Asymptotically almost all λ-terms are strongly normalizing. Preprint: arXiv: math. LO/0903.5505 v3 (2010)
9. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. 1 edn. Cambridge University Press, New York, NY, USA (2009)
10. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences. (2014) Published electronically at https://oeis.org/.
11. Tarau, P.: On Logic Programming Representations of Lambda Terms: de Bruijn Indices, Compression, Type Inference, Combinatorial Generation, Normalization. In Pontelli, E., Son, T.C., eds.: Proceedings of the Seventeenth International Symposium on Practical Aspects of Declarative Languages PADL'15, Portland, Oregon, USA, Springer, LNCS 8131 (June 2015) 115–131

12. Tarau, P.: Ranking/Unranking of Lambda Terms with Compressed de Bruijn Indices. In Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V., eds.: Proceedings of the 8th Conference on Intelligent Computer Mathematics, Washington, D.C., USA, Springer, LNAI 9150 (July 2015) 118–133
13. Tarau, P.: On a Uniform Representation of Combinators, Arithmetic, Lambda Terms and Types. In Albert, E., ed.: PPDP'15: Proceedings of the 17th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, New York, NY, USA, ACM (July 2015) 244–255
14. Tarau, P.: On Type-directed Generation of Lambda Terms. In De Vos, M., Eiter, T., Lierler, Y., Toni, F., eds.: 31st International Conference on Logic Programming (ICLP 2015), Technical Communications, Cork, Ireland, CEUR (September 2015) available online at http://ceur-ws.org/Vol-1433/.
15. Tarau, P.: A logic programming playground for lambda terms, combinators, types and tree-based arithmetic computations. CoRR **abs/1507.06944** (2015)
16. Stanley, R.P.: Enumerative Combinatorics. Wadsworth Publ. Co., Belmont, CA, USA (1986)
17. Statman, R.: Intuitionistic propositional logic is polynomial-space complete. Theor. Comput. Sci. **9** (1979) 67–72
18. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12** (1 2012) 67–96
19. Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. Theory and Practice of Logic Programming **12** (1 2012) 5–34
20. Fetscher, B., Claessen, K., Palka, M.H., Hughes, J., Findler, R.B.: Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In: Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. (2015) 383–405
21. Grygiel, K., Lescanne, P.: Counting and generating terms in the binary lambda calculus (extended version). CoRR **abs/1511.05334** (2015)
22. Bendkowski, M., Grygiel, K., Lescanne, P., Zaionc, M.: A natural counting of lambda terms. In Freivalds, R.M., Engels, G., Catania, B., eds.: SOFSEM 2016: Theory and Practice of Computer Science - 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings. Volume 9587 of Lecture Notes in Computer Science., Springer (2016) 183–194
23. Genitrini, A., Kozik, J., Zaionc, M.: Intuitionistic vs. classical tautologies, quantitative comparison. In Miculan, M., Scagnetto, I., Honsell, F., eds.: Types for Proofs and Programs, International Conference, TYPES 2007, Cividale del Friuli, Italy, May 2-5, 2007, Revised Selected Papers. Volume 4941 of Lecture Notes in Computer Science., Springer (2007) 100–109