Fundamenta Informaticae XX (2023) 1–31
DOI 10.3233/FI-2016-0000
IOS Press

Deriving Efficient Sequential and Parallel Generators for Closed Simply-Typed Lambda Terms and Normal Forms

Paul Tarau
Department of Computer Science and Engineering
University of North Texas
paul.tarau@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 currently 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 published 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 counts by deriving progressively faster sequential Prolog programs that generate and/or count the set of closed simply-typed lambda terms of sizes up to 14. Similar counts for *closed simply-typed normal forms* are also derived up to size 14.

Finally, we devise several parallel execution algorithms, based on generating code to be uniformly distributed among the available cores, that push the counts for simply typed terms up to size 15 and simply typed normal forms up to size 16. As a remarkable feature, our parallel algorithms are linearly scalable with the number of available cores.

Keywords: *logic programming transformations, type inference, simply-typed lambda terms and normal forms, sequential and parallel combinatorial generation algorithms, Prolog multi-threading*

1. Introduction

This paper is an extended and updated version of [1], with new material centered around parallel algorithms bringing speed-ups linear in the number of processors used.

Generation of lambda terms [2] 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. Among them, simply-typed lambda terms [3, 4] 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 [5, 6], lambda terms that are *inhabitants* of simple types can be seen as proofs for tautologies in the *implicational fragment of intuitionistic propositional logic* which, in turn, correspond to the types. Generation of large simply-typed lambda terms can also help with automation of testing and debugging compilers for functional programming languages [7].

Recent work on the combinatorics of lambda terms [8, 9, 10, 11], relying on recursion equations, generating functions and techniques from analytic combinatorics [12] has provided counts for several families of lambda terms and clarified important properties like their asymptotic density. With the techniques provided by generating functions [12], 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, [13].

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 definite clause grammars have been shown to provide compact combinatorial generation algorithms for various families of lambda terms [14, 15, 16, 17].

For the case of simply-typed lambda terms, we have pushed (in the unpublished draft [18]) the counts in sequence A220471 of [13, 8] to cover sizes `11` and `12`, each requiring 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 [18], and one more orders of magnitude using parallel algorithms. Using similar techniques, we achieve the same, for the special case of simply-typed normal forms with a two order of magnitude gain via parallelization.

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 algorithm 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 an efficient generator for simply-typed closed normal forms. Section 6 aggregates our experimental performance data for sequential execution. Section 7 describes several generic algorithms for parallelization of the generation of simply typed lambda terms and normal forms. Section 8 evaluates their performance and scalability improvements. Section 9 discusses possible extensions and future improvements. Section 10 overviews related work and section 11 concludes the paper.

The paper is structured as a literate Prolog program. The code has been tested with SWI-Prolog 7.21.40 (including multi-thread execution) and YAP 6.3.4 (sequential execution only). It is also available as a separate file at `http://www.cse.unt.edu/~tarau/research/2018/parlgen.pro`.

2. Deriving a generator for lambda terms

Lambda terms can be seen as Motzkin trees [19], 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 [2] 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 Prolog 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. Such terms are generated, for instance, when one converts a term representation using de Bruijn indices to one using named variables, as shown in [18].

Lambda terms might contain *free variables* not associated to any binders. Such terms are 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 nodes, with S represented in unary notation with the functor `s/1`.

```
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, with leaves assumed of size 1 are counted by the sequence `A001006` in [13]. Alternatively, as in our case, when leaves are assumed of size 0, we obtain binary-unary trees with `S` internal nodes, counted by the entry `A006318` (Large Schröder Numbers) of [13].

In Prolog, a convenient way to automate defining chains of arguments operating on an evolving state in a backtrackable way, is by using Definite Clause Grammars (DCGs), which transform a clause defined with "`-->`" like

```
a0 --> a1,a2,...,an.
```

into

```
a0(S0,Sn):-a1(S0,S1),a2(S1,S2),...,an(Sn-1,Sn).
```

In our case, this expands a clause like:

```
motzkin(l(X))-->down,motzkin(X).
```

into

```
motzkin(l(X),S1,S4):-down(S1,S2),motzkin(X,S2,S3).
```

We will use this mechanism repeatedly in the paper.

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 (and 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, as the DCG notation can be eliminated by threading two extra arguments controlling the size of the terms, a fact mostly relevant as a witness for the expressiveness of this subset of Prolog.

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 trees 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 [13, 8] 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 [5], the language of types is isomorphic with that of *the implicational fragment of intuitionistic propositional 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 [13]).

Example 3. All type skeletons for N=2 and N=3.

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

?- 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 [13].

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 intuitionistic implicational logic, among which tautologies will correspond to simple types, as being generated by the predicate `maybe_type/3`.

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

Example 5. Well-formed formulas of the implicational fragment of intuitionistic propositional 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 product of the Catalan number of size n and Bell number of size $n+1$, (A289679 in [13, 1]) .

4. Merging the two worlds: generating simply-typed 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 is 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 `PSPACE` complete [20].

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 [8], this operation is available in most Prologs as a built-in predicate (called `unify_with_occurs_check/2`), optimized in SWI-Prolog [21] 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 lambda terms to their types. Each logic variable is then bound to a lambda 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 lambda 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 [13, 8].

```
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. Interleaving 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 by 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/2`. 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.

Example 8. A simply-typed 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
```

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 [13, 8]. Interestingly, by *interleaving* generation of closed terms and type inference in the predicate `typed_lambda`, the time to generate all the closed simply-typed terms is actually shorter than the time to generate all closed terms of the same size, e.g., 10.313 vs. 20.763 seconds for size 10 (see section 6). As, via the Curry-Howard isomorphism, closed simply typed terms correspond to proofs of tautologies in the implicational fragment of intuitionistic propositional logic, co-generation of terms and types corresponds to co-generation of tautologies and their proofs for proofs of a given length.

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 the inhabitants they are derived from. This simplification brings us an additional 1.5x speed-up.

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

One more (easy) step, giving a 3x speed-up, makes reaching counts for sizes 13 and 14 achievable: on a faster machine, using a slightly faster Prolog, with a similar unify_with_occurs_check built-in, like YAP [22], with the value for size 14 computed 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. As normalization preserves typability, generating them is relevant for the study simple typed lambda terms.

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 [13, 8] that counts closed normal forms of various sizes.

The predicate nf_with_type/3 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`.

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

It works by calling the DCG-expanded `typed_nf` 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
```

Note that if we want to just collect the set of types having inhabitants of a given size, the *preservation of typability under β-reduction* property [4] would allow us to work with the (smaller) set of simply-typed terms in normal form. Like in the case of general lambda terms, we can drop the generation of actual lambda terms and generate only the inhabitable types, shown here, with the DCG-expansion made explicit.

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

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

Size	closed λ-terms	gen, then infer	gen + infer	inhabitants	typed normal form
5	0.000	0.000	0.001	0.000	0.000
6	0.002	0.006	0.002	0.002	0.001
7	0.017	0.059	0.016	0.010	0.005
8	0.161	0.628	0.122	0.079	0.032
9	1.745	7.607	1.067	0.713	0.252
10	20.763	98.259	10.313	6.947	2.059

Figure 2. Timings (in seconds) for our generators up to size 10 (on a 2017 iMacPro with Xeon W Processor)

```
% types with inhabitants in normal form, directly
inh_nf_direct_with_succ(S,T):-inh_nf_direct(T,[],S,0).

inh_nf_direct(P,Ps,N1,N2):-inh_nf_no_left_lambda_direct(P,Ps,N1,N2).
inh_nf_direct((P->Q),Ps,s(N1),N2):-inh_nf_direct(Q,[P|Ps],N1,N2).

inh_nf_no_left_lambda_direct(P,[Q|Ps],N,N):-hypo_type(P,[Q|Ps]).
inh_nf_no_left_lambda_direct(Q,Ps,s(N1),N3) :-
  inh_nf_no_left_lambda_direct((P->Q),Ps,N1,N2),
  inh_nf_direct(P,Ps,N2,N3).
```

Size	slicedRun-typed-4Gb	mparRun-typed	Speed-up
6	0.014	0.005	2.8
7	0.042	0.013	3.231
8	0.197	0.061	3.23
9	1.075	0.302	3.56
10	5.428	1.432	3.791
11	30.684	7.456	4.115
12	169.755	82.958	2.046

Figure 3. YAP vs. SWI-Prolog sequential execution times (in seconds)

6. Performance of sequential execution

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 its actual complexity depends on the size of the terms involved. Therefore, figure **2** gives actual timings for the same operations above `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`" covers the algorithm that first generates lambda terms and then infers their types. The column "`gen + infer`" gives performance data for the significantly faster 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 results for the fast, merged generation and type inference for terms in normal form.

Note that the performance gap between the last two columns comes from the fact that the typed normal form generator builds the actual lambda term, while the the code checking the existence of inhabitants avoids that.

Note also that performance of generating typed terms or inhabitable types is significantly better than just generating closed terms. This is explained by the fact that early failure in occurrence checking acts as a constraint that avoids term building as early as possible.

As our sequential code is highly portable, we have hoped that using a Prolog known to be, in general, faster than SWI-Prolog can provide additional speed-up.

Figure 3 compares the performance of the YAP system [22], version 6.5.0, and SWI-Prolog 8.1.3, both with optimization turned on (-O for SWI-Prolog and -J4 for YAP) and and both compiled on the same 18-core iMac-Pro machine running OS-X 10.15.2.

Surprisingly, SWI-Prolog turned out to be slightly faster, possibly due to faster backtracking on its virtual machine or its more efficient memory reclamation. This also hints towards the fact, known to this author since the early 90's [23, 24] that the usual assumptions about performance of Prolog implementations do not extend in an obvious way to "OR-intensive" programs like our term generators.

7. Parallel term generation and counting

7.1. Some parallelization challenges

Prolog has a rich history of virtual machines customized for parallel programming, overviewed in [25]. Unfortunately, after a few hundred papers and a dozen of systems, no implementation has survived in a usable Prolog system. A shared feature of these bygone systems is their fine-grained shared-memory parallel execution model, that contrasts with the fairly language independent message-passing multi-threading execution model prevalent today.

As we adopt the robust and well-designed multi-threading API implementation of SWI-Prolog[1] [26], which uses message passing to communicate between threads, we will need to devise a coarse-grained parallelization mechanism ensuring that maximum performance is extracted from the underlying native threads. While expressive and easy to work with, message queues involve double copying from the heaps of the communicating threads. Clearly, that precludes any attempt to parallelize fine-grained execution steps. On the other hand, generating and filtering terms in the presence of backtracking, hints toward the need for devising a coarse-grained model of *OR-parallel* execution, as combinatorial generation problems are typically written in an *"OR-intensive"* programming style [24], ensuring the compact memory footprint and scalability required when dealing with trillions of generated terms.

This also means that we need to minimize communication overhead and synchronization costs while avoiding single points of contention. At the same time, we will need to balance the load as uniformly as possible to keep running the native threads at full speed with as little task switching as possible.

Next, we devise a stepwise refinement process that will progressively achieve all these objectives.

7.2. The "generate and execute" pattern

To get some real speed-up from parallel execution, given the granularity of thread-based parallel programming, with stack, trail, heap local to each thread, one needs to find a way to split the code execution such that all available native threads can be kept busy, while communication and synchronization costs are kept low.

One way to do this would be to generate closed terms and delegate type-inference to the threads. But then we would lose the 1-2 orders of magnitude speed-up coming from interleaving generation and type inference. Thus, we need to ensure that at least some type inference steps are interleaved with generation.

We can achieve this by modifying the predicate `typed_lambda` given in subsection 4.2, to generate, at each step, a constraint, `hypo(X,P,Ps)` stating that, hypothetically, for some member `X:Q` of `Ps` the types `P` and `Q` will unify. The constraint is specified by the predicate `hypo/3`. As the interleaved term generation and type inference proceeds, constraints are added to a DCGs stream by the predicate `add_hypo/5`.

```
hypo(X,P,Ps):-member(X:Q,Ps),unify_with_occurs_check(P,Q).
```

[1] http://www.swi-prolog.org/man/threads.html

```
add_hypo(X,P,Ps,(hypo(X,P,Ps),Gs),Gs).
```

Thus, with the DCG transformation maintaining the accumulation of the `hypo/3` constraints into a conjunction of Prolog goals, we obtain typed/3, which, otherwise, mimics the execution mechanism of `type_lambda` in subsection 4.2.

```
typed(X,P,[Q|Ps],N,N)-->add_hypo(X,P,[Q|Ps]).
typed(l(X,A),(P->Q),Ps,s(N1),N2)-->typed(A,Q,[X:P|Ps],N1,N2).
typed(a(A,B),Q,Ps,s(N1),N3)-->
  typed(A,(P->Q),Ps,N1,N2),
  typed(B,P,Ps,N2,N3).
```

By initializing the conjunction stream with `true` and collecting the accumulated constraint goals into the conjunction `Gs`, we obtain the predicate `typed/3` returning as its argument `Gs`, *executable code*, ready to be used in sequential or parallel mode.

```
typed(N,X:T,Gs):-n2s(N,S),typed(X,T,[],S,0,Gs,true).
```

Thus, the sequential use, as defined by `typed/2`, is equivalent to the predicate `typed_lambda/3` in section 4.2.

```
typed(N,X:T):-typed(N,X:T,Gs),call(Gs).
```

Example 13. Generating the executable constraints:

```
?- typed(4,X:T,Gs).
X = l(A, l(B, l(C, l(D, E)))),
T =  (F->G->H->I->J),
Gs =  (hypo(E, J, [D:I, C:H, B:G, A:F]), true) ;
X = l(A, l(B, l(C, a(D, E)))),
T =  (F->G->H->I),
Gs =  (hypo(D,  (J->I), [C:H, B:G, A:F]), hypo(E, J, [C:H, B:G, A:F]), true) ;
.....
```

One can see that the first stage of the computation derives a possible type template. When executing the generated constraints, one can either refine these templates, or discard them when the unifications triggered in `hypo/3` fail.

Example 14. Generating and running the executable constraints:

```
?- typed(4,X:T,Gs),call(Gs).
X = l(A, l(B, l(C, l(D, D)))),
T =  (E->F->G->H->H),
Gs =  (hypo(D, H, [D:H, C:G, B:F, A:E]), true) ;
X = l(A, l(B, l(C, l(D, C)))),
T =  (E->F->G->H->G),
Gs =  (hypo(C, G, [D:H, C:G, B:F, A:E]), true) ;
.....
```

In fact, if one partially evaluates the use of `member/2` in predicate `hypo/3` into a set of equivalent disjunctions, it becomes clear that we have compiled the interleaved generation and type inference of `typed_lambda/3` into a conjunction of disjunctions, ready to be run on independent threads, with non-determinism brought by the disjunctive components (or the action of member/2), and the filtering for correctness of the inferred types provided by the conjunction list.

As the derivation steps from closed normal forms to simply typed normal forms are similar to those described for general lambda terms, we will describe here only the last step.

The same mechanism can be used for generating simply typed normal forms, except that the generator will avoid placing lambda nodes on the left branch of an application node.

```
tnf(N,X:T):-tnf(N,X:T,Gs),Gs.

tnf(N,X:T,Gs):-n2s(N,S),tnf(X,T,[],S,0,Gs,true).

tnf(X,P,Ps,N1,N2)-->tnf_no_left_lambda(X,P,Ps,N1,N2).
tnf(l(X,A),(P->Q),Ps,s(N1),N2)-->tnf(A,Q,[X:P|Ps],N1,N2).

tnf_no_left_lambda(X,P,[Q|Ps],N,N)-->add_hypo(X,P,[Q|Ps]).
tnf_no_left_lambda(a(A,B),Q,Ps,s(N1),N3)-->
  tnf_no_left_lambda(A,(P->Q),Ps,N1,N2),
  tnf(B,P,Ps,N2,N3).
```

We will next see how to fit this *generate and execute* pattern, into SWI-Prolog's actual multi-threading constructs. Note that keeping it as general as possible will not only simplify our logic, but also result in reusable code, beneficial to easily parallelize similar combinatorial generation problems.

7.3. Concurrent execution with SWI-Prolog's `concurrent_maplist/3`

SWI-prolog offers some high-level parallel execution predicates in its library `threads.pl`, among which, the `concurrent_maplist` predicate, taking a list of goals and running them in parallel, offers a clear declarative semantics.

This gives us a very concise code snippet with a good speed-up over sequential execution. First, we can use SWI-Prolog's aggregate library to count the solutions of a goal.

```
sols(Goal,SolCount):-aggregate_all(count,Goal,SolCount).
```

Then `concur_gen/3` will create a list of `Exec` goals by running the generator `Gen` to be used as inputs of `concurrent_maplist/3` applying `sols/2` to each `Exec` goal on the list `Execs`.

```
concur_gen(Exec,Gen,Sols):-
  findall(Exec,Gen,Execs),
  concurrent_maplist(sols,Execs,AllSols),
  sum_list(AllSols,Sols).
```

As in our case the length of the list generated by `findall` grows exponentially with the size of the terms, at sizes 14 and 15 we expect billions and trillions of such goals.

A way to solve the intractable space requirements of `concur_gen/3` is to use SWI-Prolog's `findnsols/4` predicate, that works like `findall/3` but backtracks over slices of length at most `N`, in-

stead of building the complete list of solutions on the heap. The predicate `sliced_gen/4` implements this idea.

```
sliced_gen(SliceSize,Exec,Gen,TotalSols):-
  aggregate_all(sum(Sum),
  (
    findnsols(SliceSize,Exec,Gen,Execs),
    concurrent_maplist(sols,Execs,Sols),
    sum_list(Sols,Sum)
  ),
  TotalSols).
```

Note that this time, the memory consumption is controlled by `SliceSize` and will not grow exponentially with the size of the generated terms. By defining

```
sliced_gen(Exec,Gen,TotalSols):-
  SliceSize is 2^20,
  sliced_gen(SliceSize,Exec,Gen,TotalSols).
```

one can manage, in exchange for a minor drop in performance to work with terms up to size 15 in less than 4Gb of total stack space.

Still, our combinatorial generation programs, when run sequentially, can work in constant space in Prolog by exploring their search space on backtracking. This brings us back to search for a way to avoid building any intermediate list of solutions. Thus, we will need to design a high-level multithreading mechanism that is aware that our generator as well as the executables it produces, both encapsulate OR-intensive, nondeterministic code. At the same time, to avoid generating billions of suspended threads, we need to ensure that a thread-pool with an optimal number of workers is used.

7.4. Feed the birds: OR-intensive multithreading with minimal communication

We could not resist this simple analogy: one just spreads grains to a flock of birds that eagerly pick up and consume the grains, with threads playing the role of birds and tasks the role of grains.

We will start by computing a good guess on the number of available native threads on a given machine, assuming hyperthreading with two threads per core (roughly equivalent to 1.5 real cores) and some other work also happening on the machine.

```
thread_count(ThreadCnt):-
  prolog_flag(cpu_count,MaxThreads),
  ThreadCnt is max(2,ceiling((2/3)*MaxThreads)).
```

Next, we design our worker, fetching tasks from the `Queue` associated to each thread and stopping when there are no more tasks. To avoid a contentious global counter, the worker will increment a local counter each time the nondeterministic `Goal` succeeds.

```
nondet_worker():-
  thread_self(Queue),
  C=c_(0),
  repeat,
    thread_get_message(Queue,Goal),
```

```
    ( Goal='$stop',!,arg(1,C,K),thread_exit(K)
    ; Goal,
      ctr_inc(C),
      fail
    ).

ctr_inc(C):-arg(1,C,K),succ(K,SK),nb_setarg(1,C,SK).
```

Note the uses of the efficient but "impure" [2] `nb_setarg/3` to maintain the state of the counter. It works by updating (without backtracking like `setarg/3` would) the value of at given argument position in a compound term. At the end, the worker returns the total count with `thread_exit/1`.

To return to the bird analogy, we want to ensure that no grains are lost and all birds are happily collecting all the grains they can eat.

The predicate `nondet_gen/6` allows fine-tuning several parameters controlling the resources involved. It starts by creating an optimal number of threads, all running `nondet_worker/0`. Then it collects their thread identifiers and runs the nondeterministic generator `ExecGen`. The generated `Exec` goals are placed in the message queues of each thread. Fairness is ensured by `next_thread_id` that iterates, modulo the number of threads, over each thread. At the end, `thread_join` collects their results returned by `thread_exit/1`, after waiting until all the threads terminate. Finally, the results, representing success counts of the Goals executed by the workers are summed up.

```
nondet_gen(ThreadCnt,MaxMes,StackLimit,Exec,ExecGen, Res):-
  % create and start ThreadCnt  worker threads
  findall(Id,
    (
      between(1,ThreadCnt,_),
      thread_create(nondet_worker(),Id,[
        queue_max_size(MaxMes),
        stack_limit(StackLimit)
      ])
    ),
  Ids),
  ThreadArray=..[thread|Ids],
  Ctr=c(1),
  ( call(ExecGen),
    % uniformly distribute tasks
    next_thread_id(MaxMes,Ctr,ThreadCnt,ThreadArray,Id),
    thread_send_message(Id,Exec),
    fail
  ; % send as many stops as threads, but AFTER the work is done
    forall(member(Id,Ids),thread_send_message(Id,'$stop'))
  ),
  maplist(thread_join,Ids,Rs),
  maplist(arg(1),Rs,Ks), % collect results
  sum_list(Ks,Res). % sum-them up
```

[2] As we are designing here on top of a procedural multithreading API, restricting us to side-effect free Prolog is not a realistic ideal anymore.

The predicate `next_thread_id` keeps a count of each call to it and picks a thread identifier `Id` by rotating over all the available threads, modulo their number. At the same time, it skips over threads that have full message queues on which its caller would block otherwise.

```
next_thread_id(MaxMes,Ctr,ThreadCnt,ThreadArray,Id):-
  repeat,
    arg(1,Ctr,K),succ(K,SK),
    NewK is SK mod MaxMes,
    I is 1+(K mod ThreadCnt),
    nb_setarg(1,Ctr,NewK),
    arg(I,ThreadArray,Id),
    queue_size(Id,Size),
    Size<MaxMes,
  !.
```

The heuristic behind avoiding to block on a full queue is that suspension costs are relatively high and several workers might be ready to take on more work, while waiting on one's full queue to unblock.

The predicate `nondet_gen/3` sets some practical values for the parameters of the algorithm as follows:

```
nondet_gen(Exec, ExecGen, SolCount):-
 thread_count(ThreadCnt),
 MaxMes=1000000,
 prolog_flag(stack_limit,StackLimit),
 nondet_gen(ThreadCnt, MaxMes, StackLimit, Exec, ExecGen, SolCount).
```

The algorithm works well and computes in about a day and a half, on an 18-core iMac Pro, the number of simply-typed lambda terms of size **15** as **2,401,456,180,621** and the number of simply typed normal forms of size **15** as **417,818,246,574** in less than a day.

While the generator was constantly outperforming the 24 workers and needed to be contained by limiting the size of the message queue and/or by making it skip threads with full message queues, it is still potentially a single point of contention.

This brings us to the next step, a possible surprise to the reader, as it definitely was for us. The question, to which we expected the usual "it is impossible!" was: *Can we eliminate message queues altogether?*

7.5. Same birds, but we set them free: independent OR-parallel execution

To refine the analogy, our birds are now free to fly and pick up their grains, but they will be on their own to find them.

This brings us to devise a mechanism that *mimics the splitting of the tasks placed in each message queue*. If our threads would "magically" know which of the generated executable code is theirs, then they can run the same generator and consume just *their assigned executable*, while ignoring the others.

Let us first define, independently of any multithreading operations, how such a customized independent task works.

First, let's encapsulate the state of a counter rotating modulo `M` on values `K` ranging from `0` to `M-1`.

```
rotate(Ctr,M,K):-arg(1,Ctr,K),succ(K,SK),NewK is SK mod M,nb_setarg(1,Ctr,NewK).
```

The higher-order predicate `indep_task/5` will apply a generator `F` working on terms of size `N` and producing a goal `Gs` to be executed locally. It maintains both the `Rotor` rotating counter and the success counter `Ctr` incremented after each success of the nondeterministic executable goal `Gs`.

```
indep_task(F,N,M,I, Res):-
  Rotor=r(0),Ctr=c(0),
  ( call(F,N,_,Gs),
    rotate(Rotor,M,J),
    I=:=J, % only execute Gs for designated value I, fail otherwise
    call(Gs),
    ctr_inc(Ctr),
    fail
  ; arg(1,Ctr,Res)
  ).
```

Example 15. Count of solutions `Res` provided by execution of slice `I=7`, assuming `M=24` threads and the generator `F=typed/3`

```
?- F=typed,N=10,M=24,I=7,indep_task(F,N,M,I, Res).
F = typed,
N = 10,
M = 24,
I = 7,
Res = 414871.
```

Thus, each of our workers runs exactly the same code centered on `indep_task`, except for the selector `I` specific to each thread and assigned to them at creation time, and when done, returns its result with `thread_exit/1`.

```
indep_worker(F,N,M,I):-
  indep_task(F,N,M,I, Res),
  thread_exit(Res).
```

This makes the code generic and communication-free, except for the launching of each worker on its own thread.

The predicate `indep_run(F,N,ThreadCnt,Res)` creates and starts the workers, collects their thread identifiers, joins them when done, then collects their results and sums them up.

Thus, the selector `I` *ranging from* `0` *to* `M` *makes each worker run exactly as if it had picked a message from its message queue, except that it is now generating it on its own.*

```
indep_run(F,N,ThreadCnt,Res):-
  M is ThreadCnt-1,
  findall(Id,
    (
      between(0,M,I),
      thread_create(indep_worker(F,N,M,I),Id,[])
    ),
  Ids),
  maplist(thread_join,Ids,Es),
```

```
  maplist(arg(1),Es,Rs),
  sum_list(Rs,Res).
```

Finally, by specializing for the optimal thread count, we obtain.

```
indep_run(F,N,Res):-
  thread_count(ThreadCnt),
  indep_run(F,N,ThreadCnt,Res).
```

Example 16. Running with generator `typed/3` on terms of size `10`.

```
?- indep_run(typed,10,SolCount).
SolCount = 9006364.
```

We can expose `indep_run` with the same interface as `indep_gen`.

```
indep_gen(Exec,Gen,SolCount):-Gen=..[F,N,_,Exec],indep_run(F,N,SolCount).
```

Note that efficiency of this mechanism depends on the ratio between the effort to generate tasks and the effort to run the tasks. As such, in the case of generation of simply typed terms and their types, it is only slightly slower than dispatching tasks via message queues to each thread. On the other hand, it scales easily to cluster or cloud computing infrastructures, where the cost of sending messages between processes running on distinct computers or virtual machines would incur additional costs.

8. Performance of parallel execution

We start with a performance measuring predicate, `parRun/5`, that provides a uniform interface to run and time all the combinations of `Runner`, providing the parallelization algorithm, and `Prog`, providing the `Gen` and `Exec` pair, that, by working together, count all simply typed terms or normal forms of size `N`.

```
parRun(N,Prog,Runner,SolCount,Time):-
  Gen=..[Prog,N,_,Exec],
  time(call(Runner,Exec,Gen,SolCount),Time).
```

Its interesting instances, corresponding to the four algorithms described in section 7 are:

```
mparRun(N,Prog,SolCount,Time):-parRun(N,Prog,nondet_gen,SolCount,Time).

iparRun(N,Prog,SolCount,Time):-parRun(N,Prog,indep_gen,SolCount,Time).

concRun(N,Prog,SolCount,Time):-parRun(N,Prog,concur_gen,SolCount,Time).

slicedRun(N,Prog,SolCount,Time):-parRun(N,Prog,sliced_gen,SolCount,Time).
```

We will add also `seqRun/4` that exposes sequential execution under a similar interface.

```
seqRun(N,Prog,SolCount,Time):-
  Gen=..[Prog,N,_,Exec],
  time(sols((Gen,Exec),SolCount),Time).
```

Size	slicedRun-typed-4Gb	mparRun-typed	Speed-up
6	0.014	0.005	2.8
7	0.042	0.013	3.231
8	0.197	0.061	3.23
9	1.075	0.302	3.56
10	5.428	1.432	3.791
11	30.684	7.456	4.115
12	169.755	82.958	2.046

Figure 4. Timings (in seconds) for sequential vs. parallel generation of simply typed lambda terms on a Xeon W 18-core iMacPro

Size	slicedRun-typed-4Gb	mparRun-typed	Speed-up
6	0.014	0.005	2.8
7	0.042	0.013	3.231
8	0.197	0.061	3.23
9	1.075	0.302	3.56
10	5.428	1.432	3.791
11	30.684	7.456	4.115
12	169.755	82.958	2.046

Figure 5. Timings for sequential vs. parallel simply typed normal forms generation

We will proceed by first comparing our best-performing parallel program against its sequential equivalent and then the parallel algorithms among them.

Figure 4 shows a speed-up of `mparRun` over `seqRun` that stabilizes around a factor of 10 on an 18-core iMacPro using a total of 26 threads among which 24 are working on tasks, one thread feeding their message queues and one thread being SWI-Prolog's concurrent garbage collection thread. That keeps the machine at around 69% total CPU dedicated to our program. We have observed that adding more threads results in minor performance changes, mostly because the two threads on each hyperthreading core are known to be equivalent to around 1.5 threads running each on its own core.

Figure 5 shows the same comparison applied this time to the generation of simply typed normal forms.

Figure 6 compares `iparRun`, with threads working on independent tasks, without using any message queues with our best-performing `mparRun` predicate. Note that performance variations are minor, making the more memory-thrifty `iParRun` a valid alternative, given also that it can spread its work on a cluster or cloud infrastructure without involving communication costs at each step.

Size	slicedRun-typed-4Gb	mparRun-typed	Speed-up
6	0.014	0.005	2.8
7	0.042	0.013	3.231
8	0.197	0.061	3.23
9	1.075	0.302	3.56
10	5.428	1.432	3.791
11	30.684	7.456	4.115
12	169.755	82.958	2.046

Figure 6. Timings for two parallel algorithms for generation of simply typed lambda terms

Size	slicedRun-typed-4Gb	mparRun-typed	Speed-up
6	0.014	0.005	2.8
7	0.042	0.013	3.231
8	0.197	0.061	3.23
9	1.075	0.302	3.56
10	5.428	1.432	3.791
11	30.684	7.456	4.115
12	169.755	82.958	2.046

Figure 7. Timings for two parallel algorithms for generation of simply typed lambda terms

Figure 7 compares the slower and more memory intensive `concRun` with `mparRun`. Note that we had to lift SWI-Prolog's `stack_limit` to **16GB** to accommodate its exponentially growing memory needs.

Finally, Figure 8 compares the slightly slower `slicedRun` which avoids the memory explosion problem of `concRun`.

Our longest test was the counting of simply typed normal forms of size **16**, that took 3.27 days to complete.

As at this size we only wanted to count the terms, given the expected trillions of them, we derived the predicate `inh_nf` from `tnf`, by removing the actual lambda term arguments, a technique outlined for sequential execution in subsection 4.3 and giving in the sequential case a 40%-50% speed-up.

```
inh_nf(N,T,Gs):-n2s(N,S),inh_nf(T,[],S,0,Gs,true).

inh_nf(P,Ps,N1,N2)-->inh_nf_no_left_lambda(P,Ps,N1,N2).
inh_nf((P->Q),Ps,s(N1),N2)-->inh_nf(Q,[P|Ps],N1,N2).
```

Size	slicedRun-typed-4Gb	mparRun-typed	Speed-up
6	0.014	0.005	2.8
7	0.042	0.013	3.231
8	0.197	0.061	3.23
9	1.075	0.302	3.56
10	5.428	1.432	3.791
11	30.684	7.456	4.115
12	169.755	82.958	2.046

Figure 8. Timings for two parallel algorithms for generation of simply typed lambda terms

```
inh_nf_no_left_lambda(P,[Q|Ps],N,N) --> add_hypo_type(P,[Q|Ps]).
inh_nf_no_left_lambda(Q,Ps,s(N1),N3) -->
  inh_nf_no_left_lambda((P->Q),Ps,N1,N2),
  inh_nf(P,Ps,N2,N3).
```

We have derived the auxiliary predicates `add_hypo_type` and `hypo_type` by also trimming the first arguments of `add_hypo` and `hypo`.

```
add_hypo_type(P,Ps,(hypo_type(P,Ps),Gs),Gs).

hypo_type(P,Ps):-member(Q,Ps),unify_with_occurs_check(P,Q).
```

We used the `nondet_gen` generator and allowed message queue sizes of up to a million, resulting in a maximum memory footprint of 12GB.

Example 17. Computing the number of simply typed normal forms of size **16**.

```
?- parRun(16,inh_nf,nondet_gen,SolCount,TimeInSecs).
SolCount = 5,612,087,926,963,
TimeInSecs = 282768.107.
```

As an indication of how robust its multi-threading subsystem is, at the end of the test, SWI-Prolog shut down all worker threads and trimmed back its memory footprint to the usual few megabytes default.

We were surprised, by running `statistics/0` at the end, to notice that a nominal **24x** speed-up has been achieved, as the 24 finished threads used 6570418.889 seconds and $6848345.203/282768.107 = 24.218944900317204$. While some of the total time of the threads was spent on managing their message queues and task switching, this indicates that the Xeon W 18 core CPU has been used very close to its maximum capacity.

9. Discussion

The parallel execution mechanisms that we have derived for these problems are generic, in the following sense:

1. they apply to any combinatorial generation problem, where one can split the generation mechanism into two layers, where the second is an executor of nondeterministic code generated by the first, which is also a nondeterministic generator
2. benefits are maximized when the instances of second layer are computationally intensive and are run on separate threads
3. if one wants to just count the number of answers produced by a combinatorial generator, inter-thread communication can be drastically reduced, by allocating to each computation in the second layer its own slice, filtered from an integrated first layer generator, provided that re-executing it on each thread is inexpensive.

An interesting open problem is if these performance improvements 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. As future work we plan to explore the possibility of relying on the type discipline in [27] that hints on how to deal with type environments for de Bruijn indices, where the interaction between the weakening rule and the rules for lambda abstractions determine the type of a de Bruijn index or the closure calculus of [28] that combines de Bruijn indices with de Bruijn levels, such that the latter remain independent from the context within the closure's environment.

We have not seen any obvious way to improve these results using constraint programing systems, partly because the key "inner loop" operation is unification with occurs check, with computations ranging over Prolog terms rather than over objects of a finite constraint domain. On the other hand, for a given size, grounding to propositional formulas for SAT-solvers or Answer-Set Programming systems seem worth exploring. The main problem faced there is that of expressing unification with occurs check in terms of a propositional encoding, under the assumption that sizes of both terms and type-expressions are bounded.

We have not discussed here the use of similar techniques to improve the Boltzmann samplers described to [29], but clearly interleaving type checking with the probability-driven building of the terms would improve their performance, by excluding terms with ill-typed subterms as early as possible, during the large number of retries needed to overcome the asymptotically 0-density of simply-typed terms in the set of closed terms [9].

Several concepts of size have been used in the literature, for reasons ranging from simplifying evaluation procedures [30] to matching the structure of the terms naturally occurring in actual programs [31]. As a byproduct, some size definitions also result in better convergence conditions of formal series in analytic combinatorics [32]. Our techniques can be easily adapted to a different size definition like the ones in [31, 32] where variables in de Bruijn notation have a size proportional to the distance to their binder.

Interestingly, if one wants to match as closely as possible the intuition that in actual programs most lambdas will bind more than one variable, the cost of lambda constructors should be higher than that of application nodes, which correlate n to $n+1$ with the number of available leaves. The size definition used in this paper, with equal cost for applications and lambdas comes close to that as it

ensures at least a one-to-one ratio between lambda nodes and the leaf variables they might bind. On the other hand, definitions adding extra weight to variables or applications are likely to generate less interesting terms of a given size by implicitly devaluating the lambda binders.

Our generic parallel execution algorithms assume a split of a given combinatorial generation or search program into a code generator, to be run sequentially and a second stage execution of the generated code on multiple threads. While this is an "eureka" step that requires some intuition on the most effective way to decompose such algorithms, it is facilitated by the compositional nature of these algorithms, that can be seen as successive refinements of simpler generators with additional layers of "decorations" driven by independent combinatorial mechanisms.

Note also that while we have specialized our generators to produce only solution counts, a simple change would be to make them collect the actual solutions and then return the with `thread_exit`.

10. Related work

The classic reference for lambda calculus is [2]. Various instances of typed lambda calculi are overviewed in [4].

The combinatorics and asymptotic behavior of various classes of lambda terms are extensively studied in [8, 11]. Distribution and density properties of random lambda terms are described in [9].

Generation of random simply-typed lambda terms and its applications to generating functional programs from type definitions is covered in [33].

Asymptotic density properties of simple types (corresponding to tautologies in implicational fragment of intuitionistic propositional logic) have been studied in [34] with the surprising result that "almost all" classical tautologies are also intuitionistic ones.

In [7] 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). A statistical exploration of the structure of the simple types of lambda terms of a given size in [17] gives indications that some types frequent in human-written programs are among the most frequently inferred ones.

Generators for closed simply-typed lambda terms, as well as their normal forms, expressed as functional programming algorithms, are given in [8], 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 [18] 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, we have covered linear, affine linear terms as well as terms of bounded unary height and in binary lambda calculus encoding. In [18] type inference algorithms are also given for SK and Rosser's X-combinator expressions. A similar (but slower) program for type inference using de Bruijn notation is also given in [18], without however describing the step-by-step derivation steps leading to it, as done in this paper.

Parallel algorithms for generating random simply typed lambda terms and normal forms are given in [35], but working with a different concept of size, amenable to the use of analytic methods and

Boltzmann samplers. The key difference with the parallel algorithms described here is that the algorithms described here cover exhaustive generation of terms of a given size, a much harder problem than the filtering from typability of closed lambda terms provided by a Boltzmann sampler.

Typability of special families of lambda terms are investigated in [36] and [37], suggesting other possible splittings of combinatorial generation algorithms into generator/executable pairs, with potential uses for parallelization as instances of the algorithms described in this paper.

In [38] a general constraint logic programming framework is defined for size-constrained generation of data structures as well as a program-transformation mechanism. While our fine-tuned interleaving of term generation and type inference directly provides the benefits of a CLP-based scheme, the program transformation techniques described in [38] are worth exploring for possible performance improvements. In [39] a general Haskell-based framework for generating enumerable structures is introduced. While clearly useful for arbitrary free structures, the fine-grained interleaving of generation and type inference of this paper do not seem to be embeddable in it with similar performance gains.

11. Conclusion

In [1], that this paper extends, 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 higher than previously published results. The parallel algorithms given in this paper add one order of magnitude to the generation of simply typed lambda terms and two orders of magnitudes to the generation of simply typed normal forms.

Our sequential execution algorithms have 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 only as counters of solutions, as in this case 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. For our sequential programs, 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.

With the extra speed-up brought by our parallel algorithms, two of the integer sequences in [13] can be further extended. Sequence **A220471**[3], counting the number of simply typed lambda terms of size `N` [8] has **2,401,456,180,621** terms of size **15** and **A289681**[4], counting the set of simply typed normal forms of size `N` [1] has **417,818,246,574** terms of size **15** and **5,612,087,926,963** terms of size **16**.

Our parallel execution algorithms, have been designed to be generic. Beyond their instantiations for generating simply typed lambda terms and normal forms, they are likely to be reusable for a several similar combinatorial generation or search problems.

An interesting application, we have started to work on, is the use of simply typed normal forms, via the Curry-Howard correspondence [5, 6], as known-to-be-provable formulas of implicational in-

[3] `https://oeis.org/A220471`
[4] `https://oeis.org/A289681`

tuitionistic propositional calculus. As such, they provide a very large supply of positive examples allowing automated testing of the soundness of intuitionistic theorem provers [40].

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 anonymous reviewers of LOPSTR'16 and Fundamenta Informaticae for their constructive suggestions and the participants of the 9th Workshop Computational Logic and Applications (`https://cla.tcs.uj.edu.pl/`) for enlightening discussions 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] Tarau P. A Hiking Trip Through the Orders of Magnitude: Deriving Efficient Generators for Closed Simply-Typed Lambda Terms and Normal Forms. In: Hermenegildo MV, Lopez-Garcia P (eds.), Logic-Based Program Synthesis and Transformation: 26th International Symposium, LOPSTR 2016, Edinburgh, UK, Revised Selected Papers. Springer LNCS, volume 10184. ISBN 978-3-319-63139-4, 2017 pp. 240–255. doi:10.1007/978-3-319-63139-4_14. , Best paper award.

[2] Barendregt HP. The Lambda Calculus Its Syntax and Semantics, volume 103. North Holland, revised edition, 1984.

[3] Hindley JR, Seldin JP. Lambda-calculus and combinators: an introduction, volume 13. Cambridge University Press Cambridge, 2008.

[4] Barendregt HP. Lambda Calculi with Types. In: Handbook of Logic in Computer Science, volume 2. Oxford University Press, 1991.

[5] Howard W. The Formulae-as-types Notion of Construction. In: Seldin J, Hindley J (eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic Press, London, 1980.

[6] Wadler P. Propositions as types. *Commun. ACM*, 2015. **58**:75–84.

[7] Palka MH, 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. ACM, New York, NY, USA, 2011 pp. 91–97.

[8] Grygiel K, Lescanne P. Counting and generating lambda terms. *J. Funct. Program.*, 2013. **23**(5):594–628.

[9] David R, Raffalli C, Theyssier G, Grygiel K, Kozik J, Zaionc M. Some properties of random lambda terms. *Logical Methods in Computer Science*, 2009. **9**(1).

[10] Bodini O, Gardy D, Gittenberger B. Lambda-terms of Bounded Unary Height. In: ANALCO. SIAM, 2011 pp. 23–32.

[11] 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.

[12] Flajolet P, Sedgewick R. Analytic Combinatorics. Cambridge University Press, New York, NY, USA, 1 edition, 2009. ISBN 0521898064, 9780521898065.

[13] Sloane NJA. The On-Line Encyclopedia of Integer Sequences. 2020. Published electronically at https://oeis.org/.

[14] Tarau P. On Logic Programming Representations of Lambda Terms: de Bruijn Indices, Compression, Type Inference, Combinatorial Generation, Normalization. In: Pontelli E, Son TC (eds.), Proceedings of the Seventeenth International Symposium on Practical Aspects of Declarative Languages PADL'15. Springer, LNCS 8131, Portland, Oregon, USA, 2015 pp. 115–131.

[15] 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. Springer, LNAI 9150, Washington, D.C., USA, 2015 pp. 118–133.

[16] 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. ACM, New York, NY, USA, 2015 pp. 244–255.

[17] 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. CEUR, Cork, Ireland, 2015 Available online at http://ceur-ws.org/Vol-1433/.

[18] Tarau P. A Logic Programming Playground for Lambda Terms, Combinators, Types and Tree-based Arithmetic Computations. *CoRR*, 2015. **abs/1507.06944**. URL `http://arxiv.org/abs/1507.06944`.

[19] Stanley RP. Enumerative Combinatorics. Wadsworth Publ. Co., Belmont, CA, USA, 1986. ISBN 0-534-06546-5.

[20] Statman R. Intuitionistic Propositional Logic is Polynomial-Space Complete. *Theor. Comput. Sci.*, 1979. **9**:67–72. doi:10.1016/0304-3975(79)90006-9.

[21] Wielemaker J, Schrijvers T, Triska M, Lager T. SWI-Prolog. *Theory and Practice of Logic Programming*, 2012. **12**:67–96. doi:10.1017/S1471068411000494.

[22] Costa VS, Rocha R, Damas L. The YAP Prolog system. *Theory and Practice of Logic Programming*, 2012. **12**:5–34. doi:10.1017/S1471068411000512.

[23] Tarau P. An Efficient Specialization of the WAM for Continuation Passing Binary Programs. In: Proceedings of the 1993 ILPS Conference. MIT Press, Vancouver, Canada, 1993 Poster.

[24] Tarau P, Demoen B. Higher-Order Programming in an OR-intensive Style. In: Hermenegildo M, Lopez P (eds.), Proceedings of the 1995 COMPULOG-NET Workshop and Area Meeting on Parallelism and Implementation Technology. 1995 .

[25] Gupta G, Pontelli E, Ali KAM, Carlsson M, Hermenegildo MV. Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 2001. **23**(4):472–602. doi:10.1145/504083.504085. URL `https://doi.org/10.1145/504083.504085`.

[26] Wielemaker J. Native Preemptive Threads in SWI-Prolog. In: Palamidessi C (ed.), Practical Aspects of Declarative Languages. Springer Verlag, Berlin, Germany, 2003 pp. 331–345. LNCS 2916.

[27] Kiselyov O. λ to SKI, Semantically - Declarative Pearl. In: Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings, volume 10818 of *Lecture Notes in Computer Science*. 2018 pp. 33–50.

[28] García-Pérez Á, Nogueira P. The full-reducing Krivine abstract machine KN simulates pure normal-order reduction in lockstep: A proof via corresponding calculus. *J. Funct. Program.*, 2019. **29**:e7.

[29] Lescanne P. Boltzmann samplers for random generation of lambda terms. *CoRR*, 2014. **abs/1404.3875**. URL `http://arxiv.org/abs/1404.3875`.

[30] Tromp J. Binary Lambda Calculus and Combinatory Logic, 2018. Published electronically at https://tromp.github.io/cl/LC.pdf.

[31] Grygiel K, Lescanne P. Counting and generating terms in the binary lambda calculus. *J. Funct. Program.*, 2015. **25**. doi:10.1017/S0956796815000271. URL `http://dx.doi.org/10.1017/S0956796815000271`.

[32] Bendkowski M, Grygiel K, Lescanne P, Zaionc M. A Natural Counting of Lambda Terms. In: Freivalds RM, 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. ISBN 978-3-662-49191-1, 2016 pp. 183–194. doi:10.1007/978-3-662-49192-8_15. URL `http://dx.doi.org/10.1007/978-3-662-49192-8_15`.

[33] Fetscher B, Claessen K, Palka MH, Hughes J, Findler RB. 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 pp. 383–405.

[34] 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. ISBN 978-3-540-68084-0, 2007 pp. 100–109. doi:10.1007/978-3-540-68103-8_7.

[35] Bendkowski M, Grygiel K, Tarau P. Random generation of closed simply typed λ-terms: A synergy between logic programming and Boltzmann samplers. *TPLP*, 2018. **18**(1):97–119. URL `https://doi.org/10.1017/S147106841700045X`.

[36] Bodini O, Tarau P. On Uniquely Closable and Uniquely Typable Skeletons of Lambda Terms. In: Fioravanti F, Gallagher JP (eds.), Logic-Based Program Synthesis and Transformation, LNCS 10855. Springer International Publishing. ISBN 978-3-319-94460-9, 2018 pp. 252–268.

[37] Tarau P. On k-colored Lambda Terms and their Skeletons. In: Calimeri F, Hamlen KW, Leone N (eds.), Practical Aspects of Declarative Languages - 20th International Symposium, PADL 2018, Los Angeles, CA, USA, January 8-9, 2018, Proceedings, volume 10702 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-319-73304-3, 2018 pp. 116–131. doi:10.1007/978-3-319-73305-0_8.

[38] Fioravanti F, Proietti M, Senni V. Efficient generation of test data structures using constraint logic programming and program transformation. *Journal of Logic and Computation*, 2015. **25**(6):1263–1283. doi:10.1093/logcom/ext071.

[39] Kuraj I, Kuncak V, Jackson D. Programming with Enumerable Sets of Structures. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015. ACM, New York, NY, USA. ISBN 978-1-4503-3689-5, 2015 pp. 37–56. doi:10.1145/2814270.2814323. URL `http://doi.acm.org/10.1145/2814270.2814323`.

[40] Tarau P. A Combinatorial Testing Framework for Intuitionistic Propositional Theorem Provers. In: Alferes JJ, Johansson M (eds.), Practical Aspects of Declarative Languages - 21th International Symposium, PADL 2019, Lisbon, Portugal, January 14-15, 2019, Proceedings, volume 11372 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-030-05997-2, 2019 pp. 115–132. doi:10.1007/978-3-030-05998-9_8.