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

August 21, 2016

Research supported by NSF grant **1423324**

Outline

Motivation

- simply-typed lambda terms enjoy a number of nice properties
  - strong normalization, set-theoretical semantics
  - via the Curry-Howard correspondence lambda terms that are inhabitants of simple types can be seen as proofs for tautologies in minimal logic
  - extended with a fix-point operator they can be used as the intermediate language for compiling Turing-complete functional languages
- generating or counting simply typed lambda terms is hard: no analytic method is currently known
- their asymptotic density in the set of closed terms is 0
- $\Rightarrow$ sparsity makes their brute-force enumeration is computation intensive
- published work covers generation and counting up to size **10**
- for each size, computational effort increases by one order of magnitude
- $\Rightarrow$ the challenge: how far we can go?
- $\Rightarrow$ we derive progressively more efficient logic programs - up to size **14**

Deriving a generator for lambda terms

A canonical representation with logic variables

- logic variables can be used in Prolog for connecting a lambda binder and its related variable occurrences
- this representation can be made canonical by ensuring that each lambda binder is marked with a distinct logic variable
- the term $\lambda a.((\lambda b.(a(b\ b)))(\lambda c.(a(c\ c))))$ is represented as
- `l(A,a(l(B, a(A,a(B,B))), l(C, a(A,a(C,C)))))`
- each lambda binder is mapped to a distinct logic variable
- scoping of logic variables is "global" to a clause
- lambda terms can be seen as Motzkin trees (also called unary-binary trees), labeled with lambda binders at their unary nodes and corresponding variables at the leaves
- $\Rightarrow$ we will derive a generator from one for Motzkin trees

Generating Motzkin trees

- Motzkin-trees have internal nodes of arities 1 or 2
- they can be seen as a skeleton of lambda terms that ignores binders and variables and their leaves
- we use successor arithmetic: 0, s(0),s(s(0)) ...
- DCG notation keeps track of size with predicated `down/2` that decrements it at each step

```
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).
```

Generating closed lambda terms

- we use successor arithmetic: 0, s(0),s(s(0)) ...
- size definition: a/2 = 1 unit, l/1 = 1 unit. leaves = 0 units

```
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).
```

- a list, initially empty of variables is built
- each lambda binder pushes a variable to it
- each leaf is constrained to correspond, to a variable
- we use the list to count binders - but it will later hold types that we infer

Example: generating lambda terms

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))) .
```

The language of simple types

Type skeletons

```
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 *Online Encyclopedia of Integer Sequences* (OEIS)
- 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) .
```

Set partitions as logic variable equalities

- set partitions are in bijection with equivalence relations!
- ⇒ they can be used to generate all the ways N logic variables can be made equal (see code in the paper)

```
?- 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].
```

- ⇒ 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) .
```

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

Merging the two worlds: generating simply-typable lambda terms

Solving the easier problem

- per-size counts of both the sets of lambda terms and their potential types are very fast growing
- computing an inhabitant of a type (if it exists) is `P-space` complete
- computing the type of a given lambda term (if it exists) is efficient (linear in practice)
- $\Rightarrow$ design a type inference algorithm that takes advantage of operations on logic variables

A type inference algorithm

- in a functional language inferring types requires implementing unification with occurs-check
- this operation is available in Prolog as a built-in predicate often optimized to proceed incrementally, only checking that no new cycles are introduced during the unification step
- `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
- 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).
```

Lambda terms of size up to 3 and their types

first, the naive way: generate and then infer type

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

?- 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).
```

Interleaving term generation and type inference

Term generation + type inference as a program transformation

- two changes to `infer_type` to turn it into an efficient generator for simply-typed lambda terms
- we need to add an argument to control the size of the terms and ensure termination, by calling `down/2` for internal nodes
- we need to generate the mapping between binders and variables
- $\Rightarrow$ borrow the `member/2`-based mechanism used in the predicate `lambda/4` for generating closed lambda terms

Interleaving term generation and type inference

```
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).
```

- `typed_lambda/5` relies on Prolog's DCG notation to thread together the steps controlled by the predicate `down`
- `member/2` enumerates values for `variable:type` pairs ranging over the list of available pairs `Vs`
- `unify_with_occurs_check` ensures that unification of candidate types does not create cycles

One more trim: generating inhabited types

- 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
- we make the DCG-processing of the size computations explicit, in the last two arguments

```
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).
```

Counts for simply-typable terms up to `N=14`

- the multiset of generated types has the same count as the set of their inhabitants
- this 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: using a faster Prolog, with a similar `unify_with_occurs_check` built-in, ⇒ YAP
- 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
```

- the last value computed in less than a day

Generating simply-typed normal forms

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

```
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).
```

Generating simply-typed normal forms

- we can define the more efficient combined generator and type inferrer

```
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).
```

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
```

- what if we would want to just collect the set of types having inhabitants of a given size?
- a nice property: *preservation of typability under β-reduction* !
- we can work with the (smaller) set of simply-typed terms in normal form

Experimental data

Experimental data: LIPS

Size	closed λ-terms	gen, then infer	gen + infer	inhabitants	typed NF
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: Number of logical inferences used by our generators, as counted by SWI-Prolog

Experimental data: timings

Size	closed λ-terms	gen, then infer	gen + infer	inhabitants	typed NF
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

Figure: Timings (in seconds) for our generators up to size 10 (on a 2015 MacBook, with 1.3 GHz Intel Core M processor)

Discussion

Can we go a few more steps higher?

- `term_hash` based indexing and tabling-based dynamic programming algorithms, using de Bruijn terms?
- 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
- parallel execution: easy improvement - for each skeleton, flesh it up with corresponding terms on separate threads
- no obvious way to improve these results using constraint programing systems as the “inner loop” computation is unification with occurs check with computations ranging over Prolog terms rather than being objects of a constraint domain
- 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.

Extending the techniques

- adapt to a different size definition(e.g., where variables in de Bruijn notation have a size proportional to the distance to their binder)
- improve the Boltzmann samplers: 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
- likely to reduce effort during the large number of retries needed to overcome the asymptotically 0-density of simply-typed terms in the set of closed terms

Related work

- various instances of typed lambda calculi are overviewed in [1]
- 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**
- asymptotic density properties of simple types (corresponding to tautologies in minimal logic) have been studied in [4] with the surprising result that "almost all" classical tautologies are also intuitionistic ones
- several concepts of term size have been used in the literature [6, 2]
- in [3] a general constraint logic programming framework is defined for size-constrained generation of data structures as well as a program-transformation mechanism

Related work - continued

H. P. Barendregt.
Lambda calculi with types.
In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991.

M. Bendkowski, K. Grygiel, P. Lescanne, and M. Zaionc.
A Natural Counting of Lambda Terms.
In R. M. Freivalds, G. Engels, and B. Catania, editors, *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*, pages 183–194. Springer, 2016.

F. Fioravanti, M. Proietti, and V. Senni.
Efficient generation of test data structures using constraint logic programming and program transformation.
Journal of Logic and Computation, 25(6):1263–1283, 2015.

A. Genitrini, J. Kozik, and M. Zaionc.
Intuitionistic vs. Classical Tautologies, Quantitative Comparison.
In M. Miculan, I. Scagnetto, and F. Honsell, editors, *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*, pages 100–109. Springer, 2007.

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

K. Grygiel and P. Lescanne.
Counting and Generating Terms in the Binary Lambda Calculus (Extended version).
CoRR, abs/1511.05334, 2015.

Conclusions

- a language as simple as Horn Clause Prolog can handle hard combinatorial generation problems when the synergy between sound unification, backtracking and and DCGs is put at work
- deriving progressively more efficient logic programs is relatively easy for combinatorial generation tasks
- interleaving generation, closedness checking and type inference works well when we apply our constraints as early as possible
- similar techniques can be improve performance for random generation of lambda terms (e.g., via Boltzmann sampling)