On Type-directed Generation of Lambda Terms

PAUL TARAU
Department of Computer Science and Engineering
(*e-mail:* `tarau@cs.unt.edu`)

submitted TBD; revised TBD; accepted TBD

Abstract

We describe a Prolog-based combined lambda term generator and type-inferrer for closed well-typed terms of a given size, in de Bruijn notation. By taking advantage of Prolog's unique bidirectional execution model and sound unification algorithm, our generator can build "customized" closed terms of a given type. This relational view of terms and their types enables the discovery of interesting patterns about frequently used type expressions occurring in well-typed functional programs. Our study uncovers the most "popular" types that govern function applications among a about a million small-sized lambda terms and hints toward practical uses to combinatorial software testing. It also shows the effectiveness of Prolog as a meta-language for modeling properties of lambda terms and their types.

KEYWORDS: lambda calculus, de Bruijn notation, type inference, generation of lambda terms, logic programming as a meta-language.

1 Introduction

Properties of logic variables, unification with occurs-check and exploration of solution spaces via backtracking facilitate compact algorithms for inferring types or generate terms for various calculi. This holds in particular for lambda terms (Barendregt 1984). Lambda terms provide a foundation to modern functional languages, type theory and proof assistants and have been lately incorporated into mainstream programming languages including Java 8, C# and Apple's Swift.

While possibly one of the most heavily researched computational objects, lambda terms offer an endless stream of surprises to anyone digging just deep enough below their intriguingly simple surface.

This paper focuses on the synergy between combinatorial generation of lambda terms and type inference, both favored by the use of Prolog as meta-language for representing them. While we will take advantage of Prolog's logic variables for type inference, we will retain the "nameless" de Bruijn representation (Bruijn 1972) for term generation as it provides a canonical representation to α-equivalent terms. We will also focus on *closed* terms (terms without free variables) as they are the ones used in lambda calculus based programming languages like Haskell and ML or proof assistants like Coq or Agda.

Prolog's ability to support "relational" queries enables us to easily explore the population of de Bruijn terms up to a given size and answer questions like the following:

1. How many distinct types occur for terms up to a given size?
2. What are the most popular types?
3. What are the terms that share a given type?
4. What is the smallest term that has a given type?
5. What smaller terms have the same type as this term?

The paper is organized as follows. Section 2 introduces the de Bruijn notation for lambda terms and describes a type inference algorithm working on them. Section 3 describes a generator for Motzkin trees and derives a generator for lambda terms in de Bruijn form from it. Section 4 introduces an algorithm combining term generation and type inference. Section 5 uses our combined term generation and type inference algorithm to discover frequently occurring type patterns. Section 6 describes a type-directed algorithm for the generation of closed typable lambda terms. Section 7 discusses related work and section 8 concludes the paper.

The paper is structured as a literate Prolog program. The code, tested with SWI-Prolog 6.6.6 and YAP 6.3.4., is at `http://www.cse.unt.edu/~tarau/research/2015/dbt.pro`.

2 Type inference for lambda terms in de Bruijn notation

As lambda terms represent functions, inferring their types provides information on what kind of argument(s) they can be applied to. For simple types, type inference is decidable (Hindley and Seldin 2008) and it uses unification to recursively propagate type information between application sites of variable occurrences covered by a given lambda binder. We will describe next a type inference algorithm using de Bruijn indices in Prolog - a somewhat unusual choice, given that logic variables can play the role of lambda binders directly. One of the reasons we chose them is that they will be simpler to manipulate at meta-language level, as they handle object-level variables implicitly. At the same time this might be useful for other purposes, as we are not aware of any Prolog implementation of type inference with this representation of lambda terms.

2.1 De Bruijn Indices

De Bruijn indices (Bruijn 1972) provide a *name-free* representation of lambda terms. All terms that can be transformed by a renaming of variables (α-conversion) will share a unique representation. Variables following lambda abstractions are omitted and their occurrences are marked with positive integers *counting the number of lambdas until the one binding them* is found on the way up to the root of the term. We represent them using the constructor `a/2` for application, `l/1` for lambda abstractions (that we will call shortly *binders*) and `v/1` for marking the integers corresponding to the de Bruijn indices.

For instance, the term `l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C)))))` is represented as `l(a(l(a(v(1),a(v(0),v(0)))),l(a(v(1),a(v(0),v(0))))))`, given that `v(1)` is bound by the outermost lambda (two steps away, counting from `0`) and the occurrences of `v(0)` are bound each by the closest lambda, represented by the constructor `l/1`.

One can define the size of a lambda expression in de Bruijn form as the number of its internal nodes, as implemented by the predicate `dbTermSize`.

```
dbTermSize(v(_),0).
dbTermSize(l(A),R):-
  dbTermSize(A,RA),
  R is RA+1.
dbTermSize(a(A,B),R):-
  dbTermSize(A,RA),
  dbTermSize(B,RB),
  R is 1+RA+RB.
```

A lambda term is called *closed* if it contains no free variables. The predicate `isClosed` defines this property for de Bruijn terms.

```
isClosed(T):-isClosed1(T,0).

isClosed1(v(N),D):-N<D.
isClosed1(l(A),D):-D1 is D+1,
  isClosed1(A,D1).
isClosed1(a(X,Y),D):-
  isClosed1(X,D),
  isClosed1(Y,D).
```

Besides being closed, lambda terms interesting for functional languages and proof assistants, are also well-typed. We will start with an algorithm inferring types directly on the de Bruijn terms.

2.2 A type inference algorithm for Bruijn terms

Simple types will be defined here also as binary trees built with the constructor "`->/2`" with empty leaves, representing the unique primitive type "`o`". Types can be seen as as a "binary tree approximation" of lambda terms, centered around ensuring their safe and terminating evaluation (strong normalization), as it is well-known (e.g., (Barendregt 1991)) that lambda terms that have simple types are strongly normalizing. When a term `X` has a type `T` we say that the type `T` is *inhabited* by the term `X`.

While in a functional language inferring types requires implementing unification with occur check, as shown for instance in the appendix of (Grygiel and Lescanne 2013), this is readily available in Prolog.

The predicate `boundTypeOf/3` works by associating the same logical variable, denoting its type, to each of its occurrences. As a unique logic variable is associated to each location corresponding via its de Bruijn index to the same binder, types are consistently inferred. This is ensured by the use of the built-in `nth0(I,Vs,V0)` that unifies `V0` with the `I`-th element of the list `Vs`. Note that unification with occurs-check needs to be used to avoid cycles in the inferred type formulas.

```
boundTypeOf(v(I),V,Vs):-
  nth0(I,Vs,V0),
  unify_with_occurs_check(V,V0).
boundTypeOf(a(A,B),Y,Vs):-
  boundTypeOf(A,(X->Y),Vs),
  boundTypeOf(B,X,Vs).
boundTypeOf(l(A),(X->Y),Vs):-
  boundTypeOf(A,Y,[X|Vs]).
```

At this point, most general types are inferred by `boundTypeOf` as fresh variables, similar to multi-parameter polymorphic types in functional languages, if one interprets logic variables as universally quantified.

Example 1
Type inferred for the `S` combinator $\lambda x0.\ \lambda x1.\ \lambda x2.((x0\ x2)\ (x1\ x2))$ in de Bruijn form.

```
?- X=l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),boundTypeOf(X,T,0).
X = l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),
T = ((A->B->C)-> (A->B)->A->C).
```

However, as we are only interested in simple types of closed terms with only one basic type, we will bind uniformly the leaves of our type tree to the constant "o" representing our only primitive type, by using the predicate `bindType/1`.

```
boundTypeOf(A,T):-boundTypeOf(A,T0,[]),bindType(T0),!,T=T0.

bindType(o):-!.
bindType((A->B)):-
  bindType(A),
  bindType(B).
```

Example 2
Simple type inferred for the `S` combinator and failure to assign a type to the `Y` combinator $\lambda x0.(\ \lambda x1.(x0\ (x1\ x1))\ \lambda x2.(x0\ (x2\ x2)))$.

```
?- boundTypeOf(l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),T).
T = ((o-> (o->o))-> ((o->o)-> (o->o))).
?- boundTypeOf(l(a(l(a(v(1), a(v(0), v(0)))),
                      l(a(v(1), a(v(0), v(0)))))),T).
false.
```

3 Deriving a generator for lambda terms in de Bruijn form

As a first step toward deriving a generator for lambda terms in de Bruijn form, we will describe a generator for Motzkin-trees.

3.1 Generation of Motzkin trees

Motzkin-trees (also called binary-unary trees) have internal nodes of arities 1 or 2. Thus they can be seen as an abstraction of lambda terms that ignores de Bruijn indices at the leaves. The predicate `motzkinTree/2` generates Motzkin trees with L internal and leaf nodes.

```
motzkinTree(L,Tree):-motzkinTree(Tree,L,0).

motzkinTree(u)-->down.
motzkinTree(l(A))-->down,motzkinTree(A).
motzkinTree(a(A,B))-->down,
  motzkinTree(A),
  motzkinTree(B).

down(From,To):-From>0,To is From-1.
```

Note the work of the predicate `down/2` that, in combination with Prolog's DCG notation, counts downward and left-to-right the nodes available to build the tree. One can also see it as abstracting away the predecessor operation. Motzkin-trees are counted by the sequence `A001006` in (Sloane 2014). If we replace the first clause of `motzkinTree/2` with `motzkinTree(u)-->[]`, we obtain binary-unary trees with `L` internal nodes, counted by the entry `A006318` (Large Schröder Numbers) of (Sloane 2014).

3.2 Generation of de Bruijn terms

We can derive a generator for closed lambda terms in de Bruijn form by extending the Motzkin-tree generator to keep track of the lambda binders. When reaching a leaf `v/1`, one of the available binders (expressed as a de Bruijn index) will be assigned to it nondeterministically.

The predicate `genDBterm/4` generates closed de Bruijn terms with a fixed number of internal (non-index) nodes, as counted by entry `A220894` in (Sloane 2014).

```
genDBterm(v(X),V)-->
  {down(V,V0)},
  {between(0,V0,X)}.
genDBterm(l(A),V)-->down,
  {up(V,NewV)},
  genDBterm(A,NewV).
genDBterm(a(A,B),V)-->down,
  genDBterm(A,V),
  genDBterm(B,V).

up(K1,K2):-K2 is K1+1.
```

The range of possible indices is provided by Prolog's built-in integer range generator `between/3`, that provides values from `0` to `V0`. Note also the use of `down/2` abstracting away the predecessor operation and `up/2` abstracting away the successor operation. Together, they control the amount of available nodes and the incrementing of de Bruijn indices at each lambda node.

Our generator of de Bruijn terms is exposed through two interfaces: `genDBterm/2` that generates closed de Bruijn terms with exactly `L` non-index nodes and `genDBterms/2` that generates terms with up to `L` non-index nodes, by not enforcing that exactly `L` internal nodes must be used.

```
genDBterm(L,T):-genDBterm(T,0,L,0).

genDBterms(L,T):-genDBterm(T,0,L,_).
```

Like in the case of Motzkin trees, inserting a `down` operation in the first clause of `genDBterm/4` will enumerate terms counted by sequence `A135501` in (Sloane 2014).

Example 3
Generation of terms with up to 2 internal nodes.

```
?- genDBterms(2,T).
T = l(v(0)) ;
T = l(l(v(0))) ;
```

```
T = l(l(v(1))) ;
T = l(a(v(0), v(0))).
```

4 Combining term generation and type inference

One could combine a generator for closed terms and a type inferrer in a “generate-and-test” style as follows:

```
genTypedTerm1(L,Term,Type):-
  genDBterm(L,Term),
  boundTypeOf(Term,Type).
```

Note that 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 efficient querying about *what terms inhabit a given type*, as one would expect from Prolog’s multi-directional execution model.

4.1 Generating closed well-typed terms of a given size

One can derive, from the type inferrer `boundTypeOf`, a more efficient generator for de Bruijn terms with a given number of internal nodes.

Like the predicate `motzkinTree`, the predicate `genTypedTerm/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 `nth0` that enumerates values for both `I` and `V` ranging over the list of available variables `Vs`, as well as the use of `unify_with_occurs_check` to ensure that unification of candidate types does not create cycles.

```
genTypedTerm(v(I),V,Vs)-->
  {
   nth0(I,Vs,V0),
   unify_with_occurs_check(V,V0)
  }.
genTypedTerm(a(A,B),Y,Vs)-->down,
  genTypedTerm(A,(X->Y),Vs),
  genTypedTerm(B,X,Vs).
genTypedTerm(l(A),(X->Y),Vs)-->down,
  genTypedTerm(A,Y,[X|Vs]).
```

Two interfaces are offered: `genTypedTerm` that generates de Bruijn terms of with exactly `L` internal nodes and `genTypedTerms` that generates terms with `L` internal nodes or less.

```
genTypedTerm(L,B,T):-
  genTypedTerm(B,T,[],L,0),
  bindType(T).

genTypedTerms(L,B,T):-
  genTypedTerm(B,T,[],L,_),
  bindType(T).
```

As expected, the number of solutions, computed as the sequence $1,2,9,40,238,1564,\ldots$ for sizes for sizes $1,2,3,\ldots$, matches entry `A220471` in (Sloane 2014).

Example 4
Generation of well-typed closed de Bruijn terms of size 3.

```
?- genTypedTerm(3,Term,Type).
Term = a(l(v(0)), l(v(0))),Type = (o->o) ;
Term = l(a(v(0), l(v(0)))),Type = (((o->o)->o)->o) ;
Term = l(a(l(v(0)), v(0))),Type = (o->o) ;
Term = l(a(l(v(1)), v(0))),Type = (o->o) ;
Term = l(l(a(v(0), v(1)))),Type = (o-> ((o->o)->o)) ;
Term = l(l(a(v(1), v(0)))),Type = ((o->o)-> (o->o)) ;
Term = l(l(l(v(0)))),Type = (o-> (o-> (o->o))) ;
Term = l(l(l(v(1)))),Type = (o-> (o-> (o->o))) ;
Term = l(l(l(v(2)))),Type = (o-> (o-> (o->o))) .
```

4.2 *Querying the generator for specific types*

Coming with Prolog's unification and non-deterministic search, is the ability to make more specific queries by providing a type pattern, that selects only terms that match it, while generating terms and inferring their types.

The predicate `queryTypedTerm` finds closed terms of a given type of size exactly `L`.

```
queryTypedTerm(L,QueryType,Term):-
  genTypedTerm(L,Term,QueryType),
  boundTypeOf(Term,QueryType).
```

Similarly, the predicate `queryTypedTerm` finds closed terms of a given type of size `L` or less.

```
queryTypedTerms(L,QueryType,Term):-
  genTypedTerms(L,Term,QueryType),
  boundTypeOf(Term,QueryType).
```

Note that giving the query type ahead of executing `genTypedTerm` would unify with more general "false positives", as type checking, contrary to type synthesis, proceeds bottom-up. This justifies filtering out the false positives simply by testing with the deterministic predicate `boundTypeOf` at the end. Despite the extra call to `boundTypeOf`, the performance improvements are significant, as shown in Figure 1. The figure also shows that when the slow generate-and-test predicate `genTypedTerm1` is used, the result (in "logical-inferences-per-second") does not depend on the pattern, contrary to the fast `queryTypedTerm` that prunes mismatching types while inferring the type of the terms as it generates them.

Example 5
Terms of type `o->o` of size 4.

```
?- queryTypedTerm(3,(o->o),Term).
Term = a(l(v(0)), l(v(0))) ;
Term = l(a(l(v(0)), v(0))) ;
Term = l(a(l(v(1)), v(0))) .

?- queryTypedTerms(12,(o->o)->o,T).
false.
```

Note that the last query, taking about a minute, shows that no closed terms of type `(o->o)->o` exist up to size 12.

Size	Slow `o->o`	Slow `o->o->o`	Fast `o->o`	Fast `o->o->o`	Fast `o`
1	39	39	38	27	15
2	126	126	60	109	36
3	552	552	240	200	88
4	3,108	3,108	634	1,063	290
5	21,840	21,840	3,213	3,001	1,039
6	181,566	181,566	12,721	19,598	4,762
7	1,724,131	1,724,131	76,473	81,290	23,142
8	18,307,585	18,307,585	407,639	584,226	133,554
9	213,940,146	213,940,146	2,809,853	3,254,363	812,730

Fig. 1. Performance of algorithms when querying generators with type patterns given in advance

4.3 Same-type siblings

Given a closed well-typed lambda term, we can ask what other terms of the same size or smaller share the same type. This can be interesting for finding possibly alternative implementations of a given function or for generation of similar siblings in genetic programming.

The predicate `typeSiblingOf` lists all the terms of the same or smaller size having the same type as a given term.

```
typeSiblingOf(Term,Sibling):-
  dbTermSize(Term,L),
  boundTypeOf(Term,Type),
  queryTypedTerms(L,Type,Sibling).
```

Example 6

```
?- typeSiblingOf(l(l(a(v(0),a(v(0),v(1))))),T).
T = l(l(a(v(0), v(1)))) ; % <= smaller sibling
T = l(l(a(v(0), a(v(0), v(1))))) .
```

5 Discovering frequently occurring type patterns

The ability to run "relational queries" about terms and their types extends to compute interesting statistics, giving a glimpse at their distribution.

5.1 The "Popular" type patterns

As types can be seen as an approximation of their inhabitants, we expect them to be shared among distinct terms. As we can enumerate all the terms for small sizes and infer their types, we would like to know what are the most frequently occurring ones. This can be meaningful as a comparison base for types that are used in human-written programs of comparable size. In approaches like (Palka et al. 2011), where types are used to direct the generation of random terms, focusing on the most frequent types might help with generation of more realistic random tests.

Figure 2 describes counts for terms and their types for small sizes. It also shows the first two most frequent types with the count of terms they apply to.

Figure 3 shows the "most popular types" for the about 1 million closed well-typed terms up to size 9 and the count of their inhabitants.

Term size	Types	Terms	Ratio	1-st frequent	2-nd frequent
1	1	1	1.0	1: `o->o`	
2	1	2	0.5	2: `o->o->o`	
3	5	9	0.555	3: `o->o->o->o`	3: `o->o`
4	16	40	0.4	14: `o->o->o`	4: `o->o->o->o->o`
5	55	238	0.231	38: `o->o->o->o`	31: `o->o`
6	235	1564	0.150	201: `o->o->o`	80: `o->o->o->o->o`
7	1102	11807	0.093	732: `o->o->o->o`	596: `o->o`
8	5757	98529	0.058	4632: `o->o->o`	2500: `o->o`
9	33251	904318	0.036	20214: `o->o->o->o`	19855: `(o->o)->o->o`

Fig. 2. Counts for terms and types for sizes 1 to 9 and the first two most frequent types

Count	Type
23095	`o->o->o`
22811	`(o->o)->o->o`
22514	`o->o->o->o`
21686	`o->o`
18271	`o-> (o->o)->o`
14159	`(o->o)->o->o->o`
13254	`((o->o)->o)-> (o->o)->o`
12921	`o-> (o->o)->o->o`
11541	`(o->o)-> ((o->o)->o)->o`
10919	`(o->o->o)->o->o->o`

Fig. 3. Most frequent types, out of a total of `33972` distinct types, of `1016508` terms up to size 9.

We can observe that, like in some human-written programs, functions representing binary operations of type `o->o->o` are the most popular. Ternary operations `o->o->o->o` come third and unary operations `o->o` come fourth. Somewhat surprisingly, a higher order function type `(o->o)->o->o` applying a function to an argument to return a result comes second and multi-argument variants of it are also among the top 10.

5.2 Growth sequences of some popular types

We can make use of our generator's efficient specialization to a given type to explore empirical estimates for some types interesting to human programmers.

Contrary to the total absence of the type `(o->o)->o` among terms of size up to 12, "binary operations" of type `o->(o->o)` turn out to be quite frequent, giving, by increasing sizes, the sequence [0, 2, 0, 14, 12, 201, 445, 4632, 17789, 158271, 891635].

Transformers of type `o->o`, by increasing sizes, give the sequence [1, 0, 3, 3, 31, 78, 596, 2500, 18474, 110265]. While type `(o->o)->o` turns our to be absent up to size 12, the type `(o->o)->(o->o)`, describing *transformers of transformers* turns out to be quite popular, as shown by the sequence [0, 0, 1, 1, 18, 52, 503, 2381, 19855, 125599]. The same turns out to be true also for `(o->o)->((o->o)->(o->o))`, giving [0, 0, 0, 0, 2, 6, 96, 505, 5287, 36769] and `((o->o)->(o->o)) -> ((o->o)->(o->o))` giving [0, 0, 0, 0, 0, 6, 23, 432, 2450, 29924]. One might speculate that homotopy type theory (The Univalent Foundations Program 2013), that focuses on such transformations and transformations of transformations etc. has a rich population of lambda terms from which to chose interesting inhabitants of such types!

Another interface, generating closed simply-typed terms of a given size, restricted to have at most a given number of free de Bruijn indices, is implemented by the predicate `genTypedWithSomeFree`.

```
genTypedWithSomeFree(Size,NbFree,B,T):-
   between(0,NbFree,NbVs),
   length(FreeVs,NbVs),
   genTypedTerm(B,T,FreeVs,Size,0),
   bindType(T).
```

The first 9 numbers counting closed simply-typed terms with at most one free variable (not yet in (Sloane 2014)), are [3, 10, 45, 256, 1688, 12671, 105743, 969032, 9639606].

Note that, as our generator performs the early pruning of untypable terms, rather than as a post-processing step, enumeration and counting of these terms happens in a few seconds.

6 Generating closed typable lambda terms by types

In (Palka et al. 2011) 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 GHC bugs.

We can organize in a similar way the interface of our combined generator and type inferrer.

6.1 Generating type trees

The predicate genType generates binary trees representing simple types with a single base type `‘‘o’’`.

```
genType(o)-->[].
genType((X->Y))-->down,
  genType(X),
  genType(Y).
```

It provides two interfaces, for generating types of exactly size N or up to size N.

```
genType(N,X):-genType(X,N,0).

genTypes(N,X):-genType(X,N,_).
```

Example 7
Type trees with up to 2 internal nodes (and up to 4 leaves).

```
?- genTypes(3,T).
?- genTypes(2,T).
T = o ;
T = (o->o) ;
T = (o->o->o) ;
T = ((o->o)->o) .
```

Next, we will combine this type generator with the generator that efficiently produces terms matching each type pattern.

6.2 Generating lambda terms by increasing type sizes

The predicate genByType first generates types (seen simply as binary trees) with genType and then uses the unification-based querying mechanism to generate all closed well-typed de Bruijn terms with fewer internal nodes then their binary tree type.

```
genByType(L,B,T):-
  genType(L,T),
  queryTypedTerms(L,T,B).
```

Example 8
Enumeration of closed simply-typed de Bruijn terms with types of size 3 and terms of a given type with at most 3 internal nodes.

```
?- genByType(3,B,T).
B = l(l(l(v(0)))),T = (o->o->o->o) ;
B = l(l(l(v(1)))),T = (o->o->o->o) ;
B = l(l(l(v(2)))),T = (o->o->o->o) ;
B = l(l(a(v(0), v(1)))),T = (o-> (o->o)->o) ;
B = l(l(a(v(1), v(0)))),T = ((o->o)->o->o) ;
B = l(a(v(0), l(v(0)))),T = (((o->o)->o)->o) .
```

Given that various constraints are naturally interleaved by our generator we obtain in a few seconds the sequence counting these terms having types up to size 8, [1, 2, 6, 18, 84, 376, 2344, 15327]. Intuitively this means that despite of their growing sizes, types have an increasingly large number of inhabitants of sizes smaller than their size. This is somewhat contrary to what we see in human-written code, where types are almost always simpler and smaller than the programs inhabiting them.

7 Related work

The classic reference for lambda calculus is (Barendregt 1984). Various instances of typed lambda calculi are overviewed in (Barendregt 1991).

Originally introduced in (Bruijn 1972), the de Bruijn notation makes terms equivalent up to α-conversion and facilitates their normalization (Kamareddine 2001).

Combinatorics of lambda terms, including enumeration, random generation and asymptotic behavior has seen an increased interest recently (see for instance (David et al. 2009; Bodini et al. 2011; Grygiel and Lescanne 2013; David et al. 2010; Grygiel et al. 2013)), partly motivated by applications to software testing, given the widespread use of lambda terms as an intermediate language in compilers for functional languages and proof assistants. In (Palka et al. 2011), types are used to generate random terms for software testing. The same naturally "goal-oriented" effect is obtained in the generator/type inferrer for de Bruijn terms in subsection 4.2, by taking advantage of Prolog's ability to backtrack over possible terms, while filtering against unification with a specific pattern.

Of particular interest are the results of (Grygiel and Lescanne 2013) where recurrence relations and asymptotic behavior are studied for several families of lambda terms. Empirical evaluation of the density of closed simply-typed general lambda terms described in (Grygiel and Lescanne 2013) indicates extreme sparsity for large sizes. However, the problem of their exact asymptotic behavior is still open.

8 Conclusions

We have described Prolog-based term and type generation and as well as type-inference algorithms for de Bruijn terms. Among the possible applications of our techniques we mention compilation and test generation for lambda-calculus based languages and proof assistants.

By taking advantage of Prolog's unique bidirectional execution model and unification algorithm (including support for cyclic terms and occurs-check), we have merged generation and type inference in an algorithm that can build "customized closed terms of a given type". This "relational view" of terms and their types has enabled the discovery of interesting patterns about the type expressions occurring in well-typed programs. We have uncovered the most "popular" types that govern function applications among a about a million small-sized lambda terms.

The paper has also introduced a number of algorithms that, at our best knowledge, are novel, at least in terms of their logic programming implementation, among which we mention the type inference for de Bruijn terms using unification with occurs-check in subsection 2.2 and the integrated generation and type inference algorithm for closed simply typed de Bruijn terms in section 4. Besides the ability to efficiently query for inhabitants of specific types, our algorithms also support a from of "query-by-example" mechanism, for finding (possibly smaller) terms inhabiting the same type as the query term.

We have also observed some interesting phenomena about frequently occurring types, that seem to be similar to those in human-written programs and we have computed growth sequences for the number of inhabitants of some "popular" types, for which we have not found any study in the literature.

While a non-strict functional language like Haskell could have been used for deriving similar algorithms, the synergy between Prolog's non-determinism, DCG transformation and the availability unification with occurs-check made the code embedded in the paper significantly simpler and arguably clearer.

Acknowledgement

This research has been supported by NSF grant `1423324`.

References

BARENDREGT, H. P. 1984. *The Lambda Calculus Its Syntax and Semantics*, Revised ed. Vol. 103. North Holland.

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

BODINI, O., GARDY, D., AND GITTENBERGER, B. 2011. Lambda-terms of bounded unary height. In *ANALCO*. SIAM, 23–32.

BRUIJN, N. G. D. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae 34*, 381–392.

DAVID, R., GRYGIEL, K., KOZIK, J., RAFFALLI, C., THEYSSIER, G., AND ZAIONC, M. 2010. Asymptotically almost all λ-terms are strongly normalizing. *Preprint: arXiv: math. LO/0903.5505 v3*.

DAVID, R., RAFFALLI, C., THEYSSIER, G., GRYGIEL, K., KOZIK, J., AND ZAIONC, M. 2009. Some properties of random lambda terms. *Logical Methods in Computer Science 9,* 1.

GRYGIEL, K., IDZIAK, P. M., AND ZAIONC, M. 2013. How big is BCI fragment of BCK logic. *J. Log. Comput. 23,* 3, 673–691.

GRYGIEL, K. AND LESCANNE, P. 2013. Counting and generating lambda terms. *J. Funct. Program. 23,* 5, 594–628.

HINDLEY, J. R. AND SELDIN, J. P. 2008. *Lambda-calculus and combinators: an introduction*. Vol. 13. Cambridge University Press Cambridge.

KAMAREDDINE, F. 2001. Reviewing the Classical and the de Bruijn Notation for calculus and Pure Type Systems. *Journal of Logic and Computation 11,* 3, 363–394.

PALKA, M. H., CLAESSEN, K., RUSSO, A., AND HUGHES, J. 2011. 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, 91–97.

SLOANE, N. J. A. 2014. The On-Line Encyclopedia of Integer Sequences. Published electronically at https://oeis.org/.

THE UNIVALENT FOUNDATIONS PROGRAM. 2013. *Homotopy Type Theory*. Institute of Advanced Studies, Princeton. `http://homotopytypetheory.org/2013/06/20/the-hott-book/`.