On logic programming representations of lambda terms: de Bruijn indices, compression, type inference, combinatorial generation, normalization

Paul Tarau

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

Abstract. We introduce a compressed de Bruijn representation of lambda terms and define its bijections to standard representations. Compact combinatorial generation algorithms are given for several families of lambda terms, including open, closed, simply typed and linear terms as well as type inference and normal order reduction algorithms. We specify our algorithms as a literate Prolog program. In the process, we rely in creative ways on unification of logic variables, cyclic terms, backtracking and definite clause grammars.
Keywords: *lambda calculus, de Bruijn indices, lambda term compression, type inference, normalization, combinatorics of lambda terms.*

1 Introduction

Lambda terms [1] 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. Generation of lambda terms 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.

Prolog's underlying backtracking and unification make it an ideal tool for defining compact combinatorial generation algorithms for various families of lambda terms. Of particular interest are representations that are canonical up to alpha-conversion (variable renamings) among which the most well-known ones are de Bruijn's indices [2], representing bound variables as the number of binders to traverse to the lambda abstraction binding them.

However, a sequence of binders in de Bruijn notation, can be seen as a natural number expressed in unary notation. This suggests introducing a compressed representation of the binders that puts in a new light the underlying combinatorial structure of lambda terms and highlights their connection to the *Catalan* *family* of combinatorial objects [3], among which binary trees are the most well known. The proposed compressed de Bruijn notation also simplifies generation of some families of lambda terms.

At the same time, the use of Prolog's unification of logic variables is instrumental in designing compact algorithms for inferring simple types or for generating linear, linear affine or lambda terms with bounded unary height as well as in implementing normalization algorithms.

To be able to use the most natural representation for each of the proposed algorithms, we implement bijective transformations between lambda terms in standard as well as de Bruijn and compressed de Bruijn representation.

The paper is organized as follows. Section 2 introduces the compressed de Bruijn terms and bijective transformations from them to standard lambda terms. Section 3 describes generation of binary trees and mappings from lambda terms to binary trees representing their inferred types and and their applicative skeletons. Section 4 describes generators for several classes of lambda terms, including closed, simply typed, linear, affine as well as terms with bounded unary height and terms in the binary lambda calculus encoding. Section 5 describes a normal order reduction algorithm for lambda terms relaying on their de Bruijn representation. Section 6 discusses related work and section 7 concludes the paper.

The paper is structured as a literate Prolog program. The code has been tested with SWI-Prolog 6.6.6 and YAP 6.3.4. It is also available as a separate file at `http://www.cse.unt.edu/~tarau/research/2015/dbx.pro`.

2 A compressed de Bruijn representation of lambda terms

.

We represent standard lambda terms [1] in Prolog using the constructors `a/2` for applications and `l/2` for lambda abstractions. Variables bound by the lambdas as well as their occurrences are represented as *logic variables*. As an example, the lambda term $\lambda x0.(\lambda x1.(x0\ (x1\ x1))\ \lambda x2.(x0\ (x2\ x2)))$ will be represented as `l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C)))))`.

2.1 De Bruijn Indices

De Bruijn indices [2] 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))))))`, corresponding to the fact 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`.

From de Bruijn to lambda terms with canonical names The predicate `b2l` converts from the de Bruijn representation to lambda terms whose canonical names are provided by logic variables. We will call them terms in *standard notation.*

```
b2l(DeBruijnTerm,LambdaTerm):-b2l(DeBruijnTerm,LambdaTerm,_Vs).

b2l(v(I),V,Vs):-nth0(I,Vs,V).
b2l(a(A,B),a(X,Y),Vs):-b2l(A,X,Vs),b2l(B,Y,Vs).
b2l(l(A),l(V,Y),Vs):-b2l(A,Y,[V|Vs]).
```

Note the use of the built-in `nth0/3` that associates to an index `I` a variable `V` on the list `Vs`. As we initialize in `b2l/2` the list of logic variables as a free variable `_Vs`, free variables in open terms, represented with indices larger than the number of available binders will also be consistently mapped to logic variables. By replacing `_Vs` with `[]` in the definition of `b2l/2`, one could enforce that only closed terms (having no free variables) are accepted.

From lambda terms with canonical names to de Bruijn terms Logic variables provide canonical names for lambda variables. An easy way to manipulate them at meta-language level is to turn them into special "$VAR/1" terms - a mechanism provided by Prolog's built-in `numbervars/3` predicate. Given that "$VAR/1" is distinct from the constructors lambda terms are built from (`l/2` and `a/2`), this is a safe (and invertible) transformation. To avoid any side effect on the original term, in the predicate `l2b/2` that inverts `b2l/2`, we will uniformly rename its variables to fresh ones with Prolog's `copy_term/2` built-in. We will adopt this technique through the paper each time our operations would mutate an input argument otherwise.

```
l2b(StandardTerm,DeBruijnTerm):-
  copy_term(StandardTerm,Copy),
  numbervars(Copy,0,_),
  l2b(Copy,DeBruijnTerm,_Vs).

l2b('$VAR'(V),v(I),Vs):-once(nth0(I,Vs,'$VAR'(V))).
l2b(a(X,Y),a(A,B),Vs):-l2b(X,A,Vs),l2b(Y,B,Vs).
l2b(l(V,Y),l(A),Vs):-l2b(Y,A,[V|Vs]).
```

Note the use of `nth0/3`, this time to locate the index `I` on the (open) list of variables `_Vs`. By replacing `_Vs` with `[]` in the call to `l2b/3`, one can enforce that only closed terms are accepted.

Example 1 *illustrates the bijection defined by predicates* `l2b` *and* `b2l`*.*

```
?- LT=l(A,l(B,l(C,a(a(A,C),a(B,C))))),l2b(LT,BT),b2l(BT,LT1),LT=LT1.
LT = LT1, LT1 = l(A, l(B, l(C, a(a(A, C), a(B, C))))),
BT = l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))).
```

2.2 Going one step further: compressing the blocks of lambdas

Iterated lambdas (represented as a block of constructors `l/1` in the de Bruijn notation) can be seen as a successor arithmetic representation of a number that counts them. So it makes sense to represent that number more efficiently in the usual binary notation. Note that in de Bruijn notation blocks of lambdas can wrap either applications or variable occurrences represented as indices. This suggests using just two constructors: `v/2` indicating in a term `v(K,N)` that we have `K` lambdas wrapped around variable `v(N)` and `a/3`, indicating in a term `a(K,X,Y)` that `K` lambdas are wrapped around the application `a(X,Y)`.

We call the terms built this way with the constructors `v/2` and `a/3` *compressed de Bruijn terms*.

2.3 Converting between representations

We can make precise the definition of compressed deBruijn terms by providing a bijective transformation between them and the usual de Bruijn terms.

From de Bruijn to compressed The predicate `b2c` converts from the usual de Bruijn representation to the compressed one. It proceeds by case analysis on `v/1, a/2, l/1` and counts the binders `l/1` as it descends toward the leaves of the tree. Its steps are controlled by the predicate `up/2` that increments the counts when crossing a binder.

```
b2c(v(X),v(0,X)).
b2c(a(X,Y),a(0,A,B)):-b2c(X,A),b2c(Y,B).
b2c(l(X),R):-b2c1(0,X,R).

b2c1(K,a(X,Y),a(K1,A,B)):-up(K,K1),b2c(X,A),b2c(Y,B).
b2c1(K, v(X),v(K1,X)):-up(K,K1).
b2c1(K,l(X),R):-up(K,K1),b2c1(K1,X,R).

up(From,To):-From>=0,To is From+1.
```

From compressed to de Bruijn The predicate `c2b` converts from the compressed to the usual de Bruijn representation. It reverses the effect of `b2c` by expanding the `K` in `v(K,N)` and `a(K,X,Y)` into `K` `l/1` binders (no binders when `K=0`). The predicate `iterLam/3` performs this operation in both cases, and the predicate `down/2` computes the decrements at each step. We will reuse the predicates `up/2` and `down/2` that can be seen as abstracting away the successor/predecessor operation.

```
c2b(v(K,X),R):-X>=0,iterLam(K,v(X),R).
c2b(a(K,X,Y),R):-c2b(X,A),c2b(Y,B),iterLam(K,a(A,B),R).

iterLam(0,X,X).
iterLam(K,X,l(R)):-down(K,K1),iterLam(K1,X,R).

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

Example 2 *illustrates the bijection defined by the predicates* `b2c` *and* `c2b`*.*

```
?- BT=l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),b2c(BT,CT),c2b(CT,BT1).
BT = BT1, BT1 = l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),
CT = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))) .
```

A convenient way to simplify defining chains of such conversions is by using Prolog's DCG transformation. For instance, the predicate `c2l/2` (which expands to something like `c2l(X,Z):-c2b(X,Y),b2l(Y,Z)`), converts from compressed de Bruijn terms and standard lambda terms using de Bruijn terms as an intermediate step, while `l2c/2` works the other way around.

```
c2l --> c2b,b2l.
l2c --> l2b,b2c.
```

2.4 Open and closed terms

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.

Closed terms can be easily identified by ensuring that the lambda binders on a given path from the root outnumber the de Bruijn index of a variable occurrence ending the path. The predicate `isClosed` does that for compressed de Bruijn terms.

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

isClosed(v(K,N),S):-N<S+K.
isClosed(a(K,X,Y),S1):-S2 is S1+K,isClosed(X,S2),isClosed(Y,S2).
```

3 Binary trees, lambda terms and and types

We can see our compressed de Bruijn terms as binary trees decorated with integer labels. The binary trees provide a skeleton that describes the applicative structure of the underlying lambda terms. At the same time, types in the *simple typed lambda calculus* [4] share a similar binary tree structure.

Binary trees are among the most well-known members of the Catalan family of combinatorial objects [3], that has at least 58 structurally distinct members, covering several data structures, geometric objects and formal languages.

Generating binary trees We will build binary trees with the constructor `->/2` for branches and the constant `o` for its leaves. This will match the usual notation for simple types [4] of lambda terms that can be represented as binary trees.

A generator / recognizer of binary trees of a fixed size (seen as the number of internal nodes, counted by entry `A000108` in [5]) is defined by the predicate `scat/2`.

```
scat(N,T):-scat(T,N,0).

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

Note the creative use of Prolog's DCG-grammar transformation. After the DCG expansion, the code for `scat/3` becomes something like:

```
scat(o,K,K).
scat((X->Y),K1,K3):-down(K1,K2),scat(X,K2,K3),scat(K3,K4).
```

Given that down(K1,K2) unfolds to `K1>0,K2 is K1-1` it is clear that this code ensures that the total number of nodes `N` passed by `scat/2` to `scat/3` controls the size of the generated trees. We will reuse this pattern through the paper, as it simplifies the writing of generators for various combinatorial objects. It is also convenient to standardize on the number of *internal nodes* as defining the *size* of our terms.

Example 3 *illustrates trees with 3 internal nodes (built with the constructor "*`->/2`*") generated by* `scat/2`*.*

```
?- scat(3,BT).
BT = (o->o->o->o) ;
BT = (o-> (o->o)->o) ;
BT = ((o->o)->o->o) ;
BT = ((o->o->o)->o) ;
BT = (((o->o)->o)->o) .
```

Note the right associative constructor "`->`" reducing the use of parentheses.

3.1 Type Inference with logic variables

Simple types, represented as binary trees built with the constructor "`->/2`" with empty leaves representing the unique primitive type "`o`", can be seen as a "Catalan approximation" of lambda terms, centered around ensuring their safe and terminating evaluation (strong normalization).

While in a functional language inferring types requires implementing unification with occur check, as shown for instance in [6], this operation is available in Prolog as a built-in. Also a "post-mortem" verification that unification has not introduced any cycles is provided by the built-in `acyclic_term/1`.

The predicate `extractType/2` works by turning each logical variable `X` into a pair `_:TX`, where `TX` is a fresh variable denoting its type. As logic variable bindings propagate between binders and occurrences, this ensures that types are consistently inferred.

```
extractType(_:TX,TX):-!. % this matches all variables
extractType(l(_:TX,A),(TX->TA)):-extractType(A,TA).
extractType(a(A,B),TY):-extractType(A,(TX->TY)),extractType(B,TX).
```

Instead of (inefficiently) using unification with occur-check at each step, we ensure that at the end, our term is still *acyclic*, by using the built-in ISO-Prolog predicate `acyclic_term/1`.

```
hasType(CTerm,Type):-
  c2l(CTerm,LTerm),
  extractType(LTerm,Type),
  acyclic_term(LTerm),
  bindType(Type).
```

At this point, most general types are inferred by `extractType` as fresh variables, somewhat similar to multi-parameter polymorphic types in functional languages, if one interprets logic variables as universally quantified. However, as we are only interested in simple types, 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`.

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

We can also define the predicate `typeable/1` that checks if a lambda term is well typed, by trying to infer and then ignoring its inferred type.

```
typeable(Term):-hasType(Term,_Type).
```

Example 4 *illustrates typability of the term corresponding to the* `S` *combinator* $\lambda x0.\ \lambda x1.\ \lambda x2.((x0\ x2)\ (x1\ x2))$ *and untypabilty of the term corresponding to the* `Y` *combinator* $\lambda x0.(\ \lambda x1.(x0\ (x1\ x1))\ \ \lambda x2.(x0\ (x2\ x2)))$*, in de Bruijn form.*

```
?- hasType(a(3,a(0,v(0,2),v(0,0)),a(0,v(0,1),v(0,0))),T).
T = ((o->o->o)-> (o->o)->o->o).
?- hasType(
   a(1,a(1,v(0,1),a(0,v(0,0),v(0,0))),a(1,v(0,1),a(0,v(0,0),v(0,0)))),T).
false.
```

4 Generating special classes of lambda terms

To generate lambda terms of a given size, we can write generators similar to the ones for binary trees in section 3. Moreover, we have the choice to use generators for standard, de Bruijn or compressed de Bruijn terms and then bijectively morph the resulting terms in the desired representation, as outlined is section 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 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,T):-motzkinTree(T,L,0).

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

Motzkin-trees are counted by the sequence `A001006` in [5]. 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 [5].

4.1 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 `genDB/4` generates closed de Bruijn terms with a fixed number of internal (non-index) nodes, as counted by entry `A220894` in [5].

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

The range of possible indices is provided by Prolog's built-in integer range generator `between/3` that provides values from `0` to `V0`.

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

```
genDB(L,T):-genDB(T,0,L,0).
genDBs(L,T):-genDB(T,0,L,_).
```

Like in the case of Motzkin trees, a slight modification of the first clause of `genDB/4` will enumerate terms counted by sequence `A135501` in [5].

Example 5 *illustrates the generation of terms with up to* 2 *internal nodes.*

```
?- genDBs(2,T).
T = l(v(0)) ;
T = l(l(v(0))) ;
T = l(l(v(1))) ;
T = l(a(v(0), v(0))) ;
```

4.2 Generators for closed terms in compressed de Bruijn form

A generator for compressed de Bruijn terms can be derived by using `DCG` syntax to compose a generator for closed de Bruijn terms `genDB` and `genDBs` and a transformer to compressed terms `b2c/2`.

```
genCompressed --> genDB,b2c.
genCompresseds--> genDBs,b2c.
```

4.3 Generators for closed terms in standard notation

```
genStandard-->genDB,b2l.
genStandards-->genDBs,b2l.
```

Example 6 *illustrates generators for closed terms in compressed de Bruijn and standard notation with logic variables providing lambda variable names.*

```
?- genCompressed(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)).

?- genStandard(2,T).
T = l(_G3434, l(_G3440, _G3440)) ;
T = l(_G3434, l(_G3440, _G3434)) ;
T = l(_G3437, a(_G3437, _G3437)).
```

4.4 Generating closed lambda terms in standard notation

With logic variables representing binders and their occurrences, one can also generate lambda terms in standard notation directly. The predicate `genLambda/2` equivalent to `genStandard/2`, builds a list of logic variables as it generates binders. When generating a leaf, it picks nondeterministically one of the binders among the list of binders available, `Vs`. As usual, the predicate `down/2` controls the number of internal nodes.

```
genLambda(L,T):-genLambda(T,[],L,0).

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

4.5 Generating typeable terms

The predicate `genTypeable/2` generates closed typeable terms of size `L`. These are counted by entry `A220471` in [5].

```
genTypeable(L,T):-genCompressed(L,T),typeable(T).
genTypeables(L,T):-genCompresseds(L,T),typeable(T).
```

Example 7 *illustrates a generator for closed typeable terms.*

```
?- genCompressed(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)).
```

4.6 Generating normal forms

Normal forms are lambda terms that cannot be further reduced. A normal form should not be an application with a lambda as its left branch and, recursively, its subterms should also be normal forms. The predicate `nf/4` defines this inductively and generates all normal forms with `L` internal nodes in de Bruijn form.

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

As we standardize our generators to produce compressed de Bruijn terms, we combine `nf/4` and the converter `b2c/2` to produce normal forms of size exactly `L` (predicate `nf/2`) and with size up to `L` (predicate `nfs/2`).

```
nf(L,T):-nf(B,0,L,0),b2c(B,T).
nfs(L,T):-nf(B,0,L,_),b2c(B,T).
```

Example 8 *illustrates normal forms with exactly 2 non-index nodes.*

```
?- nf(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)) .
```

The number of solutions of our generator replicates entry `A224345` in [5] that counts closed normal forms of various sizes.

4.7 Generation of linear lambda terms

Linear lambda terms [7] restrict binders to *exactly one* occurrence.

The predicate `linLamb/4` uses logic variables both as leaves and as lambda binders and generates terms in standard form. In the process, binders accumulated on the way down from the root, must be split between the two branches of an application node. The predicate `subset_and_complement_of/3` achieves this by generating all such possible splits of the set of binders.

```
linLamb(X,[X])-->[].
linLamb(l(X,A),Vs)-->down,linLamb(A,[X|Vs]).
linLamb(a(A,B),Vs)-->down,
  {subset_and_complement_of(Vs,As,Bs)},
  linLamb(A,As),linLamb(B,Bs).
```

At each step of `subset_and_complement_of/3`, `place_element/5` is called to distribute each element of a set to exactly one of two disjoint subsets.

```
subset_and_complement_of([],[],[]).
subset_and_complement_of([X|Xs],NewYs,NewZs):-
  subset_and_complement_of(Xs,Ys,Zs),
  place_element(X,Ys,Zs,NewYs,NewZs).
place_element(X,Ys,Zs,[X|Ys],Zs).
place_element(X,Ys,Zs,Ys,[X|Zs]).
```

As usual, we standardize the generated terms by converting them with `l2c` to compressed de Bruijn terms.

```
linLamb(L,CT):-linLamb(T,[],L,0),l2c(T,CT).
```

Example 9 *illustrates linear lambda terms for* `L=3`*.*

```
?- linLamb(3,T).
T = a(2, v(0, 1), v(0, 0)) ;
T = a(2, v(0, 0), v(0, 1)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = a(1, v(1, 0), v(0, 0)) ;
T = a(0, v(1, 0), v(1, 0)) .
```

4.8 Generation of affine linear lambda terms

Linear affine lambda terms [7] restrict binders to *at most one* occurrence.

```
afLinLamb(L,CT):-afLinLamb(T,[],L,0),l2c(T,CT).

afLinLamb(X,[X|_])-->[].
afLinLamb(l(X,A),Vs)-->down,afLinLamb(A,[X|Vs]).
afLinLamb(a(A,B),Vs)-->down,
  {subset_and_complement_of(Vs,As,Bs)},
  afLinLamb(A,As),afLinLamb(B,Bs).
```

Example 10 *illustrates generation of affine linear lambda terms in compressed de Bruijn form.*

```
?- afLinLamb(3,T).
T = v(3, 0) ;
T = a(2, v(0, 1), v(0, 0)) ;
T = a(2, v(0, 0), v(0, 1)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = a(1, v(1, 0), v(0, 0)) ;
T = a(0, v(1, 0), v(1, 0)) ;
```

Clearly all linear terms are affine. It is also known that all affine terms are typeable.

4.9 Generating lambda terms of bounded unary height

Lambda terms of bounded unary height are introduced in [8] where it is argued that such terms are naturally occurring in programs and it is shown that their asymptotic behavior is easier to study.

They are specified by giving a bound on the number of lambda binders from a de Bruijn index to the root of the term.

```
boundedUnary(v(X),V,_D)-->{down(V,V0),between(0,V0,X)}.
boundedUnary(l(A),V,D1)-->down,
  {down(D1,D2),up(V,NewV)},
  boundedUnary(A,NewV,D2).
boundedUnary(a(A,B),V,D)-->down,
  boundedUnary(A,V,D),boundedUnary(B,V,D).
```

The predicate `boundedUnary/5` generates lambda terms of size `L` in compressed de Bruijn form with unary hight `D`.

```
boundedUnary(D,L,T):-boundedUnary(B,0,D,L,0),b2c(B,T).
boundedUnarys(D,L,T):-boundedUnary(B,0,D,L,_),b2c(B,T).
```

Example 11 *illustrates terms of unary height 1 with size up to 3.*

```
?- boundedUnarys(1,3,R).
R = v(1, 0) ;
R = a(1, v(0, 0), v(0, 0)) ;
R = a(1, v(0, 0), a(0, v(0, 0), v(0, 0))) ;
R = a(1, a(0, v(0, 0), v(0, 0)), v(0, 0)) ;
R = a(0, v(1, 0), v(1, 0)) .
```

4.10 Generating terms in binary lambda calculus encoding

Generating de Bruijn terms based on the size of their binary lambda calculus encoding [9] works by using a DCG mechanism to build the actual code as a list `Cs` of `0` and `1` digits and specifying the size of the code in advance.

```
blc(L,T,Cs):-length(Cs,L),blc(B,0,Cs,[]),b2c(B,T).

blc(v(X),V)-->{between(1,V,X)},encvar(X).
blc(l(A),V)-->[0,0],{NewV is V+1},blc(A,NewV).
blc(a(A,B),V)-->[0,1],blc(A,V),blc(B,V).
```

Note that de Bruijn binders are encoded as `00`, applications as `01` and de Bruijn indices in unary notation are encoded as `00...01`. This operation is preformed by the predicate `encvar/3`, that, in DCG notation, uses `down/2` at each step to generate the sequence of `1` terminated `0` digits.

```
encvar(0)-->[0].
encvar(N)-->{down(N,N1)},[1],encvar(N1).
```

Example 12 *illustrates generation of 8-bit binary lambda terms (*`Cs`*) together with their compressed de Bruijn form (*`T`*).*

```
?- blc(8,T,Cs).
T = v(3, 1),
Cs = [0, 0, 0, 0, 0, 0, 1, 0] ;
T = a(1, v(0, 1), v(0, 1)),
Cs = [0, 0, 0, 1, 1, 0, 1, 0] .
```

Note that while not bijective, the binary encoding has the advantage of being a self-delimiting code. This facilitates its use in an unusually compact interpreter.

5 Normalization of lambda terms

Evaluation of lambda terms involves *β-reduction*, a transformation of a term like `a(l(X,A),B)` by replacing every occurrence of `X` in `A` by `B`, under the assumption that `X` does not occur in `B` and *η-conversion*, the transformation of an application term `a(l(X,A),X)` into `A`, under the assumption that X does not occur in `A`.

The first tool we need to implement normalization of lambda terms is a safe substitution operation. In lambda-calculus based functional languages this can be achieved through a HOAS (Higher-Order Abstract Syntax) mechanism, that borrows the substitution operation from the underlying "meta-language". To this end, lambdas are implemented as functions which get executed (usually lazily) when substitutions occur. We refer to [10] for the original description of this mechanism, widely used these days for implementing embedded domain specific languages and proof assistants in languages like Haskell or ML.

While logic variables offer a fast and easy way to perform *substitutions*, they do not offer any elegant mechanism to ensure that substitutions are *capture-free*. Moreover, no HOAS-like mechanism exists in Prolog for borrowing anything close to *normal order reduction* from the underlying system, as Prolog would provide, through meta-programming, only a *call-by-value* model.

We will devise here a simple and safe interpreter for lambda terms supporting normal order β-reduction by using de Bruijn terms, which also ensures that terms are unique up to α-equivalence. As usual, we will omit η-conversion, known to interfere with things like type inference, as the redundant argument(s) that it removes might carry useful type information.

The predicate `beta/3` implements the β-conversion operation corresponding to the binder `l(A)`. It calls `subst/4` that replaces in `A` occurrences corresponding the the binder `l/1`.

```
beta(l(A),B,R):-subst(A,0,B,R).
```

The predicate `subst/4` counts, starting from `0` the lambda binders down to an occurrence `v(N)`. Replacement occurs at at level `I` when `I=N`.

```
subst(a(A1,A2),I,B,a(R1,R2)):-I>=0,
  subst(A1,I,B,R1),
  subst(A2,I,B,R2).
subst(l(A),I,B,l(R)):-I>=0,I1 is I+1,subst(A,I1,B,R).
subst(v(N),I,_B,v(N1)):-I>=0,N>I,N1 is N-1.
subst(v(N),I,_B,v(N)):-I>=0,N<I.
subst(v(N),I,B,R):-I>=0,N=:=I,shift_var(I,0,B,R).
```

When the right occurrence `v(N)` is reached, the term substituted for it is shifted such that its variables are marked with the new, incremented distance to their binders. The predicate `shift_var/4` implements this operation.

```
shift_var(I,K,a(A,B),a(RA,RB)):-K>=0,I>=0,
  shift_var(I,K,A,RA),
  shift_var(I,K,B,RB).
shift_var(I,K,l(A),l(R)):-K>=0,I>=0,K1 is K+1,shift_var(I,K1,A,R).
shift_var(I,K,v(N),v(M)):-K>=0,I>=0,N>=K,M is N+I.
shift_var(I,K,v(N),v(N)):-K>=0,I>=0,N<K.
```

Normal order evaluation of a lambda term, if it terminates, leads to a unique normal form, as a consequence of the Church-Rosser theorem, elegantly proven in [2] using de Bruijn terms. Termination holds, for instance, in the case of simply typed lambda terms. Its implementation is well known; we will follow here the algorithm described in [11]. We first compute the *weak head normal form* using `wh_nf/2`.

```
wh_nf(v(X),v(X)).
wh_nf(l(E),l(E)).
wh_nf(a(X,Y),Z):-wh_nf(X,X1),wh_nf1(X1,Y,Z).
```

The predicate `wh_nf1/3` does the case analysis of application terms `a/2`. The key step is the β-reduction in its second clause, when it detects an "eliminator" lambda expression as its left argument, in which case it performs the substitution of its binder, with its right argument.

```
wh_nf1(v(X),Y,a(v(X),Y)).
wh_nf1(l(E),Y,Z):-beta(l(E),Y,NewE),wh_nf(NewE,Z).
wh_nf1(a(X1,X2),Y,a(a(X1,X2),Y)).
```

The predicate `to_nf` implements normal order reduction. It follows the same skeleton as `wh_nf`, which is called in the third clause to perform reduction to weak head normal form, starting from the outermost lambda binder.

```
to_nf(v(X),v(X)).
to_nf(l(E),l(NE)):-to_nf(E,NE).
to_nf(a(E1,E2),R):-wh_nf(E1,NE),to_nf1(NE,E2,R).
```

Case analysis of application terms for possible β-reduction is performed by `to_nf1/3`, where the second clause calls `beta/3` and recurses on its result.

```
to_nf1(v(E1),E2,a(v(E1),NE2)):-to_nf(E2,NE2).
to_nf1(l(E),E2,R):-beta(l(E),E2,NewE),to_nf(NewE,R).
to_nf1(a(A,B),E2,a(NE1,NE2)):-to_nf(a(A,B),NE1),to_nf(E2,NE2).
```

The predicates `to_nf` provides a Turing-complete lambda calculus interpreter working on de Bruijn terms. It is guaranteed to compute a normal form, if it exists. The predicate `evalStandard/2` works on standard lambda terms, that in converts to de Bruijn terms and then back after evaluation. The predicate `evalCompressed/2` works in a similar way on compressed de Bruijn terms. We express them as a composition of functions (first argument in, second out) using Prolog's DCG notation.

```
evalStandard-->l2b,to_nf,b2l.
evalCompressed-->c2b,to_nf,b2c.
```

Example 13 *illustrates evaluation of the lambda term* $SKK =$ $((\ \lambda x0.\ \lambda x1.\ \lambda x2.((x0\ x2)\ (x1\ x2))\ \ \lambda x3.\ \lambda x4.x3)\ \ \lambda x5.\ \lambda x6.x5)$ *in compressed de Brijn form, resulting in the definition of the identity combinator* $I = \lambda x0.x0$.

```
?- S=a(3,a(0,v(0,2),v(0,0)),a(0,v(0,1),v(0,0))),K=v(2,1),
     evalCompressed(a(0,a(0,S,K),K),R).
S = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))),
K = v(2, 1),
R = v(1, 0).
```

6 Related work

The classic reference for lambda calculus is [1]. Various instances of typed lambda calculi are overviewed in [4]. De Bruijn's notation for lambda terms is introduced in [2]. The compressed de Bruijn representation of lambda terms proposed in this paper is novel, to our best knowledge.

The combinatorics and asymptotic behavior of various classes of lambda terms are extensively studied in [6, 7, 12]. Distribution and density properties of random lambda terms are described in [13].

Lambda terms of bounded unary height are introduced in [8]. John Tromp's binary lambda calculus is only described through online code and the Wikipedia entry at [9].

Generators for closed and well-typed lambda terms, as well as their normal forms, expressed as functional programming algorithms, are given in [6], derived from combinatorial recurrences. However, they are significantly more complex than the ones described here in Prolog. On the other hand, we have not found in the literature generators for linear, linear affine terms and lambda terms of bounded unary height. Normalization of lambda terms and its confluence properties are described in [1] and [14] with functional programming algorithms given in [11] and HOAS-based evaluation first described in [10].

In a logic programming context, unification of simply typed lambda terms has been used in as the foundation of the programming language λProlog [15, 16] and applied to higher order logic programming [17].

7 Conclusion

We have described compact (and arguably elegant) combinatorial generation algorithms for several important families of lambda terms. Besides the newly introduced a compressed form of de Bruijn terms we have used ordinary de Bruijn terms as well as a canonical representation of lambda terms relying on Prolog's logic variables. In each case, we have selected the representation that was more appropriate for tasks like combinatorial generation, type inference or normalization. We have switched representation as needed, though bijective transformers working in time proportional to the size of the terms. Our combinatorial generation algorithms match the corresponding sequence of counts by size, given in [5] as an empirical validation of their correctness. Our techniques, combining unification of logic variables with Prolog's backtracking mechanism and DCG grammar notation, recommend logic programming as an unusually 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.

References

1. Barendregt, H.P.: The Lambda Calculus Its Syntax and Semantics. Revised edn. Volume 103. North Holland (1984)
2. Bruijn, N.G.D.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. Indagationes Mathematicae **34** (1972) 381–392
3. Stanley, R.P.: Enumerative Combinatorics. Wadsworth Publ. Co., Belmont, CA, USA (1986)
4. Barendregt, H.P.: Lambda calculi with types. In: Handbook of Logic in Computer Science. Volume 2. Oxford University Press (1991)
5. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences. (2014) Published electronically at https://oeis.org/.
6. Grygiel, K., Lescanne, P.: Counting and generating lambda terms. J. Funct. Program. **23**(5) (2013) 594–628
7. Grygiel, K., Idziak, P.M., Zaionc, M.: How big is BCI fragment of BCK logic. J. Log. Comput. **23**(3) (2013) 673–691
8. Bodini, O., Gardy, D., Gittenberger, B.: Lambda-terms of bounded unary height. In: ANALCO, SIAM (2011) 23–32
9. Wikipedia: Binary lambda calculus — wikipedia, the free encyclopedia (2015) [Online; accessed 20-February-2015].
10. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. PLDI '88, New York, NY, USA, ACM (1988) 199–208
11. Sestoft, P.: Demonstrating lambda calculus reduction. In Mogensen, T.A., Schmidt, D.A., Sudborough, I.H., eds.: The Essence of Computation. Springer-Verlag New York, Inc., New York, NY, USA (2002) 420–435
12. 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)
13. David, R., Raffalli, C., Theyssier, G., Grygiel, K., Kozik, J., Zaionc, M.: Some properties of random lambda terms. Logical Methods in Computer Science **9**(1) (2009)
14. Kamareddine, F.: Reviewing the Classical and the de Bruijn Notation for calculus and Pure Type Systems. Journal of Logic and Computation **11**(3) (2001) 363–394
15. Miller, D.: Unification of simply typed lambda-terms as logic programming. In: Proc. Int. Conference on Logic Programming (Paris), MIT Press (1991) 255–269
16. Nadathur, G., Mitchell, D.: System Description: Teyjus A Compiler and Abstract Machine Based Implementation of λProlog. In: Automated Deduction CADE-16. Volume 1632 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1999) 287–291
17. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press, New York, NY, USA (2012)