Ranking/unranking of lambda terms with compressed de Bruijn indices

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. Our compressed terms facilitate derivation of size-proportionate ranking and unranking algorithms of lambda terms and their inferred simple types. We specify our algorithms as a literate Prolog program.
Keywords: *lambda calculus, de Bruijn indices, lambda term compression, combinatorics of lambda terms, ranking and unranking of lambda terms, bijective Gödel numberings 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 newly designed programming language Swift. Ranking and unranking of lambda terms (i.e., their bijective mapping to unique natural number codes) 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.

Of particular interest are lambda term representations that are canonical up to alpha-conversion (variable renamings) among which are de Bruijn's indices [2], representing each bound variable as the number of binders to traverse to the lambda abstraction binding this variable.

A joke about the de Bruijn indices representation of lambda terms is that it can be used to tell apart Cylons from humans [3]. Arguably, the compressed de Bruijn representation that we introduce in this paper is taking their fictional use one step further. To alleviate the legitimate fears of our (most likely, for now, human) reader, these representations will be bijectively mapped to conventional ones.

A merit of our compressed representation is to simplify the underlying combinatorial structure of lambda terms, by exploiting their connection to the Catalan family of combinatorial objects [4]. This leads to algorithms that focus on their (bijective) natural number encodings - known to combinatorialists as *ranking/unranking* functions [5] and to logicians as *Gödel numberings* [6]. Among

the most obvious practical applications, such encodings can be used to generate random terms for testing tools like compilers or source-to-source program transformers. At the same time, as our encodings are "size-proportionate", they provide a compact serialization mechanism for lambda terms. To derive a bijection to $\mathbb{N}$ that is size-proportionate we will first extract a "Catalan skeleton" abstracting away the recursive structure of the compressed de Bruijn term, then implement a bijection from it to $\mathbb{N}$. The "content" fleshing out the term, represented as a list of natural numbers, will have its own bijection to $\mathbb{N}$ by using a generalized Cantor tupling / untupling function, that will also help pairing / unpairing the code of the skeleton and the code of the content of the term.

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 mappings from lambda terms to Catalan families of combinatorial objects, with focus on binary trees representing their inferred types and their applicative skeletons. These mappings lead in section 4 to size-proportionate ranking and unranking algorithms for lambda terms and their inferred types. Section 5 discusses related work and section 6 concludes the paper. The code in the paper has been tested with SWI-Prolog 6.6.6 and YAP 6.3.4. It is also available at `http://www.cse.unt.edu/~tarau/research/2015/dbr.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.

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(1,X,R).

b2c1(K,a(X,Y),a(K,A,B)):-b2c(X,A),b2c(Y,B).
b2c1(K, v(X),v(K,X)).
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.

```
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.
```

A convenient way to simplify defining chains of such conversions is by using Prolog's 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).
```

For instance, the predicate `c2l/2` which can be seen as specifying a composition of two functions, expands to something like `c2l(X,Z):-c2b(X,Y),b2l(Y,Z)` The predicate 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 Catalan approximations of lambda terms

We can see our compressed de Bruijn terms as binary trees decorated with integer labels. The underlying binary trees provide a skeleton that describes the applicative structure of the terms.

The Catalan family of combinatorial objects Binary trees are among the most well-known members of the Catalan family of combinatorial objects [4], that has at least 58 structurally distinct members, covering several data structures, geometric objects and formal languages.

3.1 Type Inference with logic variables

Simple types - seen as binary trees built by 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 safe and terminating evaluation (strong normalization) of lambda terms.

While in a functional language inferring types requires implementing unification with occur check, as shown, for instance, in [7], this is readily available in Prolog. 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):-!.
extractType(l(_:TX,A),(TX->TA)):-
  extractType(A,TA).
extractType(a(A,B),TY):-
  extractType(A,(TX->TY)),
  extractType(B,TX).
```

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

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

At this point, most general types are inferred by `extractType` as fresh variables, similar to multi-parameter polymorphic types in functional languages. 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 ignore its inferred type.

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

Example 1 *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.
```

3.2 Generating closed well-typed terms of a given size

One can derive, along the lines of the type inferrer `hasType`, a generator working directly on de Bruijn terms with a given number of internal nodes, by controlling their creation with the predicate `down/2`.

The predicate `genTypedB/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.

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

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

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

genTypedBs(L,B,T):-genTypedB(B,T,[],L,_),bindType(T).
```

As expected, the number of solutions of `genTypedB`, $1, 2, 9, 40, 238, 1564, \ldots$ for sizes $1, 2, 3, \ldots$, matches entry `A220471` in [8].

Example 2 *Generation of all well-typed closed de Bruijn terms of size 2.*

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

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 of a given type.

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

```
queryTypedB(L,Term,QueryType):-
  genTypedB(L,Term,Type),
  Type=QueryType.
```

Example 3 *Terms of type* `x>x` *of size 4.*

```
?- queryTypedB(4,Term,(o->o)).
Term = a(l(l(v(0))), l(v(0))) ;
Term = l(a(l(v(1)), l(v(0)))) ;
Term = l(a(l(v(1)), l(v(1)))) .

?- queryTypedB(10,Term,((o->o)->o)).
false.
```

Note that the last query, taking about half a minute, shows that no closed terms of type `(o->o)->o` exist up to size 10. Generating closed terms that match a specific type is likely to be useful for combinatorial testing.

We will explore next a mechanism for generating terms and types by defining a bijection to $\mathbb{N}$.

3.3 Size-proportionate encodings

In the presence of a bijection between two, usually infinite sets of data objects, it is possible that representation sizes on one side or the other are exponentially larger that on the other. Well-known encodings like Ackerman's bijection for hereditarily finite sets to natural numbers fall in this category.

We will say that a bijection is *size-proportionate* if the representation sizes for corresponding terms on its two sides are "close enough" up to a constant factor multiplied with at most the logarithm of any of the sizes.

Definition 1. *Given a bijection between sets of terms of two datatypes denoted M and N, $f : M \to N$, let $m(x)$ be the representation size of a term $x \in M$ and $n(y)$ be the representation size of $y \in N$. Then f is called* size-proportionate *if* $|m(x) - n(y)| \in O(log(max(m(x), n(y))))$.

Informally we also assume that the constants involved are small enough such that the printed representation of two data objects connected by the bijections is about the same.

3.4 The language of balanced parentheses

Binary trees are in a well-known size-proportionate bijection with the language of balanced parentheses [4], from which we will borrow an efficient ranking/unranking bijection. The reversible predicate `catpar/2` transforms between binary trees and lists of balanced parentheses, with `0` denoting the open parentheses and `1` denoting the closing one.

```
catpar(T,Ps):-catpar(T,0,1,Ps,[]).
catpar(X,L,R) --> [L],catpars(X,L,R).
catpars(o,_,R) --> [R].
catpars((X->Xs),L,R)-->catpar(X,L,R),catpars(Xs,L,R).
```

Example 4 *illustrates the work of the reversible predicate* `catpar/2`.

```
?- catpar(((o->o)->o->o),Ps),catpar(T,Ps).
Ps = [0, 0, 0, 1, 1, 0, 1, 1],T = ((o->o)->o->o) .
```

Note the extra opening/closing parentheses, compared to the usual definition of Dyck words [4], that make the sequence self-delimiting.

3.5 A bijection from the language of balanced parenthesis lists to $\mathbb{N}$

This algorithm follows closely the procedural implementation described in [5].

The code of the helper predicates called by `rankCatalan` and `unrankCatalan` is provided in `http://www.cse.unt.edu/~tarau/research/2015/dbr.pro`. The details of the algorithms for computing `localRank` and `localunRank` are described at `http://www.cse.unt.edu/~tarau/research/2015/dbrApp.pdf`.

The predicate `rankCatalan` uses the Catalan numbers computed by `cat` in `rankLoop` to shift the ranking over the ranks of smaller sequences, after calling `localRank`.

```
rankCatalan(Xs,R):-
  length(Xs,XL),XL>=2,
  L is XL-2, I is L // 2,
  localRank(I,Xs,N),
  S is 0, PI is I-1,
  rankLoop(PI,S,NewS),
  R is NewS+N.
```

The predicate `unrankCatalan` uses the Catalan numbers computed by `cat` in `unrankLoop` to shift over smaller sequences, before calling `localUnrank`.

```
unrankCatalan(R,Xs):-
  S is 0, I is 0,
  unrankLoop(R,S,I,NewS,NewI),
  LR is R-NewS,
  L is 2*NewI+1,
  length(As,L),
  localUnrank(NewI,LR,As),
  As=[_|Bs],
  append([0|Bs],[1],Xs).
```

The following example illustrates the ranking and unranking algorithms:

```
?- unrankCatalan(2015,Ps),rankCatalan(Ps,Rank).
Ps = [0,0,1,0,1,0,1,0,0,0,0,1,0,1,1,1,1,1],Rank = 2015
```

3.6 Ranking and unranking simple types

After putting together the bijections between binary trees and balanced parentheses with the ranking/unranking of the later we obtain the size-proportionate ranking/unranking algorithms for simple types.

```
rankType(T,Code):-
  catpar(T,Ps),
  rankCatalan(Ps,Code).

unrankType(Code,Term):-
  unrankCatalan(Code,Ps),
  catpar(Term,Ps).
```

Example 5 *illustrates the ranking and unranking of simple types.*

```
?- I=100, unrankType(I,T),rankType(T,R).
I = R, R = 100,T = (((o->o)-> (o->o->o)->o)->o) .
```

As there are $O(\frac{4^n}{n^{\frac{3}{2}}})$) binary trees of size n corresponding to 2^n natural numbers of bitsize up to n and our ranking algorithm visits them in lexicographic order, it follows that:

Proposition 1. *The bijection between types and their ranks is size-proportionate.*

3.7 Catalan skeletons of compressed de Bruijn terms

As compressed de Bruijn terms can be seen as binary trees with labels on their leaves and internal nodes, their "Catalan skeleton" is simply the underlying binary tree. The predicate `cskel/3` extracts this skeleton as well as the list of the labels, in depth-first order, as encountered in the process.

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

cskel(v(K,N),o)-->[K,N].
cskel(a(K,X,Y),(A->B))-->[K],cskel(X,A),cskel(Y,B).
```

The predicates `toSkel` and `fromSkel` add conversion between this binary tree and lists of balanced parenthesis by using the (reversible) predicate `catpar`.

```
toSkel(T,Skel,Vs):-
  cskel(T,Cat,Vs,[]),
  catpar(Cat,Skel).

fromSkel(Skel,Vs, T):-
  catpar(Cat,Skel),
  cskel(T,Cat,Vs,[]).
```

Example 6 *illustrates the Catalan skeleton* `Skel` *and the list of variable labels* `Vs` *extracted from a compressed de Bruijn term corresponding to the* `S` *combinator.*

```
?- T = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))),
   toSkel(T,Skel,Vs),fromSkel(Skel,Vs,T1).
T = T1, T1 = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))),
Skel = [0,0,0,1,1,0,1,1],Vs = [3,0,0,2,0,0,0,0,1,0,0] .
```

3.8 The generalized Cantor k-tupling bijection

As we we have already solved the problem of ranking and unranking lists of balanced parentheses, the remaining problem is that of finding a bijection between the lists of labels collected from the nodes of a compressed de Bruijn term and natural numbers.

We will use the generalized Cantor bijection between $\mathbb{N}^n$ and $\mathbb{N}$ as the first step in defining this bijection. The formula, given in [9] p.4, looks as follows:

$$K_n(x_1,\ldots,x_n) = \sum_{k=1}^{n} \binom{k-1+s_k}{k} \text{ where } s_k = \sum_{i=1}^{k} x_i \tag{1}$$

Note that $\binom{n}{k}$ represents the number of subsets of k elements of a set of n elements, that also corresponds to the binomial coefficient of x^k in the expansion of $(x+y)^n$, and $K_n(x_1,\ldots,x_n)$ denotes the natural number associated to the tuple $(x_1,\ldots,x_n)$. It is easy to see that the generalized Cantor n-tupling function defined by equation (1) is a polynomial of degree n in its arguments.

The bijection between sets and sequences of natural numbers We recognize in the equation (1) the *prefix sums* s_k incremented with values of k starting at 0. It represents the "set side" of the bijection between sequences of n natural numbers and sets of n natural numbers described in [10]. It is implemented in the online Appendix as the bijection `list2set` together with its inverse `set2list`. For example, `list2set` transforms `[2,0,1,5]` to [2, 3, 5, 11] as `3=2+0+1,5=3+1+1,11=5+5+1` and `set2list` transforms it back by computing the differences between consecutive members, reduced by 1.

3.9 The $\mathbb{N}^n \to \mathbb{N}$ bijection

The bijection $K_n : \mathbb{N}^n \to \mathbb{N}$ is basically just summing up a set of binomial coefficients. The predicate `fromCantorTuple` implements the the $\mathbb{N}^n \to \mathbb{N}$ bijection in Prolog, using the predicate `fromKSet` that sums up the binomials in formula 1 using the predicate `untuplingLoop`, as well as the sequence to set transformer `list2set`.

```
fromCantorTuple(Ns,N):-
  list2set(Ns,Xs),
  fromKSet(Xs,N).

fromKSet(Xs,N):-untuplingLoop(Xs,0,0,N).

untuplingLoop([],_L,B,B).
untuplingLoop([X|Xs],L1,B1,Bn):-L2 is L1+1,
  binomial(X,L2,B),B2 is B1+B,
  untuplingLoop(Xs,L2,B2,Bn).
```

3.10 The $\mathbb{N} \to \mathbb{N}^n$ bijection

We split our problem in two simpler ones: inverting `fromKSet` and then applying `set2list` to get back from sets to lists.

We observe that the predicate `untuplingLoop` used by `fromKSet` implements the sum of the combinations $\binom{X_1}{1} + \binom{X_2}{2} + \ldots + \binom{X_K}{K} = N$, which is nothing but the representation of N in the *combinatorial number system of degree* K due to [11]. Fortunately, efficient conversion algorithms between the conventional and the combinatorial number system are well known, [12].

We are ready to implement the Prolog predicate `toKSet(K,N,Ds)`, which, given the degree `K`, indicating the number of "combinatorial digits", finds and repeatedly subtracts the greatest binomial smaller than `N`. It calls the predicate `combinatoriallDigits` that returns these "digits" in increasing order, providing the canonical set representations that `set2list` needs.

```
toKSet(K,N,Ds):-combinatoriallDigits(K,N,[],Ds).

combinatoriallDigits(0,_,Ds,Ds).
combinatoriallDigits(K,N,Ds,NewDs):-K>0,K1 is K-1,
```

```
  upperBinomial(K,N,M),M1 is M-1,
  binomial(M1,K,BDigit),N1 is N-BDigit,
  combinatoriallDigits(K1,N1,[M1|Ds],NewDs).

upperBinomial(K,N,R):-S is N+K,
  roughLimit(K,S,K,M),L is M // 2,
  binarySearch(K,N,L,M,R).
```

The predicate `roughLimit` compares successive powers of 2 with binomials $\binom{I}{K}$ and finds the first `I` for which the binomial is between successive powers of 2.

```
roughLimit(K,N,I, L):-binomial(I,K,B),B>N,!,L=I.
roughLimit(K,N,I, L):-J is 2*I,
  roughLimit(K,N,J,L).
```

The predicate `binarySearch` finds the exact value of the combinatorial digit in the interval `[L,M]`, narrowed down by `roughLimit`.

```
binarySearch(_K,_N,From,From,R):-!,R=From.
binarySearch(K,N,From,To,R):-Mid is (From+To) // 2,binomial(Mid,K,B),
  splitSearchOn(B,K,N,From,Mid,To,R).

splitSearchOn(B,K,N,From,Mid,_To,R):-B>N,!,
  binarySearch(K,N,From,Mid,R).
splitSearchOn(_B,K,N,_From,Mid,To,R):-Mid1 is Mid+1,
  binarySearch(K,N,Mid1,To,R).
```

The predicates `toKSet` and `fromKSet` implement inverse functions, mapping natural numbers to canonically represented sets of `K` natural numbers.

```
?- toKSet(5,2014,Set),fromKSet(Set,N).
Set = [0, 3, 4, 5, 14], N = 2014 .
```

The efficient inverse of Cantor's N-tupling is now simply:

```
toCantorTuple(K,N,Ns):-
  toKSet(K,N,Ds),
  set2list(Ds,Ns).
```

Example 7 *illustrates the work of the generalized cantor bijection, on some large numbers:*

```
?- K=1000,pow(2014,103,N),toCantorTuple(K,N,Ns),fromCantorTuple(Ns,N).
K = 1000, N = 208029545585703688484419851459547264831381665...567744,
Ns = [0, 0, 2, 0, 0, 0, 0, 0, 1|...] .
```

As the image of a tuple is a polynomial of degree n it means that the its bitsize is within constant factor of the sum of the bitsizes of the members of the tuple, thus:

Proposition 2. *The bijection between* $\mathbb{N}^n$ *and* $\mathbb{N}$ *is size-proportionate.*

4 Ranking/unranking of compressed de Bruijn terms

We will implement a size-proportionate bijective encoding of compressed de Bruijn terms following the technique described in [13]. The algorithm will split a lambda tree into its *Catalan skeleton* and the list of atomic objects labeling its nodes. In our case, the Catalan skeleton abstracts away the applicative structure of the term. It also provides the key for decoding unambiguously the integer labels in both the leaves (two integers) and internal nodes (one integer). Our ranking/unranking algorithms will rely on the encoding/decoding of the Catalan skeleton provided by the predicates `rankCatalan/2` and `unrankCatalan/2` as well as for the encoding/decoding of the labels, provided by the predicates `toCantorTuple/3` and `fromCantorTuple/2`.

The predicate `rankTerm/2` defines the bijective encoding of a (possibly open) compressed de Bruijn term.

```
rankTerm(Term,Code):-
  toSkel(Term,Ps,Ns),
  rankCatalan(Ps,CatCode),
  fromCantorTuple(Ns,VarsCode),
  fromCantorTuple([CatCode,VarsCode],Code).
```

The predicate `rankTerm/2` defines the bijective decoding of a natural number into a (possibly open) compressed de Bruijn term.

```
unrankTerm(Code,Term):-
  toCantorTuple(2,Code,[CatCode,VarsCode]),
  unrankCatalan(CatCode,Ps),
  length(Ps,L2),L is (L2-2) div 2, L3 is 3*L+2,
  toCantorTuple(L3,VarsCode,Ns),
  fromSkel(Ps,Ns,Term).
```

Note that given the unranking of `CatCode` as a list of balanced parentheses of length `2*L+2`, we can determine the number `L` of internal nodes of the tree and the number `L+1` of leaves. Then we have `2*(L+1)` labels for the leaves and `L` labels for the internal nodes, for a total of `3L+2`, value needed to decode the labels encoded as `VarsCode`.

It follows from Prop. 1 and Prop. 2 that:

Proposition 3. *A compressed de Bruijn terms is size-proportionate to its rank.*

Example 8 *illustrates the "size-proportionate" encoding of the compressed de Bruijn terms corresponding to the combinators* `S` *and* `Y`.

```
?- T = a(3,a(0,v(0,2),v(0,0)),a(0,v(0, 1),v(0,0))),
   rankTerm(T,R),unrankTerm(R,T1).
T = T1,T1 = a(3,a(0,v(0,2),v(0,0)),a(0,v(0, 1),v(0,0))),
R = 56493141 .

?- T=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)))),
   rankTerm(T,R),unrankTerm(R,T1).
T=T1,T1=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)))),
R = 261507060 .
```

4.1 Generation of lambda terms via unranking

While direct enumeration of terms constrained by number of nodes or depth is straightforward in Prolog, an unranking algorithm is also usable for term generation, including generation of random terms.

Generating open terms in compressed de Bruijn form Open terms are generated simply by iterating over an initial segment of $\mathbb{N}$ with the built-in `between/3` and calling the predicate `unrankTerm/2`.

```
ogen(M,T):-between(0,M,I),unrankTerm(I,T).
```

Reusing unranking-based open term generators for more constrained families of lambda terms works when their asymptotic density is relatively high. The extensive quantitative analysis available in the literature [7, 14, 15] indicates that density of closed and typed terms decreasing very quickly with size, making generation by filtering impractical for very large terms.

The predicate `cgen/2` generates closed terms by filtering the results of `ogen/2` with the predicate `isClosed` and `tgen` generates typeable terms by filtering the results of `cgen/2` with `typeable/2`.

```
cgen(M,IT):-ogen(M,IT),isClosed(IT).

tgen(M,IT):-cgen(M,IT),typeable(IT).
```

Generation of random lambda terms Generation of random lambda terms, resulting from the unranking of random integers of a give bit-size, is implemented by the predicate `ranTerm/3`, that applies the predicate `Filter` repeatedly until a term is found for which the predicate `Filter` holds.

```
ranTerm(Filter,Bits,T):-X is 2^Bits,N is X+random(X),M is N+X,
  between(N,M,I),
   unrankTerm(I,T),call(Filter,T),
  !.
```

Random open terms are generated by `ranOpen/2`, random closed terms by the predicate `ranClosed`, random typeable term by `ranTyped` and closed typeable terms by `closedTypeable/2`.

```
ranOpen(Bits,T):-ranTerm(=(_),Bits,T).

ranClosed(Bits,T):-ranTerm(isClosed,Bits,T).

ranTyped(Bits,T):-ranTerm(closedTypeable,Bits,T).

closedTypeable(T):-isClosed(T),typeable(T).
```

Open terms based on unranking random numbers of 3000 bits of size above 1000, closed terms of size above 55 for 150 bits and closed typeable terms of size above 13 for 30 bits can be generated within a few seconds. The limited scalability for

closed and well-typed terms is a consequence of their low asymptotic density, as shown in [14, 7]. We refer to [7] for algorithms supporting random generation of large lambda terms.

Example 9 *illustrates generation of some closed and well-typed terms in compressed de Bruijn form.*

```
?- ranClosed(10,T).
T = a(1, a(0, v(0, 0), v(0, 0)), a(0, a(0, v(0, 0), v(0, 0)), v(1, 0))).

?- ranTyped(20,T).
T = a(3, v(3, 1), v(2, 0)).
```

5 Related work

The classic reference for lambda calculus is [1]. Various instances of typed lambda calculi are overviewed in [16]. 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.

While Gödel numbering schemes for lambda terms have been studied in several theoretical papers on computability, we are not aware of any size proportionate bijective encoding as the one described in this paper.

The combinatorics and asymptotic behavior of various classes of lambda terms are extensively studied in [7, 14, 15]. Distribution and density properties of random lambda terms are described in [14].

6 Conclusions

The most significant contributions of this paper are the size-proportionate ranking/unranking algorithm for lambda terms and the compressed de Bruijn representation that facilitated it. The ability to encode lambda terms bijectively can be used as a "serialization" mechanism in functional programming languages and proof assistants using them as an intermediate language.

Besides the newly introduced 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. We have switched representation as needed, though bijective transformers working in time proportional to the size of the terms. Our techniques, combining unification of logic variables with Prolog's backtracking mechanism and Definite Clause Grammar notation, suggest that logic programming is a suitable meta-language for the manipulation of various families of lambda terms and the study of their combinatorial and computational properties.

Acknowledgement We thank the anonymous referees of **Calculemus'15** for their constructive criticisms and valuable suggestions that have helped improving the paper. This research was supported by NSF research grant **1423324**.

References

1. Barendregt, H.P.: The Lambda Calculus Its Syntax and Semantics. Revised edn. Volume 103. North Holland (1984)
2. de Bruijn, N.G.: 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. McBride, C.: I am not a number, I am a classy hack (2010) *Blog entry:* `http://mazzo.li/epilogue/index.html%3Fp=773.html`.
4. Stanley, R.P.: Enumerative Combinatorics. Wadsworth Publ. Co., Belmont, CA, USA (1986)
5. Kreher, D.L., Stinson, D.: Combinatorial Algorithms: Generation, Enumeration, and Search. The CRC Press Series on Discrete Mathematics and its Applications. CRC PressINC (1999)
6. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
7. Grygiel, K., Lescanne, P.: Counting and generating lambda terms. J. Funct. Program. **23**(5) (2013) 594–628
8. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences. (2014) Published electronically at https://oeis.org/.
9. Cegielski, P., Richard, D.: On arithmetical first-order theories allowing encoding and decoding of lists. Theoretical Computer Science **222**(1-2) (1999) 55–75
10. Tarau, P.: An Embedded Declarative Data Transformation Language. In: Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009, Coimbra, Portugal, ACM (September 2009) 171–182
11. Lehmer, D.H.: The machine tools of combinatorics. In: Applied combinatorial mathematics. Wiley, New York (1964) 5–30
12. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions. Addison-Wesley Professional (2005)
13. Tarau, P.: Compact Serialization of Prolog Terms (with Catalan Skeletons, Cantor Tupling and Gödel Numberings) . Theory and Practice of Logic Programming **13**(4-5) (2013) 847–861
14. 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)
15. 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)
16. Barendregt, H.P.: Lambda calculi with types. In: Handbook of Logic in Computer Science. Volume 2. Oxford University Press (1991)

Appendix

Helper predicates for ranking and unranking balanced parentheses expressions

The predicate `cat` computes efficiently the n-th Catalan number $\frac{1}{n+1}\binom{2n}{n}$:

```
cat(0,1).
cat(N,R):-N>0, PN is N-1, SN is N+1,cat(PN,R1),R is 2*(2*N-1)*R1//SN.
```

The predicate `binDif` computes the difference of two binomials.

```
binDif(N,X,Y,R):- N1 is 2*N-X,R1 is N - (X + Y) // 2, R2 is R1-1,
  binomial(N1,R1,B1),binomial(N1,R2,B2),R is B1-B2.
```

The predicate `localRank` computes, by binary search the rank of sequences of a given length.

```
localRank(N,As,NewLo):- X is 1, Y is 0, Lo is 0,
  binDif(N,0,0,Hi0),Hi is Hi0-1,
  localRankLoop(As,N,X,Y,Lo,Hi,NewLo,_NewHi).
```

After finding the appropriate range containing the rank with `binDif`, we delegate the work to the predicate `localRankLoop`.

```
localRankLoop(As,N,X,Y,Lo,Hi,FinalLo,FinalHi):-N2 is 2*N,X< N2,!,
  PY is Y-1, SY is Y+1, nth0(X,As,A),
  (0=:=A-> binDif(N,X,PY,Hi1),
     NewHi is Hi-Hi1, NewLo is Lo, NewY is SY
   ; binDif(N,X,SY,Lo1),
     NewLo is Lo+Lo1, NewHi is Hi, NewY is PY
  ), NewX is X+1,
  localRankLoop(As,N,NewX,NewY,NewLo,NewHi,FinalLo,FinalHi).
localRankLoop(_As,_N,_X,_Y,Lo,Hi,Lo,Hi).
```

```
rankLoop(I,S,FinalS):-I>=0,!,cat(I,C),NewS is S+C, PI is I-1,
  rankLoop(PI,NewS,FinalS).
rankLoop(_,S,S).
```

Unranking works in a similar way. The predicate `localUnrank` builds a sequence of balanced parentheses by doing binary search to locate the sequence in the enumeration of sequences of a given length.

```
localUnrank(N,R,As):-Y is 0,Lo is 0,binDif(N,0,0,Hi0),Hi is Hi0-1, X is 1,
  localUnrankLoop(X,Y,N,R,Lo,Hi,As).

localUnrankLoop(X,Y,N,R,Lo,Hi,As):-N2 is 2*N,X=<N2,!,
   PY is Y-1, SY is Y+1,
   binDif(N,X,SY,K), LK is Lo+K,
   ( R<LK -> NewHi is LK-1, NewLo is Lo, NewY is SY, Digit=0
   ; NewLo is LK, NewHi is Hi, NewY is PY, Digit=1
   ),nth0(X,As,Digit),NewX is X+1,
   localUnrankLoop(NewX,NewY,N,R,NewLo,NewHi,As).
localUnrankLoop(_X,_Y,_N,_R,_Lo,_Hi,_As).
```

```
unrankLoop(R,S,I,FinalS,FinalI):-cat(I,C),NewS is S+C, NewS=<R,
   !,NewI is I+1,
   unrankLoop(R,NewS,NewI,FinalS,FinalI).
unrankLoop(_,S,I,S,I).
```

The bijection between finite lists and sets

The bijection `list2set` together with its inverse `set2list` are defined as follows:

```
list2set(Ns,Xs) :- list2set(Ns,-1,Xs).

list2set([],_,[]).
list2set([N|Ns],Y,[X|Xs]) :-
  X is (N+Y)+1,
  list2set(Ns,X,Xs).

set2list(Xs,Ns) :- set2list(Xs,-1,Ns).

set2list([],_,[]).
set2list([X|Xs],Y,[N|Ns]) :-
  N is (X-Y)-1,
  set2list(Xs,X,Ns).
```

The following examples illustrate this bijection:

```
?- list2set([2,0,1,4],Set),set2list(Set,List).
Set = [2, 3, 5, 10],
List = [2, 0, 1, 4].
```

As a side note, this bijection is mentioned in [12] with indications that it might even go back to the early days of the theory of recursive functions.

Binomial Coefficients, efficiently

Binomial coefficients are given by the formula $\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\ldots(n-(k-1))}{k!}$. By performing divisions as early as possible to avoid generating excessively large intermediate results, one can derive the `binomialLoop` tail-recursive predicate:

```
binomialLoop(_,K,I,P,R) :- I>=K, !, R=P.
binomialLoop(N,K,I,P,R) :- I1 is I+1, P1 is ((N-I)*P) // I1,
   binomialLoop(N,K,I1,P1,R).
```

The predicate `binomial(N,K,R)` computes $\binom{N}{K}$ and unifies the result with `R`.

```
binomial(_N,K,R):- K<0,!,R=0.
binomial(N,K,R) :- K>N,!, R=0.
binomial(N,K,R) :- K1 is N-K, K>K1, !, binomialLoop(N,K1,0,1,R).
binomial(N,K,R) :- binomialLoop(N,K,0,1,R).
```