Compact Serialization of Prolog Terms (with Catalan Skeletons, Cantor Tupling and Gödel Numberings)

PAUL TARAU

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

submitted April 10, 2013; revised June 23, 2013; accepted July 5, 2013

Abstract

We describe a compact serialization algorithm mapping Prolog terms to natural numbers of bit-sizes proportional to the memory representation of the terms. The algorithm is a " no bit lost" bijection, as it associates to each Prolog term a unique natural number and each natural number corresponds to a unique syntactically well-formed term.

To avoid an exponential explosion resulting from bijections mapping term trees to natural numbers, we separate the symbol content and the syntactic skeleton of a term that we serialize compactly using a ranking algorithm for Catalan families.

A novel algorithm for the generalized Cantor bijection between $\mathbb{N}$ and $\mathbb{N}^k$ is used in the process of assigning polynomially bounded Gödel numberings to various data objects involved in the translation.

KEYWORDS: bijective serialization of Prolog terms, ranking/unranking of Catalan families, generalized Cantor *n*-tupling bijection, combinatorial number system

1 Introduction

Serialization - seen as a reversible transformation of an arbitrary data object to a bitstring (or equivalently, to a natural number of an unbounded size) is widely used in today's programming languages to persistently store objects or send them over networks in a compact form that bypasses the slow parsing and security risks of storing or sending them in source form.

While specific Prolog systems provide saving of terms in a binary format, no principled serialization mechanism exists, partly because of the comparatively old age of Prolog's initial design and partly because of Prolog's powerful metaprogramming and source-to-source transformation mechanisms that work directly on Prolog's standard term representation.

Traditional serialization mechanisms, as provided by languages like Java or Scala, encode object graphs to byte files or streams *injectively*, i.e., the encoding can be inverted to the original data object. Such encodings are by no means *surjective* , as not every possible byte string configuration corresponds to a valid object. Clearly, such injective-only serialization mechanisms waste the information-theoretic capabilities of their range by not using every possible bit to store useful information.

While arguably a bijective serialization mechanism is "mission impossible" for object oriented languages like Java, that rely on a reflection API of close to one hundred methods, it makes sense for a language with a more uniform syntactic and semantic structure like Prolog or Lisp/Scheme.

While we show in this paper that a serialization mechanism that is both *bijective* and *size-proportionate* [1] is possible for Prolog terms, significant challenges, requiring creative algorithmic solutions, need to be handled in the process.

First, as a fairly routine step, a bijective mapping from Prolog terms to canonical members of a term algebra (section 2), provides an intermediate representation that simplifies our algorithms.

Next, mapping of Prolog atoms (section 3) to unique natural numbers (subsection 3.2), as well as tagging and untagging data types of unbounded size, require the use of bijective numeration systems, for which we develop incremental, one digit at a time conversion algorithms (subsection 3.1).

The main difficulty comes from the fact that encoding bijectively the tree structure of Prolog terms, results easily in an exponential size explosion. We tackle this by separating the tree skeleton and the symbol content of Prolog terms (section 4).

A bijective encoding of sequences of length k requires a mapping between $\mathbb{N}^k$ and $\mathbb{N}$ provided by a polynomial formula - and it is conjectured that the generalization of Cantor's pairing function to k-tuples is the only one available. An efficient algorithm to invert it will be provided for this purpose (section 5).

At the end (section 6), the bijective encoding will associate a unique $n \in \mathbb{N}$ to a term and for all $n \in \mathbb{N}$ a unique term will be generated from it, as illustrated by the following examples:

```
?- encodeTerm(f(X,g(a,0,X),[1,2]),N),decodeTerm(N,T).
N = 678547916890513735116076, T = f(A, g(a, 0, A), [1, 2]) .

?- decodeTerm(314159,T),encodeTerm(T,N).
T = '.'(c(A, A, A, [](A))), N = 314159 .
```

As the reader might notice, we used the first few digits of π for our second example to emphasize that *any* $n \in \mathbb{N}$ decodes to a syntactically valid Prolog term.

Finally, section 7 estimates time complexity of our algorithms and illustrates our findings with a set of tests on large term encodings and decodings, section 8 discusses related work and section 9 concludes the paper. The **Appendix** contains routine algorithms that ensure that the Prolog code (tested with SWI-Prolog 6.2.6) in the paper is self-contained. The code is also available as a separate file from `http://logic.cse.unt.edu/tarau/Research/2013/serpro.pl`.

2 A canonical representation of Prolog terms

To facilitate operating on Prolog terms, we put them in a bijection with a term algebra providing a canonical ground representation. For flexibility, the representation is designed to be extensible and it can, with a few easy changes, accommodate new basic data types.

The term algebra can be seen as defined by the recursive data type:

```
data Term = Fun Nat [Term] | Leaf TypeTag Nat
```

where `Nat` is an arbitrary length natural number and `TypeTag` is a small natural number classifying simple data types corresponding to Prolog's `var`, `atom`, etc. types and `[Term]` stands for a list of terms. The predicates `toCanonical` and `fromCanonical` provide the conversion.

[1] We call "size proportionate" an encoding that ensures that the bitsize of the representations is (roughly) within a constant factor on both sides. Speed-wise, as arbitrary size integer arithmetic and binary search are involved, in practice this results in a low polynomial worse case conversion performance, depending on the size of the terms, choice of traditional, Kratsuba or FFT-multiplication for large integers, and practical factors like the memory management of arbitrary size integer packages like GMP or the underlying Prolog system's garbage collector.

```
toCanonical(P,T):-copy_term(P,CP),numbervars(CP,0,_),prolog2TermAlgebra(CP,T).

fromCanonical(T,P):-termAlgebra2prolog(T,P).
```

Note that Prolog variables are grounded as usual with `numbervars` and `copy_term` is used to prevent side effects on the original term.

The predicates `prolog2TermAlgebra` and `termAlgebra2prolog` recurse over the structure of Prolog and respectively their canonical counterparts and separate the compound and simple terms corresponding to the union type `Term`.

```
prolog2TermAlgebra(P,T):-P='$VAR'(_),!,toTypeValue(P,T).
prolog2TermAlgebra(P,T):-atomic(P),!,toTypeValue(P,T).
prolog2TermAlgebra(P,fun(FCode,Ts)):-compound(P),P=..[F|Ps],atom2nat(F,FCode),
  maplist(prolog2TermAlgebra,Ps,Ts).
```

```
termAlgebra2prolog(leaf(0,N),P):-!,P='$VAR'(N).
termAlgebra2prolog(T,P):-T=leaf(_,_),!,fromTypeValue(T,P).
termAlgebra2prolog(fun(FCode,Ts),P):-nat2atom(FCode,F),
  maplist(termAlgebra2prolog,Ts,Ps),P=..[F|Ps].
```

Note the use of the bijection `atom2nat / nat2atom` for associating unique codes to Prolog atoms. It will be described in detail in subsection 3.2. The predicates `toTypeValue` and `fromTypeValue` provide the encoding of simple types and `totalTypes` indicates the number of type tags marking the simple types.

```
toTypeValue('$VAR'(N),TV):-!,TV=leaf(0,N).
toTypeValue(P,TV):-atom(P),!,atom2nat(P,Code),TV=leaf(1,Code).
toTypeValue(P,TV):-integer(P),!,encodeSign(P,Code),TV=leaf(2,Code).
```

```
fromTypeValue(leaf(0,N),P):-!,P='$VAR'(N).
fromTypeValue(leaf(1,Code),P):-!,nat2atom(Code,P).
fromTypeValue(leaf(2,Code),P):-decodeSign(Code,P).

totalTypes(3).
```

The predicates `encodeSign` and `decodeSign` are used to bijectively convert signed integers to natural numbers.

```
encodeSign(0,Code):-!,Code=0.
encodeSign(P,Code):-P>0,!,Code is 2*P+1.
encodeSign(N,Code):-N<0,Code is 2*(-N)+2.

decodeSign(0,Z):-!,Z=0.
decodeSign(Code,P):-Code>0,Code mod 2=:=1,!,P is (Code-1) // 2.
decodeSign(Code,N):-Code>0,Code mod 2=:=0,N is (2-Code) // 2.
```

Note that, for simplicity, we have not considered floating point or rational numbers but the encoding is easily extensible to accommodate them, by modifying the predicate `totalTypes` and adding new conversion rules.

3 Bijective encodings of Prolog atoms

Prolog provides a mapping between its symbols and their lists of character codes (strings). To obtain an encoding of strings that is linear in their bitsize, we need a general mechanism to map arbitrary combinations of `k` symbols to natural numbers.

3.1 Encoding numbers in bijective base-k

The conventional numbering system does not provide a bijection between arbitrary combinations of digits and natural numbers, given that leading 0s are ignored. For this purpose we need to use *numbers in bijective base-k* representation[2].

We start with the mapping from a list of digits in `[0..k-1]` to a natural number defined by the predicate `fromBBase`.

```
fromBBase(_,[],0).
fromBBase(B,[D|Ds],NewN):-fromBBase(B,Ds,N),putBDigit(B,D,N,NewN).
```

The predicate `toBBase` reverses the mapping provided by `fromBBase`.

```
toBBase(_,0,[]).
toBBase(B,N,[D|Ds]):-N>0,getBDigit(B,N,D,M),toBBase(B,M,Ds).
```

Both predicates proceed *one digit at a time*, by calling `putBDigit` and `getBDigit`.

```
putBDigit(B,D,M,R):-R is 1+D+B*M.

getBDigit(B,N,Digit,NewN):-Q is N // B,D is N mod B,
  getBDigit1(D,Q,B,Digit,NewN).

getBDigit1(0,Q,B,Digit,NewN):-Digit is B-1,NewN is Q-1.
getBDigit1(D,Q,_B,Digit,Q):-D>0,Digit is D-1.
```

Note that the predicates `fromBBase` and `toBBase` are parametrized by the base of numeration (`Base>0`) which should be the same when encoding and decoding. Also, while not very useful, `Base=1` is acceptable and it will encode `n` as a sequence of `n` 0s. The following holds:

Proposition 1
The predicates `toBBase` and `fromBBase` define a bijection between natural numbers and their bijective base-k representation.

We illustrate the use of these predicates with an example:

```
?- toBBase(7,2014,Ds),fromBBase(7,Ds,N).
Ds = [4, 6, 4, 4], N = 2014 .
```

This encoding will turn out to be useful for symbols of a finite alphabet. At the same time, the incremental steps `putBDigit` and `getBDigit` will be used as an unconventional mechanism to *tag a data object of arbitrary size* reversibly when one wants to define a generic union type.

```
?- TotalTags=3,putBDigit(TotalTags,1,1234567890,Tagged),
   getBDigit(TotalTags,Tagged,Tag,Untagged).
TotalTags = 3, Tagged = 3703703672,
Tag = 1, Untagged = 1234567890.
```

3.2 Encoding strings

Strings can be seen just as a notational equivalent of lists of natural numbers written in bijective base-*k*. For simplicity, (and to avoid unprintable characters as a result of applying the inverse

[2] We refer to (Salomaa 1973), pages 90-91 for the definition of this representation, in the context of formal languages, where it is used under the name *m-adic encoding*.

mapping), we assume that our strings naming function symbols are built using lower case ASCII characters. One can easily customize our code to cover UTF-8 or UTF-32 Unicode characters, if needed, by simply redefining the predicates `c0` and `c1`. Alternatively, an implementor with access to symbol codes in the Prolog system's symbol table, might want to bypass this and just provide the codes, if she/he can avoid the potential conflict with symbol garbage collection.

```
c0(A):-[A]="a".
c1(Z):-[Z]="z".

base(B):-c0(A),c1(Z),B is 1+Z-A.
```

The predicates `string2nat` and `nat2string` define the bijective base-k encodings between Prolog strings and natural numbers.

```
string2nat(".",N):-!,N=0.
string2nat("[]",N):-!,N=1.
string2nat(Cs,N):-Cs=[_|_],base(B), maplist(chr2ord,Cs,Ns),
  fromBBase(B,Ns,N0),N is N0+1.
```

```
nat2string(0,Cs):-!,Cs=".".
nat2string(1,Cs):-!,Cs="[]".
nat2string(N0,Cs):-N0 > 1,N is N0-1,
  base(B),toBBase(B,N,Xs),maplist(ord2chr,Xs,Cs).
```

Note that the additional symbols "." and "[]", used for Prolog list construction are also mapped uniquely to `0` and `1`, as special cases. After defining

```
chr2ord(C,O):-c0(A),C>=A,c1(Z),C=<Z,O is C-A.
ord2chr(O,C):-O>=0,base(B),O<B,c0(A),C is A+O.
```

we obtain an encoder for Prolog strings working as follows:

```
?- Cs = "hello", string2nat(Cs,N), nat2string(N,CsAgain).
Cs = [104, 101, 108, 108, 111], N = 7073803,
CsAgain = [104, 101, 108, 108, 111] .
```

This brings us to the encoding of Prolog atoms as natural numbers:

```
atom2nat(Atom,Nat):-atom_codes(Atom,Cs),string2nat(Cs,Nat).

nat2atom(Nat,Atom):-nat2string(Nat,Cs),atom_codes(Atom,Cs).
```

The following holds:

Proposition 2
The predicates `atom2nat` and `nat2atom` define a bijection between Prolog atoms on a fixed alphabet and natural numbers.

4 "Catalan skeletons" of Prolog terms

We now turn to encodings focusing on the separation of the structure and the content of Prolog terms. The connection between balanced parenthesis languages and a large number of different data types (among which we find multi-way and binary trees) in the *Catalan family* is known to combinatorialists (Kreher and Stinson 1999; Liebehenschel 2000). We will first map a term to a "skeleton" representing its structure as a list of balanced parentheses and then rework the mapping to obtain a bijective representation.

4.1 Extracting the Catalan skeleton of a term

The predicate `term2bitpars` converts a term `T` to a sequence of balanced parentheses (represented as `0` and `1` digits) `Ps`, and a list of atomic terms and Prolog variables `As`, seen as a symbol table that stores the "content" of the terms.

```
term2bitpars(T,Ps,As):-toCanonical(T,CT),term2bitpars(CT,Ps,[],As,[]).

term2bitpars(T,[0,1|Ps],Ps)-->{T=leaf(_,_)},!,[T].
term2bitpars(T,[0|Ps],NewPs)-->{T=fun(F,Xs)},[fun(F)],args2bitpars(Xs,Ps,NewPs).

args2bitpars([],[1|Ps],Ps)-->[].
args2bitpars([X|Xs],Ps,NewPs)-->term2bitpars(X,Ps,XPs),
  args2bitpars(Xs,XPs,NewPs).
```

The encoding is reversible, i.e., the term T can be recovered:

```
bitpars2term(Ps,As,T):-bitpars2term(CT,Ps,[],As,[]),fromCanonical(CT,T).

bitpars2term(T,[0,1|Ps],Ps)-->!,[T].
bitpars2term(T,[0|Ps],NewPs)-->[fun(F)],bitpars2args(Xs,Ps,NewPs),{T=fun(F,Xs)}.

bitpars2args([],[1|Ps],Ps)-->[].
bitpars2args([X|Xs],Ps,NewPs)-->bitpars2term(X,Ps,XPs),bitpars2args(Xs,XPs,NewPs).
```

Proposition 3
The predicates `term2bitpars` and `bitpars2term` define a bijection between a canonical term `T` and the set of pairs consisting of the bitlist representing the "structure" of `T` and the list of constants and variables representing its "content".

Note the mapping is only *injective* if its range is extended to the set of all finite bitstrings (i.e., the regular language $\{0,1\}^*$), as not all such strings represent sequences of balanced parentheses. The two transformations work as follows:

```
?-  TA=f(g(a,X),X,42),term2bitpars(TA,Ps,As),bitpars2term(Ps,As,TB).
TA = f(g(a, X), X, 42), Ps = [0, 0, 0, 1, 0, 1, 1, 0, 1|...],
As = [fun(7), fun(8), leaf(1, 2), leaf(0, 0), leaf(0, 0), leaf(2, 42)],
TB = f(g(a, A), A, 42) .
```

We describe next a *bijective* encoding to "Catalan skeletons" which abstract away the structure of a Prolog term as a unique natural number code.

4.2 A bijective structure encoding scheme

Providing a bijective encoding of the Catalan skeleton of a term is more challenging. For instance, a straightforward implementation (Tarau 2011), results in exponential representation sizes for sequences representing deep trees (and long Prolog lists in particular). Fortunately, algorithms for mapping sequences of balanced parentheses (also known as words in a Dyck language) are well known to combinatorialists (Kreher and Stinson 1999; Liebehenschel 2000). While some creativity is involved in translating to Prolog its `while` and `for` loops, the procedural algorithm of (Kreher and Stinson 1999), encapsulated in the predicates `rankCatalan` and `unrankCatalan`, is fairly routine and we refer to the **Appendix** for its details.

By combing the converters between terms to lists of parentheses with a bijection provided by `rankCatalan` and unrankCatalan we obtain:

```
bijEncodeStructure(T,N,As):-term2bitpars(T,Ps,As),rankCatalan(Ps,N).

bijDecodeStructure(N,As,T):-unrankCatalan(N,Ps),bitpars2term(Ps,As,T).
```

Proposition 4
The predicates `bijEncodeStructure` and `bijDecodeStructure` define a bijection between terms and pairs of natural numbers (representing the "structure" of the terms) and lists of constants and variables representing the "content" of the term.

The following example illustrates that the encoding is indeed reversible:

```
?- TA=f(a,g(X,Y),g(Y,X)),bijEncodeStructure(TA,N,As),bijDecodeStructure(N,As,TB).
TA = f(a, g(X, Y), g(Y, X)), N = 566,
As = [fun(7),leaf(1,2),fun(8),leaf(0,0),leaf(0,1),fun(8),leaf(0,1),leaf(0,0)],
TB = f(a, g(A, B), g(B, A)) .
```

Note also the succinctness, by comparison to the usual Prolog heap representations, of the "Catalan skeleton" `N` of the term, both on a long list and on a deep term.

```
?- bijEncodeStructure([a,b,a,b,a,b,a,b,a,b,a,b,a,b,a,b,a,b,a,b,a,b],N,As),
  bijDecodeStructure(N,As,T).
N = 722026845467271530176602,
As = [fun(0), leaf(1, 2), fun(0), leaf(1, 3), fun(0), leaf(1, 2)|...],
T = [a, b, a, b, a, b, a, b, a|...] .

?- bijEncodeStructure(f(f(f(f(f(f(f(f(f(f(a)))))))))),N,As),
   bijDecodeStructure(N,As,T).
N = 6918, As = [fun(7), fun(7), fun(7),...],T = f(f(f(f(f(...)))))))))) .
```

Finally, note that given a fixed (but possibly infinite) set of constant symbols and variables, this mapping can be seen as a bijection between natural numbers and terms built using variables in the given set and the constant symbols in the set as names of functions of arbitrary arities. Therefore, this encoding covers the case of term algebras with infinite signatures.

To complete the encoding of Prolog terms as natural numbers, we will need an encoding of the symbol and variable tables that is size-proportionate as well as a mechanism to merge the skeleton and content encodings together. Section 5 will provide the means to do this.

5 The generalized Cantor k-tupling bijection

The generalization of Cantor's pairing function is mentioned in two relatively recent papers (Cegielski and Richard 1999; Lisi 2007) with a possible attribution in (Cegielski and Richard 1999) to Skolem as a first reference.

The formula, given in (Cegielski and Richard 1999) p.4, looks as follows:

$$K_n(x_1,\ldots,x_n) = \sum_{k=1}^{n} \binom{k-1+x_1+\ldots+x_k}{k} \tag{1}$$

where $\binom{n}{k}$ 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, and a conjecture, attributed in (Cegielski and Richard 1999) to Rudolf Fueter (1923), states that it is the only one, up to a permutation of the arguments.

Computing the direct function given by equation 1 poses no challenges. So our problem reduces to *finding an efficient, linear or low polynomial algorithm for computing the inverse*. Toward this end we will focus next on an improvement that brings the performance of the algorithm given in (Tarau 2012) to a practical range, with the related predicates given in the **Appendix**.

5.1 The bijection between sets and sequences of natural numbers

After rewriting the formula for the $\mathbb{N}^n \to \mathbb{N}$ bijection as:

$$K_n(x_1,\ldots,x_n) = \sum_{k=1}^{n} \binom{k-1+s_k}{k} \tag{2}$$

where $s_k = \sum_{i=1}^{k} x_i$, we recognize 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 (Tarau 2009). It is implemented in the **Appendix** as the bijection `list2set` together with its inverse `set2list`.

5.2 The $\mathbb{N}^k \to \mathbb{N}$ bijection

The bijection $K_n : \mathbb{N}^n \to \mathbb{N}$ is basically just summing up a set of binomial coefficients. The predicate `binomial` used in this process is given in the **Appendix**. The predicate `fromCantorTuple` implements the the $\mathbb{N}^k \to \mathbb{N}$ 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`, given in the **Appendix**.

```
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).
```

5.3 The $\mathbb{N} \to \mathbb{N}^K$ bijection

We derive the efficient Cantor tupling bijection along the lines of (Tarau 2012), but with two significant improvements on performance.

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 (Lehmer 1964). Fortunately, efficient conversion algorithms between the conventional and the combinatorial number system are well known, (Buckles and Lybanon 1977; Knuth 2005).

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).
```

The improvement over (Tarau 2012)'s linear search-based algorithm is concentrated in the predicate `upperBinomial` which uses `roughLimit` to estimate an upper and lower bound on the value of the combinatorial digit and binary search for the actual value in this narrower range.

```
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 two 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).
```

Together, the predicates `roughLimit` and `binarySearch` provide a significant speed-up and scalability to very large numbers by replacing the linear search used in (Tarau 2012).

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).
```

The following example illustrates that it works as expected, including on very 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|...] .
```

6 The final encoding: a size-proportionate bijection from Prolog terms to $\mathbb{N}$

Our bijection between Prolog terms and $\mathbb{N}$ is obtained by putting all the pieces together.

The predicate `encodeTerm` first splits a Prolog terms into a Catalan skeleton `Ps` represented as sequence of balanced parentheses by calling `term2bitpars` and a list of canonically represented symbols.

Next, `rankCatalan` provides the natural number encoding of the skeleton, and `encodeSyms` provides encodings for each of its (canonically represented) symbols.

Finally, fromCantorTuple applies the generalized Cantor bijection to the encoded symbols, as well as the final pairing of the skeleton and symbol encodings. It uses fromCantorTuple twice, first to glue together the encoding of the symbols and then to pair together the skeleton code N and the symbol list code M.

```
encodeTerm(T,Code):-
  term2bitpars(T,Ps,Xs),rankCatalan(Ps,N),encodeSyms(Xs,Ns),
  fromCantorTuple(Ns,M),fromCantorTuple([N,M],Code).
```

The predicate decodeTerm proceeds (in reverse order) by inverting each of the previous steps. Note that K is *inferred* as being exactly the half of the length L of the Catalan skeleton, as symbols correspond one-to-one to the open parentheses. This is needed by the predicate toCantorTuple to split its SymsCode into exactly K symbol codes Ns.

```
decodeTerm(Code,T):-
  toCantorTuple(2,Code,[SkelCode,SymsCode]),unrankCatalan(SkelCode,Ps),
  length(Ps,L),K is L // 2,
  toCantorTuple(K,SymsCode,Ns),decodeSyms(Ns,Ps,Syms),bitpars2term(Ps,Syms,T).
```

The predicate encodeTerm relies on encodeSyms, which, in the case of leaf objects, merges together the type tags and the corresponding data by using putBDigit and, in the case of function terms, just passes on their canonical encodings.

```
encodeSyms([],[]).
encodeSyms([leaf(T,C)|Xs],[N|Ns]):-!,
  totalTypes(B),putBDigit(B,T,C,N1),N is N1-1,encodeSyms(Xs,Ns).
encodeSyms([fun(N)|Xs],[N|Ns]):-encodeSyms(Xs,Ns).
```

The predicate decodeTerm relies on decodeSyms, which, in the case of leaf objects, splits the encodings into type tags and the corresponding data, by using getBDigit, and in the case of function objects, it just extracts their encoding. Note that the Catalan skeleton is used to disambiguate between atomic symbols (corresponding to an opening and closing parentheses sequence 0,1) and function encodings (corresponding to an opening parenthesis 0).

```
decodeSyms([],[],[]).
decodeSyms([N|Ns],[0,1|Ps],[X|Xs]):-!,
  totalTypes(B),N1 is N+1,getBDigit(B,N1,T,C),
  X=leaf(T,C),decodeSyms(Ns,Ps,Xs).
decodeSyms([N|Ns],[0|Ps],[X|Xs]):-!,X=fun(N),decodeSyms(Ns,Ps,Xs).
decodeSyms(Ns,[1|Ps],Xs):-decodeSyms(Ns,Ps,Xs).
```

The following example illustrates their use:

```
?- encodeTerm(f(0,X,g(X,h(X)),a,b,1),N),decodeTerm(N,T).
N = 8571185322691882145456, T = f(0, A, g(A, h(A)), a, b, 1) .
```

7 Sketch of time complexity evaluation

A precise time complexity analysis, given all the algorithms involved is beyond the scope of this paper. However, we will sketch here an estimate on the overall time complexity of the encoding end decoding operations, that we will support it empirically with a few simple tests.

A large number of putBDigit and getBDigit operations happen close to the "leaves" of the term trees and their operands often fit in a 64 bit word, but their impact would be more significant if encoding very long symbols. Note that they could be also speeded up if one implements division and multiplication by 2 as bitshifts.

bitsize	100	200	300	400	500	600	700	800	900	5000
decodeTerm	7	21	51	76	134	170	245	324	413	40666
encodeTerm	4	11	23	37	59	82	111	142	178	15025

Fig. 1. Effort in thousands of logical inferences for bitsizes from 100 to 900 and 5000.

bitsize	100	200	300	400	500	600	700	800	900	5000
decodeTerm	3	16	48	70	132	169	261	342	460	21472
encodeTerm	2	7	23	42	64	95	153	232	283	27317

Fig. 2. Effort in milliseconds of CPU-time for bitsizes from 100 to 900 and 5000.

Conversion effort to/from canonical forms is proportional to the size of the terms.

The inner loops of the Cantor tupling/untupling functions are dominated by binomial coefficient computations. Our algorithm does k multiplications and integer division for $\binom{n}{k}$. While a state of the art arbitrary length integer package like the GMP library used by SWI-Prolog switches from "traditional multiplication" to Karatsuba and then FFT-multiplication, depending on bitsize, we can estimate that the "traditional multiplication" provides a conservative upper bound at $O(N^2)$. The use of binary search might increase that by a $O(log(N))$ multiplicative factor. As these operations dominate on both the encoding and decoding side, we can estimate that raw CPU-times for our algorithms are low polynomial, typically not much higher than quadratic in the bitsize of any of the two sides.

Empirical tests confirm these estimates on complexity, as shown in the figures 1 and 2 both in terms of logical inferences and CPU-time. The SWI-Prolog queries used in the tests are variants of

```
?- X is 2^100,time(decodeTerm(X,T)),time(encodeTerm(T,N)),fail.
```

with exponents of 2 ranging from `100` to `900` and then 5000 to illustrate that the algorithms scale up for larger bitsizes.

Also, the fact that effort growths in `decodeTerm` (subject to range narrowing and binary search) and `encodeTerm` (straightforward deterministic computations) are *proportionate*, indicates clear improvements over the inverse Cantor bijection algorithm of (Tarau 2012).

8 Related work

In (Tarau 2012), a step by step derivation of the inverse of the Cantor bijection is given, without the (critical) optimizations provided here by the predicates `upperBinomial`, `roughLimit` and `binarySearch` which avoid linear search when computing the digits of the combinatorial number system used in the $\mathbb{N} \rightarrow \mathbb{N}^k$ bijection.

A Prolog-based paper (Tarau 2011) with a similar scope has been presented by the author at the CICLOPS'2011 workshop with no formal proceedings. However the algorithms presented there only work for encoding small terms as they are not based on the polynomial Cantor encoding and do not use a size-proportionate encoding of the Catalan skeletons. Moreover, the bijective encodings of (Tarau 2011) leave the symbols and variables occurring in a Prolog term unencoded. A Haskell-based paper, also focusing on bijective encodings, that does not guarantee size-proportionality, has appeared in the informal proceedings of TFP'2010.

This paper can be seen as an application to the data transformation framework (Tarau 2009)

which helps gluing together the pieces needed for the derivation of our bijective encoding of term algebras.

We have not found in the literature an encoding scheme for term algebras that is *bijective* and is computable both ways in *space proportional to the size of the inputs*.

On the other hand, *ranking* functions for sequences can be traced back to Gödel numberings (Gödel 1931; Hartmanis and Baker 1974) associated to formulas. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms (Martinez and Molinero 2003; Knuth 2006).

We have found the first reference to the generalization of Cantor's pairing function to n-tuples in (Cegielski and Richard 1999), and benefited from the extensive study of its properties in (Lisi 2007).

Combinatorial number systems can be traced back to (Lehmer 1964) and one can find efficient conversion algorithms to conventional number systems in (Knuth 2005) and (Buckles and Lybanon 1977).

Finally, the "once you have seen it, obvious" `list2set / set2list` bijection is borrowed from (Tarau 2009), but not unlikely to be common knowledge of people working in combinatorics or recursion theory.

9 Conclusion

We have described a compact bijective serialization mechanism for Prolog terms that can be seen, more generally, as a bijective Gödel numbering scheme for term algebras with an an infinite number of variable and functions symbols.

As novel contributions, that together ensure that the two sides of the bijections are proportionate in size, we mention the separate encoding of the Catalan skeleton and the symbol content of a term, the improved algorithm for inverting the generalized Cantor bijection from $\mathbb{N}^k$ to $\mathbb{N}$ and the use of bijective base-k numbers for encoding and tagging various data types.

Our algorithms have applications ranging from generation of random instances to exchanges of structured data between declarative languages and/or theorem provers and proof assistants.

Tabling algorithms can make use of our encodings (seen as perfect hashing functions) for variant checking. A similar use can help the grounding stages interfacing logic and constraint programming systems with SAT and ASP solvers.

We foresee also applications as a generalized serialization mechanism usable to encode complex information streams with heterogeneous subcomponents - for instance as a mechanism for sending serialized objects over a network.

Finally, given that our encodings are *bijective*, they can be used to generate random terms, which in turn, can be used to represent random code fragments. This could have applications ranging from generation of random tests to representation of populations in genetic programming and candidate generation in inductive logic programming systems.

Acknowledgement

This research has been supported by NSF research grant 1018172. We thank the anonymous reviewers of ICLP'13 for their thoughtful comments and their suggestions that have helped improve this paper.

References

BUCKLES, B. P. AND LYBANON, M. 1977. Generation of a Vector form the Lexicagraphical Index [G6]. *ACM Transactions on Mathematical Software 5,* 2 (June), 180–182.

CEGIELSKI, P. AND RICHARD, D. 1999. On arithmetical first-order theories allowing encoding and decoding of lists. *Theoretical Computer Science 222,* 12, 55 – 75.

GÖDEL, K. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik 38*, 173–198.

HARTMANIS, J. AND BAKER, T. P. 1974. On Simple Goedel Numberings and Translations. In *ICALP* (2002-02-01), J. Loeckx, Ed. Lecture Notes in Computer Science, vol. 14. Springer, Berlin Heidelberg, 301–316.

KNUTH, D. E. 2005. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional.

KNUTH, D. E. 2006. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. Addison-Wesley Professional.

KREHER, D. L. AND STINSON, D. 1999. *Combinatorial Algorithms: Generation, Enumeration, and Search*. The CRC Press Series on Discrete Mathematics and its Applications. CRC PressINC.

LEHMER, D. H. 1964. The machine tools of combinatorics. In *Applied combinatorial mathematics*. Wiley, New York, 5–30.

LIEBEHENSCHEL, J. 2000. Ranking and unranking of a generalized Dyck language and the application to the generation of random trees. *Séminaire Lotharingien de Combinatoire 43*, 19.

LISI, M. 2007. Some remarks on the Cantor pairing function. *Le Matematiche 62,* 1.

MARTINEZ, C. AND MOLINERO, X. 2003. Generic algorithms for the generation of combinatorial objects. In *MFCS*, B. Rovan and P. Vojtas, Eds. Lecture Notes in Computer Science, vol. 2747. Springer, Berlin Heidelberg, 572–581.

SALOMAA, A. 1973. *Formal Languages*. Academic Press, New York.

TARAU, P. 2009. An Embedded Declarative Data Transformation Language. In *Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009*. ACM, Coimbra, Portugal, 171–182.

TARAU, P. 2011. Bijective Term Encodings. In *CICLOPS'2011*. Lexington, KY.

TARAU, P. 2012. Deriving a Fast Inverse of the Generalized Cantor N-tupling Bijection. In *28-th International Conference on Logic Programming - Technical Communications (ICLP'12)*, A. Dovier and V. S. Costa, Eds. Budapest, Hungary.