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

Paul Tarau[1]

[1] Department of Computer Science and Engineering
University of North Texas

ICLP 2013

Motivation

- serialization: a *reversible* transformation of an arbitrary data object to a bitstring (same as encoding to an arbitrary size natural number, just an instance of a *“Gödel numbering”* scheme)
- traditional serialization mechanisms encode to byte streams *injectively* but the encodings are *not surjective* , as not every possible bitstring configuration corresponds to a syntactically well formed object:
  - $decode(encode(term)) = term$
  - $encode(decode(number)) = number$???
- the challenges motivating this paper are:
  1. can we make serialization bijective, such that every bitstring corresponds uniquely to a *syntactically well formed Prolog term*?
  2. can we ensure that the encoding is size-proportionate (i.e, uses space proportional to the “printed” representation of the term)?
  3. can we do it fast enough to be practical?

Why are bijective term encodings useful?

1. they can be used for the generation of terms in random testing and in Prolog such terms can also represent code fragments
2. tabling algorithms can make use of bijective encodings (seen as perfect hashing functions)
3. the encodings can benefit to the grounding stages interfacing logic and constraint programming systems with SAT and ASP solvers
4. usable to represent populations in genetic programming and candidate generation in inductive logic programming systems
5. they are usable for exchanges of structured data between systems
6. they are usable for sending serialized objects over a network

Why this is not as easy as it seems?

Why does this seem easy?

- injective-only (but size-proportionate) mappings are widely used:
  - serialization in OO languages and networking protocols
  - terms in compiled form are usually more compact than in source form

Why it is *not easy* to design encodings that are bijective and size-proportionate?

- terms have a combinatorialy very rich tree structure (there is a very large number of trees of a given size)
- $\Rightarrow$ the mapping from terms to numbers tends to explode exponentially
- encoding of long symbols can place relatively large numbers at the leaves of a tree which will increase very quickly when propagated up

$\Rightarrow$ a reusable excuse for computer scientists to do what they want to do:

We choose to go to the moon in this decade and do the other things. Not because they are easy, but because they are hard. J.F. Kennedy

A look-ahead to how our bijective term encoding will work

Prolog code at `http://logic.cse.unt.edu/tarau/research/2013/serpro.pl`

- our encoding will associate a unique $n \in \mathbb{N}$ to a term
- for all $n \in \mathbb{N}$ a unique term will be generated from it

first, observe that the encoding is size-proportionate:

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

next, let's use *the first few digits of* π , to emphasize that *any* $n \in \mathbb{N}$ decodes to a syntactically valid Prolog term:

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

Outline

A canonical representation of Prolog terms

Our term type in Haskell notation is:

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

The conversion API:

```
toCanonical(PrologTerm, CanonicalTerm) :- ...

fromCanonical(CanonicalTerm, PrologTerm) :- ...
```

- `toCanonical` recurses over the structure of Prolog terms and builds a canonical representation
- `fromCanonical` reverses this transformation (up to a variant term)

```
?- toCanonical(f(X,g(X),a,10),T), fromCanonical(T,P).
T = fun(7, [leaf(0, 0), fun(8, [leaf(0, 0)]),
            leaf(1, 2), leaf(2, 21)]),
P = f(A, g(A), a, 10) .
```

Encoding numbers in bijective base-k

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

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

They work one digit at a time, incrementally, using:

```
putBDigit(Base,Digit,Before, After):-After is 1+Digit+Base*Before.
getBDigit(Base,Before,Digit, After):- ...
```

- they will encode strings made of symbols of a finite alphabet
- the incremental steps `putBDigit` and `getBDigit` will be also used to *tag a data object of arbitrary size*, reversibly:

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

Bijective string encodings

Strings can be seen as numbers in bijective base-k where k is the size of the alphabet.

```
?- 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).
```

Proposition

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

“Catalan skeletons” of Prolog terms

- The separation of the “structure” and the “content” of Prolog terms is needed to avoid “heavy leaves” that result in very large codes for term trees when encodings propagate to the roots.
- ⇒ Catalan families: combinatorial structures isomorphic to the Dyck language of the balanced parenthesis and a large number of different data types (among which multi-way and binary trees)

$$f(a,g(b,X),h(X,10)) \Rightarrow (()(()())(()())) + [f,a,g,b,X,h,X,10]$$

`TERM` `SKELETON` `SYMBOLS`

Extracting the Catalan skeleton of a term

- convert a Prolog term `T` to/from Catalan skeleton `Ps` + list of symbols `As`
- `Ps` is represented as list of parentheses, "`(`" ⇒ 0 and "`)`" ⇒ 1

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

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

```
?-  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) .
```

A bijective encoding of the Catalan skeletons

Proposition

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.*

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

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

`rankCatalan` and `unrankCatalan` (given in the appendix) are well known algorithms, using binary search (still faster with arrays, see `http://code.google.com/p/bijective-goedel-numberings/`)

Examples of bijective encodings of Catalan skeletons

`bijEncodeStructure` and `bijDecodeStructure` define together a bijective encoding of the skeletons, assuming that the same symbol list is available when decoding

long list:

```
?- 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 = 7220268454672715301766O2, % <<<<<<<<<<<<<<<<<<<<<< CODE
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|...] .
```

deep term:

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

The bijection between sets and sequences of natural numbers

To complete the encoding of Prolog terms as natural numbers, we will need an encoding of the symbol lists that is size-proportionate, as well as a mechanism to merge the skeleton and content encodings together.
a first step is:

- lists and sets of natural numbers (represented canonically) can be morphed into each other using a simple bijection
- see PPDP'2009: An Embedded Declarative Data Transformation Language for various such morphings – (also, code in appendix)

```
?- list2set([2,0,1,2],Set).
Set = [2, 3, 5, 8].

?- set2list([2, 3, 5, 8],List).
List = [2, 0, 1, 2].
```

The generalized Cantor k-tupling bijection

- we need a bijection between $\mathbb{N}^k$ and $\mathbb{N}$ to aggregate / separate between a list of codes and a code
- it is conjectured that the only one given by a polynomial formula is the generalized Cantor n-tupling bijection:

$$K_n(x_1,\ldots,x_n) = \binom{n-1+x_1+\ldots+x_n}{n} + \ldots + \binom{1+x_1+x_2}{2} + \binom{x_1}{1} \quad (1)$$

- $\binom{n}{k}$: binomial coefficient, "n choose k"
- $\binom{n}{k}$ is easy to compute, but care is taken to use a tail recursive predicate (see the appendix of the paper)
- EXAMPLE: $K_3(x_1,x_2,x_3) = \binom{2+x_1+x_2+x_3}{3} + \binom{1+x_1+x_2}{2} + \binom{x_1}{1}$
- $K_3(2,0,3) = \binom{2+2+0+3}{3} + \binom{1+2+0}{2} + \binom{2}{1} = \binom{7}{3} + \binom{3}{2} + \binom{2}{1}$
- $K_3(2,0,3) = 35+3+2 = 40$

```
?- from_cantor_tuple([2,0,3],N).
N = 40.
```

The case $n = 2$: Cantor's pairing function

- $K_2(x_1, x_2) = x_1 + \frac{(x_1+x_2+1)(x_1+x_2)}{2}$
- $K'_2(x_1, x_2) = x_2 + \frac{(x_1+x_2+1)(x_1+x_2)}{2}$ (symmetric alternative)
- K_2 and K'_2 are the only ones known to be polynomials in $x_1,\ x_2$
- the inverse of $K_2(x_1, x_2)$ has a simple closed formula - involving an integer square root operation, but we have found no algorithm for computing the inverse in the n>2 case in the literature
- the naive inversion algorithm - "enumerate until you hit the right one" becomes quickly intractable
- we needed to invent one – see our ICLP'12 paper for the step-by-step derivation of the algorithm, but 2 further optimizations given in this paper are needed to make it practical

The generalized Cantor tupling: from $\mathbb{N}^K$ to $\mathbb{N}$

quite simple: morph lists to sets, then sum up binomials

```
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).
```

The inverse of Cantor's pairing function: a geometric view

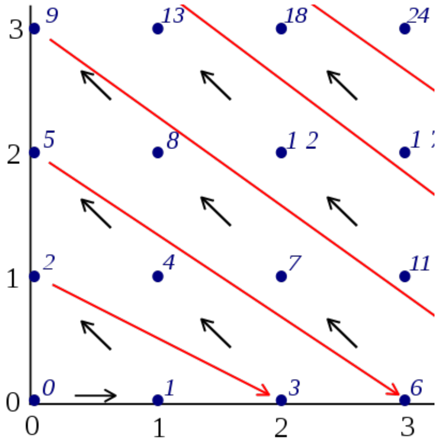

Figure : Path connecting pairs associated to successive natural numbers by the inverse of Cantor's pairing function (credit: Wikipedia)

A fast algorithm for the inverse $\mathbb{N} \to \mathbb{N}^K$ bijection

- we split our problem in two simpler ones: inverting `fromKSet` and then applying `set2list` to get back from sets to lists
- the predicate `untuplingLoop` used by `fromKSet` implements the sum of the combinations $\binom{X_1}{1} + \binom{X_2}{2} + \ldots + \binom{X_K}{K} = N$, the representation of N in the *combinatorial number system of degree K*
- conversion algorithms between the conventional and the combinatorial number system are well known, except that one should be careful to ensure they scale up when very large numbers are involved

Finding the combinatorial digits efficiently

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

combinatorialDigits(0,_,Ds,Ds).
combinatorialDigits(K,N,Ds,NewDs):-K>0,K1 is K-1,
  upperBinomial(K,N,M),  % <<<<<<<<<<< key optimization here
  M1 is M-1,
  binomial(M1,K,BDigit),
  N1 is N-BDigit,
  combinatorialDigits(K1,N1,[M1|Ds],NewDs).
```

inside `upperBinomial`:

- the predicate `roughLimit` narrows the range for the largest digit
- the predicate `binarySearch` finds the largest digit
- `combinatorialDigits` iterates to the next digit

The optimizations for finding the largest combinatorial digit

```
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 efficient $\mathbb{N} \to \mathbb{N}^K$ inverse mapping at work

The predicates `toKSet` and `fromKSet` implement inverse functions, mapping between natural numbers and 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 = 2080295455857036884841 9...567744,
Ns = [0, 0, 2, 0, 0, 0, 0, 0, 1|...] .
```

Putting it all together: a size-proportionate bijection from Prolog terms to $\mathbb{N}$

- `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
- `decodeTerm` proceeds (in reverse order) by inverting each of the previous steps

The final code

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

note `encodeSyms` and `decodeSyms` that collect encodings from the list canonical forms of symbols and variables

```
decodeTerm(Code,T):-
  toCantorTuple(2,Code,[SkelCode,SymsCode]),
  unrankCatalan(SkelCode,Ps),
  length(Ps,L),K is L // 2, % <<<<< K = how many symbols we have
  toCantorTuple(K,SymsCode,Ns),
  decodeSyms(Ns,Ps,Syms),bitpars2term(Ps,Syms,T).
```

```
?- encodeTerm(f(0,X,g(X,h(X)),a,b,1),N),decodeTerm(N,T).
N = 857118532269188214545 6, T = f(0, A, g(A, h(A)), a, b, 1) .
```

Complexity estimates

- dominated by large binomial computations, sensitive to GMP's multiplication algorithm
- low polynomial, typically not much higher than quadratic in the bitsize of any of the two sides
- size-proportionate
- note that effort growth for the two sides of the bijection is similar

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

Figure : 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

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

Conclusion

- our 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
- unique applications of the bijective aspect: easy generation of Prolog terms from (random) numbers, with uses for automated testing, possibly also for inductive logic programming and genetic algorithms
- Prolog code at `http://logic.cse.unt.edu/tarau/research/2013/serpro.pl`
- Scala and Haskell based open source projects: at: `http://code.google.com/p/bijective-goedel-numberings/`
- work supported by NSF research grant 1018172