Bijective Term Encodings

Paul Tarau

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

Abstract. We encode/decode Prolog terms as unique natural numbers. Our encodings have the following properties: a) are bijective b) natural numbers always decode to syntactically valid terms c) they work in low polynomial time in the bitsize of the representations d) the bitsize of our encodings is within constant factor of the syntactic representation of the input.
We describe encodings of term algebras with finite signature as well as algorithms that separate the "structure" of a term, a natural number encoding of a list of balanced parenthesis, from its "content", a list of atomic terms and Prolog variables.
The paper is organized as a literate Prolog program available from `http://logic.cse.unt.edu/tarau/research/2011/bijenc.pl`.
Keywords: *natural number encodings of term algebras with finite signatures bijective Gödel numberings for Prolog terms ranking/unranking functions for tuples and lists Catalan skeletons of Prolog terms*

1 Introduction

A *ranking/unranking* function defined on a data type is a bijection to/from the set of natural numbers (denoted $\mathbb{N}$). When applied to formulas or proofs, ranking functions are usually called *Gödel numberings* as they have originated in arithmetization techniques used in the proof of Gödel's incompleteness results [1,2]. In Gödel's original encoding [1], given that primitive operation and variable symbols in a formula are mapped to exponents of distinct prime numbers, factoring is required for decoding, which is therefore intractable for formulas of non-trivial size. As this mapping is not a surjection, there are codes that decode to syntactically invalid formulas. This key difference also applies to alternative Gödel numbering schemes (like Gödel's beta-function), while ranking/unranking functions, as used in combinatorics, are bijective mappings.

Besides codes associated to formulas, a wide diversity of common computer operations, ranging from data compression and serialization to data transmissions and cryptographic codes are essentially bijective encodings between data types. They provide a variety of services ranging from free iterators and random objects to data compression and succinct representations. Tasks like serialization and persistence are facilitated by simplification of reading or writing operations without the need of special-purpose parsers.

The main focus of this paper is designing an efficient bijective Gödel numbering scheme (i.e. a ranking/unranking bijection) for *term algebras*, essential building blocks for various data types and programming language constructs.

The resulting Gödel numbering algorithm, the main contribution of the paper, enjoys the following properties:

1. the mapping is bijective
2. natural numbers always decode to syntactically valid terms
3. it works in time low polynomial in the bitsize of the representations
4. the bitsize of our encoding is within constant factor of the syntactic representation of the input.

These properties ensure that our algorithm can be applied to derive compact serialized representations for various formal systems and programming language constructs.

2 Tuple Encodings

We will now define a few primitive operations in terms of a small set of bitwise primitives with known asymptotic complexity. Assuming a copying implementation of arbitrary size integers each of the following operations are at most linear in the bitsize of their operand `N` and some can be considered constant time when this operand fits in a machine word as well as when an efficient mutable implementation of arbitrary length integers is used.

```
first_bit(N,Bit):- Bit is 1 /\ N.
times_exp2(N,K,R):-R is N << K.
div_by_exp2(N,K,R):-R is N >> K.
predecessor(N,R):-R is N-1.
successor(N,R):-R is N+1.
```

First we define the `k_deflate` and `k_inflate` operations. `k_deflate` can be seen as collecting each k-th bit from a number's binary representation and aggregates the result into a new natural number. `k_inflate` can be seen as building a new natural number by inserting `0`s in every position except in each k-th position where the bits of its argument `N` are placed. However, we avoid direct bitlist manipulation by expressing them in terms of the previously defined arbitrary length integer operations.

```
k_deflate(_,0,0).
k_deflate(K,N,R):-N>0,
  div_by_exp2(N,K,A),
  k_deflate(K,A,B),
  times_exp2(B,1,C),
  first_bit(N,D),
  R is C\/D.
```

```
k_inflate(_,0,0).
k_inflate(K,N,R):-N>0,
  div_by_exp2(N,1,A),
  k_inflate(K,A,B),
  times_exp2(B,K,C),
  first_bit(N,D),
  R is C\/D.
```

The following example illustrates their use:

```
?- k_inflate(3,42,X),k_deflate(3,X,Y).
X = 33288,
Y = 42 .
```

We can define a bijective decomposition of a natural number `N` as a tuple of `K` natural numbers in terms of `k_deflate` and our primitive bitwise operations.

The function `to_tuple:` $Nat \to Nat^k$ converts a natural number to a k-tuple by splitting its bit representation into k groups, from which the k members in the tuple are finally rebuilt. This operation can be seen as a transposition of a bit matrix obtained by expanding the number in base 2^k:

```
to_tuple(K,N,Ns):-K>0,
  predecessor(K,K1),
  numlist(0,K1,Ks),
  maplist(div_by_exp2(N),Ks,Ys),
  maplist(k_deflate(K),Ys,Ns).
```

Note the use of the SWI-Prolog library predicates `numlist` that generates a list of integers in increasing order and `maplist` that applies a closure to lists of corresponding arguments. To convert a k-tuple back to a natural number we will merge their bits, k at a time. This operation can be seen as the transposition of a bit matrix obtained from the tuple, seen as a number in base 2^k, but we implement it more efficiently in terms of bitwise operations on integers. Note the use of the SWI-Prolog library predicate `sumlist` that computes the sum of a list of numbers.

```
from_tuple(Ns,N):-
  length(Ns,K),K>0,
  predecessor(K,K1),
  maplist(k_inflate(K),Ns,Xs),
  numlist(0,K1,Ks),
  maplist(times_exp2,Xs,Ks,Ys),
  sumlist(Ys,N).
```

The following example shows the mapping of `42` to a 3-tuple and the encoding back to `42`.

```
?- to_tuple(3,42,T),from_tuple(T,N).
T = [2, 1, 2],
N = 42 .
```

Fig. 1 shows multiple steps of the same decomposition, with shared nodes collected in a DAG. Note that markers on edges indicate argument positions.

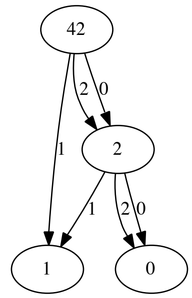

Fig. 1: 42 after repeated 3-tuple expansions

Note that one can now define *pairing functions*, i.e. *bijections between natural numbers and pairs of natural numbers*, as specializations of the tupling/untupling predicates:

```
to_pair(N,A,B):-to_tuple(2,N,[A,B]).
```

```
from_pair(X,Y,Z):-from_tuple([X,Y],Z).
```

One can observe that `to_pair` and `from_pair` are the same as the functions defined in Steven Pigeon's PhD thesis on Data Compression [3] and also known as Morton-codes with uses in indexing of spatial databases [4].

3 Gödel numberings of a term algebra with finite signature

Traditionally a term algebra is defined over a finite set of functions symbols of given arities. Constants can be singled out as a special set or considered function symbols of arity 0. Term algebras are *free magmas*[1] induced by a set of variables and a set of function symbols of various arities (0 included), called `signature`, that are closed under the operation of inserting terms as arguments of function symbols. In various logic formalisms a term algebra is called a Herbrand Universe.

Having a bijective encoding over a signature and a (finite) set of variables (seen as input "wires") is also useful for synthesizing code over a set of functions - for instance a library of logic gates, in the case of circuit synthesis, as well as for generating random terms of a given signature for testing purposes.

Besides being bijective, it is useful if the mapping relates terms to numbers of comparable representation size and if it works in linear or low polynomial time to be useful for practical applications.

We will denote `Vs, CSyms, FSyms` the sets of variables, constant symbols and function symbol/arity pairs, respectively, that parameterize the converter from terms to codes `term2nat/5` and the converter from codes to terms, `nat2term/5`.

Given that `Vs, CSyms` are finite, we map them bijectively to the ranges `[0..LV-1]` (variables), `[LV..LV+LC-1]` (constants). The predicate `term2nat` precomputes these values and then calls the recursive converter `t2n`.

[1] See `http://wikipedia.org/wiki/Free_object`

```
term2nat(Vs,CSyms,FSyms,T, X):-
  length(CSyms,LC),
  length(FSyms,LF),
  length(Vs,LV),
  LVC is LV+LC,
  LVC>0,
  t2n(LV,LC,LF,LVC,Vs,CSyms,FSyms,T, X).
```

The predicate `t2n` uses `lookup_var` and the built-in `nth0/3` to look-up indices associated to variable, constant and function symbols. For compound terms, these values are combined with values computed recursively on their arguments and then merged using `from_tuple` into natural numbers.

Note that the index `L` of the function symbol `F/K` computed by `nth0/3` is multiplied with the length `LF` of the list of function symbols `FSyms`. This operation will be reversed using modulo and quotient when converting back.

```
t2n(LV,_LC,_LF,_LVC,Vs,_CSyms,_FSyms,V, X):-var(V),!,
  lookup_var(I,Vs,V),
  I>=0,I<LV,X=I.
t2n(LV,_LC,_LF,_LVC,_Vs,CSyms,_FSyms,C, X):-atomic(C),!,
  nth0(I,CSyms,C),
  X is I+LV.
t2n(LV,LC,LF,LVC,Vs,CSyms,FSyms,T, X):-compound(T),
  T=..[F|Ts],
  nth0(L,FSyms,F/K),
  K>0,
  length(Args,K),
  P=..[t2n,LV,LC,LF,LVC,Vs,CSyms,FSyms],
  maplist(P,Ts,Args),
  from_tuple(Args,N),
  X is LVC+LF*N+L.
```

```
lookup_var(N,Xs,X):-lookup_var(X,Xs,0,N).
```

```
lookup_var(X,[Y|_],N,N):-X==Y.
lookup_var(X,[_|Xs],N1,N3):-
  N2 is N1+1,
  lookup_var(X,Xs,N2,N3).
```

The predicate `nat2term` reverses the process, using the same lists `Vs,CSyms,FSyms` to map variables, constants and function symbols to natural number codes, by calling the recursive converter `n2t/8`.

```
nat2term(Vs,CSyms,FSyms,X, T):-X>=0,
  length(CSyms,LC),
  length(FSyms,LF),
  length(Vs,LV),
  LVC is LV+LC,LVC>0,
  n2t(LV,LC,LF,LVC,Vs,CSyms,FSyms,X, T).
```

The recursive converter `n2t` uses dictionaries `Vs, CSyms, FSyms` to map natural numbers to the corresponding, functions, constant and variable terms, uniformly. Note the use of the library predicate `nth0` that associates an index, starting at 0, to a term on a list. It also uses Prolog's `univ` to build a closure `P` that with help from `maplist` applies it recursively.

```
n2t(LV,_LC,_LF,_LVC,Vs,_CSyms,_FSyms,X, V):-X<LV,!,
  nth0(X,Vs,V).
n2t(LV,_LC,_LF,LVC,_Vs,CSyms,_FSyms,X, C):-LV=<X,X<LVC,!,
  X0 is X-LV,
  nth0(X0,CSyms,C).
n2t(LV,LC,LF,LVC,Vs,CSyms,FSyms,X, T):-X>=LVC,
  X0 is X-LVC,
  N is X0 // LF,
  L is X0 mod LF,
  nth0(L,FSyms,F/K),
  K>0,
  to_tuple(K,N,Args),
  P=..[n2t,LV,LC,LF,LVC,Vs,CSyms,FSyms],
  maplist(P,Args,Ts),
  T=..[F|Ts].
```

Note the use of the predicate `to_tuple` with length `K` based on the arity of each function symbol, which splits the natural number `N` in a list of codes `Args` to be used recursively to build the subterms associated to the function symbol `F/K`.

A first example shows that starting from a term `T` we obtain a natural number from which the same term `T` is recovered. Note that the two side of the transformer are parameterized by the same lists of variables, constants and function symbols.

```
?- T=f(a,f(X,g(Y))),Vs=[X,Y],Cs=[a],Fs=[f/2,g/1],
   term2nat(Vs,Cs,Fs,T,N),nat2term(Vs,Cs,Fs,N,T_again).
T = f(a, f(X, g(Y))),
Vs = [X, Y],
Cs = [a],
Fs = [f/2, g/1],
N = 17439,
T_again = f(a, f(X, g(Y))) .
```

The next example shows that starting from any natural number e.g. `2012` we obtain a term that in turn is converted back to the same number.

```
?- N=2012,Vs=[X,Y],Cs=[a,b],Fs=[f/2,g/1],
   nat2term(Vs,Cs,Fs,N,T),term2nat(Vs,Cs,Fs,T,N_again).
N = 2012,
Vs = [X, Y],
Cs = [a, b],
Fs = [f/2, g/1],
T = f(f(Y, b), f(b, a)),
N_again = 2012 .
```

Finally, the following example (where `'->'` is seen as logical implication), hints towards an application to circuit synthesis. When combined with a fast bitstring-based boolean evaluator (see [5]) terms associated with natural numbers can be tried out to see if the result of their boolean evaluation matches a given specification.

```
?- N=2012,Vs=[A,B],Cs=[0],Fs=['->'/2],
   nat2term(Vs,Cs,Fs,N,T),term2nat(Vs,Cs,Fs,T,N_again).
N = 2012,
Vs = [A, B],
Cs = [0],
Fs = [ (->)/2],
T = (((B->A)->0->A)-> (0->A)->B),
N_again = 2012 .
```

One can generate random terms with a given signature based on a natural number of a given bitsize as follows.

```
ranterm(Bits,Vs,Cs,Fs, T):-
  N is random(2^Bits),
  nat2term(Vs,Cs,Fs,N,T).
```

This can be useful in generating random arithmetic expressions or boolean functions for testing purposes.

```
?- Vs=[A,B,C],ranterm(100,Vs,[],['+'/2,'*'/2],T).
Vs = [A, B, C],
T = (B+ (C+A))* ((A+B)*A* ((A+A)*C))+ (A+B+A*A+ (B+ (A+A))*(B+A))+
    (B*A*B* (B+B)* (C+ (C+B))+ (A* (A+A)+C*A)* ((B+A)* ((A+A)*C))) .

?- Vs=[A,B,C,D],ranterm(50,Vs,[0,1],[and/2,or/2,not/1],T).
Vs = [A, B, C, D],
T = and(not(not(or(or(not(0), A), or(and(B, B), A)))),
    or(or(and(A, A), or(D, A)), not(or(C, not(B))))) .
```

4 Bijective encodings of Prolog atoms

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

4.1 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*[2]. First we start with the mapping from list of digits in `[0..k-1]` to a natural number defined by the predicate `from_bbase/3`

[2] We refer to `http://en.wikipedia.org/wiki/Bijective_numeration` for the historical origins of the concept and the properties of this number representation.

```
from_bbase(Base,Xs,R):-
  maplist(successor,Xs,Xs1),
  from_base1(Base,Xs1,R).
```

```
from_base1(_Base,[],0).
from_base1(Base,[X|Xs],R):-X>0,X=<Base,
  from_base1(Base,Xs,R1),
  R is X+Base*R1.
```

```
to_bbase(Base,N,Xs):-
  to_base1(Base,N,Xs1),
  maplist(predecessor,Xs1,Xs).

to_base1(_,0,[]).
to_base1(Base,N,[D1|Ds]):-N>0,
   Q is N//Base,
   D is N mod Base,
   (D=:=0->D1=Base;D1=D),
   (D=:=0->Q1 is Q-1;Q1=Q),
   (Q1=:=0->Ds=[];to_base1(Base,Q1,Ds)).
```

Note that the predicates `from_bbase` and `to_bbase` are parametrized by the base of numeration which should be the same when encoding and decoding.

```
?- to_bbase(7,2012,Ds),from_bbase(7,Ds,N).
Ds = [2, 6, 4, 4],
N = 2012 .
```

This encoding will turn out to be useful for symbols of a finite alphabet.

4.2 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 mapping) we will assume that our strings naming functions are built only using lower case ASCII characters.

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

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

Next, we define the bijective base-k encodings

```
string2nat(Cs,N):-
  base(B),
  maplist(chr2ord,Cs,Ns),
  from_bbase(B,Ns,N).
```

```
nat2string(N,Cs):-N >= 0,
  base(B),
  to_bbase(B,N,Xs),
  maplist(ord2chr,Xs,Cs).
```

```
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 strings working as follows:

```
?- Cs="hello",string2nat(Cs,N),nat2string(N,CsAgain).
Cs = [104, 101, 108, 108, 111],
N = 7073802,
CsAgain = [104, 101, 108, 108, 111] .

?- nat2string(2012,Cs),string2nat(Cs,N).
Cs = [106, 121, 98],
N = 2012 .
```

And finally we can obtain a bijective encoding of Prolog atoms as

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

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

5 “Catalan skeletons” of Prolog terms

We will 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 [6,7]. We will start by mapping a term to a “skeleton” representing its structure as a list of balanced parentheses.

5.1 An injective-only structure encoding

We sketch here an encoding mechanism that might also be useful to Prolog implementors interested in designing alternative heap representations for new Prolog runtime systems or abstract machine architectures as well as hashing mechanisms for ground terms or variant checking for tabling.

First we provide an encoding that separates the “structure” of a term `T`, expressed as a balanced parenthesis language[3] representation `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,[0,1],[T]):-var(T).
term2bitpars(T,[0,1],[T]):-atomic(T).
term2bitpars(T,Ps,As):-compound(T),term2bitpars(T,Ps,[],As,[]).
```

[3] A member of the *Catalan family* of combinatorial objects.

```
term2bitpars(T,Ps,Ps)-->{var(T)},[T].
term2bitpars(T,Ps,Ps)-->{atomic(T)},[T].
term2bitpars(T,[0|Ps],NewPs)-->{compound(T),T=..Xs},
  args2bitpars(Xs,Ps,NewPs).
```

```
args2bitpars([],[1|Ps],Ps)-->[].
args2bitpars([X|Xs],[0|Ps],NewPs)-->
  term2bitpars(X,Ps,[1|XPs]),
  args2bitpars(Xs,XPs,NewPs).
```

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

```
bitpars2term([0,1],[T],T).
bitpars2term([P,Q,R|Ps],As,T):-bitpars2term(T,[P,Q,R|Ps],[],As,[]).
```

```
bitpars2term(T,Ps,Ps)-->[T].
bitpars2term(T,[0|Ps],NewPs)-->
  bitpars2args(Xs,Ps,NewPs),{T=..Xs}.
```

```
bitpars2args([],[1|Ps],Ps)-->[].
bitpars2args([X|Xs],[0|Ps],NewPs)-->
  bitpars2term(X,Ps,[1|XPs]),
  bitpars2args(Xs,XPs,NewPs).
```

The two transformations work as follows:

```
?- term2bitpars(f(g(a,X),X,42),Ps,As),
   bitpars2term(Ps,As,T).
Ps = [0,0,1,0,0,0,1,0,1,0,1,1,1,0,1,0,1,1],
As = [f,g,a,X,X,42],
T=f(g(a,X),X,42) .
```

By using this encoding one can further aggregate bitlists into natural numbers with `term2inj_code` by converting the resulting bitlists seen as bijective-base 2 digits and then convert them back with `inj_code2term`.

```
term2inj_code(T,N,As):-
  term2bitpars(T,Ps,As),
  from_bbase(2,Ps,N).
```

```
inj_code2term(N,As,T):-
  to_bbase(2,N,Ps),
  bitpars2term(Ps,As,T).
```

working as follows:

```
?- term2inj_code(f(a,g(X,Y),g(Y,X)),N,As),inj_code2term(N,As,T).
N = 131364115,
As = [f, a, g, X, Y, g, Y, X],
T = f(a, g(X, Y), g(Y, X)) .
```

Note however that this encoding is injective only i.e. not every natural number is a code of a term.

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

5.2 A Diophantine decomposition of natural numbers

First, we need a mechanism to bijectively encode/decode the actual information content of term as well as the arities associated to its function symbols.

As an immediate consequence of the unique decomposition of natural numbers in prime factors, the Diophantine equation

$$2^x(2y+1) = z \tag{1}$$

has, for any positive natural number `z` a unique solution `(x,y)`.

Using the `lsb/1` function that returns the least significant bit of a natural number (available, for instance, in SWI-Prolog and easy to emulate in other Prologs) one can define:

```
cons(X,Y, Z):-Z is ((Y<<1)+1)<<X.

decons(Z, X,Y):-Z>0, X is lsb(Z), Y is Z>>(X+1).
```

We will use these predicates to decompose a natural number `Z>0` into `X` and `Y` such that `X` is well suited to work as the length of a tuple and `Y` to provide the members of a tuple of length `X`, in a reversible way.

5.3 A bijection between natural numbers and lists

By combining `cons/3` and `decons/3` (which aggregate/separate "length" and "content") with `to_tuple` and `from_tuple` (which aggregate/separate a "content" of fixed "length"), we obtain an bijection between lists of numbers and numbers of size proportional to the bit representations of the operands.

```
nat2nats(0,[]).
nat2nats(N,Ns):-N>0,
  decons(N,L1,N1),
  L is L1+1,
  to_tuple(L,N1,Ns).
```

```
nats2nat([],0).
nats2nat(Ns,N):-
  length(Ns,L),
  L1 is L-1,
  from_tuple(Ns,N1),
  cons(L1,N1,N).
```

The following example illustrates that this encoding is a bijection:

```
?- nat2nats(2012,Ns),nats2nat(Ns,N).
Ns = [7, 7, 2],
N = 2012 .
```

5.4 A bijection between natural numbers and lists of balanced parenthesis

We can build a bijection between lists of balanced parenthesis and natural numbers by encoding sublists, recursively with `nats2nat/1` while parsing them with a DCG.

```
pars2nat(Xs,T):-pars2nat(0,1,T,Xs,[]).
```

```
pars2nat(L,R,N) --> [L],pars2nats(L,R,Xs),{nats2nat(Xs,N)}.

pars2nats(_,R,[]) --> [R].
pars2nats(L,R,[X|Xs])-->pars2nat(L,R,X),pars2nats(L,R,Xs).
```

The inverse mapping works in a similar way, using `nat2nats` to recursively generate the lists of balanced parenthesis using a DCG.

```
nat2pars(N,Xs):-nat2pars(0,1,N,Xs,[]).
```

```
nat2pars(L,R,N) --> {nat2nats(N,Xs)},[L],nats2pars(L,R,Xs).

nats2pars(_,R,[]) --> [R].
nats2pars(L,R,[X|Xs])-->nat2pars(L,R,X),nats2pars(L,R,Xs).
```

The following example illustrates that the two mappings are indeed invertible.

```
?- nat2pars(2012,Ps),pars2nat(Ps,N).
Ps = [0,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,1,0,1,1,1],
N = 2012 .
```

5.5 Bijective Catalan skeletons of Prolog terms

By combing the converters between terms to lists of parenthesis with a bijection provided by `pars2nat` and nat2pars we obtain:

```
term2code(T,N,As):-
  term2bitpars(T,Ps,As),
  pars2nat(Ps,N).
```

```
code2term(N,As,T):-
  nat2pars(N,Ps),
  bitpars2term(Ps,As,T).
```

As as the following example shows the encoding is indeed reversible:

```
?- term2code(f(a,g(X,Y),g(Y,X)),N,As),code2term(N,As,T).
N = 786632,
As = [f, a, g, X, Y, g, Y, X],
T = f(a, g(X, Y), g(Y, X)) .
```

Not also the succinctness, by comparison to the usual Prolog heap representations, of the “Catalan structure” of the term.

6 Related work

This paper can be seen as an application to the data transformation framework [8] which helps gluing together the pieces needed for the derivation of our bijective encoding of term algebras, including algebras with finite signatures, the novel contribution of this paper.

We have not found in the literature an encoding scheme for term algebras that is *bijective*, nor an encoding that is computable both ways with effort proportional to the size of the inputs.

On the other hand, *ranking* functions for sequences can be traced back to Gödel numberings [1,2] associated to formulas. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [9,10]. Pairing functions have been used in work on decision problems as early as [11]. A typical use in the foundations of mathematics is [12]. An extensive study of various pairing functions and their computational properties is presented in [13].

The closest reference on encapsulating bijections as a programming language data type is [14] and Conal Elliott's composable bijections Haskell module [15]. [16] uses a similar category theory inspired framework implementing relational algebra, also in a Haskell setting.

7 Conclusion

We have described a compact bijective Gödel numbering scheme for term algebras. The algorithm can be made to work in linear time and has applications ranging from generation of random instances to exchanges of structured data between declarative languages and/or theorem provers and proof assistants. We foresee some practical 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. Also, 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.

Acknowledgment

We thank NSF (research grant 1018172) for support.

References

1. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
2. Hartmanis, J., Baker, T.P.: On Simple Goedel Numberings and Translations. In Loeckx, J., ed.: ICALP. Volume 14 of Lecture Notes in Computer Science., Springer (1974) 301–316
3. Pigeon, S.: Contributions à la compression de données. Ph.d. thesis, Université de Montréal, Montréal (2001)

4. Lawder, J.: The application of space-filling curves to the storage and retrieval of multi-dimensional data (2000) PhD Thesis, University of London, UK.
5. Tarau, P., Luderman, B.: A Logic Programming Framework for Combinational Circuit Synthesis. In: 23rd International Conference on Logic Programming (ICLP), LNCS 4670, Porto, Portugal, Springer (September 2007) 180–194
6. Berstel, J., Boasson, L.: Formal properties of XML grammars and languages. Acta Informatica **38**(9) (2002) 649–671
7. Liebehenschel, J.: Ranking and unranking of a generalized Dyck language and the application to the generation of random trees. Séminaire Lotharingien de Combinatoire **43** (2000) 19
8. 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
9. Martinez, C., Molinero, X.: Generic algorithms for the generation of combinatorial objects. In Rovan, B., Vojtas, P., eds.: MFCS. Volume 2747 of Lecture Notes in Computer Science., Springer (2003) 572–581
10. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming). Addison-Wesley Professional (2006)
11. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
12. Cégielski, P., Richard, D.: Decidability of the theory of the natural integers with the cantor pairing function and the successor. Theor. Comput. Sci. **257**(1-2) (2001) 51–77
13. Rosenberg, A.L.: Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science **14**(1) (2003) 3–17
14. Alimarine, A., Smetsers, S., van Weelden, A., van Eekelen, M., Plasmeijer, R.: There and back again: arrows for invertible programming. In: Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM Press (2005) 86–97
15. Conal Elliott: Module: Data.Bijections Haskell source code library at: http://haskell.org/haskellwiki/TypeCompose.
16. Kahl, W., Schmidt, G.: Exploring (finite) Relation Algebras using Tools written in Haskell. Technical Report 2000-02, Fakultät für Informatik, Universität der Bundeswehr München (October 2000)