Bijective Size-preserving Gödel Numberings for Term Algebras

Paul Tarau

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

Abstract. We introduce Gödel numbering algorithms that encode/decode elements of term algebras with infinite and finite signatures as unique natural numbers. In contrast with Gödel's original encoding and various alternatives in the literature, our encodings are bijective and ensure that natural numbers always decode to syntactically valid terms. At the same time, our algorithms work in low polynomial time in the bitsize of the representations and the bitsize of our encodings is within a constant factor of the the syntactic representation of the input.
Our encodings rely in an essential way on a novel algorithm for the fast computation of an inverse of the generalized Cantor tupling function, conjectured to be (up to a permutation of arguments) the only bijection between $\mathbb{N}^k$ and $\mathbb{N}$ based on a polynomial formula.
As applications, the algorithms provide compact serialized representations for various formal system and programming language constructs. In the special case of finite signature algebras applications to circuit synthesis and genetic programming are possible.
The paper is organized as a literate Haskell program available from `http://logic.cse.unt.edu/tarau/research/2013/cgoedel.hs`.
Keywords: *natural number encodings of terms, bijective Gödel numberings, ranking/unranking functions for tuples, bijective base-k encodings, generalized Cantor tupling/untupling bijection, combinatorial numbers system*

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 with large or infinite signatures. 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 iterators and generation of 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. A Term in our term algebra is:

- a variable labeled with a natural number
- a constant symbol function symbol labeled with a natural number applied to a finite sequence of Terms

The set of variables as well as the set of constant and function symbols (called the *signature* of the algebra) can be finite or infinite.

Term algebras can be seen as a generic syntax for well-formed expressions in languages like predicate or lambda calculus as well as a generic syntax for "proof-terms" in proof assistants like Coq.

By restricting Goedel numberings to only well formed terms and ensuring that they are bijective, "no bit is lost" through the mapping to natural numbers, e.g. optimal information theoretical succinctness is achieved.

To achieve size-proportionate encoding s we use polynomial bijections provided by the generalized Cantor tupling function and its inverse for which we design a fast algorithm based on combinatorial number systems.

The resulting Gödel numbering algorithm (with a special treatment for finite signatures and variable sets) is the main contribution of the paper. It enjoys the following properties:

1. the mapping is bijective
2. natural numbers always decode to syntactically valid terms
3. it works in low polynomial time 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 algorithms can be applied to derive compact serialized representations for various formal systems and programming language constructs.

In combination with a symbol table for function symbols and heap position of variables used as labels, this can provide a possibly practical serialization algorithm for data types that can be seen as term algebras - including XML files and object graphs. In the special case of finite signature algebras applications to circuit synthesis and genetic programming are possible.

The paper is organized as follows: Section 2 provides the implementation of the mapping between n-tuples to natural numbers, using a *list-to-set bijection*

that reduces the computation of the inverse of the Cantor n-tupling function to a well-known combinatorial problem. Section 3 describes a bijection from arbitrary lists of natural numbers to $\mathbb{N}$ using the generalized Cantor $\mathbb{N}^k \to \mathbb{N}$ bijection. Section 4 presents the bijection between term algebras with infinite signatures and $\mathbb{N}$, while section 5 presents the bijection between term algebras with finite numbers of variables constants and function symbols and $\mathbb{N}$.

Section 6 discusses related work and section 7 concludes the paper.

The paper is written as a literate Haskell program, available as a separate file from `http://logic.cse.unt.edu/tarau/research/2013/cgoedel.hs`.

We group our code in a self-contained module `Goedel`, importing only a few library modules:

```
module Goedel where
import Data.List
import Data.Char
```

All our computations are over the set of natural numbers $\mathbb{N}$ approximated as a Haskell arbitrary size `Integer`:

```
type N = Integer
```

2 Implementing the Generalized Cantor n-tupling Bijection

Tupling/untupling functions are a natural generalization of pairing/unpairing operations. They are called *ranking/unranking* functions by combinatorialists as they map bijectively various combinatorial objects to $\mathbb{N}$ (ranking) and back (unranking).

The generalization of Cantor's pairing function is mentioned in two relatively recent papers [3, 4] with a possible attribution in [3] to Skolem as a first reference.

The formula, given in [3] 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 $\binom{n}{k}$, also called "binomial coefficient" denotes the number of subsets of n with k elements as well as the coefficient of x^k in the expansion of the binomial $(x + y)^n$.

This natural generalization of Cantor's pairing bijection described in [3] is introduced using geometric considerations that make it obvious that it defines a bijection $K_n : \mathbb{N}^n \to \mathbb{N}$. More precisely, they observe that the enumeration in $\mathbb{N}^2$ of integer coordinate pairs laying on the anti-diagonals $x_1 + x_2 = c$ can be lifted to points with integer coordinates laying on hyperplanes of the form $x_1 + x_2 + \ldots + x_k = c$. The same result, using a slightly different formula is proven algebraically, by induction in [4].

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 [3] to Rudolf Fueter (1923), states that it is the only one, up to a permutation of the arguments.

An efficient implementation of the function $K_n : \mathbb{N}^k \to \mathbb{N}$ is the easy part, just summing up a set of binomial coefficients.

On the other hand, the problem of inverting it means finding a solution of corresponding *Diophantine equation* $z = K_n(x_1, \ldots, x_n)$ and proving that it is unique.

As an inductive proof that K_n is a bijection is given in [4] (Theorem 2.1), we know that a solution exists and is unique, so the problem reduces to computing the only solution of the Diophantine equation.

While solving an arbitrary Diophantine equation is Turing-equivalent, things do not look that bad in this case, as an enumeration of all tuples $x_1, \ldots, x_n$ for $0 \leq x_i \leq z$ provides an obvious but dramatically inefficient solution. Developing a fast algorithm for that is one of the contributions of this paper.

We will proceed by providing first a few simple building blocks for our main algorithms.

2.1 Binomial Coefficients, efficiently

Computing binomial coefficients efficiently is well-known

$$\binom{k}{n} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\ldots(n-(k-1))}{k!} \tag{2}$$

However, we will need to make sure that we avoid unnecessary computations and reduce memory requirements by using a tail-recursive loop. After simplifying the slow formula in the first part of the equation (2) with the faster one based on falling factorial $n(n-1)\ldots(n-(k-1))$, and performing divisions as early as possible to avoid generating excessively large intermediate results, one can derive the `binomialLoop` tail-recursive function:

```
binomial :: N→N→N
binomial n k | n<k = 0
binomial n k = binomialLoop (min k (n-k)) 0 1 where
  binomialLoop k i p | i ≥ k = p
  binomialLoop k i p =
    binomialLoop k (i+1) ((n - i) * p `div` (i+1))
```

Note that, as a simple optimization, when $n - k \leq k$, the faster computation of $\binom{n}{n-k}$ is used to reduce the number of steps in `binomial`.

The embedding function `binomial(n,k)` computes $\binom{n}{k}$ and returns the result.

2.2 The bijection between sets and sequences of natural numbers

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

$$K_n(x_1, \ldots, x_n) = \sum_{k=1}^{n} \binom{k - 1 + s_k}{k} \tag{3}$$

where $s_k = \sum_{i=1}^{k} x_i$, we recognize the *prefix sums* s_k incremented with values of k starting at 0.

At this point, as our key "eureka step", we recognize here the "set side" of the bijection between sequences of n natural numbers and sets of n natural numbers described in [5], where a general framework for bijective data transformations provides such conversion algorithms between a large number of fundamental data types.

We can compute the bijection `list2set` together with its inverse `set2list` as

```
list2set xs = tail (scanl f (-1) xs) where f x y = x+y+1

set2list ms = zipWith g ms (-1:ms) where g x y = x-y-1
```

The following example illustrates how it works:

```
*Goedel> list2set [2,0,1,3]
[2,3,5,9]
*Goedel> set2list it
[2,0,1,3]
```

As a side note, this bijection is mentioned in [6] and implicitly in [3], with indications that it might even go back to the early days of the theory of recursive functions.

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

The function `fromCantorTuple` implements the $\mathbb{N}^k \to \mathbb{N}$ bijection by combining the sequence-to-set transformer `list2set` and the function `fromKSet` that sums up the binomials.

```
fromCantorTuple ns = fromKSet (list2set ns)

fromKSet xs = sum (zipWith binomial xs [1..])
```

This shifts the problem of computing its inverse from lists to sets, an apparently minor use of a bijective data type transformation, that will turn out to be the single most critical step toward our solution.

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

We have now split our problem in two simpler ones: inverting the untupling of sets and then applying `set2list` to get back from sets to lists.

The next "eureka step" at this point is to observe that untupling of sets corresponds to 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*, [7], originally due to Lehmer [8].

Conversion algorithms between the conventional and the combinatorial number system are well known, [9, 6]. For instance, theorem **L** in [6] describes the precise position of a given sequence in the lexicographic order enumeration of all sequences of length k.

Theorem 1 (Knuth) *The combination* $[c_k, \ldots c_2, c_1]$ *is visited after exactly* $\binom{c_k}{k} + \ldots + \binom{c_2}{2} + \binom{c_1}{1}$ *other combinations have been visited.*

The function `binomialDigits` implements a greedy search algorithm, by subtracting the combination containing the most significant "digit" at each step from the variable `n`. At a given step, it carries on the result in the tail-recursive loop together with the decreased value of `k`, used in the binomial.

```
binomialDigits :: N→N→[N]
binomialDigits 0 _ = []
binomialDigits k n | k > 0 =
    (m-1) : binomialDigits (k-1) (n-bdigit) where
      m = upperBinomial k n
      bdigit = binomial (m-1) k
```

It relies on the function `upperBinomial` which finds the largest "binomial digit" to be subtracted from n, one at a time. Note that it its efficiency (critical for the performance of our algorithms) comes from first finding an upper limit on this "digit" provided by `roughLimit` and then doing binary search within the resulting narrow interval.

```
upperBinomial :: N→N→N
upperBinomial k n = binarySearch l m where
  m = roughLimit k (n+k) k
  l = m `div` 2

  binarySearch from to | from==to = from
  binarySearch from to =
    if binomial mid k > n
      then binarySearch from mid
      else binarySearch (mid+1) to where
        mid = (from + to) `div` 2

  roughLimit k n i | binomial i k > n = i
  roughLimit k n i = roughLimit k n (2*i)
```

The efficient inverse of Cantor's N-tupling is now the composition of the functions `set2list` and `toKSet`.

```
toCantorTuple k n = set2list (toKSet k n)

toKSet k n = reverse (binomialDigits k n)
```

The function `toKSet` computes the inverse of `fromKSet`. Note that we reverse the intermediate result `xs` to ensure that `set2list` receives it in increasing order - our canonical representation for sets. The following example illustrates that the algorithms work as expected:

```
*Goedel> fromCantorTuple [7,0,0]
119
*Goedel> toCantorTuple 3 119
[7,0,0]
*Goedel> map (toCantorTuple 3) [0..6]
[[0,0,0],[0,0,1],[0,1,0],[1,0,0],[0,0,2],[0,1,1],[1,0,1]]
*Goedel> map fromCantorTuple it
[0,1,2,3,4,5,6]
*Goedel>
```

3 A bijection from natural numbers to finite lists of natural numbers

We will use the generalized Cantor bijection to devise a bijection from of a tuple and the tuple itself into a natural number. Note that one could simply use `toCantorTuple 2` to pair the length and the code of the tuple. However, as typically the length is significantly smaller than the code we will *dilute* its share in the resulting code, by an arbitrary but fixed factor, `diluter`.

```
diluter = 10
```

The function `nats2nat` first converts the tuple `ns` to a code `m`. Then, `m` is "diluted" to a tuple `ms` before being re-encoded together with the length `l` in the final code `n`.

```
nats2nat :: [N]→N
nats2nat [] = 0
nats2nat ns | l ≥0 = n where
  l = pred (genericLength ns)
  m = fromCantorTuple ns
  ms = toCantorTuple diluter m
  n = fromCantorTuple (l:ms)
```

The function `nat2nats` reverses each of the encoding operations, relying on the same fixed value of the `diluter`.

```
nat2nats :: N→[N]
nat2nats 0 = []
nat2nats m | m>0 = ns where
  (l:ms) = toCantorTuple (diluter + 1) m
  n = fromCantorTuple ms
  ns = toCantorTuple (l+1) n
```

One can observe that the bijection is size-proportionate both ways, i.e. the bit-sizes of its two sides are within a constant factor.

```
*Goedel> nats2nat [2,0,1,3,2,0,1,4,9,777,888,0,999]
78579351238994638235868454056982992335
*Goedel> nat2nats it
[2,0,1,3,2,0,1,4,9,777,888,0,999]
```

4 The bijection between a term algebra and $\mathbb{N}$

4.1 Term algebras

Term algebras are *free magmas* 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.

We will represent a function's arguments as a list and assume its arity is *implicitly* given as the length of the list.

```
data Term var const = Var var | Fun const [Term var const]
   deriving (Eq,Show,Read)
```

A Variation of the encoding assuming finite sets of variables, constants and symbols with fixed arities will be considered in section 5. Our definition here, as specified by the polymorphic `data Term`, corresponds more closely to programming language constructs like Herbrand terms or λ-terms over an "open", possibly infinite set of function symbols and variables.

4.2 Encoding in a term algebra with function symbols represented as natural numbers

First, we will separate encodings of variable and function symbols. We can map them, respectively, to even and odd numbers. To deal with function arguments, we will use the bijective encoding of sequences recursively.

```
nterm2code :: Term N N → N

nterm2code (Var i) = 2*i
nterm2code (Fun cName args) = code where
  cs = map nterm2code args
  fc = nats2nat (cName:cs)
  code = 2*fc-1
```

The inverse is computed by recursing over the sequence associated with a natural number by the `nat2nats` combinator:

```
code2nterm :: N → Term N N

code2nterm n | even n = Var (n `div` 2)
code2nterm n = Fun cName args where
  k = (n+1) `div` 2
  cName:cs = nat2nats k
  args = map code2nterm cs
```

Note that the size of the outputs is proportional to the size of the inputs.

```
*Goedel> nterm2code (Fun 0 [Fun 1 [],Fun 2 [],Var 0,Fun 2 [Var 0,Var 1]])
3469780176190289
*Goedel> code2nterm it
Fun 0 [Fun 1 [],Fun 2 [],Var 0,Fun 2 [Var 0,Var 1]]
```

We will next extend this encoding for the case of more realistic term algebras where function symbols are encoded as strings. 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.

4.3 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. An encoder for *numbers in bijective base-k* [10] that provides such a bijection is implemented as the function `toBijBase` which, like the decoder `fromBijBase`, works one digit a time, using the functions `getBijDigit` and `putBijDigit`. They are both parameterized by the base `b` of the numeration system.

```
toBijBase :: N→N→[N]
toBijBase _ 0 = []
toBijBase b n | n>0 = d : ds where
  (d,m) = getBijDigit b n
  ds = toBijBase b m

fromBijBase :: N→[N]→N
fromBijBase _ [] = 0
fromBijBase b (d:ds) = n where
  m = fromBijBase b ds
  n = putBijDigit b d m
```

The function `getBijDigit` extracts one bijective base-b digit from natural number `n`. It also returns the "left-over" information content as the second component of an ordered pair.

```
getBijDigit :: N→N→(N,N)
getBijDigit b n | n>0 = if d == 0 then (b-1,q-1) else (d-1,q) where
     (q,d) = quotRem n b
```

The function `putBijDigit` integrates a bijective base-`b` digit `d` into a natural number `m`:

```
putBijDigit :: N→N→N→N
putBijDigit b d m | 0 ≤ d && d < b = 1+d+b*m
```

The encoding/decoding to bijective base b works as follows:

```
*Goedel> toBijBase 4 2013
[0,2,0,2,2,0]
```

```
*Goedel> fromBijBase 4 it
2013
*Goedel> map (toBijBase 2) [0..7]
[[],[0],[1],[0,0],[1,0],[0,1],[1,1],[0,0,0]]
```

This encoding will turn out to be useful in uniquely encoding symbols and strings of a finite alphabet.

4.4 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 between `c0` and `c1`.

```
c0='a'
c1='z'
```

The base of the numeration system is then

```
base = 1+ord c1-ord c0
```

Next, we define the bijective base-k encodings

```
string2nat :: String → N
string2nat cs = fromBijBase (fromIntegral base)
  (map (fromIntegral . chr2ord) cs)

nat2string :: N → String
nat2string n | n ≥ 0 = map (ord2chr . fromIntegral)
  (toBijBase (fromIntegral base) n)
```

Note that `chr2ord` and `ord2chr` are providing the bijection between our alphabet and a an initial segment of $\mathbb{N}$.

```
chr2ord c | c≥c0 && c≤c1 = ord c - ord c0
ord2chr o | o≥0 && o<base = chr (ord c0+o)
```

We obtain an encoder/decoder to $\mathbb{N}$ working as follows:

```
*Goedel> string2nat "humptydumptysatonawall"
64581665088508438276502585654 70

*Goedel> nat2string it
"humptydumptysatonawall"
```

4.5 Encoding in a term algebra with function symbols represented as strings

The only change from the `Term N N` encoder is applying encoding/decoding to strings.

```
sterm2code :: Term N String → N

sterm2code (Var i) = 2*i
sterm2code (Fun name args) = 2*fc-1 where
  cName = string2nat name
  cs = map sterm2code args
  fc = nats2nat (cName:cs)
```

The inverse is computed as follows:

```
code2sterm :: N → Term N String

code2sterm n | even n = Var (n `div` 2)
code2sterm n = Fun name args where
  k = (n+1) `div` 2
  cName:cs = nat2nats k
  name = nat2string cName
  args = map code2sterm cs
```

The following example illustrate how it works.

```
*Goedel> nterm2code (Fun 0 [Fun 1 [],Fun 2 [],Var 0,Fun 2 [Var 0,Var 1]])
3469780176190289

*Goedel> code2nterm it
Fun 0 [Fun 1 [],Fun 2 [],Var 0,Fun 2 [Var 0,Var 1]]

*Goedel> sterm2code (Fun "b" [Fun "a" [],Var 0])
1277

*Goedel> code2sterm it
Fun "b" [Fun "a" [],Var 0]

*Goedel> sterm2code (Fun "forall" [Var 0, Fun "f" [Var 0]])
10798625256283010558209603 3

*Goedel> code2sterm it
Fun "forall" [Var 0,Fun "f" [Var 0]]

*Goedel> map (as sterm nat) [0..7]
[Var 0,Fun "a" [],Var 1,Fun "b" [],Var 2,Fun "c" [],Var 3,Fun "d" []]
```

5 Fixed signature term algebras

Term algebras with a finite signature and a finite number of variables model things like circuits made of a fixed library of gates and a fixed number of input wires. These algorithms are likely to be useful for random generation of boolean formulas, of use in circuit synthesis or genetic programming.

For flexibility, it is convenient to single out constants as they might have different types that the function symbols. The data type `FTerm` describes our variables, constants and function terms.

```
data FTerm a b c = V a | C b | F c [FTerm a b c] deriving (Eq,Read,Show)
```

The function `nat2term` is parameterized by a "dictionary" consisting of finite lists of variables, constants and function symbol-arity pairs `vars, consts` and `funs`. Its work is done by the local function `encode` that uses position indices in the lists `vars, consts` and `funs` to assign small natural numbers to the respective objects and recursively encodes arguments of compound terms. The function `fromCantorTuple` is used to to "fuse" together the encodings of the arguments that are propagated up in the recursion together with the bijective base-k digit corresponding to the encoding of the function symbols, provided by the function `putBijDigit`.

```
term2nat vars consts funs t | lv+lc>0 && lf>0 = encode t where
  lv = genericLength vars
  lc = genericLength consts
  lf = genericLength funs

  encode (V x) = n where n = indexOf x vars
  encode (C c) = lv+n where n = indexOf c consts
  encode (F f xs) = n where
    d = indexOf (f,k) funs
    k = genericLength xs
    ns = map encode xs
    m = fromCantorTuple ns
    n' = putBijDigit lf d m
    n = n'+lv+lc-1

  indexOf x (y:xs) | x==y = 0
  indexOf x (_:xs) = 1 + indexOf x xs
```

The function `nat2term` is parameterized by the same "dictionary" consisting of finite lists of variables, constants and functions symbol / arity pairs `vars, consts` and `funs`. Its work is done by the local function `decode` that reconstructs variable and constant symbols from their small natural number indices and recursively rebuilds a compound term by first extracting with `getBijDigit` the index of its function symbol-arity pair `(f,k)` and then splitting the remaining information content into `k` arguments using the `(toCantorTuple k)` : $\mathbb{N} \to \mathbb{N}^k$ bijection.

```
nat2term vars consts funs n | lv+lc>0 && lf>0 = decode n where
  lv = genericLength vars
  lc = genericLength consts
  lf = genericLength funs

  decode n | 0≤n && n < lv = V (vars `genericIndex` n)
  decode n | lv ≤n && n < lc+lv = C (consts `genericIndex` (n-lv))
  decode n | lc+lv≤n = F f (map decode ns) where
    n' = 1+n-(lc+lv)
    (d,m) = getBijDigit lf n'
    (f,k) = funs `genericIndex` d
```

```
    ns = toCantorTuple k m
    --r = F f (map decode ns)
```

The encoding/decoding algorithms are generic and can be parameterized by sets of variables, constants and function-arity pairs:

```
vars = ["x","y","z"]
consts = [0,1]
funs = [("~",1),("*",2),("+",2),("if",3)]
```

We can specialize our encoding/decoding functions to this signature as follows:

```
t2n :: FTerm String Integer String → N
t2n = term2nat vars consts funs

n2t :: N → FTerm String Integer String
n2t = nat2term vars consts funs
```

The following example illustrates the encoding and decoding of term describing simple boolean formulas.

```
*Goedel> n2t 123456
F "if" [V "x",C 0,F "~" [F "if" [V "x",V "x",V "y"]]]
*Goedel> t2n it
123456
```

The function `tshow` provides a more human readable output can be associated to a formula.

```
tshow t = toString t where
  s x = filter (/='"') (show x)
  toString (V x) = s x
  toString (C c) = s c
  toString (F f (x:y:[])) =
    "("++(toString x) ++ " " ++ s f ++ " " ++ (toString y)++")"
  toString (F f xs) = s f ++ "("++zs++")" where
    ys = map toString xs
    zs = intercalate "," ys
```

Note that it it works generically, on arbitrary signature. The following examples illustrate it:

```
Goedel> tshow (n2t 1234567890)
"(~(if((x * y),1,0)) * ((y * z) + (z * x)))"

*Goedel> map (tshow . n2t) [0..8]
["x", "y", "z", "0", "1", "~(x)", "(x * x)", "(x + x)", "if(x,x,x)"]
```

6 Related work

This paper makes use of the main ideas of the embedded data transformation framework introduced in [11] which helps gluing together all the pieces needed for

the derivation of our bijective encoding of term algebras, 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 [12, 13]. A typical use in the foundations of mathematics is [14].

Recent work [15, 16] in the context of functional programming on connecting heterogeneous data types through bijective mappings and natural number encodings deals with special cases of term algebras. In particular [15] provides a general description of an algorithm for encoding data type bijectively.

Numeration systems on regular languages have been studied recently, e.g. in [17] and specific instances of them are also known as bijective base-k numbers [10].

There are a large number of papers referring to the original "Cantor pairing function" among which we mention the surprising result that, together with the successor function it defines a decidable subset of arithmetic [14].

Combinatorial number systems can be traced back to [8] and one can find efficient conversion algorithms to conventional number systems in [6] and [9].

Finally, the "once you have seen it, obvious" `list2set / set2list` bijection is borrowed from [5], but not unlikely to be common knowledge of people working in combinatorics or recursion theory.

7 Conclusion

We have described bijective Gödel numbering schemes for term algebras with infinite and fixed signatures and variable sets. The algorithms work in low polynomial time and have 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. Moreover, our fixed signature algorithm is usable for generating random circuits or enumerate small circuits implementing a given boolean functions using a fixed library of primitive functions.

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., Berlin Heidelberg, Springer (1974) 301–316
3. Cegielski, P., Richard, D.: On arithmetical first-order theories allowing encoding and decoding of lists. Theoretical Computer Science **222**(12) (1999) 55 – 75
4. Lisi, M.: Some remarks on the Cantor pairing function. Le Matematiche **62**(1) (2007)
5. 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
6. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions. Addison-Wesley Professional (2005)
7. Wikipedia: Combinatorial number system — wikipedia, the free encyclopedia (2011) [Online; accessed 21-March-2012].
8. Lehmer, D.H.: The machine tools of combinatorics. In: Applied combinatorial mathematics. Wiley, New York (1964) 5–30
9. Buckles, B.P., Lybanon, M.: Generation of a Vector form the Lexicagraphical Index [G6]. ACM Transactions on Mathematical Software **5**(2) (June 1977) 180–182
10. Wikipedia: Bijective numeration — wikipedia, the free encyclopedia (2012) [Online; accessed 2-June-2012].
11. Tarau, P.: A Groupoid of Isomorphic Data Transformations. In Carette, J., Dixon, L., Coen, C.S., Watt, S.M., eds.: Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009 , Grand Bend, Canada, Springer, LNAI 5625 (July 2009) 170–185
12. 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., Berlin Heidelberg, Springer (2003) 572–581
13. 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)
14. 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
15. Vytiniotis, D., Kennedy, A.: Functional Pearl: Every Bit Counts. ICFP 2010 : The 15th ACM SIGPLAN International Conference on Functional Programming (September 2010) ACM Press.
16. Kobayashi, N., Matsuda, K., Shinohara, A.: Functional Programs as Compressed Data. ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (January 2012) ACM Press.
17. Rigo, M.: Numeration systems on a regular language: arithmetic operations, recognizability and formal power series. Theoretical Computer Science **269**(12) (2001) 469 – 498