Compact Gödel Numberings for Term Algebras, SAT-Problems and Self-delimiting Codes

Paul Tarau

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

Abstract

We introduce a Gödel numbering algorithm that encodes/decodes elements of a term algebra as unique natural numbers. In contrast with Gödel's original encoding and various alternatives in the literature, our encoding has the following properties: a) is bijective b) natural numbers always decode to syntactically valid terms c) it works in linear time d) the bitsize of our encoding is within constant factor of the syntactic representation of the input. Our algorithm can be applied to derive compact serialized representations for various formal systems and programming language constructs. The compact sequence encoder needed for the term algebra is then extended to signed integers to obtain a bijective encoding of SAT-problems. Finally we derive self-delimiting codes by lifting sequence isomorphisms to their hereditarily finite counterpart and then mapping them to a bitstring representation in a balanced parenthesis language and compare their information theoretic efficiency with Elias omega code.

The paper is organized as a literate Haskell program available from `http://logic.cse.unt.edu/tarau/research/2009/goedel.zip`.

1. Introduction

A *ranking/unranking* function defined on a data type is a bijection to/from the set of natural numbers (denoted $\mathbb{N}$ through the paper). When applied to formulae 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 [Gödel 1931, Hartmanis and Baker 1974]. In Gödel's original encoding [Gödel 1931], 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 formulae of non-trivial size. As this mapping is not a surjection, there are codes that decode to syntactically invalid formulae. This key difference, also applies to alternative Gödel numbering schemes (like Gödel's beta-function), while ranking/unranking function, as used in combinatorics, are bijective mappings.

Besides codes associated to formulae a wide diversity of common computer operations, ranging from data compression and serialization to wireless 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 linear time
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.

After a straightforward extension of sequence encodings to signed integers, we also describe an encoding of SAT-problems, useful, for instance, in generating random tests for solvers.

Self-delimiting codes are needed each time when structured data types are sent over a channel or decomposed in subcomponents for processing. Asymptotically optimal *universal codes* like the Elias omega code [Elias 1975] rely on encoding *length information* recursively i.e. delimiting is achieved by first encoding the length, then the of the length etc. Using the same data transformation framework, we introduce a new self-delimiting code. Starting with an isomorphism between natural numbers and finite sequences we derive a self-delimiting code by lifting it to its hereditarily finite counterpart and then mapping it to a bitstring representation in a balanced parenthesis language. While typically less compact than Elias omega code, the resulting code has the unique "fractal-like" property that all its subcomponents are also self-delimiting. To clarify some of the sparseness and density properties of the encoding, we compare it with Elias omega code and undelimited bitstring representations.

Through the paper, we will make use of the embedded data transformation language introduced in [Tarau 2009] and briefly overviewed in the Appendix. From a user's perspective it is typically mostly the combinator

```
as :: Encoder a → Encoder b → b → a
```

that applies a bijection defined between its first two arguments to its third argument. Encoders map various data types to *hub* object, chosen in this case to be the set of finite sequences of natural

numbers $[\mathbb{N}]$. By composing two Encoders one obtains any-to-any bijections.

The paper is organized as a literate Haskell program. Some of the tools used to produce Graphviz and Gnuplot images related to the paper as well as detailed explanation for some of the auxiliary functions given in Appendix are available as a large (104 pages) literate Haskell document at [Tarau 2009]. We group our code in a self-contained module `Goedel`, importing only two library modules:

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

2. Ranking/unranking functions for finite sequences

DEFINITION 1. *A* ranking/unranking *function defined on a data type is a bijection to/from* $\mathbb{N}$.

In the case of finite sequences of natural numbers such functions aggregate their elements in a single natural number. Clearly, this property is critical for encoding the arguments of function symbols. Gödel's original primes based encoding, mapping integers in a sequence to exponents of consecutive primes, as well as his *beta* function based on the Chinese Remainder Theorem not only create comparably large numbers but also lead to unpractical inverse functions.

2.1 Uncovering the implicit list structure of a natural number

We will first design an encoding, involving a computation linear in the bitsize of the input, uncovering a surprisingly simple "list structure" hiding inside a natural number (represented as the data type `N`, see Appendix). Given the definitions

```
cons :: N→N→N
cons x y  = (2^x)*(2*y+1)

hd :: N→N
hd n | n>0 = if odd n then 0 else 1+hd  (n `div` 2)

tl :: N→N
tl n = n `div` 2^((hd n)+1)
```

one can connect natural numbers to sequences as follows:

```
as_nats_nat :: N→[N]
as_nats_nat 0 = []
as_nats_nat n = hd n : as_nats_nat (tl n)

as_nat_nats :: [N]→N
as_nat_nats [] = 0
as_nat_nats (x:xs) = cons x (as_nat_nats xs)
```

It is shown in [Tarau 2009] (see also subsection 3.6) that the following holds:

PROPOSITION 1. `as_nat_nats` *is a bijection from finite sequences of natural numbers to natural numbers and* `as_nats_nat` *is its inverse.*

A generic ranking/unranking mechanism can be defined as the `Encoder`:

```
nat1 :: Encoder N
nat1 = Iso as_nats_nat as_nat_nats
```

While quite simple, this encoder has the disadvantage that bitsize of the encoding can be exponential in the bitsize of the elements of a sequence:

```
*Goedel> as nat1 nats [50,20,50]
5316911983139665852799595575850827776
```

We shall therefore build a more intricate encoding, while making sure that the computations it involves are tractable.

2.2 Pairing functions as Encoders

An important type of isomorphism, originating in Cantor's work on infinite sets connects natural numbers and pairs of natural numbers.

DEFINITION 2. *An isomorphism* $f : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ *is called a* pairing function *and its inverse* f^{-1} *is called an* unpairing *function.*

Given the definitions:

```
unpair z = (hd (z+1), tl (z+1))
pair (x,y) = (cons x y)-1
```

shifting by 1 turns `hd` and `tl` in total functions on $\mathbb{N}$ such that $unpair\ 0 = (0, 0)$ i.e. the following holds:

PROPOSITION 2.
$unpair : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ *is a bijection and* $pair = unpair^{-1}$.

Note that unlike `hd` and `tl`, `unpair` is defined for all natural numbers:

```
*Goedel> map unpair [0..7]
[(0,0),(1,0),(0,1),(2,0),(0,2),(1,1),(0,3),(3,0)]
```

As the cognoscenti might notice, this is in fact a classic *pairing/unpairing function* that has been used, by Pepis, Kalmar and Robinson in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem in [Pepis 1938, Kalmar 1939, Robinson 1950].

2.3 Tuple Encodings

Tupling/untupling functions are a natural generalization of pairing/unpairing operations. The function `to_tuple`: $N \to N^k$ 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 = map (from_base 2) (
    transpose (
      map (to_maxbits k) (
        to_base (2^k) n
      )
    )
  )
```

To convert a k-tuple back to a natural number we merge their bits, k at a time. This operation uses the transposition of a bit matrix obtained from the tuple, seen as a number in base 2^k, with help from bit crunching functions given in APPENDIX:

```
from_tuple ns = from_base (2^k) (
    map (from_base 2) (
      transpose (
        map (to_maxbits l) ns
      )
    )
  ) where
      k=genericLength ns
      l=max_bitcount ns
```

The following example shows the decoding of 42, its decomposition in bits (right to left), the formation of a 3-tuple and the encoding of the tuple back to 42.

```
*Goedel> to_base 2 42
[0,1,0,1,0,1]
```

```
*Goedel> to_tuple 3 42
[2,1,2]
*Goedel> to_base 2 2
[0,1]
*Goedel> to_base 2 1
[1]
*Goedel> from_tuple [2,1,2]
42
```

Fig. 1 shows multiple steps of the same decomposition, with shared nodes collected in a DAG.

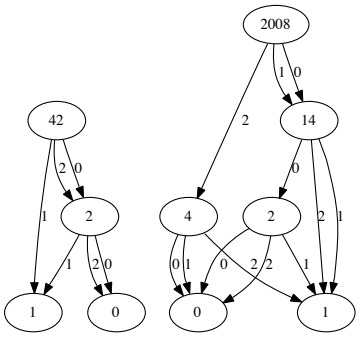

Figure 1. Repeated 3-tuple expansions: *42* and *2008*

Using the functions `pair` and `unpair` we define the Encoder

```
type N2 = (N,N)

n2 :: Encoder N2
n2 = compose (Iso pair unpair) nat
```

to obtain a pairing/unpairing isomorphism `n2` between and $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

We can now also define alternative `hd, tl, cons` and pairing/unpairing operations:

```
hd' = head . nat2ftuple
tl' = ftuple2nat . tail . nat2ftuple
cons' h t  = ftuple2nat (h:(nat2ftuple t))

pair' (x,y) = (cons' x y)-1
unpair' z = (hd' z', tl' z') where z'=z+1
```

As an interesting example for transporting structures between data types, one could also define `tl'` using the combinator `borrow_from` given in Appendix) as:

```
tl'' n = borrow_from nats tail nat n
```

Fig. 2 shows the values of `z=pair' (x,y)` for $x, y \in [0..2^6 - 1]$.

An unusual thing about this pairing function is that it reaches very large values strictly when its both arguments are powers of 2, for instance:

```
*Goedel> pair' (4,4)
4103
*Goedel> pair' (3,4)
279
*Goedel> pair' (4,3)
73
```

We can also specialize our tuple encoding/decoding bijections for $n = 2$ to obtain the pairing/unpairing Encoder also described in function [Pigeon 2001] and related to the Moser-De Bruijn sequence (A000695, [Sloane 2006]):

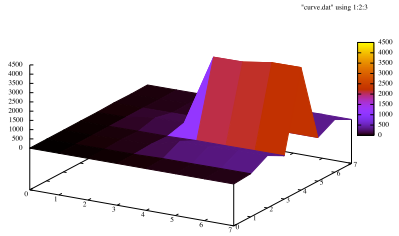

Figure 2. Values of z=pair' (x,y)

```
nat2 :: Encoder (N,N)
nat2 = compose (Iso bitpair bitunpair) nat

bitpair (x,y) = from_tuple [x,y]
bitunpair z = (x,y) where [x,y] = to_tuple 2 z

*Goedel> as nat2 nat 2009
(61,26)
*Goedel> as nat nat2 it
2009
```

Note that in contrast with `unpair`, `bitunpair` splits evenly the bits of its argument.

This can be generalized by allowing an arbitrary division in `k+l` groups with `k` aggregating into the first element and `l` aggregating as the second element of the pair as follows:

```
pairKL k l (x,y) = from_tuple (xs ++ ys) where
   xs = to_tuple k x
   ys = to_tuple l y

unpairKL k l z = (x,y) where
  zs=to_tuple (k+l) z
  xs=genericTake k zs
  ys=genericDrop k zs
  x=from_tuple xs
  y=from_tuple ys
```

Fig. 3 shows the values of `z=pairKL 3 2 n` for $n \in [0..2^6-1]$.

Fig. 4 shows the path obtained by connecting pairs `(x,y) = unpairKL 2 3` for $n \in [0..15]$.

2.4 Encoding Finite Functions as Tuples

As finite sets can be put in a bijection with an initial segment of N, a `finite function` can be seen as a function defined on an initial segment of N with values in N. We can encode and decode a finite

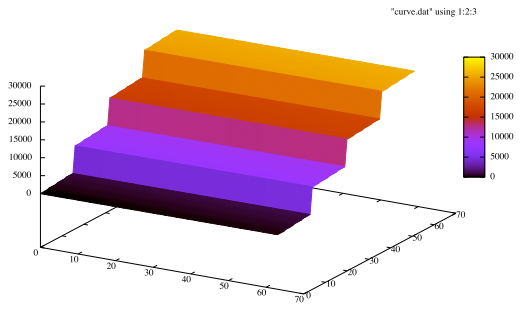

Figure 3. Values of pairKL

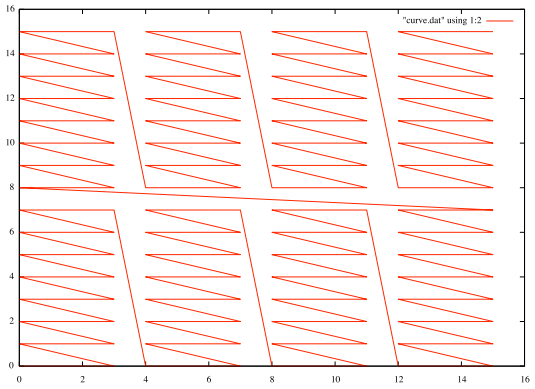

Figure 4. Path connecting unpairKL

function from $[0..k-1]$ to N (seen as the list of its values), as a natural number:

```
ftuple2nat [] = 0
ftuple2nat ns = succ (pair (pred k,t)) where
  k=genericLength ns
  t=from_tuple ns

nat2ftuple 0 = []
nat2ftuple kf = to_tuple (succ k) f where
  (k,f)=unpair (pred kf)
```

This suggests the following alternative encoder for finite functions:

```
nat :: Encoder N
nat = Iso nat2ftuple ftuple2nat


*Goedel> as nats nat 2008
[3,2,3,1]
*Goedel> as nat nats it
2008
```

The following holds:

PROPOSITION 3. *The encoder* `nat` *works in space and time proportional to the bitsize of its input.*

3. Designing an efficient bijective Gödel numbering scheme

With all the building blocks in place, we can now proceed with the design of a compact bijective Gödel numbering algorithm.

3.1 Term Algebras

Term algebras are *free magmas* induced by a 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 an arguments of function symbols. In various logic formalisms a term algebra is called a Herbrand Universe.

We will represent function arguments as lists and assume their arity is implicitly given as the length of the the lists:

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

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

Let's first instantiate the term algebra `Term var const` as:

```
type NTerm = Term N N
```

First, we will have to 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=as nat nats (cName:cs)
  code = 2*fc-1
```

The inverse is computed as follows:

```
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 = as nats nat k
  args = map code2nterm cs


*Goedel> as nterm nat 55
Fun 1 [Fun 0 [],Var 0]
*Goedel> as nat nterm it
55
```

We can encapsulate our transformers as the Encoder:

```
nterm :: Encoder NTerm
nterm = compose (Iso nterm2code code2nterm) nat
```

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

3.3 Encoding strings

Strings can be seen just as a notational equivalent of lists of natural numbers written in base n. For simplicity (an to avoid unprintable as a result of applying the inverse mapping) we will assume that we use only lower case characters to name functions.

We obtain an Encoder `string` immediately as:

```
string :: Encoder String
string = compose (Iso string2nat nat2string) nat

base = ord 'z'- ord 'a'
chr2ord c | c≥'a' && c≤'z' = ord c - ord 'a'

ord2chr o | o≥0 && o≤base = chr (ord 'a'+o)
```

```
string2nat cs = from_base
  (fromIntegral base)
  (map (fromIntegral . chr2ord) cs)

nat2string n = map
  (ord2chr . fromIntegral)
  (to_base (fromIntegral base) n)


*Goedel> as nat string "hello"
5647607
*Goedel> as string nat it
"hello"
```

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

We can now instantiate our term algebra to have function symbols range over strings.

```
type STerm = Term N String
```

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

```
sterm2code :: Term N String → N

sterm2code (Var i) = 2*i
sterm2code (Fun name args) = code where
  cName=as nat string name
  cs=map sterm2code args
  fc=as nat nats (cName:cs)
  code=2*fc-1
```

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 = as nats nat k
  name = as string nat cName
  args = map code2sterm cs
```

We can encapsulate our transformers as the Encoder:

```
sterm :: Encoder STerm
sterm = compose (Iso sterm2code code2sterm) nat


*Goedel> as sterm nat 55
Fun "b" [Fun "a" [],Var 0]
*Goedel> as nat sterm it
55
*Goedel> as nat sterm
   (Fun "forall" [Var 0, Fun "f" [Var 0]])
54497608016617424332069 83
*Goedel> as sterm nat it
Fun "forall" [Var 0,Fun "f" [Var 0]]
```

3.5 Mapping terms to arbitrary bitstrings

Term algebras are free magmas generated through fairly complex substitution operations. Their underlying data representation involve ordered trees. Can we design a bijective mapping to, arguably, the simplest possible free magma - the set of strings on $\{0,1\}$? The answer is affirmative, provided that we obtain a mapping from *arbitrary* bitstrings to natural numbers.

We will first "rebuild" $\mathbb{N}$ itself through a more computer oriented view: as arbitrary bitstrings i.e. as elements of the regular language $\{0,1\}^*$. First we need a decoder/encoder:

```
bits :: Encoder [N]
bits = compose (Iso as_nat_bits as_bits_nat) nat

as_bits_nat = drop_last . (to_base 2) . succ where
  drop_last = reverse . tail . reverse

as_nat_bits bs = pred (from_base 2 (bs ++ [1]))
```

It works as follows:

```
*Goedel> as bits nat 42
[1,1,0,1,0]
*Goedel> as nat bits [1,1,0,1,0]
42
```

Note that the bit order is from smaller to larger exponents of 2 and that final 1 digits (used as delimiters in conventional computer representations of binary numbers) have been removed. This ensures that every combination of 0 and 1 in $\{0,1\}^*$ represents a number. Note also that this encoding is isomorphic to the so called *bijective base 2* representation. Using the `as` combinator we obtain:

```
nterm2bits = as bits nterm
bits2nterm = as nterm bits

sterm2bits = as bits sterm
bits2sterm = as sterm bits


*Goedel> as nterm bits
  [0,0,0,1,0,1,0,0,1,1,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,
   0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Fun 0 [Fun 1 [Var 0,Fun 1 [Var 0]],Var 1]
*Goedel> as bits nterm
    (Fun 0 [Fun 1 [Var 0,Fun 1 [Var 0]],Var 1])
[0,0,0,1,0,1,0,0,1,1,0,1,0,0,0,0,0,1,0,0,0,0,0,1,
 0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

The following is a consequence of the fact that each of the encoding steps are linear in the bitsize of their input and run in linear time and preserve syntactic validity by structural induction.

PROPOSITION 4. *The Gödel numbering algorithms implemented by the encoders* `nterm` *and* `sterm` *are bijective and work in linear time and space linear in the bitsize of their input. Moreover, all natural numbers decode to syntactically valid terms.*

3.6 Deriving representations of finite sequences from one-solution Diophantic equations

Let's first observe that

PROPOSITION 5. $\forall z \in \mathbb{N} - \{0\}$ *the diophantic equation*

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

has exactly one solution $x, y \in \mathbb{N}$.

This follows immediately from the unicity of the decomposition of a natural number as a multiset of prime factors.

This equation justifies the fact that the `cons, hd, tl` functions described in subsection 2.1 can be used to uncover a list structure inside natural numbers. The mechanism can be generalized to obtain a family of finite sequence encoders by choosing an arbitrary base `b` instead of 2. Care should be taken in this case to *rebase* i.e. switch from/to base `b-1` to ensure that we obtain a bijection. Such rebasing operations are similar to those used in generating the sequences used in Goodstein's theorem [Goodstein 1944, Tarau 2009]. First we implement the head and tail operations `xhd` and `xtl` as follows:

```
xhd :: N→N→N
xhd b n | b>1 && n>0 =
  if n `mod` b > 0 then 0 else 1+xhd b (n `div` b)
```

```
xtl :: N→N→N
xtl b n = y where
  y'= n `div` b^(xhd b n)
  q = y' `div` b
  m = y' `mod` b -- rebase to b
  y = (b-1)*q+m-1
```

The `xcons` operation aggregates back the head and tail:

```
xcons :: N→N→N→N
xcons b x y  | b>1 = (b^x)*y' where
   q =  y `div` (b-1)
   m = y `mod` (b-1) -- rebase to b-1
   y' = b*q + (m+1)
```

The following examples show the operations for base 3:

```
*Goedel> xcons 3 10 20
1830519
*Goedel> xhd 3 1830519
10
*Goedel> xtl 3 1830519
20
```

We can extend the mechanism to derive *pairing/unpairing* functions as follows:

```
xunpair b n = (xhd b n',xtl b n') where n'=n+1

xpair b (x,y) = (xcons b x y)-1
```

One can see that we obtain a family of bijections $f_b : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ for every base b.

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

Such rebasing operations are similar to those used in generating the sequences used in Goodstein's theorem [Goodstein 1944, Tarau 2009].

4. Emulating primality with multiset encodings

We will now explore some prime number based encodings, for comparison purposes as well as to glean useful generalizations from the similarities between Gödel's original encoding mechanism and the one proposed in the previous section.

4.1 Encoding finite multisets with primes

A factorization of a natural number is uniquely described as a multiset of primes. This suggest exploring the analogy between the prime-based and alternative multiset encodings of natural numbers.

We will use the fact that each prime number is uniquely associated to its position in the infinite stream of primes, to obtain a bijection from multisets of natural numbers to natural numbers. Note that this mapping is the same as the *prime counting function* traditionally denoted $\pi(n)$, which associates to n the number of primes smaller or equal to n, restricted to primes. We assume defined a prime generator `primes` and a factoring function `to_factors` (see Appendix).

The function `nat2pmset` maps a natural number to the multiset of prime positions in its factoring. Note that we map `0` to `[]` and shift `n` to `n+1` to accomodate `1`, to which prime factoring operations do not apply.

```
nat2pmset 0 = []
nat2pmset n =
  map (to_pos_in (h:ts)) (to_factors (n+1) h ts) where
      (h:ts)=genericTake (n+1) primes
```

The function `pmset2nat` maps back a multiset of positions of primes to the result of the product of the corresponding primes. Again, we map `[]` to `0` and shift back by `1` the result.

```
pmset2nat [] = 0
pmset2nat ns = (product ks)-1 where
  ks=map (from_pos_in ps) ns
  ps=primes
  from_pos_in xs n = xs !! (fromIntegral n)
```

The operations `nat2pmset` and `pmset2nat` form an isomorphism that, using the combinator language defined in [Tarau 2009] (also briefly reviewed in the Appendix) provides any-to-any encodings between various data types. This gives the Encoder `pmset` for prime encoded multisets as follows:

```
pmset :: Encoder [N]
pmset = compose (Iso pmset2nat nat2pmset) nat


*Goedel> as pmset nat 2008
[3,3,12]
*Goedel> as nat pmset it
2008
```

Note that the mappings from a set or sequence to a number work in time and space linear in the bitsize of the number. On the other hand, as prime number enumeration and factoring are involved in the mapping from numbers to multisets, this encoding is intractable for all but small values.

4.2 Revisiting Gödel's original encoding of finite sequences

Assuming that function symbols, variables and punctuation in a formula language have been mapped to consecutive natural numbers, Gödel's original prime number based encoding can be implemented as follows:

```
nats2goedel ns =  product xs where
  xs=zipWith (^) primes ns
```

i.e. to to the $2^{n_0} * 3^{n_1} * \ldots p_i^{n_i} * \ldots$ for $n_i \in ns$. We can reverse the process:

```
goedel2nats n = combine ds xs where
  pss=group (to_primes n)
  ps=map head pss
  xs=map genericLength pss
  ds=as nats set (map pi' ps)

  combine [] [] = []
  combine (b:bs) (x:xs) =
    replicate (fromIntegral b) 0 ++ x:(combine bs xs)
```

The encoding/decoding so far works as follows:

```
*Goedel> goedel2nats 2009
[0,0,0,2,0,0,0,0,0,0,0,1]
*Goedel> nats2goedel [0,0,0,2,0,0,0,0,0,0,0,1]
2009
*Goedel> nats2goedel [0,0,0,2,0,0,0,0,0,0,0,1,0,0,0]
2009
```

Reversing it requires factoring and, as seen from the previous example, needs a small fix: to avoid 0's after the last element in a sequence being ignored, we have to add how many of them are found at the end of the sequence, as part of the code. To accomodate 0 and 1, we treat 0 as a special case and shift by 1 by applying `succ` before calling `goedel2nats`.

```
goedel :: Encoder [N]
goedel = compose (Iso nats2g g2nats) nat

nats2g [] = 0
nats2g ns = pred (nats2goedel (z:ns)) where
```

```
  z=countZeros (reverse ns)

g2nats 0 = []
g2nats n = ns ++ (replicate (fromIntegral z) 0) where
  (z:ns)=goedel2nats (succ n)

countZeros (0:ns) = 1+countZeros ns
countZeros _ = 0
```

With the fix, Gödel's original encoding now works as a bijection $\mathbb{N} \to [\mathbb{N}]$:

```
*Goedel> as goedel nat 2009
[1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]
*Goedel> as nat goedel it
2009
```

A prime number based encoding similar to Gödel's original encoding can be derived from the encoder `pmset`, more directly, as follows:

```
goedel' :: Encoder [N]
goedel' = compose (Iso nats2gnat gnat2nats) nat

nats2gnat = pmset2nat . as_mset_nats

gnat2nats = as_nats_mset . nat2pmset

*Goedel> as goedel' nat 2009
[0,1,1,16]
*Goedel> as nat goedel' it
2009
```

4.3 Exploring the analogy between multiset decompositions and factoring

As natural numbers can be uniquely represented as multisets of prime factors and, independently, they can also be represented as a multiset with the Encoder `mset` (described in the Appendix), the following question arises naturally:

Can in any way the "easy to reverse" encoding `mset` *emulate or predict properties of the the difficult to reverse factoring operation?*

The first step is to define an analog of the multiplication operation in terms of the computationally easy multiset encoding `mset`. Clearly, it makes sense to take inspiration from the fact that factoring of an ordinary product of two numbers can be computed by concatenating the multisets of prime factors of its operands.

```
mprod _ 0 _ = 0
mprod _ _ 0 = 0
mprod msetEncoder n m = mshift
  (borrow_from2 msetEncoder sconcat nat) n m

mshift f x y = 1+(f (x-1) (y-1))
sconcat xs ys = sort (xs ++ ys)
```

PROPOSITION 6. *If f is a multiset encoder then* $< N, mprodf, 0 >$ *is a commutative monoid i.e.* `mprod` *is defined for all pairs of natural numbers and it is associative, commutative and has 1 as an identity element.*

The proposition follows immediately from the associativity and commutativity of the sorted concatenation operation. We can derive an exponentiation operation as a repeated application of `mprod`:

```
mexp _ n 0 = 1
mexp msetEncoder n k =
  mprod msetEncoder n (mexp msetEncoder n (k-1))
```

Here are a few examples showing that `mprod` has properties similar to ordinary multiplication and exponentiation.

Associativity:

```
*Goedel> mprod mset 41 (mprod mset 33 88)
75557881740317128131585
*Goedel> mprod mset (mprod mset 41 33)  88
75557881740317128131585
```

Commutativity:

```
*Goedel> mprod mset 33 46
2199158521921
*Goedel> mprod mset 46 33
2199158521921
```

Zero and unit elements:

```
*Goedel> mprod mset 0 12
0
*Goedel> mprod mset 1 23
23
```

One can also note that the "square" function, while generally growing comparably to the ordinary one does not exhibit montonocity.

```
*Goedel> map (λx→mexp mset x 2) [0..15]
[0,1,3,9,7,33,19,25,23,129,67,73,71,97,83,89]
```

For `msetEncoder = pmset` we obtain, as expected, ordinary multiplication:

```
*Goedel> mprod pmset 7 5
35
*Goedel> mprod pmset 4 6
24
*Goedel> map (λx→mexp pmset x 2) [0..15]
[0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225]
```

Besides the connection with products, natural mappings worth investigating are the analogies between *multiset intersection* and `gcd` of the corresponding numbers or between *multiset union* and the `lcm` of the corresponding numbers. One can define:

```
mgcd msetEncoder x y = mshift
  (borrow_from2 msetEncoder msetInter nat) x y

mlcm msetEncoder x y = mshift
  (borrow_from2 msetEncoder msetUnion nat) x y

mDif msetEncoder x y = mshift
  (borrow_from2 msetEncoder msetDif nat) x y

mSymDif msetEncoder x y = mshift
  (borrow_from2 msetEncoder msetSymDif nat) x y

mdivides msetEncoder x y =
  x==(mgcd msetEncoder x y)
```

and note that properties similar to usual arithmetic operations hold, i.e for any multiset Encoder f

$$mprod\ f(mgcd\ f\ x\ y)(mlcm\ f\ x\ y) \equiv mprod\ f\ x\ y \qquad (2)$$

Also, divisibility is emulated correctly by `mdivides` i.e. for instance

```
*Goedel> mdivides pmset 4 12
True
*Goedel> mdivides pmset 5 12
False
```

We are now ready to "emulate" primality in our multiset monoid by defining `is_mprime msetEncoder` as a recognizer for *multiset primes* and `mprimes msetEncoder` as a generator of their infinite stream:

```
is_mprime msetEncoder p | p<2 = False
is_mprime msetEncoder p  =
```

```
  []==[n|n←[2..p-1],mdivides msetEncoder n p]

mprimes msetEncoder =
  filter (is_mprime msetEncoder) [2..]
```

Trying out `mprimes` gives:

```
*Goedel> take 15 (mprimes pmset)
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
*Goedel> take 15 (mprimes mset)
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]
```

One can see that conventional primes are indeed recovered when the encoder `pmset` is used. Also, surprisingly, every even number turns out to be "prime" when the encoder `mset` is used.

Another key difference between our "emulated" multiplicative arithmetics and the conventional one is that we do not have an obvious equivalent of addition. In its simplest form, this would mean defining a successor and predecessor function.

5. Encoding SAT problems

We will now look into a different problem, also with origins in representing logic formulae - finding a bijective encoding for Boolean Satisfiability (SAT) problems.

This has practical implications - for instance in generating instances needed as test data for a SAT solver or for learning algorithms [Felici et al. 2002], interfacing SAT solvers with constraint programming systems [Hawkins and Stuckey 2006] or for compressing problems used for large circuit verifications.

A SAT problem is specified as a set of clauses, where each clause is a lists of literals represented as integers. Signs of the literals are used to indicate positive or negated occurrences of propositional variables in range [1..n]. Therefore, the first step is to extend our techniques to deal with signed integers.

5.1 Encoding signed integers

To encode signed integers one can map positive numbers to even numbers and strictly negative numbers to odd numbers. This gives the Encoder:

```
z:: Encoder Z
z = compose (Iso z2nat nat2z) nat

nat2z n = if even n then n `div` 2 else (-n-1) `div` 2
z2nat n = if n<0 then -2*n-1 else 2*n
```

5.2 Extending pairing/unpairing to signed integers

Given the bijection from $\mathbb{N}$ to $\mathbb{Z}$ one can easily extend pairing/unpairing operations to signed integers. We obtain the Encoder:

```
type Z = Integer
type Z2 = (Z,Z)

z2 :: Encoder Z2
z2 = compose (Iso zpair zunpair) nat

zpair (x,y) = (nat2z . bitpair) (z2nat x,z2nat y)
zunpair z = (nat2z n,nat2z m) where
  (n,m)= (bitunpair . z2nat) z
```

working as follows:

```
*Goedel> map zunpair [-5..5]
[(-1,1),(-2,-1),(-2,0),(-1,-1),(-1,0),
 (0,0),(0,-1),(1,0),(1,-1),(0,1),(0,-2)]
*Goedel> map zpair it
[-5,-4,-3,-2,-1,0,1,2,3,4,5]

*Goedel> as z2 z (-2008)
(63,-26)
*Goedel> as z z2 it
-2008
```

Figure 5 shows the curve covering the lattice of integer coordinates generated by the function `zunpair`.

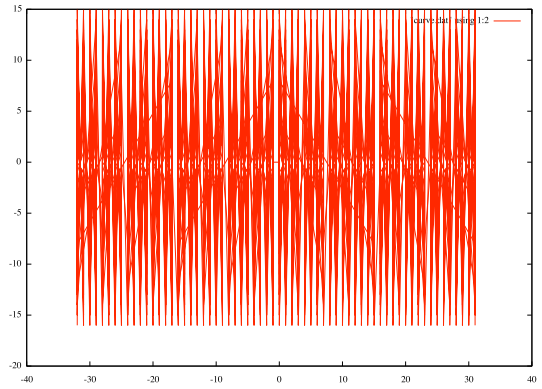

Figure 5. Curve generated by unpairing function on signed integers

5.3 Putting together an encoder for SAT problems

We are now ready to define an encoding for SAT problems given as sets of lists representing conjunctions of disjunctions of positive or negative propositional symbols. After defining:

```
set2sat = map (set2disj . (as set nat)) where
  shift0 z = if (z<0) then z else z+1
  set2disj = map (shift0. nat2z)

sat2set = map ((as nat set) . disj2set) where
  shiftback0 z = if(z<0) then z else z-1
  disj2set = map (z2nat . shiftback0)
```

we obtain the Encoder

```
sat :: Encoder [[Z]]
sat = compose (Iso sat2set set2sat) set
```

working as follows:

```
*Goedel> as sat nat 2008
[[-1],[-1,2],[1,2],[-1,2,-2]]
*Goedel> as nat sat it
2008
```

As an application, this encoding can be used to generate random SAT problems out of easier to generate random natural numbers, a technique usable to automate testing of SAT solvers.

6. Self-delimiting codes

A precise estimate of the actual size of various bitstring representations requires also counting the overhead for "delimiting" their components as this would model accurately the actual effort to transmit them over a channel or combine them in composite data structures.

6.1 Elias omega code

An asymptotically optimal mechanism for this is the use of a *universal self-delimiting code* for instance, the *Elias omega code* [Elias 1975]. To implement it, the encoder proceeds by recursively encoding the length of the string, the length of the length of the strings etc.

```
to_elias :: N → [N]
to_elias n = (to_eliasx (succ n))++[0]

to_eliasx 1 = []
to_eliasx n = xs where
  bs=to_lbits n
  l=(genericLength bs)-1
  xs = if l<2 then bs else (to_eliasx l)++bs
```

The decoder first rebuilds recursively the sequence of lengths and then the actual bitstring. It makes sense to design the decoder such that it extracts the number represented by the self-delimiting code from a sequence/stream of bits and also returns what is left after the extraction. Note also that the code uses the converters `from_base` and `to_base` given in the Appendix.

```
from_elias :: [N] → (N, [N])
from_elias bs = (pred n,cs) where (n,cs)=from_eliasx 1 bs

from_eliasx n (0:bs) = (n,bs)
from_eliasx n (1:bs) = r where
  hs=genericTake n bs
  ts=genericDrop n bs
  n'=from_lbits (1:hs)
  r=from_eliasx n' ts

to_lbits = reverse . (to_base 2)

from_lbits = (from_base 2) . reverse
```

We obtain the Encoder:

```
elias :: Encoder [N]
elias = compose (Iso (fst . from_elias) to_elias) nat
```

working as follows:

```
*Goedel> as elias nat 2008
[1,1,1,0,1,0,1,1,1,1,1,0,1,1,0,0,1,0]
*Goedel> as nat elias it
2008
```

Note also that self-delimiting codes are not *onto* the regular language $\{0,1\}^*$, therefore this Encoder cannot be used to map arbitrary bitstrings to numbers.

6.2 Unranking and ranking hereditarily finite data types

We recall that the *ranking problem* for a family of combinatorial objects is finding a unique natural number associated to each object, called its *rank*. The inverse *unranking problem* consists of generating a unique combinatorial object associated to each natural number.

We will now introduce a generic mechanism for *lifting* the isomorphism defined by a pair of ranking and unranking operations on simple data types like sequences, sets, multisets to their *hereditarily finite* counterparts through the use of recursion combinators similar to fold/unfold.

The data type representing such *hereditarily finite* structures will be a generic multi-way tree with a single leaf type `[]`.

```
data T = H [T] deriving (Eq,Ord,Read,Show)
```

The two sides of our lifting combinator are parameterized by two transformations `f` and `g` forming an isomorphism `Iso f g`:

```
unrank :: (a → [a]) → a → T
mapUnrank :: (a → [a]) → [a] → [T]

unrank f n = H (mapUnrank f (f n))
mapUnrank f ns = map (unrank f) ns

rank :: ([b] → b) → T → b
mapRank :: ([b] → b) → [T] → [b]

rank g (H ts) = g (mapRank g ts)
mapRank g ts = map (rank g) ts
```

Both combinators can be seen as a form of "structured recursion" that propagates a simpler operation guided by the structure of the data type. We can now combine the two sides of the resulting pair into an isomorphism `lift` defined with `rank` and `unrank` on the corresponding hereditarily finite data types:

```
lift :: Iso b [b] → Iso T b
lift (Iso f g) = Iso (rank g) (unrank f)

mapLift :: Iso b [b] → Iso [T] [b]
mapLift (Iso f g) = Iso (mapRank g) (mapUnrank f)
```

6.2.1 Hereditarily finite sets

Hereditarily finite sets [Takahashi 1976] will be represented as an Encoder for the tree type `T`:

```
hfs :: Encoder T
hfs=compose (lift (Iso (as set nat) (as nat set))) nat
```

Otherwise, lifting combinator induced isomorphisms work as usual with our embedded transformation language:

```
*Goedel> as hfs nat 42
H [H [],H [H [],H [H []],H [H [],H [H []]]]]
```

One can notice that we have just derived as a "free algorithm" Ackermann's encoding [Ackermann 1937] from hereditarily finite sets to natural numbers:

$$f(x) = \texttt{if } x = \{\} \texttt{ then } 0 \texttt{ else } \sum_{a \in x} 2^{f(a)}$$

together with its inverse

```
ackermann = as nat hfs
inverse_ackermann = as hfs nat
```

One can represent the action of a lifting combinator mapping a natural number into a hereditarily finite set as a directed graph with outgoing edges induced by by applying the `inverse_ackermann` function as shown in Fig. 6.

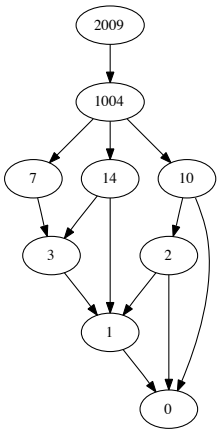

Figure 6. 2009 as a HFS

6.2.2 Hereditarily finite functions

The same tree data type can host a lifting combinator derived from finite functions instead of finite sets:

```
hff :: Encoder T
hff = compose (lift nat) nat
```

The `hff` Encoder can be seen as a "free algorithm", providing data compression/succinct representation for Hereditarily Finite Sets. Note, for instance, the significantly smaller tree size in:

```
*Goedel> as hff nat 2009
H [H [H [H [H [H []]]],H [H [],H []],H [H [H []]]]]
```

One can represent the action of a lifting combinator mapping a natural number into a hereditarily finite function as a directed ordered multi-graph as shown in Fig. 7. Note that as the mapping `as fun nat` generates a sequence where the order of the edges matters, this order is indicated with integers starting from 0 labeling the edges.

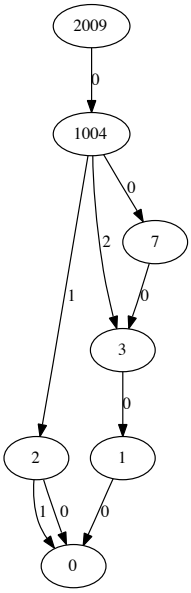

Figure 7. 2009 as a HFF

6.2.3 Hereditarily finite multisets

In a similar way, one can derive an Encoder for hereditarily finite multisets based on the `mset` isomorphism:

```
hfm :: Encoder T

hfm=compose (lift (Iso (as mset nat) (as nat mset))) nat
```

working as follows:

```
*Goedel> as hfm nat 2009
H [H [H [H [H [H []]]],H [H [H [],H [],
  H []]],H [H [H []],H [H []],H [H []]]]]
*Goedel> as nat hfm it
2009
```

One can represent the action of a lifting combinator mapping a natural number into a hereditarily finite multiset as a directed ordered multi-graph as shown in Fig. 8.

6.3 Parenthesis Language Encodings

An encoder for a parenthesis language is obtained by combining a parser and writer. As hereditarily finite functions naturally map one-to-one to parenthesis expressions expressed as bitstrings, we will choose them as target of the transformers.

```
hff_pars :: Encoder [N]
hff_pars = compose (Iso pars2hff hff2pars) hff
```

The parser recurses over a bitstring and builds a `HFF` as follows:

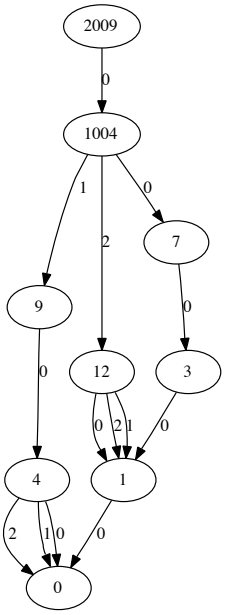

Figure 8. 2009 as a HFM

```
pars2hff cs = parse_pars 0 1 cs

parse_pars l r cs | newcs == [] = t where
  (t,newcs)=pars_expr l r cs

pars_expr l r (c:cs) | c==l = ((H ts),newcs) where
  (ts,newcs) = pars_list l r cs
  pars_list l r (c:cs) | c==r = ([],cs)
  pars_list l r (c:cs) = ((t:ts),cs2) where
    (t,cs1)=pars_expr l r (c:cs)
    (ts,cs2)=pars_list l r cs1
```

The writer recurses over a `HFF` and collects matching "parenthesis" (denoted 0 and 1) pairs:

```
hff2pars = collect_pars 0 1

collect_pars l r (H ns) =
  [l] ++ (concatMap (collect_pars l r) ns) ++ [r]
```

6.4 Parenthesis encoding of hereditarily finite types as a self-delimiting code

Like the Elias omega code, a balanced parenthesis representation is obviously self-delimiting as proven by the fact that the reader `pars_expr` defined in section 6.3 will extract a balanced parenthesis expression from a finite or infinite list while returning the part of the list left over. More precisely, the following holds:

PROPOSITION 7. *The* `hff_pars` *encoding is a self-delimiting code. If* n *is a natural number then* `hd n` *equals the code of the first parenthesized subexpression of the code of* n *and* `tl n` *equals the code of the expression obtained by removing it from the code for* n*, both of which represent self-delimiting codes.*

One can compute, for comparison purposes, with the optimal undelimited bitstring encoding `bits` and Elias code:

```
*Goedel> as bits nat 2009
[0,1,0,1,1,0,1,1,1,1]
*Goedel> as elias nat 2009
[1,1,1,0,1,0,1,1,1,1,1,0,1,1,0,1,0,0]
*Goedel> as hff_pars nat 2009
[0,0,0,0,0,0,1,1,1,1,0,0,1,0,1,1,0,0,0,1,1,1,1,1]
```

As the last example shows, the information density of a parenthesis representation is lower than the information theoretical optimal representation provided by the (undelimited) `bits` Encoder mapping natural numbers to the regular language $\{0,1\}^*$ as well as Elias code. Are there values when our encoding is better than Elias code? One can collect values that have smaller HFF codes than Elias omega codes with:

```
sparses_to m = [n|n←[0..m-1],
  (genericLength (as hff_pars nat n))
  <
  (genericLength (as elias nat n))]
```

working as follows

```
*Goedel> sparses_to (2^8)
[15,18,20,21,22,34,36,42,66,68,69,70,72,73,76,
 82,85,86,130,132,136,138,148,162,170]
```

A good way to evaluate "information density" for an arbitrary data type that is isomorphic to `N` through one of our encoders is to compute the total bitsize of its actual encoding over an interval like $[0..2^{n-1}]$. For instance,

```
hff_bitsize n= sum (map size [0..2^n-1]) where
   size k=genericLength (as hff_pars nat k)
elias_bitsize n= sum (map size [0..2^n-1]) where
   size k=genericLength (as elias nat k)
bitsize n= sum (map size [0..2^n-1]) where
   size k=genericLength (as bits nat k)
```

We obtain:

```
*Goedel> hff_bitsize 8
3768
*Goedel> elias_bitsize 8
3287
*Goedel> bitsize 8
1546
```

Knowing that the optimal (undelimited)) bit representation of all numbers in $[0..2^{n-1}]$ is given by `bitsize`, we can define a measure of information density for a bit-encoded parenthesis language seen as a representation for `HFF` as:

```
info_density_hff n = (bitsize n) / (hff_bitsize n)
```

One can see that information density progressively increases towards its upper bound 1:

```
*Goedel>  map info_density_hff [0..12]
[0.0,0.2222,0.26,...0.4581,0.4691]
```

Similarly, for Elias coding we obtain:

```
info_density_elias n = (bitsize n) / (elias_bitsize n)


*Goedel> map info_density_elias [0..12]
[0.0,0.25,0.3076...0.5099,...,0.557962824266358]
```

Also, a concept of "relative information density" for our self-delimiting encoding can be defined as:

```
relative_density_hff n =
  (elias_bitsize n) / (hff_bitsize n)
```

giving

```
*Goedel> map relative_density_hff [0..12]
[0.5,0.6666,...,0.9041,......0.8574,0.8407]
```

We can explore representation efficiency for "structured data" by comparing the size of a representation defined by a transformer `f` with the size of the self-delimiting Elias omega code. This would count for the cost of delimiting elements of a sequence as it would be needed, for instance, if the sequence is transmitted over a channel.

One can obtain an encoding `nat2sfun` for a finite sequence by encoding its length and then encoding each term. This is achieved by the generic function `nat2self` parametrized by a the transformer function $f : N \to [N]$:

```
nat2sfun n = nat2self (as nats nat) n

nat2self f n = (to_elias l) ++ concatMap to_elias ns where
  ns = f n
  l=genericLength ns
```

This function is injective (but not onto!) and its action can be reversed by first decoding the length l and then extracting self delimited sequences l times.

```
self2nat g ts = (g xs,ts') where
  (l,ns) = from_elias ts
  (xs,ts')=take_from_elias l ns

  take_from_elias 0 ns = ([],ns)
  take_from_elias k ns = ((x:xs),ns'') where
     (x,ns')=from_elias ns
     (xs,ns'')=take_from_elias (k-1) ns'

sfun2nat ns = xs where (xs,[])=self2nat (as nat nats) ns
```

After defining a bitsize measure for this encoding

```
sfun_bitsize n= sum (map size [0..2^n-1]) where
   size k=genericLength (nat2sfun k)
```

we obtain closer values:

```
*Goedel> sfun_bitsize 8
3579
*Goedel> hff_bitsize 8
3768
```

Let us perform one more experiment. First we define a few simple terms:

```
t0=Fun 0 [Var 0,Var 1]
t1=Fun 1 [t0,t0]
t2=Fun 2 [t0,t0,t1,t1,t1]
t3=Fun 3 [t0,t1,t2]
```

We can now compare the bitsize of the Elias code and our self-delimiting code as follows:

```
*Goedel> length (as elias nterm t1)
46
*Goedel> length (as hff_pars nterm t1)
50
*Goedel> length (as elias nterm t2)
225
*Goedel> length (as hff_pars nterm t2)
198
*Goedel> length (as elias nterm t3)
866
*Goedel> length (as hff_pars nterm t3)
282
```

As the size of the terms grows, our recursively self-delimiting code seems to outperforms Elias code by a significant margin.

Based on prop. 7 and our experiments, we can conclude that while typically less compact than Elias omega code, the self-delimiting code provided by our hereditarily finite function encoding, has the "fractal-like" property that recursively, all its subcomponents are also self delimited. Also, the gap between the two becomes insignificant when one additional level of self-delimiting is imposed in the case of the application of Elias code to each element of a sequence.

7. Related work

The paper makes use of the embedded data transformation language introduced in [Tarau 2009], a large unpublished draft, also organized as literate Haskell program (see a summary in the Appendix). The framework is described in more detail, in a logic programming setting in submission `#44` at this conference.

Ranking functions can be traced back to Gödel numberings [Gödel 1931, Hartmanis and Baker 1974] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [Martinez and Molinero 2003, Knuth 2006, Ruskey and Proskurowski 1990, Myrvold and Ruskey 2001]. The generic view of such transformations as hylomorphisms obtained compositionally from simpler isomorphisms, as described in this paper, originates in [Tarau 2009].

Natural Number encodings of hereditarily finite sets (that have been the main inspiration for our concept of hereditarily finite functions) have triggered the interest of researchers in fields ranging from Axiomatic Set Theory to Foundations of Logic [Takahashi 1976, Kaye and Wong 2007, Abian and Lamacchia 1978, Kirby 2007].

Pairing functions have been used in work on decision problems as early as [Robinson 1950]. A typical use in the foundations of mathematics is [Cégielski and Richard 2001]. An extensive study of various pairing functions and their computational properties is presented in [Rosenberg 2003].

The closest reference on encapsulating bijections as a Haskell data type is [Alimarine et al. 2005] and Conal Elliott's composable bijections module [Conal Elliott], where, in a more complex setting, Arrows [Hughes] are used as the underlying abstractions. [Kahl and Schmidt 2000] uses a similar category theory inspired framework implementing relational algebra, also in a Haskell setting.

Some other techniques are for sure part of the scientific commons. In that case our focus was to express them as elegantly as possible in a uniform framework.

8. Conclusion

We have described a compact bijective Gödel numbering scheme for term algebras. The algorithm works 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. A similar bijective encoding of SAT problems showed the flexibility of our data transformation framework. We have explored some basic properties of a new self-delimiting code derived from a parenthesis language representation of hereditarily finite functions. The most interesting aspect of this code is the "fractal-like" property that all its subcomponents are also self delimited. We foresee some practical applications to encode complex information streams with heterogeneous subcomponents - for instance as a mechanism for sending serialized objects over a wireless channel.

References

Alexander Abian and Samuel Lamacchia. On the consistency and independence of some set-theoretical constructs. *Notre Dame Journal of Formal Logic*, X1X(1):155–158, 1978.

Wilhelm Friedrich Ackermann. Die Widerspruchsfreiheit der allgemeinen Mengenlhere. *Mathematische Annalen*, (114):305–315, 1937.

Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 86–97, New York, NY, USA, 2005. ACM Press.

Patrick Cégielski and Denis Richard. Decidability of the theory of the natural integers with the cantor pairing function and the successor. *Theor. Comput. Sci.*, 257(1-2):51–77, 2001.

Conal Elliott. Data.Bijections Haskell Module. http://haskell.org/haskellwiki/TypeCompose.

P Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

G. Felici, K. Truemper, and F. S. Sun. A MINSAT approach for learning in logic domains. *INFORMS Journal on Computing*, 2002.

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

R. Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, (9):33–41, 1944.

Juris Hartmanis and Theodore P. Baker. On simple goedel numberings and translations. In Jacques Loeckx, editor, *ICALP*, volume 14 of *Lecture Notes in Computer Science*, pages 301–316. Springer, 1974. ISBN 3-540-06841-4. URL `http://dblp.uni-trier.de/db/conf/icalp/icalp74.html#HartmanisB74`.

Peter Hawkins and Peter J. Stuckey. A hybrid bdd and sat finite domain constraint solver. In Pascal Van Hentenryck, editor, *PADL*, volume 3819 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2006. ISBN 3-540-30947-0.

John Hughes. Generalizing Monads to Arrows. Science of Computer Programming 37, pp. 67-111, May 2000.

Wolfram Kahl and Gunther Schmidt. Exploring (finite) Relation Algebras using Tools written in Haskell. Technical Report 2000-02, Fakultät für Informatik, Universität der Bundeswehr München, October 2000. URL `http://ist.unibw-muenchen.de/Publications/TR/2000-02/`.

Laszlo Kalmar. On the reduction of the decision problem. first paper. ackermann prefix, a single binary predicate. *The Journal of Symbolic Logic*, 4(1):1–9, mar 1939. ISSN 0022-4812.

Richard Kaye and Tin Lock Wong. On Interpretations of Arithmetic and Set Theory. *Notre Dame J. Formal Logic Volume*, 48(4):497–510, 2007.

Laurence Kirby. Addition and multiplication of sets. *Math. Log. Q.*, 53(1): 52–65, 2007.

Donald Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.

Conrado Martinez and Xavier Molinero. Generic algorithms for the generation of combinatorial objects. In Branislav Rovan and Peter Vojtas, editors, *MFCS*, volume 2747 of *Lecture Notes in Computer Science*, pages 572–581. Springer, 2003.

Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79:281–284, 2001.

Jozef Pepis. Ein verfahren der mathematischen logik. *The Journal of Symbolic Logic*, 3(2):61–76, jun 1938. ISSN 0022-4812.

Stephen Pigeon. *Contributions à la compression de données.* Ph.d. thesis, Université de Montréal, Montréal, 2001.

Julia Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950. ISSN 0002-9939.

Arnold L. Rosenberg. Efficient pairing functions - and why you should care. *International Journal of Foundations of Computer Science*, 14(1):3–17, 2003.

Frank Ruskey and Andrzej Proskurowski. Generating binary trees by transpositions. *J. Algorithms*, 11:68–84, 1990.

N. J. A. Sloane. A000695, The On-Line Encyclopedia of Integer Sequences. 2006. published electronically at www.research.att.com/∼njas/sequences.

Moto-o Takahashi. A Foundation of Finite Mathematics. *Publ. Res. Inst. Math. Sci.*, 12(3):577–708, 1976.

Paul Tarau. Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell, January 2009. http://arXiv.org/abs/0808.2953, unpublished draft, 104 pages.

Appendix

Quick Overview of a Bijective Data Transformation Framework

A relatively small number of universal data types are used as basic building blocks in programming languages and their runtime interpreters, corresponding to a few well tested mathematical abstractions like sets, sequences, multisets etc. We summarize here the framework described at `http://arXiv.org/abs/0808.2953` that provides bijective any-to-any conversions between various data types together with a general mechanism for transporting their operations.

Connecting data types with a groupoid of isomorphisms

A category in which every morphism is an *isomorphism* is called a *groupoid*. We represent *isomorphism* pairs as a data type `Iso`, together with the operations `compose, itself` and `invert` providing together a (finite) *groupoid* structure.

```
data Iso a b = Iso (a→b) (b→a)

compose :: Iso a b → Iso b c → Iso a c
compose (Iso f g) (Iso f' g') = Iso (f' . f) (g . g')

itself = Iso id id

invert (Iso f g) = Iso g f
```

We will put at work these combinators by designing bijections between various data types. They transport operations and are invertible. This justifies seeing them as are *isomorphisms* between data types. Such bijections are typed, therefore `f` and `g` are composable morphisms only if the target of `f` is identical with the source of `g`. These two considerations make the "natural" structure hosting them a *groupoid*.

Connecting through a Hub

Assuming our isomorphisms form a *connected groupoid* it makes sense at this point to route them through a *hub* dat type to avoid having to provide $n * (n-1)/2$ isomorphisms.

Let us introduce our natural numbers $\mathbb{N}$ as an arbitrary length integer subtype together with a predicate `isN` implementing their canonical embedding in $\mathbb{Z}$.

```
type N = Integer
isN n = n≥0
```

A possible choice for such a hub is $[\mathbb{N}]$ - seen here as the set of finite sequences of natural numbers, or, equivalently, as the set of finite functions from an initial segment of $\mathbb{N}$. We call a connector from a data type to the hub an `Encoder`:

```
type Encoder a = Iso a [N]
```

We first define a trivial `Encoder`:

```
nats :: Encoder [N]
nats = itself
```

One can route isomorphism through the Hub with Encoders, using the combinator `as`:

```
as :: Encoder a → Encoder b → b → a
as that this x = g x where
   Iso _ g = compose that (invert this)
```

A one argument function `f` is transported between data types using the combinator `borrow_from`:

```
borrow_from :: Encoder b → (b → b) →
               Encoder a → a → a

borrow_from lender f borrower =
  (as borrower lender) . f . (as lender borrower)
```

Similarly, a two argument function `op` is transported between data types using the combinator `borrow_from2`:

```
borrow_from2 :: Encoder a → (a → a → a) →
                Encoder b → b → b → b

borrow_from2 lender op borrower x y =
    as borrower lender r where
       x'= as lender borrower x
       y'= as lender borrower y
       r = op x' y'
```

Encoding of some fundamental data types

We will now quickly put the mechanism at work and show that Encoders for some fundamental data types are surprisingly easy to build.

From finite sequences to finite multisets of natural numbers

An encoder of multisets (assumed ordered) as sequences is obtained by "summing up" a sequence with `scanl`.

```
mset :: Encoder [N]
mset = compose (Iso as_nats_mset as_mset_nats) nats

as_mset_nats ns = tail (scanl (+) 0 ns)
as_nats_mset ms = zipWith (-) (ms) (0:ms)
```

While finite multisets and sequences share a common representation $[N]$, multisets are subject to the implicit constraint that the order of their elements is immaterial i.e. they can be seen as a quotient set with respect to an equivalence relations given by reorderings. Thus they are canonically represented as non-decreasing sequences assuming an embedding into arbitrary finite sequences provided by set inclusion. The constraints inducing such injective embeddings of a data type in another can be regarded as *laws/assertions* restricting the host data type to the domain of the embedded mathematical concept. We will implicitly assume such injective embeddings, when needed.

From finite sequences to finite sets of natural numbers

An encoder of finite sets of natural numbers (assumed ordered) as sequences is obtained by adjusting the encoding of multisets so that 0s are first mapped to 1s - this ensures that all elements are different.

```
set :: Encoder [N]
set = compose (Iso as_nats_set as_set_nats) nats

as_set_nats = (map pred) . as_mset_nats . (map succ)
as_nats_set = (map pred) . as_nats_mset . (map succ)
```

Examples

We will show through a few examples the action of the data type transformation combinators.

```
*Goedel> as mset nats [2,0,1,0]
[2,2,3,3]
*Goedel> as set mset [2,2,3,3]
[2,3,5,6]
*Goedel> as nats set [2,3,5,6]
[2,0,1,0]
```

Bit crunching functions

The function bitcount computes the number of bits needed to represent an integer and max_bitcount computes the maximum bitcount for a list of integers.

```
bitcount n = head [x|x←[1..],(2^x)>n]
max_bitcount ns = foldl max 0 (map bitcount ns)
```

The following function converts a number to to binary, padded with 0s, up to maxbits.

```
to_maxbits maxbits n =
  bs ++ (genericTake (maxbits-l)) (repeat 0) where
    bs=to_base 2 n
    l=genericLength bs
```

Conversions to/from a given base are implemented as follows:

```
to_base base n | base > 1 = d :
  (if q==0 then [] else (to_base base q)) where
     (q,d) = quotRem n base

from_base base [] = 0
from_base base (x:xs) | x≥0 && x<base =
   x+base*(from_base base xs)
```

Primes

The following code implements factoring function `to_primes` a primality test (`is_prime`) and a generator for the infinite stream of prime numbers `primes`.

```
primes = 2 : filter is_prime [3,5..]

is_prime p = [p]==to_primes p

to_primes n | n>1 = to_factors n p ps where
  (p:ps) = primes

to_factors n p ps | p*p > n = [n]
to_factors n p ps | 0==n `mod` p =
   p : to_factors (n `div` p)  p ps
to_factors n p ps@(hd:tl) = to_factors n hd tl

pi_ n = primes !! (fromIntegral n)
pi' p | is_prime p= to_pos_in primes p

to_pos_in xs x =
  fromIntegral i where Just i=elemIndex x xs
```

Multiset Operations

The following functions provide multiset analogues of the usual set operations, under the assumption that multisets are represented as non-decreasing sequences.

```
msetInter xs ys = sort (msetInter' xs ys)

msetInter' [] _ = []
msetInter' _ [] = []
msetInter' (x:xs) (y:ys) | x==y =
  (x:zs) where zs=msetInter' xs ys
msetInter' (x:xs) (y:ys) | x<y = msetInter' xs (y:ys)
msetInter' (x:xs) (y:ys) | x>y = msetInter' (x:xs) ys

msetDif xs ys = sort (msetDif' xs ys)

msetDif' [] _ = []
msetDif' xs [] = xs
msetDif' (x:xs) (y:ys) | x==y = zs where
  zs=msetDif' xs ys
msetDif' (x:xs) (y:ys) | x<y = (x:zs) where
  zs=msetDif' xs (y:ys)

msetDif' (x:xs) (y:ys) | x>y = zs where
  zs=msetDif' (x:xs) ys

msetSymDif xs ys =
  sort ((msetDif xs ys) ++ (msetDif ys xs))

msetUnion xs ys = sort ((msetDif xs ys) ++
  (msetInter xs ys) ++ (msetDif ys xs))

msetIncl xs ys = xs==msetInter xs ys
```