Bijective Gödel Numberings for Term Algebras

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 in the bitsize of the representations d) the bitsize of our encoding is within constant factor of the syntactic representation of the input.
The algorithm can be applied to derive compact serialized representations for various formal systems and programming language constructs. The paper is organized as a literate Haskell program available from `http://logic.cse.unt.edu/tarau/research/2009/fgoedel.hs`.
Keywords: *ranking/unranking functions, natural number encodings of terms, Gödel numberings, computational mathematics in Haskell*

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

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

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

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

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

Through the paper, we will make use of the embedded data transformation language introduced in [3, 4] 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 [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 [4]. We group our code in a self-contained module `Goedel`, importing only two library modules:

```
module Goedel where
import Data.List
import Data.Bits
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* 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  = shiftL (1 .|. (shiftL y 1)) (fromIntegral x)

hd :: N→N
hd n | n>0 = if 1==n .&. 1 then 0 else succ (hd  (shiftR n 1))

tl :: N→N
tl n = shiftR n (fromIntegral (succ (hd n)))
```

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 easy to see that the following holds:

Proposition 1 *cons x $y = 2^x(2y+1)$ and together with hd and tl it defines a bijection between $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$;* `as_nat_nats` *is a bijection from finite sequences of natural numbers to natural numbers and* `as_nats_nat` *is its inverse.*

Using the groupoid of data type isomorphisms introduced in [3] (see also the Appendix), 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 and fairly efficient due to its implementation using bitshift operations, this encoder has the disadvantage that the bitsize of the encoding can be *exponential in the bitsize of the elements of a sequence*:

```
*Goedel> as nat1 nats [50,20,50]
531691198313966585279959557585082776
```

We shall therefore build, through the following sections, a more intricate encoding, while making sure that the computations it involves will always use time and space proportional to the bitsize of their inputs.

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 and decidability [5–7].

Using the functions `pair` and `unpair`, we define following [3], 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 $\mathbb{N}$.

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 the 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
```

Clearly, the following holds:

Proposition 3 *The functions* `from_tuple` *and* `from_tuple` *are inverse and both work in time and space proportional to the bitsize of their inputs.*

The following example shows the decoding of `42` and the encoding of the tuple back to `42`.

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

Fig. 1 shows multiple steps of the same decomposition, with shared nodes collected in a DAG and labels on the edges indicating the order in a tuple.

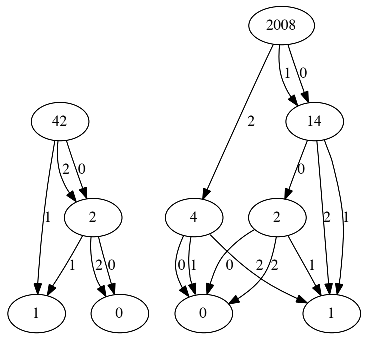

Fig. 1: Repeated 3-tuple expansions: *42* and *2008*

2.4 Encoding Finite Functions as Tuples

A `finite function` can be seen as a function defined on an initial segment of N with values in N. As finite sets can be put in a bijection with an initial segment of N, we can encode and decode a finite function from $[0..k-1]$ to N (seen as the list of its values), as a natural number, by using our pairing and tupling functions.

```
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
```

One can see that the first argument of the pairing function controls the length of the tuple while the second controls the bits defining the tuple. It follows immediately from Prop. 3 that:

Proposition 4 *The encoder* `nat` *works in space and time proportional to the bitsize of its input.*

as shown in the following example

```
*Goedel> as nat nats [2009,2010,4000,0,5000,42]
485513619123942740473456O
*Goedel> as nats nat it
[2009,2010,4000,0,5000,42]
```

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 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 the list:

```
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` parameterized by `var` and `const` as:

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

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=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. To obtain an encoding of strings linear in their bitsize we need a general mechanism to map arbitrary combinations of `k` symbols (seen as digits in base k) to natural numbers.

3.3 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* that provides such a bijection is implemented as follows:

```
bijnat :: N→Encoder [N]

bijnat a = compose (Iso (from_bbase a) (to_bbase a)) nat

from_bbase base xs = from_base' base (map succ xs)

from_base' base [] = 0
from_base' base (x:xs) | x>0 && x≤base =
   x+base*(from_base' base xs)

to_bbase base n = map pred (to_base' base n)

to_base' _ 0 = []
to_base' base n = d' : ds where
   (q,d) = quotRem n base
   d'=if d==0 then base else d
   q'=if d==0 then q-1 else q
   ds=if q'==0 then [] else to_base' base q'
```

Note that the encoder bijnat is parametrized by the base of numeration, i.e. the `as` combinator works as follows:

```
*ISO> as (bijnat 3) nat 2009
[1,2,2,0,2,0,1]
*ISO> as nat (bijnat 3) it
2009
*ISO> as (bijnat 10) nat 2009
[8,9,8,0]
*ISO> as nat (bijnat 10) it
2009
*ISO> map (as (bijnat 3) nat) [0..12]
[[],[0],[1],[2],[0,0],[1,0],[2,0],[0,1],
    [1,1],[2,1],[0,2],[1,2],[2,2]]
```

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

3.4 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 i.e. we set:

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

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

Next, we define the bijective base-k encodings

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

nat2string n = map (ord2chr . fromIntegral)
  (to_bbase (fromIntegral base) 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 `string` immediately as

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

working as follows:

```
*Goedel> as nat string "hello"
7073802
*Goedel> as string nat it
"hello"
```

3.5 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 nat sterm (Fun "b" [Fun "a" [],Var 0])
2215
*Goedel> as sterm nat it
Fun "b" [Fun "a" [],Var 0]

*Goedel> as nat sterm (Fun "forall" [Var 0, Fun "f" [Var 0]])
3869627004010296175657939 9
*Goedel> as sterm nat it
Fun "forall" [Var 0,Fun "f" [Var 0]]

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

3.6 Mapping terms to arbitrary bitstrings

Term algebras are *free magmas* generated through fairly complex substitution operations. Their underlying data representation involves 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 $\{0,1\}^*$. Let's observe that this encoding is isomorphic to the *bijective base 2* representation, so we can just instantiate the parametric encoder `bijnat`

```
bits :: Encoder [N]
bits = bijnat 2
```

working 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 in fact a *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.

Proposition 5 *The Gödel numbering algorithms implemented by the encoders* `nterm` *and* `sterm` *are bijective and work in time and space linear in the bitsize of their input. Moreover, all natural numbers decode to syntactically valid terms.*

4 Primes and 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 [3, 4] (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 2009
[0,1,2,18]
*Goedel> as nat pmset it
2009
```

Note that the mappings from a set or sequence to a number discussed previously 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.

We can also derive set and sequence encodings from this, by observing that the encoding between sets, mulitsets and sequences is independent of their mapping to natural numbers. We obtain the Encoders:

```
pnats :: Encoder [N]
pnats = compose (Iso as_mset_nats as_nats_mset) pmset

pset :: Encoder [N]
pset = compose (Iso as_nats_set as_set_nats) pnats
```

working as follows:

```
*Goedel> as pnats nat 2009
[0,1,1,16]
*Goedel> as pset pnats it
[0,2,4,21]
*Goedel> as nat pset it
2009
```

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. by computing $2^{n_0} * 3^{n_1} * \ldots p_i^{n_i} * \ldots$ 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
[0,0,0,2,0,0,0,0,0,0,0,0,1]
*Goedel> nats2goedel [0,0,0,2,0,0,0,0,0,0,0,0,1]
2009
*Goedel> nats2goedel [0,0,0,2,0,0,0,0,0,0,0,0,1,0,0,0]
2009
```

Reversing it requires factoring and, as seen from the previous example, needs a small fix: to avoid zeros 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
[1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]
*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
```

5 Related work

The paper makes use of the embedded data transformation language introduced in [3, 4] also organized as literate Haskell programs (see a summary in the Appendix).

Ranking functions can be traced back to Gödel numberings [1, 2] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [8, 9]. The generic view of such transformations as hylomorphisms obtained compositionally from simpler isomorphisms, as described in this paper, originates in [4].

Pairing functions have been used in work on decision problems as early as [7]. A typical use in the foundations of mathematics is [10]. An extensive study of various pairing functions and their computational properties is presented in [11].

The closest reference on encapsulating bijections as a Haskell data type is [12] and Conal Elliott's composable bijections module [13]. [14] 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.

6 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. 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.

References

1. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
2. Hartmanis, J., Baker, T.P.: On simple goedel numberings and translations. In Loeckx, J., ed.: ICALP. Volume 14 of Lecture Notes in Computer Science., Springer (1974) 301–316
3. 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
4. Tarau, P.: Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell (January 2009) http://arXiv.org/abs/0808.2953, unpublished draft, 104 pages.
5. Pepis, J.: Ein verfahren der mathematischen logik. The Journal of Symbolic Logic **3**(2) (jun 1938) 61–76
6. Kalmar, L.: On the reduction of the decision problem. first paper. ackermann prefix, a single binary predicate. The Journal of Symbolic Logic **4**(1) (mar 1939) 1–9
7. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
8. Martinez, C., Molinero, X.: Generic algorithms for the generation of combinatorial objects. In Rovan, B., Vojtas, P., eds.: MFCS. Volume 2747 of Lecture Notes in Computer Science., Springer (2003) 572–581
9. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
10. 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
11. Rosenberg, A.L.: Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science **14**(1) (2003) 3–17
12. Alimarine, A., Smetsers, S., van Weelden, A., van Eekelen, M., Plasmeijer, R.: There and back again: arrows for invertible programming. In: Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM Press (2005) 86–97
13. Conal Elliott: Module: Data.Bijections Haskell source code library at: http://haskell.org/haskellwiki/TypeCompose.
14. Kahl, W., Schmidt, G.: Exploring (finite) Relation Algebras using Tools written in Haskell. Technical Report 2000-02, Fakultät für Informatik, Universität der Bundeswehr München (October 2000)

Appendix

We describe here a few functions needed to make the paper a self-contained literate Haskell program.

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)
```

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 `0`s are first mapped to `1`s - 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 of encodings:

```
*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 (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
```