Efficient Bijective Gödel Numberings for Term Algebras

Paul Tarau[1]

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

Trends in Functional Programming, May 2010

Motivation

Gödel's incompleteness results (relying on Gödel numberings) had a huge impact on logic, foundations of mathematics, number theory, computer science and quite a few other fields.

- some infelicities of the original Gödel numberings:
  - encoding individual symbols rather then expression trees - using exponents of distinct prime numbers
  - computing the inverse is intractable (based on factoring)
  - encodings of syntactically ill-formed terms are possible

none of those shortcomings matter when focus is on computability only, but they do when one cares about computational complexity

Revisiting Gödel numberings - with “efficiency" in mind

- we design Gödel numberings with the following properties:
  - bijective
  - natural numbers always decode to syntactically valid terms
  - work in linear time in the bitsize of the representations
  - the bitsize of the encoding is within constant factor of the syntactic representation of the input
  - encodings on Term Algebras $\Rightarrow$ good for both code and data!

to be able to encode something as something else we need isomorphisms $\longrightarrow$ bijections that transport structures

The Groupoid of Isomorphisms

```
data Iso a b = Iso (a→b) (b→a)
from (Iso f _) = f
to (Iso _ g) = g

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
```

Proposition

`Iso` *is a groupoid: when defined,* `compose` *is associative,* `itself` *is an identity element,* `invert` *computes the inverse of an isomorphism.*

Connecting through a Hub

```
type N = Integer
isN n = n≥0
type Hub = [N]
```

We can now define an *Encoder* as an isomorphism connecting an object to *Root*

```
type Encoder a = Iso a Hub
```

This avoids having to provide $\frac{n*(n-1)}{2}$ isomorphisms!
The combinator *"as"* routes isomorphisms through two *Encoders*:

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

An Example: Lists to/from Sets

```
*Goedel> as set nats [0,1,0,0,4]
[0,2,3,4,9]
*Goedel> as nats set [0,2,3,4,9]
[0,1,0,0,4]
```

How we do it? We can map lists of natural numbers to strictly increasing sequences of natural numbers representing sets!

List		List'		Set
[0, 1,0,0,4]	$\rightarrow$	[0, 2,1,1,5]	$\rightarrow$	[0, 2,3,4,9]

$\Rightarrow$ Ackermann's encoding to $\mathbb{N}$: $2^0 + 2^2 + 2^3 + 2^4 + 2^9 = 541$

Morphing between Lists/Multisets/Sets

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

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)

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)
```

Uncovering the implicit list structure of a natural number

Proposition

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

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

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

hd, tl, cons, 0

```
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)

*Goedel> hd 2008 ⇒ 3
*Goedel> tl 2008 ⇒ 125
*Goedel> cons 3 125 ⇒ 2008
```

Morphing between $\mathbb{N}$ and $[\mathbb{N}]$

```
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)

*Goedel> as_nats_nat 2008
[3,0,1,0,0,0,0]
*Goedel> as_nat_nats [3,0,1,0,0,0,0]
2008
```

A problem - exponential in the size of the input [$\mathbb{N}$]

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

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

Pairing Functions as Encoders

Definition

An isomorphism $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ *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
```

Proposition

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

An encoder for tuples

```
to_tuple k n = map (from_base 2) (
    transpose (
      map (to_maxbits k) (
        to_base (2^k) n
      )
    )
  )
```

Simple: first bit to the first number, next bit to the next etc.

```
*Goedel> to_tuple 5 2012
[4,2,3,3,3]
```

An decoder for tuples

```
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
```

Just merging back the bits (but some padding is needed)!

```
*Goedel> from_tuple [4,2,3,3,3]
2012
```

Encoding with Tuples

1. split $n \in \mathbb{N}$ with unpair $n = 2^x(2y+1) - 1$ giving (x,y)
2. use the first element x as the length of the tuple
3. split the second element y to a *tuple* with x elements

```
nat2ftuple 0 = []
nat2ftuple n = to_tuple (succ x) y where
  (x,y)=unpair (pred n)

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

Encoding of lists proportional to the total bitsize of their elements

```
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.

A compact encoding of lists

Proposition

The encoder `nat` *works in space and time proportional to the bitsize of the largest element of the list multiplied by the length of the list.*

```
*Goedel> as nat nats [2009,2010,4000,0,5000,42]
4855136191239427404734560
*Goedel> as nats nat it
[2009,2010,4000,0,5000,42]

*Goedel> as nat1 nats [2009,2010,4000,0,5000,42]
1811020413277069844...
....2 pages more ....
......53964009455616
```

Term Algebras

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

From Terms to Natural Numbers

1. separate encodings of variable and function symbols i.e. map them, respectively, to even and odd numbers
2. to deal with function arguments, use the bijective encoding of sequences recursively

```
type NTerm = Term N N

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
```

From Natural Numbers, back to Terms

1. recurse over the sequence associated to a natural number by the `as nats nat` combinator
2. associate variables to even numbers

```
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
```

The Encoder `nterm`

We can encapsulate our transformers as the Encoder:

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

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

Encoding strings with bijective base-k numbers

```
*Goedel> 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]]

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

More realistic terms - with strings as function names

```
*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]]
```

A view as bijective base-2 bitstrings

```
*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,
 0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

*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]
```

Conclusion

- literate Haskell – a powerful tool for “experimental" theoretical computer science
- the original field for Gödel numberings is computability theory
- our Gödel numberings are“complexity aware" - possible uses in encodings relevant for complexity theory
  - encodings work in space and time proportional to the bitsize of the representations
  - natural numbers always decode to syntactically valid terms
- a possible more practical application: generate random terms - useful for QuickCheck-style testing
- also - natural numbers represent terms succinctly $\Rightarrow$ serialization of data and code, compression of terms sent over a network etc.