Hereditarily finite representations of natural numbers and self delimiting codes

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
E-mail: tarau@cs.unt.edu

Abstract. Using an isomorphism between natural numbers and finite sequences we derive self-delimiting codes by lifting it to its hereditarily finite counterpart and then mapping it back to a bitstring representation in a balanced parenthesis language. We conclude with a brief study of the information theoretic efficiency of the encodings.
The paper is organized as a self-contained literate Haskell program inviting the reader to explore its content independently.
Keywords: *ranking/unranking bijections, hereditarily finite data types, self-delimiting codes, combinatorics and computational mathematics in Haskell*

1 Introduction

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 [1] rely on encoding length information recursively i.e. delimiting is achieved by first encoding the length, then the of the length etc. In the context of a larger effort to interconnect various data types through bijective mappings organized as a groupoid of data transformations [2, 3] we will introduce here 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 back 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.

2 A ranking/unranking algorithm for finite sequences

A *ranking/unranking* function defined on a data type is a bijection to/from the set of natural numbers (denoted Nat through the paper). We start with an unusually simple but (at our best knowledge) novel ranking/unranking algorithm for finite sequences of arbitrary unbounded size natural numbers denoted $[Nat]$[1]. Given the definitions

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

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

tl :: Nat→Nat
tl n = n ‘div‘ 2^((hd n)+1)

nat2fun :: Nat→[Nat]
nat2fun 0 = []
nat2fun n = hd n : nat2fun (tl n)

fun2nat :: [Nat]→Nat
fun2nat [] = 0
fun2nat (x:xs) = cons x (fun2nat xs)
```

the following holds

Proposition 1 `fun2nat` *is a bijection from finite sequences of natural numbers to natural numbers and* `nat2fun` *is its inverse.*

This follows from the fact that `cons` and the pair `(hd, tl)` define a bijection between $Nat - \{0\}$ and $Nat \times Nat$ and that the value of `fun2nat` is uniquely determined by the number of applications of `tl` and the sequence of values returned by `hd`.

```
*ISO> hd 2008
3
*ISO> tl 2008
125
*ISO> cons 3 125
2008
```

Following [2] (summarized in the Appendix) we can define the `Encoder`

```
nat :: Encoder Nat
nat = Iso nat2fun fun2nat
```

working as follows

```
*ISO> as fun nat 2008
[3,0,1,0,0,0,0]
*ISO> as nat fun [3,0,1,0,0,0,0]
2008
```

Given the definitions:

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

[1] See Haskell types Nat and $[Nat]$ in Appendix.

the following holds

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

This follows from the fact that shifting by 1 turns `hd` and `tl` in total functions on Nat such that *unpair* $0 = (0, 0)$.

As the cognoscenti might notice, this is in fact a classic *pairing function* that has been used, by Pepis, Kalmar and Robinson in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem in [4–10] and that `hd,tl,cons,0` define on Nat an algebraic structure isomorphic to the one introduced by `car,cdr,cons,NULL` in John McCarthy's classic LISP paper [11].

3 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.

An asymptotically optimal mechanism for this is the use of a *universal self-delimiting code* for instance, the *Elias omega code* [1]. 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 :: Nat → [Nat]
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 :: [Nat] → (Nat, [Nat])
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 [Nat]
elias = compose (Iso (fst . from_elias) to_elias) nat
```

working as follows:

```
*ISO> as elias nat 2008
[1,1,1,0,1,0,1,1,1,1,1,0,1,1,0,0,1,0]
*ISO> 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.

4 Generic Unranking and Ranking Hylomorphisms

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

4.1 Hereditarily Finite Data Types

The unranking operation is seen here as an instance of a generic *anamorphism* mechanism (an *unfold* operation), while the ranking operation is seen as an instance of the corresponding catamorphism (a *fold* operation) [12, 13]. Together they form a mixed transformation called *hylomorphism*. We will use such hylomorphisms to lift isomorphisms between lists and natural numbers to isomorphisms between a derived "self-similar" tree data type and natural numbers.

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

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

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

```
unrank f n = H (unranks f (f n))
unranks f ns = map (unrank f) ns

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

Both combinators can be seen as a form of "structured recursion" that propagate a simpler operation guided by the shape of the data type. For instance, the size of a tree of type T is obtained as:

```
tsize = rank (λxs→1 + (sum xs))
```

Note also that `unrank` and `rank` work on T in cooperation with `unranks` and `ranks` working on $[T]$.

We can now combine an anamorphism+catamorphism pair into an isomorphism `hylo` defined with `rank` and `unrank` on the corresponding hereditarily finite data types:

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

hylos :: Iso b [b] → Iso [T] [b]
hylos (Iso f g) = Iso (ranks g) (unranks f)
```

The following states that isomorphisms to sequences are lifted to hereditarily finite tree data types.

Proposition 3 *If* `t` *is an isomorphism between* Nat *and* $[Nat]$ *then* `hylo` *and* `hylos` *are isomorphisms between* Nat *and* T *and* $[Nat]$ *and* $[T]$*, respectively.*

4.2 Hereditarily Finite Functions

We will use the tree data type `T` to host a hylomorphism derived from ranking/unranking of finite functions. We refer to [3] for similar hereditarily finite data types derived from ranking functions on sets, multisets and permutations.

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

One can represent the action of a hylomorphism unfolding a natural number into a hereditarily finite function as a directed ordered multi-graph as shown in Fig. 1. Note that as the mapping `as fun nat` generates a sequence where the order of the edges matters, this order is indicated by integers starting from `0` labeling the edges.

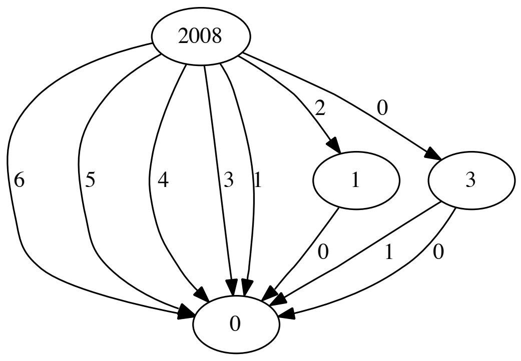

Fig. 1: 2008 as a HFF

It is also interesting to connect sequences and HFF directly - in case one wants to represent giant "sparse numbers" that correspond to sequences that would overflow memory if represented as natural numbers but have a relatively simple structure as formulae used to compute them. We obtain the Encoder:

```
hffs :: Encoder T
hffs = Iso hff2fun fun2hff

fun2hff ns = H (map (as hff nat) ns)
hff2fun (H hs) = map (as nat hff) hs
```

which can be used to generate HFFs associated to very large numbers:

```
*ISO> as hffs fun [2^65,2^131]
H [H [H [H [],H [H [],H [H []]]]],H [H [H [],H [],H [H [],H [H []]]]]]
```

5 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 [Nat]
hff_pars = compose (Iso pars2hff hff2pars) hff
```

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

```
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 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 5 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 4 *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` (see Appendix or [3]):

```
*ISO2> as bits nat 2008
[1,0,0,1,1,0,1,1,1,1]
*ISO> as hff_pars nat 2008
[0,0,0,1,0,1,1,0,1,0,0,1,1,0,1,0,1,0,1,0,1,1]
```

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 $\{0,1\}^*$

There are however cases when the parenthesis language representation is more compact. For instance,

```
*ISO> as nat bits (as hff_pars nat (2^2^16))
32639
```

while the conventional representation of the same number would have thousands of digits. This suggest defining

```
nat2parnat n = as nat bits (as hff_pars nat n)

parnat2nat n = as nat hff_pars (as bits nat n)
```

to find out that more compact representations only happen for a few numbers that are powers of two or "sparse" sums of powers of two:

```
*ISO> [x|x←[0..2^16],nat2parnat x<x]
[8192,16384,32768,32769,49152,65536]
```

As one could guess from the previous comparison with bitstrings, let's note that this is especially interesting for streams of values expressed as combinations of a few exponents of 2, as shown in Fig. 2. 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))]
```

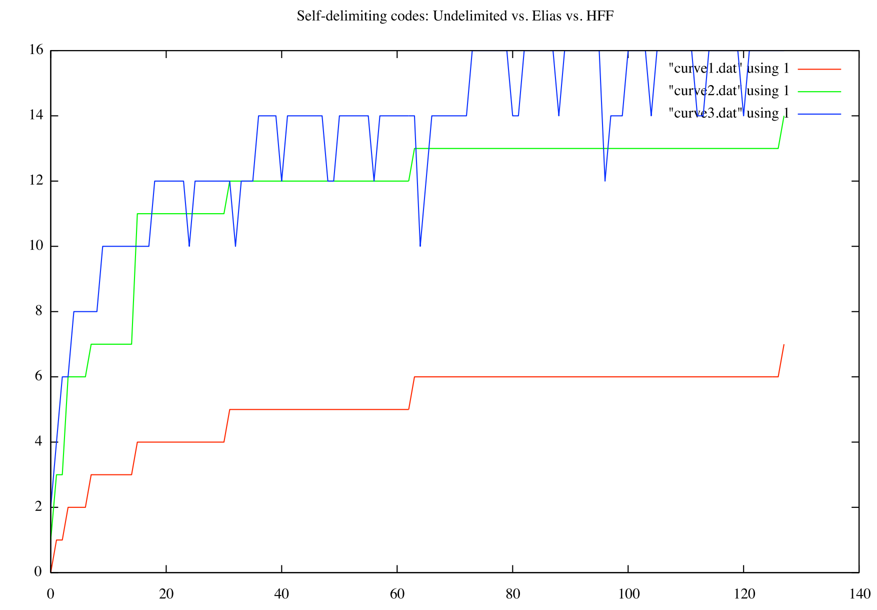

Fig. 2: Comparison of codes: curve1=Undelimited, curve2=Elias, curve3=HFF, up to 2^7

working as follows

```
*ISO> sparses_to (2^9)
[15,16,17,24,32,64,65,96,128,129,192,256,257,258,259,320,384,385,448]
```

A good way to evaluate "information density" for an arbitrary data type that is isomorphic to `Nat` 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)
```

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 the "perfect" value of `1`:

```
*ISO> map info_density_hff [0..12]
[0.0,0.1666666666666666,0.2222222222222222,0.26,0.296875,0.32802547770700635,
0.3548387096774194,0.37732558139534883,0.39600409836065575,0.41151556776556775,
0.4244180031039834,0.43526996413064,0.44448060400937256]
```

Similarly, for Elias coding we obtain:

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

*ISO> map info_density_elias [0..12]
[0.0,0.25,0.3076923076923077,0.34210526315789475,0.3877551020408163,
 0.37454545454545457,0.4,0.434695244474213,0.4703376939458473,0.486863488624052,
 0.5099136055690223,0.5342969700480853,0.557962824266358]
```

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

```
*ISO> map relative_density_hff [0..12]
[0.5,0.6666666666666666,0.7222222222222222,0.76,0.765625,0.8757961783439491,
 0.8870967741935484,0.8680232558139535,0.8419569672131147,0.8452380952380952,
 0.8323331608898086,0.8146592410798565,0.7966132951488327]
```

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 $f : Nat \to [Nat]$:

```
nat2sfun n = nat2self (as fun 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 fun) ns
```

A more elaborate concept of sparseness is derived by comparing the size of a self-delimiting code for a number `n` vs. the size of its self-delimiting representation as a finite sequence, computed as follows:

```
linear_sparseness t n =
  (genericLength (to_elias n))/(genericLength (nat2self (as t nat) n))
```

We can also extend this comparison to hereditarily finite representations, which, as we have seen, turn out to provide self-delimiting codes.

```
sparseness f n =
  (genericLength (to_elias n)) / (genericLength (as f nat n))
```

Note that no extra delimiting for subcomponents or length needs to be encoded in this case. *We can conclude that while typically less compact than Elias omega code, a self-delimiting code, as provided by our hereditarily finite function encoding, has the "fractal property" that all its subcomponents are also self delimited.*

7 Related work

Ranking functions can be traced back to Gödel numberings [14, 15] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [16–19]. The generic view of such transformations as hylomorphisms obtained compositionally from simpler isomorphisms, as described in this paper, originates in [2] and [3].

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 [20–24].

Hereditarily finite functions have been introduced in our paper [2] and extensively used for various combinatorial encodings in [3], including self-delimiting codes derived from hereditarily finite sets, multisets and permutations.

The closest reference on encapsulating bijections as a Haskell data type is [25] and Conal Elliott's composable bijections module [26], where, in a more complex setting, Arrows [27] are used as the underlying abstractions.

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 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

1. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory **21**(2) (1975) 194–203
2. Tarau, P.: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In: Proceedings of ACM SAC'09, Honolulu, Hawaii, ACM (March 2009) 1898–1903
3. 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.
4. Pepis, J.: Ein verfahren der mathematischen logik. The Journal of Symbolic Logic **3**(2) (jun 1938) 61–76
5. 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
6. Kalmar, Laszlo, Suranyi, Janos: On the reduction of the decision problem. The Journal of Symbolic Logic **12**(3) (sep 1947) 65–73
7. Kalmar, Laszlo, Suranyi, Janos: On the reduction of the decision problem: Third paper. pepis prefix, a single binary predicate. The Journal of Symbolic Logic **15**(3) (sep 1950) 161–173
8. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
9. Robinson, J.: Recursive functions of one variable. Proceedings of the American Mathematical Society **19**(4) (aug 1968) 815–820
10. Robinson, J.: Finite generation of recursively enumerable sets. Proceedings of the American Mathematical Society **19**(6) (dec 1968) 1480–1486
11. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM **3**(4) (1960) 184–195
12. Hutton, G.: A Tutorial on the Universality and Expressiveness of Fold. J. Funct. Program. **9**(4) (1999) 355–372
13. Meijer, E., Hutton, G.: Bananas in Space: Extending Fold and Unfold to Exponential Types. In: FPCA. (1995) 324–333
14. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
15. 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
16. 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
17. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
18. Ruskey, F., Proskurowski, A.: Generating binary trees by transpositions. J. Algorithms **11** (1990) 68–84
19. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters **79** (2001) 281–284
20. Takahashi, M.o.: A Foundation of Finite Mathematics. Publ. Res. Inst. Math. Sci. **12**(3) (1976) 577–708

21. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume **48**(4) (2007) 497–510
22. Abian, A., Lamacchia, S.: On the consistency and independence of some set-theoretical constructs. Notre Dame Journal of Formal Logic **X1X**(1) (1978) 155–158
23. Avigad, J.: The Combinatorics of Propositional Provability. In: ASL Winter Meeting, San Diego (January 1997)
24. Kirby, L.: Addition and multiplication of sets. Math. Log. Q. **53**(1) (2007) 52–65
25. 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
26. Conal Elliott: Data.Bijections Haskell Module. http://haskell.org/haskellwiki/TypeCompose.
27. Hughes, J.: Generalizing Monads to Arrows Science of Computer Programming 37, pp. 67-111, May 2000.

Appendix

An Embedded Data Transformation Language

We will describe briefly the embedded data transformation language used in this paper as a set of operations on a groupoid of isomorphisms. We will then extended it with a set of higher order combinators mediating the composition of the encodings and the transfer of operations between data types.

The Groupoid of Isomorphisms We implement an isomorphism between two objects X and Y as a Haskell data type encapsulating a bijection f and its inverse g. We will call the *from* function the first component (a *section* in category theory parlance) and the *to* function the second component (a *retraction*) defining the isomorphism. We can organize isomorphisms as a *groupoid* as follows:

$$\begin{tikzcd} X \arrow[r, shift left, "f = g^{-1}"] & Y \arrow[l, shift left, "g = f^{-1}"] \end{tikzcd}$$

```
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
```

Assuming that for any pair of type `Iso a b`, $f \circ g = id_a$ and $g \circ f = id_b$, we can now formulate *laws* about these isomorphisms.

The data type `Iso` has a groupoid structure, i.e. the *compose* operation, when defined, is associative, *itself* acts as an identity element and *invert* computes the inverse of an isomorphism.

Choosing a Root: Finite Sequences of Natural Numbers To avoid defining $n(n-1)/2$ isomorphisms between n objects, we choose a *Root* object to/from which we will actually implement isomorphisms. We will extend our embedded combinator language using the groupoid structure of the isomorphisms to connect any two objects through isomorphisms to/from the *Root*.

Choosing a *Root* object is somewhat arbitrary, but it makes sense to pick a representation that is relatively easy convertible to various others and scalable to accommodate large objects up to the runtime system's actual memory limits.

We will choose as our *Root* object *finite sequences of natural numbers.* They can be seen as `finite functions` from an initial segment of Nat, say $[0..n]$, to Nat. We will represent them as lists i.e. their Haskell type is $[Nat]$.

```
type Nat = Integer
type Root = [Nat]
```

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

```
type Encoder a = Iso a Root
```

together with the combinators *with* and *as* providing an *embedded transformation language* for routing isomorphisms through two *Encoders.*

```
with :: Encoder a→Encoder b→Iso a b
with this that = compose this (invert that)

as :: Encoder a → Encoder b → b → a
as that this thing = to (with that this) thing
```

The combinator `with` turns two Encoders into an arbitrary isomorphism, i.e. acts as a connection hub between their domains. The combinator `as` adds a more convenient syntax such that converters between A and B can be designed as:

```
a2b x = as A B x
b2a x = as B A x
```

$$\begin{tikzcd}
A \arrow[rr, shift left, "a2b = as\ B\ A"] \arrow[dr, shift left, "a"] & & B \arrow[ll, shift left, "b2a = as\ A\ B"] \arrow[dl, shift left, "b^{-1}"] \\
& Root \arrow[ul, shift left, "a^{-1}"] \arrow[ur, shift left, "b"] &
\end{tikzcd}$$

A particularly useful combinator that transports binary operations from an Encoder to another, `borrow_from`, can be defined as follows:

```
borrow_from :: Encoder a → (a → a → a) → Encoder b → b → b → b
borrow_from other op this x y = borrow2 (with other this) op x y
```

Given that $[Nat]$ has been chosen as the root, we will define our finite function data type *fun* simply as the identity isomorphism on sequences in $[Nat]$.

```
fun :: Encoder [Nat]
fun = itself
```

Mapping Natural Numbers to Bitstrings

This isomorphism between natural numbers and bitstrings is well known, except that conventional bit representations of integers need a twist to be mapped one-to-one to *arbitrary* sequences of `0`s and `1`s. As the usual binary representation always has `1` as its highest digit, `nat2bits` will drop this bit, given that the length of the list of digits is (implicitly) known. This transformation (a variant of the so called *bijective base n* representation), defines an isomorphism between Nat and the regular language $\{0,1\}^*$.

```
bits :: Encoder [Nat]
bits = compose (Iso bits2nat nat2bits) nat

nat2bits = drop_last . (to_base 2) . succ

drop_last bs=genericTake ((genericLength bs)-1) bs

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

bits2nat bs = pred (from_base 2 (bs ++ [1]))

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