A Haskell framework for bijective data transformations

Paul Tarau

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

Abstract

We explore in this paper, in the form of a literate Haskell program a data transformation framework implemented as a *groupoid of isomorphisms* between fundamental data types.

After introducing bijective mappings between elementary data types (natural numbers, sets, multisets, finite functions, graphs, hypergraphs, etc.) we *lift* them to hereditarily finite universes derived from *ranking/unranking* and *pairing/unpairing* operations.

An embedded higher order *combinator language* provides any-to-any encodings automatically.

Applications range from encodings of SAT-problems, list operations and monadic constructs to stream iterators on combinatorial objects, self-delimiting codes, succinct data representations and generation of random instances.

The code described in the paper is available from `http://logic.cse.unt.edu/tarau/research/2009/hISO.zip`.

Keywords data type transformations, ranking/unranking, Goedel numberings computational mathematics in Haskell, hereditarily finite data types, embedded combinator language

1. Introduction

Mathematical theories often borrow proof patterns and reasoning techniques across close and sometime not so close fields.

A relatively small number of universal data types are used as basic building blocks in programming languages and their runtime interpreters, derived from a few well tested mathematical abstractions like sets, multisets, functions, graphs, etc.

Compilers convert programs from human centered to machine centered representations. Complexity classes are defined through compilation with limited resources (time or space) to similar problems [Cook 2004].

Clearly, this as part of a more general pattern: analogical / metaphorical thinking routinely shifts entities and operations from a field to another hoping to uncover similarities in representation or use [Lakoff and Johnson 1980].

One can see the "computational universe" - in analogy with its physical or biological counterparts - as being populated by a wide diversity of data types - all fundamentally *different*. Under this view, knowledge appears as accumulation of information about similarities and differences and *abstraction* (i.e. lossy but sufficient approximation) helps managing the otherwise intractable set of attributes and behaviors.

However, this view of the "computational universe" forgets a salient difference: while we have not designed the representations and laws governing the physical or biological world, we have designed their computational counterparts. And ownership has its privileges: we can freely shift the view while fixing the object of the view, if we wish!

Under this assumption, information can take the *shape* of an arbitrary data type. And one is quickly conduced to accept that the *frame of reference* is unimportant - and that any of these shapes can be seen as the essential shape of an entity, given that *isomorphisms* can faithfully shift it from one shape to another and back without loss of information.

The cognoscenti might observe that this is not very far from what Leibniz, in *La Monadologie* [Leibniz] had already expressed about . . . *Monads*:

> *Now this interconnectedness, or this accommodation of all created things to each, and of each to all the rest, means that each simple substance has relations to all the others, which it expresses. Consequently, it is a permanent living mirror of the universe.*

As we shall see, the natural framework to organize such isomorphisms is a *groupoid* i.e. a category [Mac Lane 1998] where every morphism is an isomorphism, with objects provided by the data types and morphisms provided by their bijective transformations. It is important to keep in mind that we must not lose any information if we want to be able to shift views at will. Therefore, the only *morphisms* that matter, in this ontology, are *isomorphisms*.

This brings us into familiar territory: substitution of equals by equals - *referential transparency* - an intrinsic feature of functional programming languages. And it makes Haskell the natural choice for the first iteration of such an effort - while keeping in mind that the frame of reference can be easily shifted to a different programming paradigm.

The paper is organized as follows.

Section 2 describes a method for ranking finite sequences using a single solution diophantic equation. Section 3 introduces the general framework for the paper, in the form of an embedded data transformation language, that is applied in section 4 to obtain encodings of some basic data types. Section 5 discusses a mechanism for lifting our transformations to hereditarily finite functions and sets. After reviewing some classic pairing functions, section 6 introduces pairing/unpairing operations that are used in section 7 to provide bijective encodings of graphs and hypergraphs as natural numbers. Section 8 provides an alternative foundation by implementing arithmetic operations on sequences in $\{0,1\}^*$ in subsection 8.2 and a concept of purely functional binary numbers in subsection 8.3. Section 9 describes applications with focus on combinatorial generation, random instances, succinct representations of various data types and two encodings of complete problem domains: subsection 9.1 shows various steps involved in encoding SAT problem instances while subsection 9.6 describes natural number encodings of list operations in a simple LIPS-like programming language that is extended in subsection 9.7 with an emulation of monadic constructs. Sections 10 and 11 discuss related work, future work and conclusions.

Some of the contributions of the paper worth pointing out (with section/subsection numbers in parenthesis) are the following:

- a general framework for bijective encodings between heterogeneous data types in Haskell and an embedded combinator language providing automatic any-to-any encoding by routing through a common representation (3)
- encodings of sequences, multisets and sets derived from the use of single solution diophantic equations (2)
- a mechanism for lifting encodings to hereditarily finite data types (5.1) and its application to derive Ackermann's encoding for hereditarily finite sets (5.1.1)
- two new instances of hereditarily finite representations derived from finite function and multiset encodings (5.2.1, 5.2)
- a significant number of one-to-one encoders, claimed to be new, unless specified otherwise (through various sections the paper)
- a concept of functional binary numbers and their encodings
- a bijective natural number encoding of list processing code (2,9.6) including an emulation of monadic operations (9.7)
- a presentation of our results as a self-contained literate Haskell program directly testable for technical correctness and reusable as a public domain Haskell library

2. A possible first frame of reference: a ranking/unranking algorithm for finite sequences

One has to start somewhere - and that usually comes with a set of assumptions that are genuinely *a priori* i.e. such that most people have no trouble working with them even outside formal computer science or mathematical training.

In this sense, natural numbers are probably not just the oldest computational abstraction - but also the most widely accepted one as being "built from first principles". With their usual (second order) Peano axiomatization [Kirby and Paris 1982], natural numbers are also formally well understood - if one cares. We will, therefore, use them as our first data type, providing not only a surprisingly strong structuring mechanism - but also enough expressiveness for hosting universal data and program structures.

DEFINITION 1. *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 numberings are typically *one-to-one* but not *onto* i.e. not all natural numbers correspond to valid formulae. Originating in combinatorics, *ranking/unranking* functions are *bijective* as they relate arbitrary combinatorial objects to unique natural numbers.

We start with an unusually simple (but nevertheless *new*) ranking/unranking algorithm for finite sequences of arbitrary (i.e. unbounded size!) natural numbers. Let's first observe that

PROPOSITION 1. $\forall z \in \mathbb{N} - \{0\}$ *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.

Given the definitions

```
type N = Integer
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)

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)
```

the following holds:

PROPOSITION 2. `as_nat_nats` *is a bijection from finite sequences of natural numbers to natural numbers and* `as_nats_nat` *is its inverse.*

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

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

Note also that this isomorphism preserves "list processing" operations i.e. if one defines:

```
append 0 ys = ys
append xs ys = cons (hd xs) (append (tl xs) ys)
```

then the isomorphism commutes with operations like list concatenation:

PROPOSITION 3.
(as_nats_nat n)++(as_nats_nat m) ≡ as_nats_nat (append n m)
as_nat_nats (ns++ms) ≡ append (as_nat_nats ns) (as_nat_nats ms)

One might notice at this point that `hd,tl,cons,0` define on $\mathbb{N}$ an algebraic structure isomorphic to the one introduced by `CAR,CDR,CONS,NIL` in John McCarthy's classic LISP paper [McCarthy 1960][1].

3. Connecting the dots with a groupoid of isomorphisms

DEFINITION 2. *A category in which every morphism is an isomorphism is called a* groupoid.

We represent *isomorphism* pairs like `as_nats_nat` and `as_nat_nats` as a data type `Iso`, together with the operations `compose, itself, invert` providing the (finite) *groupoid* structure.

[1] And following John McCarthy's *eval* construct one can now build relatively easily a LISP interpreter working directly and exclusively through simple arithmetic operations on natural numbers.

```
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*.

It makes sense at this point to connect everything to a *hub* type – for instance $\mathbb{N}$ – to avoid having to provide $n * (n-1)/2$ isomorphisms. We call such a connector an `Encoder`:

```
type Encoder a = Iso a N
```

We first define a trivial `Encoder`:

```
nat :: Encoder N
nat = itself
```

and then an `Encoder` from finite sequences of (unbounded) natural numbers to $\mathbb{N}$:

```
nats :: Encoder [N]
nats = Iso as_nat_nats as_nats_nat
```

It makes sense to lift the operations `as_nats_nat` and `as_nat_nats` to route a transformation to/from an arbitrary type through the common *hub* by introducing a new combinator `as`:

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

It works as follows:

```
*ISO> as nats nat 2009
[0,2,0,1,0,0,0,0]
*ISO> as nat nats [0,2,0,1,0,0,0,0]
2009
```

Note that indeed, `as nats n` generalizes `as_nats_nat` in an intuitive way:

```
*ISO> as_nats_nat 2009
[0,2,0,1,0,0,0,0]
*ISO> as_nat_nats [0,2,0,1,0,0,0,0]
2009
```

4. 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.

4.1 From sequences to 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)
```

It works as follows:

```
*ISO> as mset nat 2009
[0,2,2,3,3,3,3,3]
*ISO> as nat mset it
2009
```

One can see how this is derived by summing up from:

```
*ISO> as nats nat 2009
[0,2,0,1,0,0,0,0]
```

While finite multisets and sequences share a common representation $[Nat]$, 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.

4.2 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)
```

It works as follows:

```
*ISO> as set nat 2009
[0,3,4,6,7,8,9,10]
*ISO> as nat set it
2009
```

These two sides of this isomorphism are instance of *ranking* and *unranking* operations.

5. 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.

5.1 Hereditarily finite data types

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)
```

5.1.1 Hereditarily finite sets

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

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

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

```
*ISO> as hfs nat 42
H [H [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. 1.

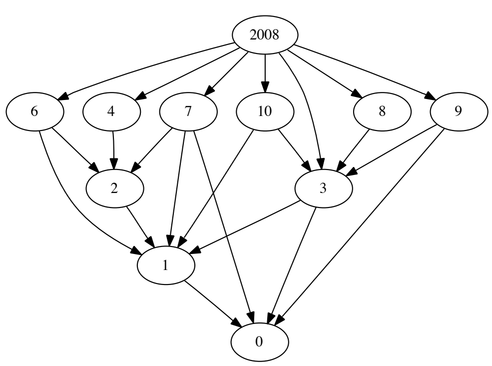

Figure 1. 2008 as a HFS

5.2 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 = lift (Iso (as mset nat) (as nat mset))
```

working as follows:

```
*ISO> as hfm nat 2008
H [H [H [],H []],H [H [],H []],
   H [H [H [H []]]],H [H [H [H []]]],
   H [H [H [H []]]],H [H [H [H []]]],H [H [H [H []]]]]
*ISO> as nat hfm it
2008
```

5.2.1 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 = lift (Iso (as nats nat) (as nat nats))
```

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:

```
*ISO> as hff nat 2008
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. 2. Note that as the mapping `as fun n` generates a sequence where the order of the edges matters, this order is indicated with integers starting from `0` labeling the edges.

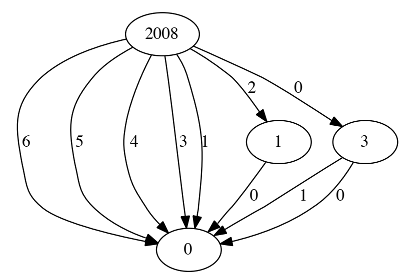

Figure 2. 2008 as a HFF

6. Pairing functions as Encoders

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

DEFINITION 3. *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 4.
$unpair : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ *is a bijection and* $pair = unpair^{-1}$.

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

```
*ISO> 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].

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

```
type N2 = (N,N)
nat2 :: Encoder N2
nat2 = Iso pair unpair
```

to obtain a pairing/unpairing isomorphism `nat2` between and $\mathbb{N}\times\mathbb{N}$ and $\mathbb{N}$. Another interesting pairing function [Pigeon 2001] can be derived from the Moser-De Bruijn sequence (A000695, [Sloane 2006])

```
inflate = (as nat set) . (map (*2)) . (as set nat)
```

as follows:

```
bpair (i,j) = inflate i + ((*2) . inflate) j
bunpair k = (deflate xs,deflate ys) where
  (xs,ys) = partition even (as set nat k)
  deflate = (as nat set) . (map (`div` 2))
```

It works as follows:

```
*ISO> map bunpair [0..7]
[(0,0),(1,0),(0,1),(1,1),(2,0),(3,0),(2,1),(3,1)]
*ISO> map bpair it
[0,1,2,3,4,5,6,7]
```

PROPOSITION 5. `bunpair` *creates a pair by separating even and odd bits of a natural number; its inverse* `bpair` *merges them back.*

Together, they provide a pairing/unpairing isomorphism `nat2'` between and $\mathbb{N}\times\mathbb{N}$ and $\mathbb{N}$:

```
nat2' :: Encoder N2
nat2' = Iso bpair bunpair
```

working as follows

```
*ISO> as nat nat2' (2008,2009)
4191170
*ISO> as nat2' nat it
(2008,2009)
```

In a way similar to hereditarily finite trees generated by applying lifting combinators, one can apply strictly decreasing[2] unpairing functions recursively. Figures 3 and 4 show the directed graphs describing recursive application of `bunpair` and `pepis_unpair`.

Given that unpairing functions are bijections from N to $N \times N$ they will progressively cover all points having natural number coordinates in their range in the plane. Figure **??** shows the curve generated by `bunpair`.

6.1 Encoding unordered pairs

To derive an encoding of unordered pairs, i.e. 2 element sets, one can combine pairing/unpairing with conversion between sequences and sets:

```
pair2unord_pair (x,y) = as set nats [x,y]
unord_pair2pair [a,b] = (x,y) where
  [x,y]=as nats set [a,b]

unord_unpair = pair2unord_pair . bunpair
unord_pair = bpair . unord_pair2pair
```

We can derive the following Encoder:

[2] except for 0 and 1, typically

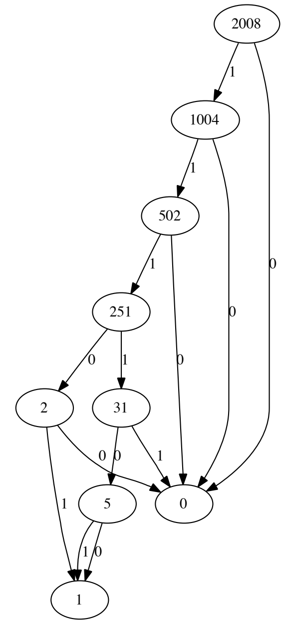

Figure 3. Graph obtained by recursive application of `unpair` for 2008

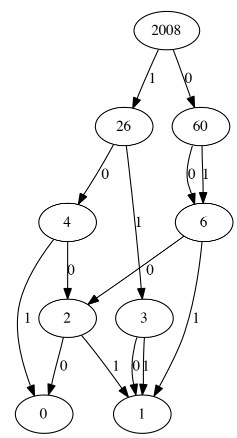

Figure 4. Graph obtained by recursive application of `bunpair` for 2008

```
set2 :: Encoder [N]
set2 = compose (Iso unord_pair unord_unpair) nat
```

working as follows:

```
*ISO> as set2 nat 2008
[60,87]
*ISO> as nat set2 [60,87]
2008
*ISO> as nat set2 [87,60]
2008
```

6.2 Encodings multiset pairs

To derive an encoding of 2 element multisets, one can combine pairing/unpairing with conversion between sequences and multisets:

```
pair2mset_pair (x,y) = (a,b) where [a,b]=as mset nats [x,y]
mset_unpair2pair (a,b) = (x,y) where [x,y] = as nats mset [a,b]
```

```
mset_unpair = pair2mset_pair . bunpair
mset_pair = bpair . mset_unpair2pair
```

We can derive the following Encoder:

```
mset2 :: Encoder N2
mset2 = compose (Iso mset_unpair2pair pair2mset_pair) nat2
```

working as follows:

```
*ISO> as mset2 nat 2009
(1,503)
*ISO> as nat mset2 it
2009
```

Figure 5 shows the curve generated by `mset_unpair` covering the lattice of points in its range.

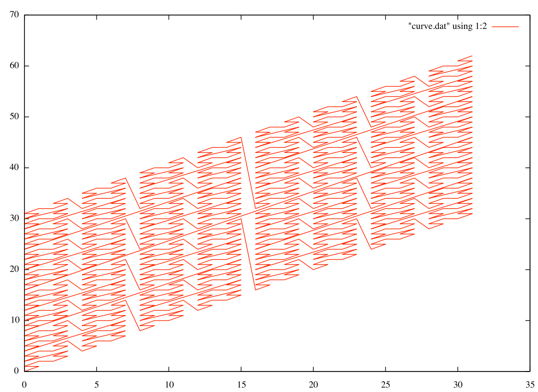

Figure 5. 2D curve connecting values of `mset_unpair n` for $n \in [0..2^{10} - 1]$

7. Encoding graphs and hypergraphs

We will now show that more complex data types like digraphs and hypergraphs have extremely simple encoders. This shows once more the importance of compositionality in the design of our embedded transformation language.

7.1 Encoding directed graphs

We can find a bijection from directed graphs (with no isolated vertices, corresponding to their view as binary relations), to finite sets by fusing their list of ordered pair representation into finite sets with a pairing function:

```
digraph2set ps = map bpair ps
set2digraph ns = map bunpair ns
```

The resulting Encoder is:

```
digraph :: Encoder [N2]
digraph = compose (Iso digraph2set set2digraph) set
```

It works as follows:

```
*ISO> as digraph nat 2008
[(1,1),(2,0),(2,1),(3,1),(0,2),(1,2),(0,3)]
*ISO> as nat digraph it
2008
```

Fig. 6 shows the digraph associated to 2008.

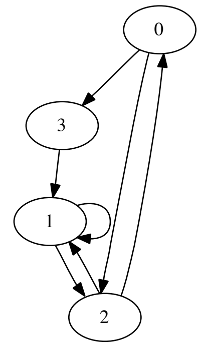

Figure 6. 2008 as a digraph

7.2 Encoding undirected graphs

We can find a bijection from undirected graphs to finite sets by fusing their list of unordered pair representation into finite sets with a pairing function on unordered pairs:

```
graph2set ps = map unord_pair ps
set2graph ns = map unord_unpair ns
```

The resulting Encoder is:

```
graph :: Encoder [[N]]
graph = compose (Iso graph2set set2graph) set
```

working as follows:

```
*ISO> as graph nat 2008
[[1,3],[2,3],[2,4],[3,5],[0,3],[1,4],[0,4]]
*ISO> as nat graph
        [[1,3],[3,2],[2,4],[5,3],[0,3],[4,1],[0,4]]
2008
```

Note that, as expected, the result is invariant to changing the order of elements in pairs like `[1,4]` and `[3,5]` to `[4,1]` and `[5,3]`.

7.3 Encoding directed multigraphs

We can find a bijection from directed multigraphs (directed graphs with multiple edges between pairs of vertices) to finite sequences by fusing their list of ordered pair representation into finite sequences with a pairing function. The resulting Encoder is:

```
mdigraph :: Encoder [N2]
mdigraph = compose (Iso digraph2set set2digraph) nats


*ISO> as mdigraph nat 2008
[(1,1),(0,0),(1,0),(0,0),(0,0),(0,0),(0,0)]
*ISO> as nat mdigraph it
2008
```

Note that the only change to the `digraph` Encoder is replacing the composition with `set` by a composition with `nats`.

7.4 Encoding undirected multigraphs

We can find a bijection from undirected multigraphs (undirected graphs with multiple edges between unordered pairs of vertices) to finite sequences by fusing their list of pair representation into finite sequences with a pairing function on unordered pairs:

The resulting Encoder is:

```
mgraph :: Encoder [[N]]
mgraph = compose (Iso graph2set set2graph) nats
```

working as follows:

```
*ISO> as mgraph nat 2008
[[1,3],[0,1],[1,2],[0,1],[0,1],[0,1],[0,1]]
*ISO> as nat mgraph it
2008
```

Note that the only change to the graph Encoder is replacing the composition with set by a composition with fun.

7.5 Encoding hypergraphs

A hypergraph (also called *set system*) is a pair $H = (X, E)$ where X is a set and E is a set of non-empty subsets of X.

We can easily derive a bijective encoding of *hypergraphs*, represented as sets of sets:

```
set2hypergraph = map (as set nat)
hypergraph2set = map (as nat set)
```

The resulting Encoder is:

```
hypergraph :: Encoder [[N]]
hypergraph = compose (Iso hypergraph2set set2hypergraph) set
```

It works as follows

```
*ISO> as hypergraph nat 2008
[[0,1],[2],[1,2],[0,1,2],[3],[0,3],[1,3]]
*ISO> as nat hypergraph it
2008
```

8. An alternative starting point: the regular language $\{0,1\}^*$

As the reader might guess - this can go on and on. In fact, one can find in [Tarau 2009] a much larger set of encodings covering more than 60 data types as diverse as BDDs, dyadic rationals, Gauss integers, DNA strands, etc.

But an important point we want to make in this paper is that the frame of reference is unimportant. In fact, the frame of reference used in [Tarau 2009] is finite functions while in this paper we picked $\mathbb{N}$ as our *hub*.

8.1 A bitstring encoder

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

```
bits :: Encoder [N]
bits = Iso as_nat_bits as_bits_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]))

to_base base n =
  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)
```

It works as follows:

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

8.2 Introducing arithmetics

We start with successor/predecessor operations:

```
s [] = [0]
s (0:xs) = 1:xs
s (1:xs) = 0:s xs

p [0] = []
p (0:xs) = 1:p xs
p (1:xs) = 0:xs
```

working as expected

```
*ISO> as nat bits (s (as bits nat 42))
43
*ISO> as nat bits (p (as bits nat 43))
42
```

A new design pattern emerges:

```
nf f = (as nat bits) . f . (as bits nat)
nf2 f x y = as nat bits (f (as bits nat x) (as bits nat y))
```

It works as follows:

```
*ISO> nf s 42
43
*ISO> nf p 43
42
```

Note that this can be seen as a functor moving operations from a datatype to another. We can further generalize it and define the *isofunctor* borrow_from:

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

It works as follows

```
*ISO> borrow_from bits nat s 42
43
*ISO> borrow_from bits nat p 43
42
```

as well as its 2-argument variant:

```
borrow_from2 lender borrower f x y =
  (as borrower lender)
  (f (as lender borrower x) (as lender borrower y))
```

We will now rebuild various arithmetic operations seen as acting on arbitrary undelimited bitstrings. We start with db and hf implementing *double* and *half* - the last one truncating toward 0:

```
db  = p . (0:)
hf = tail . s
```

Rebuilding hd (h), tl (t) and cons (c) is remarkably easy - we do not need addition or multiplication yet:

```
h :: [N]→[N]
h = h' . p

h' [] = []
h' (1:_) = []
h' (0:xs) = s (h' xs)

t :: [N]→[N]
t = t' . p
t' = hf . t''

t'' (0:xs) = t'' xs
t'' xs = xs

c x ys = s (c' x ys)
c' x xs = c'' x  (db xs)

c'' [] ys = ys
c'' xs ys = 0: c'' (p xs) ys
```

We can try them out directly on numbers using the `borrow_from` functor:

```
*ISO> borrow_from2 bits nat c 42 2009
17675748928126976
*ISO> borrow_from bits nat h 17675748928126976
42
*ISO> borrow_from bits nat t 17675748928126976
2009
```

Addition can be implemented by case analysis - i.e. treating the case when there's a carry-on 0 `sm0` and carry-on 1 `sm1` separately:

```
sm xs ys = p (sm0 xs ys)

sm0 [] [] = [0]
sm0 [] (0:ys) = 1:ys
sm0 [] (1:ys) = 0:(s ys)
sm0 (0:xs) [] = 1:xs
sm0 (1:xs) [] = 0:(s xs)
sm0 (0:xs) (0:ys) = 0:(sm0 xs ys)
sm0 (0:xs) (1:ys) = 1:(sm0 xs ys)
sm0 (1:xs) (0:ys) = 1:(sm0 xs ys)
sm0 (1:xs) (1:ys) = 0:(sm1 xs ys)

sm1 xs ys = s (sm0 xs ys)
```

Multiplication needs to handle `[]` (representing `0`) as a special case. Otherwise, it just applies the usual algorithm:

```
m [] _ = []
m _ [] = []
m xs ys = s (m1 (p xs) (p ys)) where
  m1 [] ys = ys
  m1 (0:xs) ys = 0:(m1 xs ys)
  m1 (1:xs) ys = sm0 ys (0:(m1 xs ys))
```

Both operations work as expected:

```
*ISO> sm [1,0,1,0] [0,0,1]
[0,0,0,0,0]
*ISO> map (as nat bits) [[1,0,1,0],[0,0,1]]
[20,11]
*ISO> 20+11
31
*ISO> as bits nat 31
[0,0,0,0,0]
*ISO> borrow_from2 bits nat sm 42 13
55
*ISO> borrow_from2 bits nat m 5 12
60
```

One can see that after shifting the frame of reference to the bitstring view of natural numbers we can still work with addition, multiplication etc. as usual.

8.3 Functional binary numbers

We will now make yet another shift in the frame of reference, by sketching how one can rebuild the same by designing a representation of purely functional binary numbers.

Church numerals are well known as a functional representation for Peano arithmetic. While benefiting from lazy evaluation, they implement a form of unary arithmetic that uses $O(n)$ space to represent n. This suggest devising a functional representation that mimics binary numbers. We will do this following the model described in subsection 8 to provide an isomorphism between $\mathbb{N}$ and the functional equivalent of the regular language $\{0, 1\}^*$. We will view each bit as a $\mathbb{N} \rightarrow \mathbb{N}$ transformer:

```
b x = pred x -- begin
o x = 2*x+0  -- bit 0
i x = 2*x+1  -- bit 1
e = 1        -- end
```

As the following example shows, composition of functions o and i closely parallels the corresponding bitlists:

```
*ISO> b$i$o$o$i$i$o$i$i$i$i$e
2008
*ISO> as bits nat 2008
[1,0,0,1,1,0,1,1,1,1]
```

We can follow the same model with an abstract data type:

```
data D = E | O D | I D deriving (Eq,Ord,Show,Read)
data B = B D deriving (Eq,Ord,Show,Read)
```

from which we can generate functional bitstrings as an instance of a *fold* operation:

```
funbits2nat :: B → N
funbits2nat = bfold b o i e

bfold fb fo fi fe (B d) = fb (dfold d) where
  dfold E = fe
  dfold (O x) = fo (dfold x)
  dfold (I x) = fi (dfold x)
```

Dually, we can reverse the effect of the functions b, o, i, e as:

```
b' x = succ x
o' x | even x = x `div` 2
i' x | odd x = (x-1) `div` 2
e' = 1
```

and define a generator for our data type as an *unfold* operation:

```
nat2funbits :: N → B
nat2funbits = bunfold b' o' i' e'

bunfold fb fo fi fe x = B (dunfold (fb x)) where
  dunfold n | n==fe = E
  dunfold n | even n = O (dunfold (fo n))
  dunfold n | odd n = I (dunfold (fi n))
```

The two operations form an isomorphism:

```
*ISO> funbits2nat
   (B $ I $ O $ O $ I $ I $ O $ I $ I $ I $ I $ E)
2008
*ISO> nat2funbits it
B (I (O (O (I (I (O (I (I (I (I E))))))))))
```

We can define our Encoder as follows:

```
funbits :: Encoder B
funbits = compose (Iso funbits2nat nat2funbits) nat
```

Arithmetic operations can now be performed directly on this representation. For instance, one can define a successor function as:

```
bsucc (B d) = B (dsucc d) where
  dsucc E = O E
  dsucc (O x) = I x
  dsucc (I x) = O (dsucc x)
```

Equivalently, arithmetics can be borrowed from $\mathbb{N}$:

```
*ISO> bsucc (B $ I $ O $ O $ I $ I $
           O $ I $ I $ I $ I $ E)
B (O (I (O (I (I (O (I (I (I (I E))))))))))
*ISO> as nat funbits it
2009
*ISO> borrow (with nat funbits) succ (B $ I $ O $ O $ I $
                             I $ O $ I $ I $ I $ I $ E)
B (O (I (O (I (I (O (I (I (I (I E))))))))))
*ISO> as nat funbits it
2009
```

While Haskell's C-based arbitrary length integers are likely to be more efficient for most operations, this representation, like Church numerals, has the benefit of supporting partial or delayed computations through lazy evaluation.

9. Applications

Besides their utility as a uniform basis for a general purpose data conversion library, let us point out some specific applications of our isomorphisms.

9.1 Encoding SAT problems

We will now put at use our framework in a goal driven fashion - on the practical problem of that of generating instances of a Boolean Satisfiability (SAT) problem - usable as test data for a SAT solver.

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.

9.1.1 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
```

working as follows:

```
*ISO> as set z (-42)
[0,1,4,6]
*ISO> as z set [0,1,4,6]
-42
```

9.1.2 Extending pairing/unpairing to signed integers

Given the bijection from n to 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 . bpair) (z2nat x,z2nat y)
zunpair z = (nat2z n,nat2z m) where
  (n,m)= (bunpair . z2nat) z
```

working as follows:

```
*ISO> 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)]
*ISO> map zpair it
[-5,-4,-3,-2,-1,0,1,2,3,4,5]

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

The same construction can be extended to multiset pairing functions:

```
mz2 :: Encoder Z2
mz2 = compose (Iso mzpair mzunpair) nat

mzpair (x,y) = (nat2z . mset_pair) (z2nat x,z2nat y)
mzunpair z = (nat2z n,nat2z m) where
  (n,m)= (mset_unpair . z2nat) z
```

working as follows:

```
*ISO> as mz2 z (-42)
(1,-8)
*ISO> as z mz2 it
-42
```

9.1.3 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:

```
*ISO> as sat nat 2008
[[1,-1],[2],[-1,2],[1,-1,2],[-2],[1,-2],[-1,-2]]
*ISO> 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.

9.2 Combinatorial generation

A free combinatorial generation algorithm (providing a constructive proof of recursive enumerability) for a given structure is obtained simply through an isomorphism from nat:

```
nth thing = as thing nat
nths thing = map (nth thing)
stream_of thing = nths thing [0..]

*ISO> nth set 42
[1,3,5]

*ISO> nth bits 42
[1,1,0,1,0]

*ISO> take 3 (stream_of hfs)
[H [],H [H []],H [H [H []]]]
```

9.3 Random instance generation

Combining `nth` with a random generator for natural numbers provides free algorithms for random generation of complex objects of customizable size:

```
ran thing seed largest =
  head (random_gen thing seed largest 1)

random_gen thing seed largest n = genericTake n
  (nths thing (rans seed largest))

rans seed largest =
    randomRs (0,largest) (mkStdGen seed)
```

For instance

```
*ISO> random_gen set 11 999 3
[[0,2,5],[0,5,9],[0,1,5,6]]
```

generates a list of 3 random sets.

For instance

```
*ISO>ran digraph 5 (2^31)
[(1,0),(0,1),(2,1),(1,3),(2,2),(3,2),(4,0),(4,1),
 (5,1),(6,0),(6,1),(7,1),(5,3),(6,2),(6,3)]

*ISO> ran hfs 7 30
H [H [],H [H [],H [H []]],H [H [H [H []]]]]
```

generate a random digraph and a hereditarily finite set, respectively.

Random generators for various data types are useful for further automating test generators in tools like QuickCheck [Claessen and Hughes 2002] by generating customized random tests.

An interesting other application is generating random problems or programs of a given type and size. For instance

```
*ISO> ran sat 8 (2^31)
[[-1],[1,-1],[-1,2],[1,-1,2],[-2],[1,-2],
 [-1,-2],[1,-1,-2],[2,-2],[1,2,-2],[-1,2,-2],
 [3],[1,-1,3],[1,-1,2,3],[1,-2,3],[-1,-2,3],
 [2,-2,3],[1,2,-2,3],[-1,2,-2,3]]
```

generates a random SAT-problem.

9.4 Succinct representations

Depending on the information theoretical density of various data representations as well as on the constant factors involved in various data structures, significant data compression can be achieved by choosing an alternate isomorphic representation, as shown in the following examples:

```
*ISO> as hff hfs (H [H [H []],H [H [],
        H [H []]],H [H [],H [H [H []]]]])
H [H [H []],H [H []],H [H []]]
*ISO> as nat hff (H [H [H []],H [H []],H [H []]])
42
*ISO> as nats bits [0,1,0,0,0,0,0,0,0,0,0]
[0,10]
```

In particular, mapping to efficient arbitrary length integer implementations (usually C-based libraries), can provide more compact representations or improved performance for isomorphic higher level data representations. Alternatively, lazy representations as provided by functional binary numbers for very large integers encapsulating results of some computations might turn out to be more effective space-wise or time-wise.

Using our groupoid of isomorphisms one can compare representations sharing a common datatype and make interesting conjectures about their asymptotic information density, as shown in [Tarau 2009].

9.5 Hereditarily finite data types as self-delimiting codes

We will now devise a succinct bitstring encoding of hereditarily finite data-types and apply it to derive a family of new self-delimiting codes.

9.5.1 Encoding parenthesis languages

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:

```
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]
```

This transformation maps 42 into the bitstring representation of `"((())(())(()))"` as `[0,0,0,1,1,0,0,1,1,0,0,1,1,1]`.

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

Like the Elias omega code [Elias 1975], a balanced parenthesis representation is obviously *self-delimiting* as proven by the fact that the reader `pars_expr` defined in section 9.5.1 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 6. *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, the optimal undelimited bitstring encoding provided by the Encoder `bits` (defined in subsection 8)

```
*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 that maps $\mathbb{N}$ 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. We refer to [Tarau 2009] for a comparison between this self-delimiting code and Elias omega code.

9.6 Sketch of a fully arithmetized functional programming language

We have briefly shown that shifting views over data types is quite easy - and we refer to [Tarau 2009] for other such isomorphisms between about 60 different data types. We will now focus on programming constructs. We can start with some simple list operations

```
lst nat = cons nat 0

len 0 = 0
len xs = succ (len (tl xs))
```

working as follows

```
*ISO> map lst [0..7]
[1,2,4,8,16,32,64,128]
*ISO> as nats nat 42
[1,1,1]
*ISO> len 42
3
```

Some well known higher order functions (from the Haskell prelude) are next with names prefixed with n to avoid conflicts, when needed:

```
nzipWith _ 0 _ = 0
nzipWith _ _ 0 = 0
nzipWith f xs ys =
  cons (f (hd xs) (hd ys))
       (nzipWith f (tl xs) (tl ys))

nzip xs ys = nzipWith cons xs ys

nunzip zs = (xs,ys) where
  xs = nMap hd zs
  ys = nMap tl zs
```

As the following definitions and examples show, more complex data types like associative lists are easily defined "inside" a natural number:

```
getAssoc _ 0 = 0
getAssoc k ps =
  if 0==xy then 0
  else if k== x then y
  else getAssoc k (tl ps) where
    xy=hd ps
    x=hd xy
    y=tl xy

addAssoc k v ps=cons (cons k v) ps
```

```
*ISO> as nat nats [0,1,2]
37
*ISO> as nat nats [3,4,5]
16648
*ISO> nzip 37 16648
2361183241434889715840
*ISO> getAssoc 2 it
5
*ISO> addAssoc 2 1 (addAssoc 1 2 0)
8392704
*ISO> getAssoc 1 8392704
2
*ISO> getAssoc 2 8392704
1
*ISO> getAssoc 3 8392704
0
```

The `fold` family has been shown to be able to express a large class of interesting other functions as shown in [Hutton 1999]. Various fold operations are defined along the lines of their Haskell prelude counterparts:

```
nfoldl _ z 0    =  z
nfoldl f z xs =  nfoldl f (f z (hd xs)) (tl xs)

nfoldr f z 0     =  z
nfoldr f z xs =  f (hd xs) (nfoldr f z (tl xs))

nscanl _ q 0 = lst q
nscanl f q xs = cons q (nscanl f (f q (hd xs)) (tl xs))
```

They works as follows:

```
*ISO> nfoldl (+) 0  8466
10
*ISO> as nats nat (nscanl (+) 0  8466)
[0,1,3,6,10]
```

We can now define the equivalent of `reverse`, `map` and `concat`

```
rev = nfoldl (flip cons) 0

nMap f ns = nfoldr (λx xs→cons (f x) xs) 0 ns
```

```
nconcat xss = nfoldr append 0 xss

nconcatMap f xs = nconcat (nMap f xs)
```

working as follows

```
*ISO> as nat nats [1,2,3]
274
*ISO> rev 274
328
*ISO> as nats nat 328
[3,2,1]
*ISO> nMap (λx→x^2+1) 274
524548
*ISO> as nats nat 524548
[2,5,10]
```

Following [McCarthy 1960] one can build a fully arithmetized theory of recursive functions together with a Turing-equivalent `eval` predicate along the lines of [Chaitin 1975].

9.7 A final touch: emulating monadic constructs

We can now emulate monadic constructs (except that we have $\mathbb{N}$ instead of the parametric type `Monad a`) as follows:

```
infixl 1  >>-
m >>- k = nconcatMap k m
nreturn = lst

nsequence = foldr mcons (nreturn 0) where
  mcons p q = p>>- λx → q >>- λy →nreturn (cons x y)
```

The monadic list operation:

```
*ISO>  [0,1] >>= λx→[0,1] >>= λy→ return (x,y)
[(0,0),(0,1),(1,0),(1,1)]
```

is now emulated as:

```
*ISO> as nat nats [0,1]
5
*ISO> 5 >>- λx→ 5 >>- λy→ nreturn (cons x y)
33058
```

which, after shifting to a list view (and then to a list of pairs view) becomes:

```
*ISO> as nats nat 33058
[1,3,2,6]
*ISO> map (λx→(hd x,tl x)) [1,3,2,6]
[(0,0),(0,1),(1,0),(1,1)]
```

Similarly, `nsequence` emulates Haskell's `sequence` construct as follows:

```
*ISO> nsequence [1,1,2]
2048
*ISO> map (as nats nat) [1,1,2]
[[0],[0],[1]]
*ISO> sequence it
[[0,0,1]]
*ISO> as nats nat 2048
[11]
*ISO> as nats nat 11
[0,0,1]
```

10. Related work

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. While our `Iso` data type is similar to the *Bij* data type in [Conal Elliott] and BiArrow concept of [Alimarine et al. 2005], the techniques for using such isomorphisms as building blocks of an embedded composition language centered around encodings as natural numbers are new.

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]. However the generic view of such transformations as lifting combinators obtained compositionally from simpler isomorphisms, as described in this paper, is new. Note also that Gödel numberings are typically injective but not *onto* applications, and can only be turned into bijections by exhaustive enumeration of their range. By contrast our ranking/unranking functions are designed to be "genuinely" bijective, usually with computational effort linear in the size of the data types.

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].

Natural number encodings of hereditarily finite sets have triggered the interest of researchers in fields ranging from Axiomatic Set Theory and Foundations of Logic to Complexity Theory and Combinatorics [Takahashi 1976, Kaye and Wong 2007, Kirby 2007]. Contrary to the well known hereditarily finite sets, the concepts of hereditarily finite multisets and functions as well as their encodings, are likely to be new, given that our sustained search efforts have not led so far to anything similar.

11. Conclusion

We have described encodings for various data types in a uniform framework as data type isomorphisms with a groupoid structure. The framework has been extended with lifting combinators providing generic mechanisms for encoding hereditarily finite sets, multisets and functions. In addition, by using pairing/unpairing functions we have also derived unusually simple encodings for graphs, digraphs and hypergraphs.

While we have focused on the connected groupoid providing isomorphisms to/from natural numbers, similar techniques can be used to organize bijective transformations in fields ranging from compilation and complexity theory to data compression and cryptography.

As a practical goal, we hope this work can facilitate the refactoring of the enormous ontology exhibited by various computer science and engineering fields, that have resulted over a relatively short period of evolution in unnecessarily steep learning curves limiting communication and synergy between fields.

We refer to [Tarau 2009] for implementations of a number of other encoders as well as various applications.

References

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.

Gregory J. Chaitin. A theory of program size formally identical to information theory. *J. Assoc. Comput. Mach*, 22:329–340, 1975.

Koen Claessen and John Hughes. Testing monadic code with quickcheck. *SIGPLAN Notices*, 37(12):47–59, 2002.

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

Stephen Cook. Theories for complexity classes and their propositional translations. In *Complexity of computations and proofs*, pages 1–36, 2004.

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

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

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`.

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

Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.*, 9(4):355–372, 1999.

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.

L. Kirby and J. Paris. Accessible independence results for peano arithmetic. *Bull. London. Math. Soc.*, (14):285–293, 1982.

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.

George Lakoff and Mark Johnson. *Metaphors We Live By*. University of Chicago Press, Chicago, IL, USA, 1980.

Gottfried Wilhelm Leibniz. La Monadologie. URL `http://www.philosophy.leeds.ac.uk/GMR/hmp/texts/modern/leibniz/monadology/monadology.html`. English translation by George MacDonald Ross, 1999.

Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, New York, NY, USA, 1998.

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.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367177.367199.

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.

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.