A Groupoid of Isomorphic Data Transformations

Paul Tarau

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

Abstract. As a variation on the known theme of Gödel numberings, isomorphisms defining data type transformations in a strongly typed functional language are organized as a finite groupoid using a higher order combinator language that unifies popular data types as diverse as natural numbers, finite sequences, digraphs, hypergraphs and finite permutations with more exotic ones like hereditarily finite functions, sets and permutations. [1]
Keywords: *computational mathematics in Haskell, data type transformations, ranking/unranking,Gödel numberings, higher order combinators, hylomorphisms*

1 Introduction

Analogical/metaphorical thinking routinely shifts entities and operations from a field to another hoping to uncover similarities in representation or use [1]. Compilers convert programs from human centered to machine centered representations - sometime reversibly. Complexity classes are defined through compilation with limited resources (time or space) to similar problems [2]. 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, corresponding to a few well tested mathematical abstractions like sets, functions, graphs, groups, categories etc.

In their simplest form, isomorphisms between data types show up as *encodings* to some canonical representation, for instance natural numbers. Such encodings can be traced back to Gödel numberings [3, 4] associated to formulae, but a wide diversity of common computer operations, ranging from data compression and serialization to wireless data transmissions and cryptographic codes are indirectly related.

We will show in this paper how such isomorphisms can be organized naturally as a finite groupoid *i.e. a category [5] where every morphism is an isomorphism, with objects provided by the data types and morphisms provided by their transformations.*

[1] A (very) long version of this paper is available at `http://arXiv.org/abs/0808.2953`. Like this paper, it is organized as a literate Haskell program while also including Haskell sources as a separate file.

One can see these encodings as a first step towards a "theory of everything" meant to provide a uniform view of the basic building blocks of various computational artifacts. We hope this can help refactoring the enormous ontology exhibited by 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.

The paper is organized as follows: section 2 describes our data transformation framework, section 3 introduces isomorphisms between finite sequences, sets and natural numbers and section 4 shows their lifting to hereditarily finite structures. Ranking/unranking of permutations and hereditarily finite permutations as well as Lehmer codes and factoradics are covered in section 5. Section 6 describes encodings for digraphs and hypergraphs. Sections 7 and 8 discuss related work and conclusions.

2 An Embedded Data Transformation Language

We will start by designing an embedded transformation language 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.

2.1 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 and the *to* function the second component defining the isomorphism.

$$X \underset{g = f^{-1}}{\overset{f = g^{-1}}{\rightleftarrows}} Y$$

```
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`, $f \circ g = id_b$ and $g \circ f = id_a$, we have:

Proposition 1 *The data type* `Iso` *defines a* groupoid*, i.e. for all cases when it is defined, the* compose *operation is associative,* itself *acts as an identity element and* invert *computes the inverse of an isomorphism.*

We can see the combinators `from, to, compose, itself, invert` as part of an *embedded data transformation language*. In the general case, as composition is only a partial function (i.e. $f : A \to B$ can be composed with $g : B' \to C$ only if $B = B'$), the resulting finite groupoid can be seen as a disjoint union of connected categories corresponding to the *weakly connected components* of the underlying graph.

Choosing a Root in a connected groupoid Within each connected groupoid, to avoid defining $n(n-1)/2$ isomorphisms between n objects, we can choose a *Root* object to/from which we will actually implement isomorphisms. Then we can extend our embedded combinator language using the groupoid structure of the isomorphisms to connect *any* two objects through isomorphisms to/from the *Root*.

2.2 The Gödel groupoid

Let us, from now on, focus on the connected groupoid of isomorphisms mapping various data types to *natural numbers* (Nat). It makes sense to call it the *Gödel groupoid* as traditionally such mappings have been investigated in his work on arithmetization of formulae in the proofs of his incompleteness theorems.

Within each connected groupoid, choosing a *Root* object is somewhat arbitrary, but it makes sense to pick a representation that is relatively easily convertible to various others, efficiently implementable and, last but not least, scalable to accommodate large objects up to the runtime system's actual memory limits.

With this in mind, instead of the obvious choice Nat, let us chose as our *Root* object for the Gödel groupoid as the set of *Finite Sequences of Natural Numbers*. They can also be seen as as finite functions from an initial segment of Nat, say $[0..n]$, to Nat, or as words on an alphabet with an infinite supply of symbols. We will represent them as lists i.e. their Haskell type is $[Nat]$. As we will show in subsection 3.1 such sequences will be mapped one to one to Nat while accommodating large objects more efficiently.

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

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

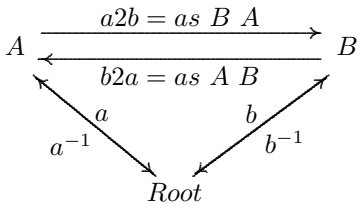
```
type Encoder a = Iso a Root
```

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.



We will provide use cases for these combinators as we populate our groupoid of isomorphisms. 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
```

3 Extending the Groupoid of Isomorphisms

We will now populate our groupoid of isomorphisms with combinators based on a few primitive encoders.

3.1 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. 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)
```

Proposition 2 `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
```

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
```

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

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

then the isomorphism commutes with operations like list concatenation:

Proposition 3 *(as fun nat n) ++ (as fun nat m) ≡ as fun nat (app n m)*
as nat fun (ns ++ ms) ≡ app (as nat fun ns) (as nat fun ms)

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 Nat such that $unpair\ 0 = (0, 0)$ i.e.

Proposition 4 $unpair : Nat \to Nat \times Nat$ *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 turns out to be 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 [6–8] and `hd,tl,cons,0` define on Nat an algebraic structure isomorphic to the one introduced by `CAR,CDR,CONS,NIL` in John McCarthy's classic LISP paper [9].

With the isomorphism defined by `pair` and `unpair` we obtain the `Encoder`:

```
type Nat2 = (Nat,Nat)
nat2 :: Encoder Nat2
nat2 = compose (Iso pair unpair) nat
```

working as follows:

```
*ISO> as nat2 nat 123
(2,15)
*ISO> as nat nat2 (2,15)
123
```

3.2 An Isomorphism to Finite Sets of Natural Numbers

We can *rank* a set represented as a list of distinct natural numbers by mapping it into a single natural number, and, reversibly, by observing that it can be seen as the list of exponents of `2` in the number's base `2` representation. We obtain the `Encoder`:

```
set :: Encoder [Nat]
set = compose (Iso set2nat nat2set) nat

nat2set n | n≥0 = nat2exps n 0 where
  nat2exps 0 _ = []
  nat2exps n x = if (even n) then xs else (x:xs) where
    xs=nat2exps (n `div` 2) (x+1)

set2nat ns = sum (map (2^) ns)
```

Note that in this case sets are sharing with sequences the underlying list representation `[Nat]`. The injection between sets represented by ordered sequences of distinct numbers and arbitrary sequences requires implementing a predicate `is_set` (see [10]) to enforce such constraint on each set argument. To keep our code simpler, we will assume in this paper that such constraints implicitly hold when required.

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 Pure Hereditarily Finite Data Types

The unranking operation is seen here as an instance of a generic *anamorphism* (an *unfold* operation), while the ranking operation is seen as an instance of the corresponding catamorphism (a *fold* operation) [11]. 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 tree data type and natural numbers.

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

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

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

```
unrank :: (a → [a]) → a → T
unranks :: (a → [a]) → [a] → Ts

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

rank :: ([b] → b) → T → b
ranks :: ([b] → b) → Ts → [b]

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 propagates a simpler operation guided by the structure of the data type. 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 Ts [b]
hylos (Iso f g) = Iso (ranks g) (unranks f)
```

As its most general type shows, `hylo` lifts an isomorphism from `b` to `[b]` to an isomorphism between trees of type `T` and `b`. In our case `b` is `Nat` but the mechanism is more general - for instance it would also work if `b` is instantiated to Church numerals or bitstrings instead of `Nat`.

Hereditarily Finite Sets Hereditarily Finite Sets [12] will be represented as an Encoder for the tree type `T`:

```
hfs :: Encoder T
hfs = compose (hylo (Iso nat2set set2nat)) nat
```

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

```
*ISO> as hfs 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 []]]],H [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 [13] 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
```

Hereditarily Finite Functions The same tree data type can host a hylomorphism derived from finite functions instead of finite sets:

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

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

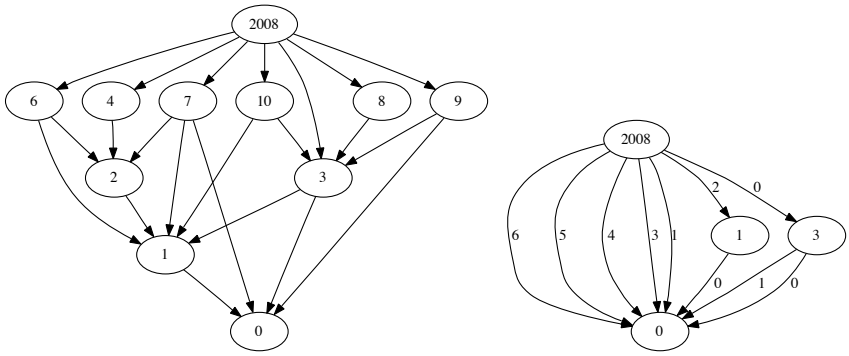
```
*ISO> as hff nat 2008
H [H [H [],H []],H [],H [H []],H [],H [],H [],H []]
```

that can be also expressed as `(((())())(())()()())` using a parenthesis language [10].

One can represent the action of a hylomorphism unfolding 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 (a)**. Similarly, a hereditarily finite function is expressed as a directed ordered multi-graph as shown in **Fig. 1 (b)**. Note that in this case 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.

5 Permutations and Hereditarily Finite Permutations

We have seen that finite sets and their derivatives represent information in an *order* independent way, focusing exclusively on information *content*. We will now look at data representations that focus exclusively on *order* in a *content* independent way - finite permutations and their hereditarily finite derivatives. To obtain an encoding for finite permutations we will first review a ranking/unranking mechanism for permutations that involves an unconventional numeric representation, *factoradics*.



(a) as a HFS (b) as a HFF

Fig. 1: Hereditarily finite representations of 2008

5.1 The Factoradic Numeral System

The factoradic numeral system [14] replaces digits multiplied by a power of a base n with digits that multiply successive values of the factorial of n. In the increasing order variant `fr` the first digit d_0 is 0, the second is $d_1 \in \{0, 1\}$ and the n-th is $d_n \in [0..n]$. For instance, $42 = 0 * 0! + 0 * 1! + 0 * 2! + 3 * 3! + 1 * 4!$. The left-to-right, decreasing order variant `fl` is obtained by reversing the digits of `fr`.

```
fr 42
  [0,0,0,3,1]
rf [0,0,0,3,1]
  42
fl 42
  [1,3,0,0,0]
lf [1,3,0,0,0]
  42
```

The function `fr` generating the factoradics of n, right to left, handles the special case of 0 and calls a local function `f` which recurses and divides with increasing values of n while collecting digits with `mod`:

```
fr 0 = [0]
fr n = f 1 n where
   f _ 0 = []
   f j k = (k `mod` j) :
           (f (j+1) (k `div` j))
```

The function `fl`, with digits left to right is obtained as follows:

```
fl = reverse . fr
```

The function `lf` (inverse of `fl`) converts back to decimals by summing up results while computing the factorial progressively:

```
rf ns = sum (zipWith (*) ns factorials) where
  factorials=scanl (*) 1 [1..]
```

Finally, `lf`, the inverse of `fl` is obtained as:

```
lf = rf . reverse
```

5.2 Ranking and unranking permutations of given size with Lehmer codes and factoradics

The Lehmer code of a permutation f of size n is defined as the sequence $l(f) = (l_1(f) \dots l_i(f) \dots l_n(f))$ where $l_i(f)$ is the number of elements of the set $\{j > i | f(j) < f(i)\}$ [15].

Proposition 5 *The Lehmer code of a permutation determines the permutation uniquely.*

The function `perm2nth` computes a `rank` for a permutation `ps` of `size>0`. It starts by first computing its Lehmer code `ls` with `perm2lehmer`. Then it associates a unique natural number `n` to `ls`, by converting it with the function `lf` from factoradics to decimals. Note that the Lehmer code `ls` is used as the list of digits in the factoradic representation.

```
perm2nth ps = (l,lf ls) where
  ls=perm2lehmer ps
  l=genericLength ls

perm2lehmer [] = []
perm2lehmer (i:is) = l:(perm2lehmer is) where
  l=genericLength [j|j←is,j<i]
```

The function `nat2perm` provides the matching *unranking* operation associating a permutation `ps` to a given `size>0` and a natural number `n`. It generates the n-th permutation of a given size.

```
nth2perm (size,n) =
  apply_lehmer2perm (zs++xs) [0..size-1] where
    xs=fl n
    l=genericLength xs
    k=size-l
    zs=genericReplicate k 0
```

The following function extracts a permutation from a "digit" list in factoradic representation.

```
apply_lehmer2perm [] [] = []
apply_lehmer2perm (n:ns) ps@(x:xs) =
   y : (apply_lehmer2perm ns ys) where
   (y,ys) = pick n ps

pick i xs = (x,ys++zs) where
  (ys,(x:zs)) = genericSplitAt i xs
```

Note also that `apply_lehmer2perm` is used this time to reconstruct the permutation `ps` from its Lehmer code, which in turn is computed from the permutation's factoradic representation.

One can try out this bijective mapping as follows:

```
*ISO> nth2perm (5,42)
  [1,4,0,2,3]
*ISO> perm2nth [1,4,0,2,3]
  (5,42)
*ISO> nth2perm (8,2008)
  [0,3,6,5,4,7,1,2]
*ISO> perm2nth [0,3,6,5,4,7,1,2]
  (8,2008)
```

5.3 A bijective mapping from permutations to natural numbers

One more step is needed to to extend the mapping between permutations of a given length to a bijective mapping from/to Nat: we will have to "shift towards infinity" the starting point of each new block of permutations in Nat as permutations of larger and larger sizes are enumerated.

First, we need to know by how much - so we compute the sum of all factorials up to $n!$.

```
sf n = rf (genericReplicate n 1)
```

This is done by noticing that the factoradic representation of [0,1,1,..] does just that.

To know by how much we have to shift our mapping, we want to decompose `n` into the distance to the last sum of factorials smaller than `n`, `n_m` and the index in the sum, `k`.

```
to_sf n = (k,n-m) where
  k=pred (head [x|x←[0..],sf x>n])
  m=sf k
```

Unranking of an arbitrary permutation is now easy - the index `k` determines the size of the permutation and `n-m` determines the rank. Together they select the right permutation with `nth2perm`.

```
nat2perm 0 = []
nat2perm n = nth2perm (to_sf n)
```

Ranking of a permutation is even easier: we first compute its size and its rank, then we shift the rank by the sum of all factorials up to its size, enumerating the ranks previously assigned.

```
perm2nat ps = (sf l)+k where
  (l,k) = perm2nth ps
```

It works as follows:

```
*ISO> nat2perm 42
  [0,2,3,1,4]
*ISO> perm2nat [0,2,3,1,4]
  42
*ISO> nat2perm 2008
  [1,4,3,2,0,5,6]
*ISO> perm2nat [1,4,3,2,0,5,6]
  2008
```

We can now define the Encoder as:

```
perm :: Encoder [Nat]
perm = compose (Iso perm2nat nat2perm) nat
```

The Encoder works as follows:

```
*ISO> as perm nat 2008
[1,4,3,2,0,5,6]
*ISO> as nat perm it
2008
*ISO> as perm nat 1234567890
[1,6,11,2,0,3,10,7,8,5,9,4,12]
*ISO> as nat perm it
1234567890
```

5.4 Hereditarily Finite Permutations

By using the generic `unrank` and `rank` functions defined in section 4 we can extend the isomorphism defined by `nat2perm` and `perm2nat` to encodings of Hereditarily Finite Permutations (HFP).

```
nat2hfp = unrank nat2perm
hfp2nat = rank perm2nat
```

The encoding works as follows:

```
*ISO> nat2hfp 42
H [H [],H [H [],H [H []]],H [H [H []],H []],
   H [H []],H [H [],H [H []],H [H [],H [H []]]]]
*ISO> hfp2nat it
  42
```

We can now define the Encoder as:

```
hfp :: Encoder T
hfp = compose (Iso hfp2nat nat2hfp) nat
```

The Encoder works as follows:

```
*ISO> as hfp nat 42
H [H [],H [H [],H [H []]],H [H [H []],H []],
   H [H []],H [H [],H [H []],H [H [],H [H []]]]]
*ISO> as nat hfp it
42
```

6 Encoding Directed Graphs and Hypergraphs

We will now show that more complex data types like digraphs and hypergraphs have extremely simple encoders.

6.1 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 pair ps
set2digraph ns = map unpair ns
```

The resulting Encoder is:

```
digraph :: Encoder [Nat2]
digraph = compose (Iso digraph2set set2digraph) set
```

working as follows:

```
*ISO> as digraph' nat 2009
[(0,0),(2,0),(0,2),(0,3),(3,0),(0,4),(1,2),(0,5)]
*ISO> as nat digraph it
2009
```

Fig. 2 shows the digraph associated to 2009.

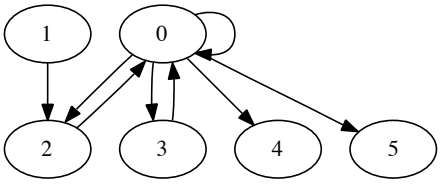


Fig. 2: 2009 as a digraph

6.2 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 derive a bijective encoding of *hypergraphs*, represented as sets of non-empty sets:

```
set2hypergraph = map (nat2set . succ)
hypergraph2set = map (pred . set2nat)
```

The resulting Encoder is:

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

working as follows

```
*ISO> as hypergraph nat 2009
[[0],[2],[0,2],[0,1,2],[3],[0,3],[1,3],[0,1,3]]
*ISO> as nat hypergraph it
2009
```

Discussion. For digraphs understood as subsets of $N \times N$, i.e. provided that a canonical mapping of vertices to an initial segment of N is assumed, our representations are clearly bijections. Once the mapping is fixed, a digraph is seen as a list of edges, each mapped to a distinct natural number using a pairing function. As all edges are distinct, the resulting list represents a set - which then is mapped to a unique natural number. Note also that digraphs can be disconnected and isolated vertices can be represented simply as vertices not occurring in the list of edges, assuming that the canonical mapping to vertices is such that the last vertex is connected to at least one other vertex. Under these assumptions, "digraphs" consisting only of isolated vertices collapse to the same encoding as the empty digraph. To avoid this problem, one can pair the representation of a digraph with a number indicating how many such vertices are considered part of the digraph after the last connected vertex occurring in an edge.

Similar reasoning applies to hypergraphs which are represented as lists of (distinct) hyperedges - i.e. sets of natural numbers that are then mapped to a unique natural number.

7 Related work

The closest reference on encapsulating bijections as a Haskell data type is [16] and Conal Elliott's composable bijections module [17], where, in a more complex setting, Arrows [18] are used as the underlying abstractions. While our `Iso` data type is similar to the *Bij* data type in [17] and BiArrow concept of [16], 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 [3, 4] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [19–21]. However the generic view of such transformations as hylomorphisms 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 [8]. A typical use in the foundations of mathematics is [22]. An extensive study

of various pairing functions and their computational properties is presented in [23].

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 [12, 24, 25]. Contrary to the well known hereditarily finite sets, the concepts of hereditarily finite functions and permutations as well as their encodings, are likely to be new, given that our sustained search efforts have not lead so far to anything similar.

8 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 hylomorphisms providing generic mechanisms for encoding hereditarily finite sets, functions and permutations. 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 Gödel groupoid providing isomorphisms to/from natural numbers and finite sequences of natural numbers, similar techniques can be used to organize bijective transformations in fields ranging from compilation and complexity theory to data compression and cryptography.

We refer to [10] for implementations of a number of other encoders, covering data types as diverse as functional binary numbers, BDDs, multigraphs, parenthesis languages, multisets, primes, Gauss integers, as well as applications ranging from succinct encodings and generation of random instances of complex data types to experiments in number theory, boolean logic and circuit minimization.

Acknowledgements

The author thanks the anonymous referees of CALCULEMUS 2009 for their helpful suggestions and comments.

References

1. Lakoff, G., Johnson, M.: Metaphors We Live By. University of Chicago Press, Chicago, IL, USA (1980)
2. Cook, S.: Theories for complexity classes and their propositional translations. In: Complexity of computations and proofs. (2004) 1–36
3. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
4. Hartmanis, J., Baker, T.P.: On Simple Goedel Numberings and Translations. In Loeckx, J., ed.: ICALP. Volume 14 of Lecture Notes in Computer Science., Berlin Heidelberg, Springer (1974) 301–316

5. Mac Lane, S.: Categories for the Working Mathematician. Springer-Verlag, New York, NY, USA (1998)
6. Pepis, J.: Ein verfahren der mathematischen logik. The Journal of Symbolic Logic **3**(2) (jun 1938) 61–76
7. 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
8. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
9. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM **3**(4) (1960) 184–195
10. Tarau, P.: Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell (January 2009) Unpublished draft, http://arXiv.org/abs/0808.2953, updated version at http://logic.cse.unt.edu/tarau/research/2010/ISO.pdf, 150 pages.
11. Meijer, E., Hutton, G.: Bananas in Space: Extending Fold and Unfold to Exponential Types. In: FPCA. (1995) 324–333
12. Takahashi, M.o.: A Foundation of Finite Mathematics. Publ. Res. Inst. Math. Sci. **12**(3) (1976) 577–708
13. Ackermann, W.F.: Die Widerspruchsfreiheit der allgemeinen Mengenlhere. Mathematische Annalen (114) (1937) 305–315
14. Knuth, D.E.: The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
15. Mantaci, R., Rakotondrajao, F.: A permutations representation that knows what "eulerian" means. Discrete Mathematics & Theoretical Computer Science **4**(2) (2001) 101–108
16. 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
17. Conal Elliott: Module: Data.Bijections Haskell source code library at: http://haskell.org/haskellwiki/TypeCompose.
18. Hughes, J.: Generalizing Monads to Arrows Science of Computer Programming 37, pp. 67-111, May 2000.
19. 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., Berlin Heidelberg, Springer (2003) 572–581
20. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
21. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters **79** (2001) 281–284
22. 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
23. Rosenberg, A.L.: Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science **14**(1) (2003) 3–17
24. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume **48**(4) (2007) 497–510
25. Kirby, L.: Addition and multiplication of sets. Math. Log. Q. **53**(1) (2007) 52–65