"Everything Is Everything" Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
Denton, TX 76203-6886, USA

This paper is an exploration of isomorphisms between elementary data types (e.g., natural numbers, sets, finite functions, graphs, hypergraphs) and their extension to hereditarily finite universes through hylomorphisms derived from ranking/unranking and pairing/unpairing operations. An embedded higher order combinator language provides any-to-any encodings automatically. A few examples of free algorithms obtained by transferring operations between data types are shown. Other applications range from stream iterators on combinatorial objects to succinct data representations and the generation of random instances.

1. Introduction

Kolmogorov-Chaitin algorithmic complexity is based on the existence of various equivalent representations of data objects, and in particular (minimal) programs that produce them in a given language and encoding [1-3]. Analogical/metaphorical thinking routinely shifts entities and operations from one field to another hoping to uncover similarities in representation or use [4]. 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 [5, 6]. Mathematical theories often borrow proof patterns and reasoning techniques across close and some time 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 well-tested mathematical abstractions like sets, functions, graphs, groups, categories, and others.

Hence, everyone explicitly or implicitly knows that ultimately, "everything is everything" through lower common denominators like bitstring representations in computer memory. From hackers and compiler writers to combinatorialists and experimental mathematicians, it is not uncommon to shapeshift between various data types. Means as

https://doi.org/10.25088/ComplexSystems.18.4.475

simple as binary editors, union types, or overlapping variable definitions are generously providing such alternate views.

A less obvious leap is that if heterogeneous objects can be seen in some way as isomorphic, then we can share them and compress the underlying informational universe by collapsing isomorphic encodings of data or programs whenever possible.

The added benefit of these “shapeshifting” data types is that the functors transporting their data content will also transport their operations, resulting in shortcuts that provide, for free, implementations of interesting algorithms. The simplest instance is the case of isomorphisms, which are reversible mappings that also transport operations. Typically, such isomorphisms show up as *encodings* to some simpler and easier to manipulate representation, for instance, natural numbers, finite sets, or finite sequences. Such encodings can be traced back to Gödel numberings [7, 8] associated to formulas, but a wide diversity of common computer operations, ranging from wireless data transmissions to cryptographic codes, qualify.

Encodings between data types provide a variety of services, ranging from free iterators and random objects to data compression and succinct representations. Tasks like serialization and persistence are facilitated by simplification of reading or writing operations without the need of special purpose parsers. Sensitivity to internal data representation format or size limitations can be circumvented without extra programming effort.

In the context of algorithmic information theory, data structures can be interpreted like graphs and program constructs like loops or recursion as compression mechanisms that focus on sharing and reusing equivalent blocks of information. In this case, maximal sharing acts as the dual of minimal program+input size. With this in mind, shapeshifting through a uniform set of encodings would extend sharing opportunities across heterogeneous data and code types.

This paper is organized as follows. Section 2 introduces the general framework for the paper, in the form of an embedded data transformation language, that is applied in Section 3 to obtain encodings of some basic data types. Section 4 discusses a mechanism for lifting our transformations to hereditarily finite functions and sets. After reviewing some classic pairing functions, Section 5 introduces pairing/unpairing operations that are used in Section 7 to provide bijective encodings of digraphs, directed acyclic graphs (DAGs), and hypergraphs as natural numbers. Section 8 describes applications that focus on combinatorial generation, experimental mathematics, random instances, and succinct representations of various data types. Sections 9 and 10 discuss related work, future work, and conclusions.

Here are the main contributions of this paper (with section/subsection numbers in parenthesis):

- 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 (2)

- efficient and simple encodings of finite sequences (3.2), digraphs, DAGs, and hypergraphs (7)
- a mechanism for lifting encodings to hereditarily finite data types (4) and its application to derive Ackermann's encoding for hereditarily finite sets
- a new instance of hereditarily finite representations derived from finite function encodings (4.1.2)
- a bijective natural number encoding of list processing code (6)
- 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. An Embedded Data Transformation Language

We start by designing an embedded transformation language as a set of operations on a groupoid of isomorphisms. We then extend it with higher order combinators mediating the composition of 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 (known as 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:

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

```
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 a b → Iso b a
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 isomorphisms that can be used to test the correctness of implementations with tools like QuickCheck [9].

Proposition 1. The data type `Iso` has a groupoid structure, that is, the `compose` operation, when it is defined, is associative, `itself` acts as an identity element, and `invert` computes the inverse of an isomorphism.

We can transport operations from one object to another with `borrow` and `lend` combinators defined as follows.

```
borrow :: Iso t s → (t → t) → s → s
borrow (Iso f g) h x = f (h (g x))
borrow2 (Iso f g) h x y = f (h (g x) (g y))
borrowN (Iso f g) h xs = f (h (map g xs))

lend :: Iso s t → (t → t) → s → s
lend = borrow . invert
lend2 = borrow2 . invert
lendN = borrowN . invert
```

The combinators `fit` and `retrofit` just transport an object x through an isomorphism and apply to it an operation `op` available on the other side.

```
fit :: (b → c) → Iso a b → a → c
fit op iso x = op ((from iso) x)

retrofit :: (a → c) → Iso a b → b → c
retrofit op iso x = op ((to iso) x)
```

We can see the combinators `from`, `to`, `compose`, `itself`, `invert`, `borrow`, `lend`, `fit`, and others as part of an *embedded data transformation language*. Note that in this design, we borrow from our strongly typed host programming language its abstraction layers and type safety mechanisms that continue to check the semantic validity of the embedded language constructs. For instance, the type checker ensures that two isomorphisms f and g can be composed if and only if the target of f is the same as the source of g.

2.2 Choosing a Root

To avoid defining $\frac{n(n-1)}{2}$ 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 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.

We will choose as our `Root` object *finite sequences of natural numbers*. They can be seen as finite functions from an initial segment of $\mathbb{N}$, say $[0\,..\,n]$, to $\mathbb{N}$. We will represent them as lists, that is, their Haskell type is `[Nat]`. Alternatively, an array representation can be chosen.

```
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 = to (with that this)
```

The combinator `with` turns two encoders into an arbitrary isomorphism, that is, it 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 in the following template.

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

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

We will provide extensive use cases for these combinators as we populate our groupoid of isomorphisms. Given that `[Nat]` has been chosen as the root, we 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 now populate our groupoid of isomorphisms with combinators based on a few primitive converters.

3.1 An Isomorphism to Finite Sets of Natural Numbers

The isomorphism is specified with two bijections, `set2fun` and `fun2set`.

```
set :: Encoder [Nat]
set = Iso set2fun fun2set
```

While finite sets and sequences share a common representation `[Nat]`, sets are subject to the implicit constraint that all their elements are distinct. Such constraints can be regarded as “laws” that we assume about a given data type when needed, restricting it to the appropriate domain of the underlying mathematical concept. This suggests that a set like {7, 1, 4, 3} could be represented by first ordering it as {1, 3, 4, 7} and then computing the differences between consecutive elements. This gives [1, 2, 1, 3], with the first element 1 followed by the increments [2, 1, 3]. To turn the set into a bijection, including 0 as a possible member of a sequence, another adjustment is needed: elements in the sequence of increments should be replaced by their predecessors. This gives [1, 1, 0, 2] as implemented by `set2fun`.

```
set2fun is | is_set is =
  map pred (genericTake l ys) where
    ns=sort is
    l=genericLength ns
    next n | n≥0 = succ n
    xs =(map next ns)
    ys=(zipWith (-) (xs++[0]) (0:xs))

is_set ns = ns==nub ns
```

It can now be verified easily that incremental sums of the successors of numbers in such a sequence return the original set in sorted form, as implemented by `fun2set`.

```
fun2set ns =
  map pred (tail (scanl (+) 0 (map next ns))) where
  next n | n≥0 = succ n
```

The resulting encoder `set` is now ready to interoperate with another encoder.

```
*ISO> as set fun [0,1,0,0,4]
[0,2,3,4,9]
*ISO> as fun set [0,2,3,4,9]
[0,1,0,0,4]
```

As the example shows, this encoding maps arbitrary lists of natural numbers representing finite functions to strictly increasing sequences of (distinct) natural numbers representing sets.

3.2 Folding Sets into Natural Numbers

We can fold a set, represented as a list of distinct natural numbers, into a single natural number, reversibly, by observing that it can be seen as the list of exponents of 2 in the number's base 2 representation.

```
nat_set = Iso nat2set set2nat

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) (succ x)

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

We standardize this pair of operations as an `Encoder` for a natural number using our `Root` as a mediator.

```
nat :: Encoder Nat
nat = compose nat_set set
```

The resulting encoder `nat` is now ready to interoperate with any other encoder.

```
*ISO> as fun nat 2008
[3,0,1,0,0,0,0]
*ISO> as set nat 2008
[3,4,6,7,8,9,10]
*ISO> as nat set [3,4,6,7,8,9,10]
2008
*ISO> lend nat reverse 2008
1135
*ISO> lend nat_set reverse 2008
2008
*ISO> borrow nat_set succ [1,2,3]
[0,1,2,3]
*ISO> as set nat 42
[1,3,5]
*ISO> fit length nat 42
3
*ISO> retrofit succ nat_set [1,3,5]
43
```

The reader might notice at this point that we have already made a full circle—finite sets can be seen as instances of finite sequences. Injective functions that are not surjections with wider and wider gaps can be generated using the fact that one of the representations is information theoretically "denser" than the other, for a given range.

```
*ISO> as set fun [0,1,2,3]
[0,2,5,9]
*ISO> as set fun (as set fun [0,1,2,3])
[0,3,9,19]
*ISO> as set fun (as set fun (as set fun [0,1,2,3]))
[0,4,14,34]
```

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) [10, 11]. Together they form a mixed transformation called *hylomorphism*. We use such hylomorphisms to lift isomorphisms between lists and natural numbers to isomorphisms between a derived "self-similar" tree data type and natural numbers. In particular, we will derive Ackermann's encoding from hereditarily finite sets to natural numbers.

The data type representing hereditarily finite structures will be a generic multiway tree with a single leaf type `[]`.

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

The two sides of our hylomorphism 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 propagate a simpler operation guided by the structure 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 Ts.

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)
```

4.1.1 Hereditarily Finite Sets

Hereditarily finite sets will be represented as an encoder for the tree type T.

```
hfs :: Encoder T
hfs = compose (hylo nat_set) nat
```

The `hfs` encoder can now borrow operations from sets or natural numbers as follows.

```
hfs_union = borrow2 (with set hfs) union
hfs_succ = borrow (with nat hfs) succ
hfs_pred = borrow (with nat hfs) pred
```

```
*ISO> hfs_succ (H [])
H [H []]
*ISO> hfs_union (H [H []] ) (H [])
H [H []]
```

Otherwise, hylomorphism 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 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
```

4.1.2 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 another free algorithm, providing data compression/succinct representation for hereditarily finite sets. Note, for instance, the significantly smaller tree size in the following.

```
*ISO> as hff nat 42
H [H [H []],H [H []],H [H []]]
```

As the cognoscenti might observe, this is explained by the fact that `hff` provides higher information density than `hfs` by incorporating order information that matters in the case of a sequence, and is ignored in the case of a set.

5. Pairing/Unpairing

A *pairing* function is an isomorphism $f : Nat \times Nat \to Nat$. Its inverse is called *unpairing*. We now introduce an unusually simple pairing function (also mentioned in [12, p. 142]). The function `bitpair` works by splitting a number's big endian bitstring representation into odd and even bits, while its inverse `bitunpair` blends the odd and even bits back together.

```
type Nat2 = (Nat,Nat)

source (x,_)= x
target (_,y)= y

bitpair :: Nat2 → Nat
bitpair (i,j) =
 set2nat ((evens i) ++ (odds j)) where
  evens x = map (2*) (nat2set x)
  odds y = map succ (evens y)

bitunpair :: Nat→Nat2
bitunpair n = (f xs,f ys) where
 (xs,ys) = partition even (nat2set n)
 f = set2nat . (map (`div` 2))
```

The transformation of the bitlists is shown in the following example with bitstrings aligned.

```
*ISO> bitunpair 2008
P 60 26

-- 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
--   60:[0,    0,    1,    1,    1,    1]
--   26:[   0,    1,    0,    1,    1   ]
```

We can derive the following encoder:

```
nat2 :: Encoder Nat2
nat2 = compose (Iso bitpair bitunpair) nat
```

working as follows.

```
*ISO> as nat2 nat 2008
(60,26)
*ISO> as nat nat2 (60,26)
2008
```

Pairing/unpairing operations are important because they can encode complex objects (when interpreted as *cons+head+tail* operations) and ultimately code, for instance, simple LISP or λ-calculus programs and interpreters.

6. Cons-Lists with Pairing/Unpairing

The simplest application of pairing/unpairing operations is the encoding of cons-lists of natural numbers, defined as the `CList` data type.

```
data CList = Atom Nat | Cons CList CList
 deriving (Eq,Ord,Show,Read)
```

First, to provide an infinite supply of atoms, we encode them as even numbers.

```
to_atom n = 2*n
from_atom a | is_atom a = a `div` 2
is_atom n = even n && n≥0
```

Next, because we want atoms and cons cells to be disjoint, we encode the latter as odd numbers.

```
is_cons n = odd n && n>0
decons z | is_cons z = bitunpair ((z-1) `div` 2)
cons x y = 2*(bitpair (x,y))+1
```

We can deconstruct a natural number by recursing over applications of the unpairing-based `decons` combinator.

```
nat2cons n | is_atom n = Atom (from_atom n)
nat2cons n | is_cons n =
 Cons (nat2cons hd)
   (nat2cons tl) where
  (hd,tl) = decons n
```

We can reverse this process by recursing with the `cons` combinator on the `CList` data type.

```
cons2nat (Atom a) = to_atom a
cons2nat (Cons h t) = cons (cons2nat h) (cons2nat t)
```

The following example shows both transformations as inverses.

```
*ISO> nat2cons 123456789
Cons
 (Atom 2512)
 (Cons
  (Cons
    (Cons
        (Cons (Atom 0) (Atom 0))
        (Cons (Atom 0) (Atom 0))
     )
     (Atom 1)
    )
    (Atom 27)
   )
*ISO> cons2nat it
123456789
```

We obtain the encoder:

```
clist :: Encoder CList
clist = compose (Iso cons2nat nat2cons) nat
```

The encoder works as follows:

```
ISO> as clist nat 101
Cons (Atom 2) (Cons (Atom 0) (Cons (Atom 0) (Atom 0)))
```

and can be used to generate random LISP-like data and code skeletons from natural numbers.

7. Directed Graphs and Hypergraphs

We now show that more complex data types, such as digraphs and hypergraphs, have extremely simple encoders. This emphasizes 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 :: [Nat2] ⟶ [Nat]
set2digraph :: [Nat] ⟶ [Nat2]

digraph2set ps = map bitpair ps
set2digraph ns = map bitunpair ns
```

Here is the resulting encoder:

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

working as follows:

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

7.2 Encoding Directed Acyclic Graphs

Encoders for DAGs can be derived from the encoding of digraphs under the assumption that they are canonically represented by pairs of edges such that the first element of the pair is strictly smaller.

After defining:

```
digraph2dag = map f where f (x,y) = (x,y+x+1)

dag2digraph = map f where f (x,y) | y>x = (x,y-x-1)
```

we obtain the encoder:

```
dag :: Encoder [Nat2]
dag = compose (Iso dag2digraph digraph2dag ) digraph
```

working as follows:

```
*ISO> as nat dag
[(0,1),(1,2),(0,2),(1,3),(2,3),(3,4),(4,5),(1,6)]
8590000191
*ISO> as dag nat it
[(0,1),(1,2),(0,2),(1,3),(2,3),(3,4),(4,5),(1,6)]
*ISO> as digraph nat 2009
[(0,0),(1,1),(2,0),(2,1),(3,1),(0,2),(1,2),(0,3)]
*ISO> as nat digraph it
2009
*ISO> as dag nat 2009
[(0,1),(1,3),(2,3),(2,4),(3,5),(0,3),(1,4),(0,4)]
*ISO> as nat dag it
2009
```

Clearly, this mapping shows that there is a bijection between the set of finite digraphs and finite DAGs that preserves the number of edges. Note also that there is an obvious mapping between DAGs and unordered graphs, without self-loops, obtained by always directing edges from vertices with lower indices to vertices with strictly higher indices.

7.3 Encoding Hypergraphs

Definition 1. A *hypergraph* (also called *set system*) is a pair $H = (X, E)$ where X is a set and E is a set of nonempty subsets of X.

We can easily derive a bijective encoding of hypergraphs, represented as sets of nonempty sets.

```
set2hypergraph :: [Nat] → [[Nat]]
hypergraph2set :: [[Nat]] → [Nat]

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
```

8. 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.

8.1 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 :: Encoder a → Nat → a
nth thing = as thing nat

nths :: Encoder a → [Nat] → [a]
nths thing = map (nth thing)

stream_of :: Encoder a → [a]
stream_of thing = nths thing [0..]

*ISO> nth set 42
[1,3,5]
*ISO> take 3 (stream_of hfs)
[H [],H [H []],H [H [H []]]]
```

8.2 Random Instance Generation

Combining nth with a random generator for nat provides free algorithms for randomly generating complex objects of customizable size.

```
random_gen thing seed largest n = genericTake n
  (nths thing (rans seed largest)) where
  rans seed largest =
    randomRs (0,largest) (mkStdGen seed)

*ISO> random_gen set 11 999 3
[[0,2,5],[0,5,9],[0,1,5,6]]
*ISO> head (random_gen hfs 7 30 1)
H [H [],H [H [],H [H []]],H [H [H [H []]]]]
```

This is useful for further automating test generators in tools like QuickCheck [9].

8.3 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
```

Sometimes, 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, such as HFF or HFS.

In other cases, like `H [H [... []... ]]` corresponding to a tower of exponents as a hereditarily finite function or set, such representations are more compact than integers or list of integers representations that would quickly overflow all available memory.

8.4 Experimental Mathematics

We can compare representations sharing a common data type to conjecture about their asymptotic information density.

For instance, after defining:

```
length_as t = fit genericLength (with nat t)
sum_as t = fit sum (with nat t)
size_as t = fit tsize (with nat t)
```

we can conjecture that finite functions are more compact than sets asymptotically:

```
*ISO> length_as set 123456789012345678901234567890
54
*ISO> length_as fun 123456789012345678901234567890
54
*ISO> sum_as set 123456789012345678901234567890
2690
*ISO> sum_as fun 123456789012345678901234567890
43
```

and then observe that the same trend applies also to their hereditarily finite derivatives:

```
*ISO> size_as hfs 123456789012345678901234567890
627
*ISO> size_as hff 123456789012345678901234567890
91
```

8.5 A Surprising Free Algorithm: strange_sort

A simple isomorphism like `nat_set` can exhibit interesting properties as a building block of more intricate mappings, such as Ackermann's encoding. Here is another simple free algorithm for sorting a list of distinct elements without the explicit use of comparison operations.

```
strange_sort :: [Nat] → [Nat]
strange_sort = (from nat_set) . (to nat_set)

*ISO> strange_sort [2,9,3,1,5,0,7,4,8,6]
[0,1,2,3,4,5,6,7,8,9]
```

This algorithm emerges as a consequence of the commutativity of addition and the unicity of the decomposition of a natural number as a sum of powers of 2. Note that this property is derived from the particular choice of the isomorphism between natural numbers and sets.

8.6 Other Applications

A fairly large number of useful algorithms in fields ranging from data compression, coding theory, and cryptography to compilers, circuit design, and computational complexity involve bijective functions between heterogeneous data types. Their systematic encapsulation in a generic API that coexists well with strong typing can bring significant simplifications to various software modules with the added benefits of reliability and easier maintenance. In a genetic programming context [13], the use of isomorphisms between bitvectors/natural numbers on one side, and trees/graphs representing HFSs, HFFs on the other side, looks like a promising phenotype-genotype connection. Mutations and crossovers in a data type close to the problem domain are transparently mapped to numerical domains where evaluation functions can be computed easily. In the context of software transaction memory implementations (e.g., Haskell's STM [14]), encodings through isomorphisms are subject to efficient shortcuts; for example, undo operations in the case of transaction failures can be performed by applying inverse transformations without needing to save the intermediate chain of data structures involved.

9. Related Work

The closest reference on encapsulating bijections as a data type is [15] and Conal Elliott's composable bijections Haskell module [16], where, in a more complex setting, Arrows [17] are used as the underlying abstractions. While our `Iso` data type is similar to the *Bij* data type in [16] and BiArrow concept of [15], 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 [7, 8] associated to formulas. Together with their inverse unranking functions they are also used in combinatorial generation algorithms [18-21]. However, the generic view of such transformations as hylomorphisms obtained compositionally from simpler isomorphisms, as described in this paper, is new.

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 [22-27]. Computational and data representation aspects of finite set theory have been described in logic programming and theorem proving contexts in [28, 29].

Pairing functions have been used in work on decision problems as early as [30, 31]. A typical use in the foundations of mathematics is [32]. An extensive study of various pairing functions and their computational properties is presented in [33].

10. Conclusion

We have shown the expressiveness of Haskell as a metalanguage for executable mathematics by describing encodings for functions and finite sets in a uniform framework as data type isomorphisms with a groupoid structure. Haskell's higher order functions and recursion patterns have helped the design of an embedded data transformation language. The framework has been extended with hylomorphisms providing generic mechanisms for encoding hereditarily finite sets and hereditarily finite functions. In the process, a few surprising free algorithms have emerged, including Ackermann's encoding from hereditarily finite sets to natural numbers. A longer version of this paper, covering isomorphisms to data types as parenthesis languages, dyadic rationals, functional binary numbers, permutations, binary decision diagrams, and their hereditarily finite derivatives is available at arXiv.org/abs/0808.2953. A *Mathematica* notebook, combining a superset of the code in the paper with interactive visualizations of various transformations, is available at http://logic.cse.unt.edu/tarau/research/2009/isoNKS.nb.

References

[1] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, New York: Springer-Verlag New York, Inc., 1993.

[2] G. Chaitin, "A Theory of Program Size Formally Identical to Information Theory," *Journal of Association for Computing Machinery*, **22**, 1975 pp. 329-340.

[3] C. Calude and A. Salomaa, "Algorithmically Coding the Universe," in *Developments in Language Theory, World Scientific*, ed. G. Rozenberg and A. Salomaa (Singapore, 1994), pp. 472-492.

[4] G. Lakoff and M. Johnson, *Metaphors We Live By*, Chicago, IL: University of Chicago Press, 1980.

[5] S. Cook, "Theories for Complexity Classes and Their Propositional Translations," *Quaderni di Matematica*, ed. Jan Krajicek, **13**, 2004 pp. 1-36.

[6] S. Cook and A. Urquhart, "Functional Interpretations of Feasibly Constructive Arithmetic," *Annals of Pure and Applied Logic*, **63**, 1993 pp. 103-200.

[7] K. Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," *Monatshefte für Mathematik und Physik*, **38**, 1931 pp. 173-198.

[8] J. Hartmanis and T. Baker, "On Simple Gödel Numberings and Translations," in *Lecture Notes in Computer Science Vol. 14 (ICALP 1974),* Saarbrücken, Germany (J. Loeckx, ed.), London: Springer, 1974 pp. 301-316.

[9] K. Claessen and J. Hughes, "Testing Monadic Code with Quickcheck," *SIGPLAN Notices*, **37**(12), 2002 pp. 47-59.

[10] G. Hutton, "A Tutorial on the Universality and Expressiveness of Fold," *Journal of Functctional Programming*, **9**(4), 1999 pp. 355-372.

[11] E. Meijer and G. Hutton, "Bananas in Space: Extending Fold and Unfold to Exponential Types," in *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA 1995)*, La Jolla, CA, (J. Williams, ed.), New York: ACM, 1995 pp. 324-333.

[12] S. Pigeon, *Contributions á la compression de données,* Ph.D. dissertation, Université de Montréal, Montréal 2001.

[13] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press, 1992.

[14] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, "Composable Memory Transactions," *Communications of the ACM*, **51**(8), 2008 pp. 91-100.

[15] A. Alimarine, S. Smetsers, A. van Weelden, M. van Eekelen, and R. Plasmeijer, "There and Back Again: Arrows for Invertible Programming," in *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Tallin, Estonia (D. Leijen, ed.), New York: ACM Press, 2005 pp. 86-97.

[16] C. Elliott. "Module: Data.Bijections." haskell.org/haskellwiki/TypeCompose.

[17] J. Hughes, "Generalizing Monads to Arrows," *Science of Computer Programming*, **37**(1-3), 2000 pp. 67-111.

[18] C. Martinez and X. Molinero, "Generic Algorithms for the Generation of Combinatorial Objects," in *Lecture Notes in Computer Science Vol. 2747 (MFCS 2003)*, Bratislava, Slovak Republic (B. Rovan and P. Vojtas, eds.), Berlin: Springer, 2003 pp. 572-581.

[19] D. Knuth, *The Art of Computer Programming, Vol. 4*, Indianapolis, IN: Addison Wesley, 2009.

[20] F. Ruskey and A. Proskurowski, "Generating Binary Trees by Transpositions," *Journal of Algorithms*, **11**(1), 1990 pp. 68-84.

[21] W. Myrvold and F. Ruskey, "Ranking and Unranking Permutations in Linear Time," *Information Processing Letters*, **79**(6), 2001 pp. 281-284.

[22] M. Takahashi, "A Foundation of Finite Mathematics," *Publications of the Research Institute for Mathematical Sciences*, **12**(3), 1976 pp. 577-708.

[23] R. Kaye and T. Wong, "On Interpretations of Arithmetic and Set Theory," *Notre Dame Journal of Formal Logic*, **48**(4), 2007 pp. 497-510.

[24] A. Abian and S. Lamacchia, "On the Consistency and Independence of Some Set-Theoretical Constructs," *Notre Dame Journal of Formal Logic*, **X1X**(1), 1978 pp. 155-158.

[25] J. Aviga. "The Combinatorics of Propositional Provability." *ASL Winter Meeting (Association for Symbolic Logic 1997)*, San Diego, CA, 1997. www.andrew.cmu.edu/user/avigad/Talks/tautologies.pdf.

[26] L. Kirby, "Addition and Multiplication of Sets," *Mathematical Logic Quarterly*, **53**(1), 2007 pp. 52-65.

[27] A. Leontjev and V. Sazonov, "Capturing LOGSPACE over Hereditarily-Finite Sets," in *Lecture Notes in Computer Science Vol. 1762 (FoIKS 2000)*, Burg, Germany (K. Schewe and B. Thalheim, eds.), London: Springer-Verlag, 2000 pp. 156-175.

[28] C. Piazza and A. Policriti, "Ackermann Encoding, Bisimulations, and OBDDs," *Theory and Practice of Logic Programming*, **4**(5-6), 2004 pp. 695-718.

[29] L. Paulson, "A Concrete Final Coalgebra Theorem for ZF Set Theory," in *Lecture Notes in Computer Science Vol. 996 (TYPES 1994)*, Båstad, Sweden, (P. Dybjer, B. Nordström, and J. Smith, eds.), London: Springer-Verlag, 1994 pp. 120-139.

[30] J. Robinson, "General Recursive Functions," *Proceedings of the American Mathematical Society*, **1**(6), 1950 pp. 703-718.

[31] J. Robinson, "Finite Generation of Recursively Enumerable Sets," *Proceedings of the American Mathematical Society*, **19**(6), 1968 pp. 1480-1486.

[32] P. Cégielski and D. Richard, "Decidability of the Theory of the Natural Integers with the Cantor Pairing Function and the Successor," *Theoretical Computer Science*, **257**(1-2), 2001 pp. 51-77.

[33] A. Rosenberg, "Efficient Pairing Functions—and Why You Should Care," *International Journal of Foundations of Computer Science*, **14**(1), 2003 pp. 3-17.