"Everything is everything" revisited: shapeshifting data types with isomorphisms and hylomorphisms

Paul Tarau

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

Abstract. This paper is an exploration of isomorphisms between elementary data types (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 generation of random instances.
A longer version of the paper, together with its literate Haskell program is available at `http://arXiv.org/abs/0808.2953`.
Keywords: computational mathematics, ranking/unranking, Ackermann encoding, hereditarily finite sets and functions, pairing/unpairing, Haskell data representations

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 [22, 6, 4].

Analogical/metaphorical thinking routinely shifts entities and operations from a field to another hoping to uncover similarities in representation or use [20].

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 [9, 10].

Mathematical theories often borrow proof patterns and reasoning techniques accross 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.

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 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.

Sharing heterogeneous data objects faces two problems:

- some form of equivalence needs to be proven between two objects A and B before A can replace B in a data structure, a possibly tedious and error prone task
- the fast growing diversity of data types makes harder and harder to recognize sharing opportunities.

Besides, this rises the question: what guaranties do we have that sharing accross heterogeneous data types is useful and safe?

The techniques introduced in this paper provide a generic solution to these problems, through isomorphic mappings between heterogeneous data types, such that unified internal representations make equivalence checking and sharing 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 – reversible mappings that also transport operations. In their simplest form such isomorphisms show up as *encodings* – to some simpler and easier to manipulate representation – for instance natural numbers.

Such encodings can be traced back to Gödel numberings [11, 13] associated to formulae, 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, one can interpret data structures like graphs and program constructs like loops or recursion as compression mechanisms focusing on sharing and reuse of 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 accross heterogeneous data and code types.

2 An embedded data transformation language

We will start by designing an embedded transformation language as a set of operations on a group of isomorphisms. We will then extended it with as set higher order combinators mediating the composition of encodings and the transfer of operations between data types.

2.1 The group of isomorphisms

We implement an isomorphism between two objects X and Y as a Haskell data type encapsulating a bijection f and its inverse g. We will call the *from* function the first component (a *section* in category theory parlance) and the *to* function the second component (a *retraction*) defining the isomorphism. We can organize isomorphisms as a *group* 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 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 correctness of implementations with tools like QuickCheck [7].

Proposition 1 *The data type Iso has a group structure, i.e. the* compose *operation is associative,* itself *acts as an identity element and* invert *computes the inverse of an isomorphism.*

We can transport operations from an 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 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 etc.` 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 safety mechanisms that continue to check the semantic validity of the embedded language constructs.

2.2 Choosing a root

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

Choosing a *Root* object is somewhat arbitrary, but it makes sense to pick a representation that is relatively easy convertible to various others, 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 as finite functions from an initial segment of Nat, say $[0..n]$, to Nat. We will represent them as lists i.e. their Haskell type is $[Nat]$. 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
```

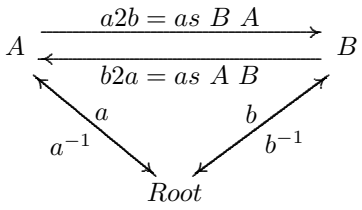
together with the combinators *with* and *as* providing an *embedded transformation language* for routing isomorphisms through two *Encoders*.

```
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, i.e. acts as a connection hub between their domains. The combinator `as` adds a more convenient syntax such that converters between A and B can be designed as:

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



We will provide extensive use cases for these combinators as we populate our group 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 group of isomorphisms

We will now populate our group 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[1]. This suggest that a set like $\{7, 1, 4, 3\}$ could be represented by first ordering it as $\{1, 3, 4, 7\}$ and then compute 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 it 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
```

[1] 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.

```
    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 will 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 full circle - as 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 Pure 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) [15, 24]. Together they form a mixed transformation called *hylomorphism*. We will use such hylomorphisms to lift isomorphisms between lists and natural numbers to isomorphisms between a derived "self-similar" tree data type and natural numbers. 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 f n = H (unranks f (f n))
unranks f ns = map (unrank f) ns

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

Both combinators can be seen as a form of "structured recursion" that propagate a simpler operation guided by the 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)
```

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
```

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:

```
*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 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 will introduce here an unusually simple pairing function (also mentioned in [28], 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
```

A reason pairing/unpairing operations are important, is that 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 encoding of cons-lists of natural numbers, defined as the 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, as we want atoms and cons cell disjoint, we will encode the later 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 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 bitpair ps
set2digraph ns = map bitunpair ns
```

The resulting Encoder is:

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

working 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
```

7.2 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 non-empty subsets of X.*

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

```
set2hypergraph = map nat2set
hypergraph2set = map set2nat
```

The resulting Encoder is:

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

working 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 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 thing = as thing nat
nths thing = map (nth thing)
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 random generation of 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 [7].

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
```

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.

8.4 Experimental Mathematics

We can compare representations sharing a common datatype 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)
```

one can conjecture that finite functions are more compact than sets asymptotically

```
*ISO> length_as set 12345678901234567890123456789O
54
*ISO> length_as fun 12345678901234567890123456789O
54
*ISO> sum_as set 12345678901234567890123456789O
2690
*ISO> sum_as fun 12345678901234567890123456789O
43
and then observe that the same trend applies also
to their hereditarily finite derivatives:
*ISO> size_as hfs 12345678901234567890123456789O
627
*ISO> size_as hff 12345678901234567890123456789O
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 like Ackermann's encoding, but let's also note a simple "free algorithm" – sorting a list of distinct elements without explicit use of comparison operations:

```
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.

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 [19] 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 (like Haskell's STM [12]), encodings through isomorphisms are subject to efficient shortcuts, as undo operations in case of transaction failure can be performed by applying inverse transformations without the need to save the intermediate chain of data structures involved.

9 Related work

The closest reference on encapsulating bijections as a data type is [2] and Connan Eliot's composable bijections Haskell module [8], where, in a more complex setting, Arrows [14] are used as the underlying abstractions. While our `Iso` data type is similar to the *Bij* data type in [8] and BiArrow concept of [2], 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 [11, 13] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [23, 18, 32, 25]. 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 [33, 16, 1, 3, 17, 21]. Computational and data representation aspects of finite set theory have been described in logic programming and theorem proving contexts in [27, 26].

Pairing functions have been used in work on decision problems as early as [29, 30]. A typical use in the foundations of mathematics is [5]. An extensive study of various pairing functions and their computational properties is presented in [31].

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 group 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 the 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 `http://arXiv.org/abs/0808.2953`.

References

1. Alexander Abian and Samuel Lamacchia. On the consistency and independence of some set-theoretical constructs. *Notre Dame Journal of Formal Logic*, X1X(1):155–158, 1978.
2. 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.
3. Jeremy Avigad. The Combinatorics of Propositional Provability. In *ASL Winter Meeting*, San Diego, January 1997.
4. Cristian Calude and Arto Salomaa. Algorithmically coding the universe. In *Developments in Language Theory, World Scientific*, pages 472–492, 1994.
5. 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.
6. Gregory J. Chaitin. A theory of program size formally identical to information theory. *J. Assoc. Comput. Mach*, 22:329–340, 1975.
7. Koen Claessen and John Hughes. Testing monadic code with quickcheck. *SIGPLAN Notices*, 37(12):47–59, 2002.
8. Connan Eliot. Data.Bijections Haskell Module. http://haskell.org/haskellwiki/TypeCompose.
9. Stephen Cook. Theories for complexity classes and their propositional translations. In *Complexity of computations and proofs*, pages 1–36, 2004.
10. Stephen Cook and Alasdair Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63:103–200, 1993.
11. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
12. Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.
13. 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.
14. John Hughes. Generalizing Monads to Arrows. Science of Computer Programming 37, pp. 67-111, May 2000.
15. Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.*, 9(4):355–372, 1999.
16. Richard Kaye and Tin Lock Wong. On Interpretations of Arithmetic and Set Theory. *Notre Dame J. Formal Logic Volume*, 48(4):497–510, 2007.
17. Laurence Kirby. Addition and multiplication of sets. *Math. Log. Q.*, 53(1):52–65, 2007.
18. Donald Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
19. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

20. George Lakoff and Mark Johnson. *Metaphors We Live By.* University of Chicago Press, Chicago, IL, USA, 1980.
21. Alexander Leontjev and Vladimir Yu. Sazonov. Capturing LOGSPACE over Hereditarily-Finite Sets. In Klaus-Dieter Schewe and Bernhard Thalheim, editors, *FoIKS*, volume 1762 of *Lecture Notes in Computer Science*, pages 156–175. Springer, 2000.
22. Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.
23. 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.
24. Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *FPCA*, pages 324–333, 1995.
25. Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79:281–284, 2001.
26. Lawrence C. Paulson. A Concrete Final Coalgebra Theorem for ZF Set Theory. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 120–139. Springer, 1994.
27. Carla Piazza and Alberto Policriti. Ackermann Encoding, Bisimulations, and OBDDs. *TPLP*, 4(5-6):695–718, 2004.
28. Stephen Pigeon. *Contributions à la compression de données.* Ph.d. thesis, Université de Montréal, Montréal, 2001.
29. Julia Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950.
30. Julia Robinson. Finite generation of recursively enumerable sets. *Proceedings of the American Mathematical Society*, 19(6):1480–1486, dec 1968.
31. Arnold L. Rosenberg. Efficient pairing functions - and why you should care. *International Journal of Foundations of Computer Science*, 14(1):3–17, 2003.
32. Frank Ruskey and Andrzej Proskurowski. Generating binary trees by transpositions. *J. Algorithms*, 11:68–84, 1990.
33. Moto-o Takahashi. A Foundation of Finite Mathematics. *Publ. Res. Inst. Math. Sci.*, 12(3):577–708, 1976.