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

Paul Tarau¹

¹Department of Computer Science and Engineering
Univ of North Texas

NKS’08
Motivation: analogies

- analogies everywhere: mathematical theories often borrow proof patterns and reasoning techniques across close and sometime not so close fields
- 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
- unified internal representations make equivalence checking and sharing possible
- Haskell code can be generated with a proof assistant (Coq)
- \(\Rightarrow\) equivalences can be formally proven
Motivation: code and data sharing

- 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.
- 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.
- Maximal sharing acts as the dual of minimal program+input size.
- Shapeshifting through a uniform set of encodings would extend sharing opportunities across heterogeneous data and code types.
Shapeshifting between datatypes: “everything is everything”

- magic made easy – but in a safe way: bijective mappings using a strongly typed language as a watchdog (Haskell)
Overview

- an exploration in a functional programming framework of isomorphisms between elementary data types
- ranking/unranking operations (bijective Gödel numberings)
- pairing/unpairing operations
- generating new isomorphisms through hylomorphisms (folding/unfolding into hereditarily finite universes)
- applications
The Group of Isomorphisms

Assumption: \( f \circ g = id_a \) and \( g \circ f = id_b \)

data \( \text{Iso} \ a \ b = \text{Iso} \ (a \to b) \ (b \to a) \)

\[
\text{from} \ (\text{Iso} \ f \ _) = f \\
\text{to} \ (\text{Iso} \ _ \ g) = g
\]

\[
\text{compose} :: \text{Iso} \ a \ b \to \text{Iso} \ b \ c \to \text{Iso} \ a \ c \\
\text{compose} \ (\text{Iso} \ f \ g) \ (\text{Iso} \ f' \ g') = \text{Iso} \ (f' \cdot f) \ (g \cdot g') \\
\text{itself} = \text{Iso} \ id \ id \\
\text{invert} \ (\text{Iso} \ f \ g) = \text{Iso} \ g \ f
\]

Proposition

\( \text{Iso} \) has a group structure: \( \text{compose} \) is associative, \text{itself} is an identity element, \( \text{invert} \) computes the inverse of an isomorphism.
Transporting Operations

\begin{align*}
\text{borrow} &:: \text{Iso } t \; s \rightarrow (t \rightarrow t) \rightarrow s \rightarrow s \\
\text{borrow} \; (\text{Iso } f \; g) \; h \; x & = f \; (h \; (g \; x)) \\
\text{borrow}2 \; (\text{Iso } f \; g) \; h \; x \; y & = f \; (h \; (g \; x) \; (g \; y)) \\
\text{borrow}N \; (\text{Iso } f \; g) \; h \; xs & = f \; (h \; (\text{map } g \; xs)) \\

\text{lend} &:: \text{Iso } s \; t \rightarrow (t \rightarrow t) \rightarrow s \rightarrow s \\
\text{lend} & = \text{borrow} \; . \; \text{invert} \\
\text{lend}2 & = \text{borrow}2 \; . \; \text{invert} \\
\text{lend}N & = \text{borrow}N \; . \; \text{invert} \\

\text{Examples will follow as we populate the universe.}
\end{align*}
Choosing a Root

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

the combinators *with* and *as* provide 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 `as`

\[ \text{as} :: \text{Encoder a} \to \text{Encoder b} \to \text{b} \to \text{a} \]

as that this = to (with that this)

\[ a2b \ x = \text{as A B x} \]
\[ b2a \ x = \text{as B A x} \]

as \([Nat]\) has been chosen as the root, we will define our finite function data type `fun` simply as the identity isomorphism on sequences in \([Nat]\):

\[ \text{fun} :: \text{Encoder [Nat]} \]
\[ \text{fun} = \text{itself} \]
Finite Functions to/from Sets

*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.
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.

*ISO> as nat set [3,4,6,7,8,9,10]
2008
*ISO> lend nat reverse 2008 -- order matters
1135
*ISO> lend nat_set reverse 2008 -- order independent
2008
*ISO> borrow nat_set succ [1,2,3]
[0,1,2,3]
*ISO> as set nat 42
[1,3,5]
The \textit{ranking problem} for a family of combinatorial objects is finding a unique natural number associated to it, called its \textit{rank}.

The inverse \textit{unranking problem} consists of generating a unique combinatorial object associated to each natural number.

\textit{Unranking anamorphism} (\textit{unfold} operation): generates an object from a simpler representation - for instance the seed for a random tree generator

\textit{Ranking catamorphism} (a \textit{fold} operation): associates to an object a simpler representation - for instance the sum of values of the leaves in a tree

Together they form a mixed transformation called \textit{hylomorphism}
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 \( \text{Iso } f \; g \):

\[
\begin{align*}
\text{unrank } f \; n &= H (\text{unranks } f \; (f \; n)) \\
\text{unranks } f \; ns &= \text{map } (\text{unrank } f) \; ns
\end{align*}
\]

\[
\begin{align*}
\text{rank } g \; (H \; ts) &= g \; (\text{ranks } g \; ts) \\
\text{ranks } g \; ts &= \text{map } (\text{rank } g) \; ts
\end{align*}
\]

“structured recursion”: propagate a simpler operation guided by the structure of the data type obtained as:

\[
tsize = \text{rank } (\lambda x \rightarrow 1 + (\text{sum } x))
\]
We can now combine an anamorphism+catamorphism pair into an isomorphism \textit{hylo} defined with \texttt{rank} and \texttt{unrank} on the corresponding hereditarily finite data types:

\begin{align*}
\text{hylo} &:: \text{Iso } b \ [b] \rightarrow \text{Iso } T \ b \\
\text{hylo} \ (\text{Iso } f \ g) & = \text{Iso } \ (\text{rank } g) \ (\text{unrank } f)
\end{align*}

\begin{align*}
\text{hylos} &:: \text{Iso } b \ [b] \rightarrow \text{Iso } Ts \ [b] \\
\text{hylos} \ (\text{Iso } f \ g) & = \text{Iso } \ (\text{ranks } g) \ (\text{unranks } f)
\end{align*}
Hereditarily finite sets

\[
\text{hfs} :: \text{Encoder } T \\
\text{hfs} = \text{compose} \ (\text{hylo} \ \text{nat_set}) \ \text{nat}
\]

\[
*\text{ISO} > \text{as} \ \text{hfs} \ \text{nat} \ 42 \\
\text{H [H [H []],H [H []],H [H []]],H [H []],H [H [H []]]]}
\]

we have just derived as a “free algorithm” Ackermann’s encoding from hereditarily finite sets to natural numbers and its inverse!

\[
\text{ackermann} = \text{as} \ \text{nat} \ \text{hfs}
\]

\[
\text{inverse_ackermann} = \text{as} \ \text{hfs} \ \text{nat}
\]

\[
f(x) = \text{if } x = \{\} \text{ then 0 else } \sum_{a \in x} 2^{f(a)}
\]
Hereditarily Finite Set associated to 42
Hereditarily Finite Set associated to 2008
Hereditarily finite functions

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

this hff Encoder can be seen as another (new this time!) “free algorithm”, providing data compression/succinct representation for hereditarily finite sets (note the significantly smaller tree size):

*ISO⟩ as hfs nat 42
  H [H [H []], H [H []], H [H [H []]]], H [H [], H [H [H [H []]]]]

*ISO⟩ as hff nat 42
  H [H [H []], H [H []], H [H []]]
Pairing/Unpairing

pairing function: isomorphism $f : \text{Nat} \times \text{Nat} \rightarrow \text{Nat}$; inverse: unpairing

type Nat2 = (Nat,Nat)
*ISO> bitunpair 2008
  (60,26)
*ISO> bitpair (60,26)
  2008
  -- 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]
*ISO> as nat2 nat 2008
  (60,26)
*ISO> as nat nat2 (60,26)
  2008
Encodings of cons-lists

*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
Encoding directed graphs

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
Encoding hypergraphs

\[
\text{set2hypergraph} = \text{map nat2set}\\
\text{hypergraph2set} = \text{map set2nat}
\]

The resulting Encoder is:

\[
\text{hypergraph} :: \text{Encoder } [[\text{Nat}]]\\
\text{hypergraph} = \text{compose } (\text{Iso } \text{hypergraph2set } \text{set2hypergraph}) \text{ set}
\]

working as follows

\[
\text{*ISO} \triangleright \text{as hypergraph nat 2008}\\
[[0,1],[2],[1,2],[0,1,2],[3],[0,3],[1,3]]
\]

\[
\text{*ISO} \triangleright \text{as nat hypergraph it}\\
2008
\]
So many encodings so little time ...

- hereditarily finite sets with (finite/infinite supply of) *urelements*
- hereditarily finite functions with *urelements*
- undirected graphs, multigraphs, multidigraphs
- permutations, hereditarily finite permutations
- BDDs, MTBDDs (multi-terminal BDDs)
- dyadic rationals
- functional binary numbers
- strings, \(\{0, 1\}\)-bitstrings
- parenthesis languages
- dyadic rationals
- DNA strands
Some Examples: BDDs

*ISO> as rbdd nat 2008
BDD 4
  (D 3
    (D 2 B0
      (D 1
        (D 0 B0 B1)
        (D 0 B1 B0)))
    (D 2
      (D 1 B1 B0)
      (D 1 B0
        (D 0 B1 B0))))
*ISO> as nat rbdd it
2008
More Examples: MTBDDs

>to_mtbdd 3 3 2008
MTBDD 3 3
(M 2
  (M 1
    (M 0 (L 2) (L 1))
    (M 0 (L 2) (L 1)))
  (M 1
    (M 0 (L 2) (L 0))
    (M 0 (L 1) (L 1))))

>from_mtbdd it
2008
Examples: permutations and HFPs

*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

*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
Examples: parenthesis languages

*ISO*\(\triangleright\) as pars nat 42
"((()))((()))((()))"

*ISO*\(\triangleright\) as hff pars it
H [H [],H [],H []]

*ISO*\(\triangleright\) as nat hff it
42

*ISO*\(\triangleright\) as bitpars nat 2008
[0,0,0,1,0,1,1,0,1,0,0,1,1,0,1,0,1,0,1,0,1,1]

*ISO*\(\triangleright\) as nat bitpars it
2008

*ISO*\(\triangleright\) as nat bits (as bitpars nat 2008)
7690599

*ISO*\(\triangleright\) map ((as nat bits) . (as bitpars nat)) [0..7]
[5,27,119,115,495,483,471,467]
DNA encodings

*ISO> as dna nat 2008
   [Adenine, Guanine, Cytosine, Thymine, Thymine, Cytosine]
*ISO> borrow (with dna nat) dna_reverse 42 42
*ISO> borrow (with dna nat) dna_reverse 2008 637
*ISO> borrow (with dna nat) dna_complement 2008 2087
*ISO> borrow (with dna nat) dna_comprev 2008 3458
*ISO> borrow (with dna bits)
       dna_comprev [1,0,1,0,1,1,0,1,0,1,0,1]
     [1,1,1,0,1,0,0,0,0,1,1,1]
Applications: a surprising “free algorithm”: strange_sort

“free algorithm” – sorting a list of distinct elements without explicit use of comparison operations:

\[
\text{strange\_sort} = (\text{from nat\_set}) \times (\text{to nat\_set})
\]

*ISO> strange\_sort [2, 9, 3, 1, 5, 0, 7, 4, 8, 6]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

a consequence of the commutativity of addition and the unicity of the decomposition of a natural number as a sum of powers of 2
Applications: succinct representations

*ISO> length_as set 123456789012345678901234567890 54
*ISO> length_as perm 123456789012345678901234567890 28
*ISO> length_as fun 123456789012345678901234567890 54
*ISO> sum_as set 123456789012345678901234567890 2690
*ISO> sum_as perm 123456789012345678901234567890 378
*ISO> sum_as fun 123456789012345678901234567890 43
Compressed representations: a measure of “structural” complexity?

*ISO> size_as hfs 123456789012345678901234567890
627
*ISO> size_as hfp 123456789012345678901234567890
276
*ISO> size_as hff 123456789012345678901234567890
91

*ISO> bdd_size $ as bdd
  nat 123456789012345678901234567890
256
*ISO> robdd_size $ as rbdd
  nat 123456789012345678901234567890
39
Applications: Random Generation

Combining \texttt{nth} with a random generator for \texttt{nat} provides free algorithms for random generation of complex objects of customizable size:

*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 []]]]
*ISO> head (random\_gen dnaStrand 1 123456789 1)
DNA\_strand P5x3 [Guanine, Thymine, Guanine, Cytosine, Cytosine, Thymine, Thymine, Thymine, Thymine, Thymine, Cytosine, Cytosine, Cytosine, Thymine, Thymine, Thymine, Thymine, Thymine, Thymine, Adenine, Thymine, Cytosine, Cytosine]

This is useful for further automating test generators in tools like QuickCheck.
Other Applications

- a promising phenotype-genotype connection in Genetic Programming: isomorphisms between bitvectors/natural numbers on one side, and trees/graphs representing HFSs, HFFs on the other side
- Software Transaction Memory: undo operations by applying inverse transformations without the need to save the intermediate chain of states
we have designed an embedded combinator language that shapeshifts datatypes at will using a small group of isomorphisms
we have shown how to lift them with hylomorphisms to hereditarily finite datatypes
a practical tool to experiment with various universal encoding mechanisms

Literate Haskell program + (very) long version of the paper at http://logic.csci.unt.edu/tarau/research/2008/fISO.zip
Open problems

- encodings are more difficult when *transitivity* is involved
  - encodings for finite posets, finite topologies?
  - encodings for finite categories?
- towards a “Theory of Everything” in Computer Science?
  - is such a theory possible? is it useful?
  - it should be easier: CS is more of a “nature independent” construct than physics!
  - our initial focus: isomorphisms between datatypes are the easy part
- can a *Theory of Everything* make Computer Science simple again?