Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell

Paul Tarau[1]

[1]Department of Computer Science and Engineering
Univ of North Texas

SAC'09

Motivation

Figure : *Shapeshifting* between datatypes: "everything is everything"

magic made easy - but also safe: we use bijective mappings between datatypes using a strongly typed language as a watchdog (Haskell)

The Challenge

- we need to be able to encode everything as everything else - the paper is about *how we can do it*
- the short answer: we need isomorphisms: bijections that transport structures
- what can that bring us?
  - we can transfer operations between datatypes
  - “free algorithms” can emerge
  - sharing opportunities across heterogeneous data types we obtain free iterators and random instance generators
  - data compression and succinct representations
  - $\Rightarrow$ a general mechanism for serialization and persistence
  - cryptography: encrypt and decrypt are bijections - all our transformations are bijections - the synergy is becoming interesting !!!

Outline

- 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 Groupoid of Isomorphisms

```
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
```

Proposition

`Iso` *is groupoid: when defined,* `compose` *is associative,* `itself` *is an identity element,* `invert` *computes the inverse of an isomorphism.*

Transporting Operations

```haskell
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
```

Examples will follow as we populate the universe.

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`

```
as :: Encoder a→Encoder b→b→a
as that this = to (with that this)
```

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

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

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 $[Nat]$:

```
fun :: Encoder [Nat]
fun = 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]
```

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.
- *unranking anamorphism* (*unfold* operation): generates an object from a simpler representation - for instance the seed for a random tree generator
- *ranking catamorphism* (a *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 *hylomorphism*

Ranking/unranking hereditarily finite datatypes

```
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
```

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

```
tsize = rank (λx→1 + (sum x))
```

Extending isomorphisms with hylomorphisms

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

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

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

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

```
ackermann = as nat hfs
inverse_ackermann = as hfs nat
```

$f(x) = \texttt{if } x = \{\} \texttt{ then } 0 \texttt{ else } \sum_{a \in x} 2^{f(a)}$

Hereditarily Finite Set associated to **2008**

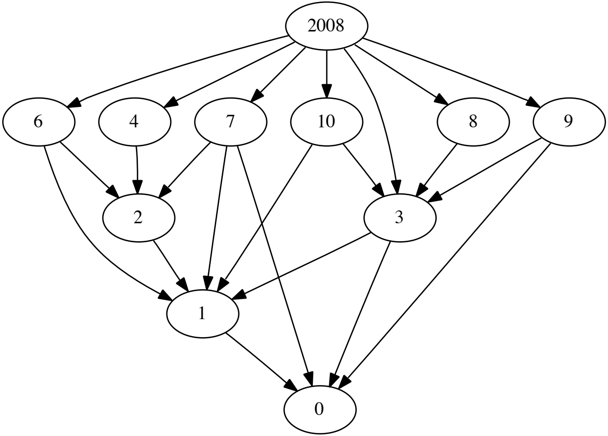

Figure : 2008 as a HFS

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

Hereditarily Finite Function associated to **2008**

Note that edges are labeled to indicate order.

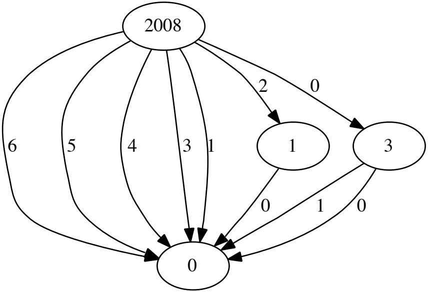

Figure : 2008 as a HFF

Pairing/Unpairing

pairing function: isomorphism $f : Nat \times Nat \rightarrow Nat$; 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
```

Recursive unpairing graph

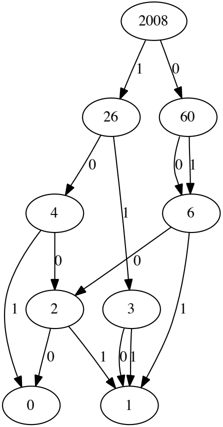

Figure : Graph obtained by recursive application of `bitunpair` for 2008

Visualizing unpairing as a plane filling curve

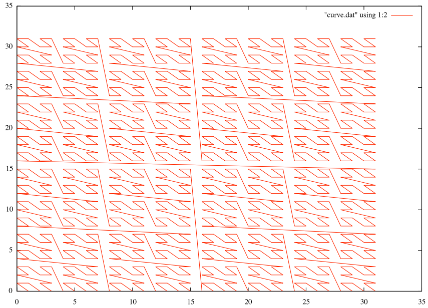

Figure : 2D curve connecting values of `bitunpair n` for $n \in [0..2^{10} - 1]$

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
```

Digraph encoding

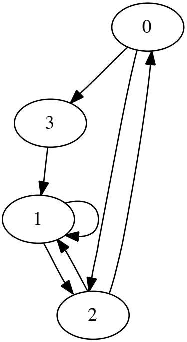

Figure : 2008 as a digraph

Encoding hypergraphs

```
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
```

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

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> as pars nat 42
"((())(())(()))"
*ISO> as hff pars it
H [H [H []],H [H []],H [H []]]
*ISO> as nat hff it
42
*ISO> 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> as nat bitpars it
2008
*ISO> as nat bits (as bitpars nat 2008)
7690599
*ISO> map ((as nat bits) .  (as bitpars nat)) [0..7]
[5,27,119,115,495,483,471,467]
```

Applications: a surprising "free algorithm": strange_sort

"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]
```

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: Random Generation

Combining `nth` with a random generator for *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 [H []]]]]
*ISO> head (random_gen dnaStrand 1 123456789 1)
DNAstrand P5x3 [Guanine,Thymine,Guanine,Cytosine,
  Cytosine,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

Conclusion

- we have designed an embedded combinator language that shapeshifts datatypes at will using a small groupoid 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?