ABSTRACT
This paper is an exploration in a functional programming framework of isomorphisms between elementary data types (natural numbers, sets, bitstrings, finite functions) and their extension to hereditarily finite universes through hylomorphisms derived from ranking and unranking operations. The paper is part of a larger effort to cover in a declarative programming paradigm some fundamental combinatorial generation algorithms along the lines of Knuth's recent work [10]. The self-contained source code of the paper, as generated from a literate Haskell program, is available at http://logic.csci.unt.edu/tarau/research/2008/sfISO.zip.

Categories and Subject Descriptors
D.3.3 [PROGRAMMING LANGUAGES]: Language Constructs and features—Data types and structures

General Terms
Languages, algorithms

Keywords
Haskell data representations, computational mathematics, ranking/unranking, Ackermann encoding, hereditarily finite sets, hereditarily finite functions

1. INTRODUCTION
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.

It is also important to organize such encodings as a flexible embedded language to accommodate any-to-any conversions without the need to write one-to-one converters.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC’09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

Toward this end we will organize our encodings as a group of isomorphisms within a (mildly) category theory inspired design.

2. 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:

\[
\begin{array}{ccc}
X & \xrightarrow{f} & Y \\
\xleftarrow{g} & & \\
\end{array}
\]

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')

invert (Iso f g) = Iso g f

We can now formulate laws about isomorphisms that can be used to test correctness of implementations with tools like QuickCheck [2].

**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 operations 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**

2.1 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 design a simple 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 a 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 \([\text{Nat}]\). Alternatively, an array representation can be chosen.

```haskell
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 operations with and as providing the embedded language for routing isomorphisms through two Encoders.

```haskell
with :: Encoder a -> Encoder b -> Iso a b
with this that = compose this (invert that)
```

```haskell
as :: Encoder a -> Encoder b -> b -> a
as that this thing = to (with that this) thing
```

With these definitions, converters between A and B will be simply:

\[
\begin{align*}
a2b \ x &= as \ A \ B \ x \\
b2a \ x &= as \ B \ A \ x
\end{align*}
\]

```latex
A \quad a2b = as \ B \ A
\quad b2a = as \ A \ B
\quad a \quad a^{-1}
\quad b \quad b^{-1}
\quad \text{Root}
```

Given that \([\text{Nat}]\) has been chosen as the root, we will define our finite function data type fun simply as the identity isomorphism on sequences in \([\text{Nat}]\).

```haskell
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.

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

While finite sets and sequences share a common representation \([\text{Nat}]\), sets are subject to the implicit constraint that all their elements are distinct. 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 \([1, 0, 0, 2]\) as implemented by set2fun:

```haskell
set2fun is | is_set is = map pred (genericTake 1 ys) where
            n = sort is
            l = genericLength ns
            next n | n ≥ 0 = succ n
            xs = (map next ns)
            ys = (zipWith (+) (xs++[0]) (0:xs))
```

```haskell
is_set 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:

```haskell
fun2set ns = map pred (genericTake l ys) where
           next n | n ≥ 0 = succ n
```

The resulting Encoder (nat) is now ready to interoperate with another Encoder:

```haskell
+ISO> as 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.

```haskell
nat_set = Iso nat2set set2nat
```

```haskell
nat2set n | n ≥ 0 = nat2exp s n 0 where
           nat2exp s 0 _ = []
           nat2exp s n x = if (even n) then xs else (x:xs) where
                           xs = nat2exp (s 'div' 2) (succ x)
```

```haskell
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:

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

The resulting Encoder (nat) is now ready to interoperate with any other Encoder:

```haskell
+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> borrow nat_set succ [1,2,3] [0,1,2,3]
```

1899
3.3 Unfolding Natural Numbers into Bitstrings

This isomorphism is well known, except that it is usually ignored that conventional bit representations of integers need a twist to be mapped one-to-one to arbitrary sequences of 0s and 1s. As the usual binary representation always has 1 as its highest digit, \texttt{nat2bits} will drop this bit, given that the length of the list of digits is (implicitly) known. This brings us a bijection between \texttt{Nat} and the regular language \{0, 1\}^*.

\begin{verbatim}
bits :: Encoder [Nat]
bits = compose (Iso bits2nat nat2bits) nat

nat2bits = drop_last . (to_base 2) . succ where
  drop_last bs = genericTake ((genericLength bs)-1) bs where
  to_base base n = d :
    (if q==0 then [] else (to_base base q)) where
    (q,d) = quotRem n base

bits2nat bs = from_base 2 (bs ++ [1]))-1 where
  from_base base [ ] = 0
  from_base base (x:xs) | x>0 & x<base = x=base*(from_base base xs)

The following examples show two conversion operations and bits borrowing a multiplication operation from nat.

*ISO* as bits nat 42
[1,1,0,1,0,0]
*ISO* as nat bits [1,1,0,1,0]
42
*ISO* borrow2 (with nat bits) (=) [1,1,0] [1,0,1,1]
[1,0,0,1,0,0,0,0]

The reader might notice at this point that we have already made full circle - as bitstrings can be seen as instances of finite sequences. Monomorphisms (injective functions) with wider and wider ranges can be generated using the fact that one of the representations is information theoretically “denser” (bitstrings in this case) than the other, for a given range:

*ISO* as bits fun [1,1]
[1,1,0]
*ISO* as bits fun $ as bits fun [1,1]
[1,1,0,1]
*ISO* as bits fun $ as bits fun $ as bits fun [1,1]
[1,1,0,1,1,0]

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 \texttt{anamorphism} mechanism (an \texttt{unfold} operation), while the ranking operation is seen as an instance of the corresponding catamorphism (a \texttt{fold} operation) \cite{8,13}. Together they form a mixed transformation or \texttt{hylo}morphism. We will use such hylomorphisms to lift isomorphisms between lists and natural numbers to isomorphisms between a “self-similar” Hereditarily Finite data type and natural numbers. In particular we will derive Ackerman’s (fairly famous) 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 \[].

\begin{verbatim}
data T = H Ts deriving (Eq, Ord, Read, Show)
type Ts = [T]
\end{verbatim}

The two sides of our hylomorphism are parameterized by two transformations \texttt{f} and \texttt{g} forming an isomorphism \texttt{Iso f g}

\begin{verbatim}
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
\end{verbatim}

Note that \texttt{unrank} and \texttt{rank} work on \texttt{Nat} in cooperation with with \texttt{unranks} and \texttt{ranks} working on \texttt{[Nat]}. We can combine together such anamorphism+catamorphism pairs into isomorphisms as a mechanism to create new bijections involving hereditarily finite data types.

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

\begin{verbatim}
hylons :: Iso b [b] \rightarrow Iso Ts [b]
hylons (Iso f g) = Iso (ranks g) (unranks f)
\end{verbatim}

4.1.1 Hereditarily Finite Sets

Hereditarily Finite Sets will be represented as an Encoder for the tree type \texttt{T}:

\begin{verbatim}
hfs :: Encoder T
hfs = compose (hylo nat_set) nat
\end{verbatim}

The \texttt{hfs} Encoder can now borrow operations from sets or natural numbers as follows:

\begin{verbatim}
hfs_union = borrow2 (with set hfs) union
hfs_succ = borrow (with nat hfs) succ
hfs_pred = borrow (with nat hfs) pred
\end{verbatim}

\begin{verbatim}
*ISO* hfs_succ (H [ ])
H [H [ ]]
*ISO* hfs_union (H [H [ ]]) (H [ ])
H [H [ ]]
\end{verbatim}

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

\begin{verbatim}
*ISO* as hfs nat 42
H [H [H [H [H [H []]]]]]
\end{verbatim}

One can notice that we have just derived as yet another “free algorithm” Ackermann’s encoding from Hereditarily Finite Sets to Natural Numbers:

\begin{verbatim}
f(x) = if x = {} then 0 else \sum_{a \in x} 2^{f(a)}
\end{verbatim}

together with its inverse:

\begin{verbatim}
ackermann = as nat hfs
inverse_ackermann = as hfs nat
\end{verbatim}

4.1.2 Hereditarily Finite Functions

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

\begin{verbatim}
hff :: Encoder T
hff = compose (hylo nat) nat
\end{verbatim}
4.2 A Hylomorphism with Atoms/Urelements

A similar construction can be carried out for the more practical case when Atoms (Urelements in Set Theory parlance) are present. Hereditarily Finite Sets with Urelements are represented as generic multiway trees with a leaf type holding urelements/atoms:

data UT a = A a | F (UTs a) deriving (Eq, Ord, Read, Show)
type UTs a = [UT a]

Atoms will be mapped to natural numbers in [0..ulimit-1]. Assuming for simplicity that ulimit is fixed, we denote this set A and denote UT the set of trees of type UT with atoms in A.

Unranking. As an adaptation of the unfold operation, natural numbers will be mapped to elements of UT with a generic higher order function unrankU f, defined from Nat to UT, parameterized by the natural number ulimit and the transformer function f:

unrankU _ n | n ≥ 0 && n < ulimit = A n
unrankU f n = F (unrankU f (f (n-ulimit)))
unranksU f ns = map (unrankU f) ns

Ranking. Similarly, as an adaptation of fold, a generic inverse mapping rankU is defined as:

rankU _ (A n) | n ≥ 0 && n < ulimit = n
rankU g (F ts) = ulimit-g (ranksU g ts))
rankU g ts = map (rankU g) ts

where rankU g maps trees to numbers and ranksU g maps lists of trees to lists of numbers.

The following proposition describes conditions under which rankU and unrankU can be used to lift isomorphisms between [Nat] and Nat to isomorphisms involving hereditarily finite structures:

**Proposition 2.** If the transformer function f : Nat → [Nat] is a bijection with inverse g, such that n ≥ ulimit ∧ f(n) = [n0,...,ni,...,nk] ⇒ ni < n, then unrank f is a bijection from Nat to UT, with inverse rank g and the recursive computations defining both functions terminate in a finite number of steps.

Proof. Note that unrankU terminates as its arguments strictly decrease at each step and rankU terminates as leaf nodes are eventually reached. That both are bijections, follows by induction on the structure of Nat and T, given that map preserves bijections and that adding/subtracting ulimit ensures that encodings of atoms and sets never overlap.
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.

5.4 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 Generic Programming context [11] 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 [5]), 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.

6. RELATED WORK

The closest reference on encapsulating bijections as a Haskell data type is [1] and Connan Eliot’s composable bijections module [3] where, in a more complex setting, Arrows [7] are used as the underlying abstractions. While our Iso data type is similar to the Bij data type in [3] and BiArrow concept of [1], 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 G"odel numberings [4, 6] associated to formulae. Together with their inverse unranking functions they are also used in combinatorial generation algorithms [12, 10]. 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 [16, 9]. Computational and Data Representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in [15, 14].

7. CONCLUSION

We have shown the expressiveness of Haskell as a meta-language for executable mathematics, by describing encodings for functions and finite sets in a uniform framework as data type isomorphisms with a group structure. 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 as well as a generalization of Ackermann’s encoding to Hereditarily Finite Sets with Urelements.

8. REFERENCES

[1] A. Alimarine, S. Smetters, 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, pages 86–97, New York, NY, USA, 2005. ACM Press.

[2] K. Claessen and J. Hughes. Testing monadic code with quickcheck. SIGPLAN Notices, 37(12):47–59, 2002.

[3] Connan Eliot. Data.Bijections Haskell Module. http://haskell.org/haskellwiki/TypeCompose.

[4] K. G"odel. Uber formal unentscheidbare S"atze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik, 38:173–198, 1931.

[5] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. Commun. ACM, 51(8):91–100, 2008.

[6] J. Hartmanis and T. P. Baker. On simple goedel numberings and translations. In J. Loeckx, editor, ICALP, volume 14 of Lecture Notes in Computer Science, pages 301–316. Springer, 1974.

[7] J. Hughes. Generalizing Monads to Arrows. Science of Computer Programming 37, pp. 67-111, May 2000.

[8] G. Hutton. A Tutorial on the Universality and Expressiveness of Fold. J. Funct. Program., 9(4):355–372, 1999.

[9] R. Kaye and T. L. Wong. On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume, 48(4):497–510, 2007.

[10] D. Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/~knuth/taocp.html.

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

[12] C. Martinez and X. Molinero. Generic algorithms for the generation of combinatorial objects. In B. Rovan and P. Vojtás, editors, MFCS, volume 2747 of Lecture Notes in Computer Science, pages 572–581. Springer, 2003.

[13] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In FPCA, pages 324–333, 1995.

[14] L. C. Paulson. A Concrete Final Coalgebra Theorem for ZF Set Theory. In P. Dybjer, B. Nordström, and J. M. Smith, editors, TYPES, volume 996 of Lecture Notes in Computer Science, pages 120–139. Springer, 1994.

[15] C. Piazza and A. Policriti. Ackermann Encoding, Bisimulations, and OBDDs. TPLP, 4(5-6):695–718, 2004.

[16] M.-o. Takahashi. A Foundation of Finite Mathematics. Publ. Res. Inst. Math. Sci., 12(3):577–708, 1976.