Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cs.unt.edu

Abstract

This paper is an exploration in a functional programming framework of isomorphisms between elementary data types (natural numbers, sets, finite functions, permutations binary decision diagrams, graphs, hypergraphs, parenthesis languages, dyadic rationals etc.) 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.

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 (Knuth 2006).

In 440 lines of Haskell code we cover 20 data types and, through the use of the embedded combinator language, provide 380 distinct bijective encodings between them.

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/fISO.zip`

Keywords *Haskell data representations, computational mathematics, ranking/unranking, Ackermann encoding, hereditarily finite sets functions and permutations, encodings of binary decision diagrams*

1. Introduction

Analogical/metaphorical thinking routinely shifts entities and operations from a field to another hoping to uncover similarities in representation or use (Lakoff and Johnson 1980).

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 (Cook 2004; Cook and Urquhart 1993).

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.

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 (Gödel 1931; Hartmanis and Baker 1974) 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.

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:

$$\begin{tikzcd} X \arrow[r, shift left, "f = g^{-1}"] & Y \arrow[l, shift left, "g = f^{-1}"] \end{tikzcd}$$

```
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
```

We can now formulate *laws* about isomorphisms that can be used to test correctness of implementations with tools like QuickCheck (Claessen and Hughes 2002).

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

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
```

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

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
    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 (nat) 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
```

[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.

```
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]
```

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, nat2bits will drop this bit, given that the length of the list of digits is (implicitly) known. This brings us a bijection between Nat and the regular language $\{0,1\}^*$.

```
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

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

from_base base [] = 0
from_base base (x:xs) | x≥0 && x<base =
   x+base*(from_base base xs)
```

Note also that, strictly speaking, this is only an isomorphism when bits are in $\{0,1\}$, therefore we shall assume this constraint as a *law* governing this Encoder. 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]
*ISO> as nat bits [1,1,0,1,0]
42
*ISO> borrow2 (with nat bits) (*) [1,1,0] [1,0,1,1]
[1,0,0,1,1,0,0,0]
```

The reader might notice at this point that we have made full circle again - as bitstrings can be seen as instances of finite sequences. Injective functions that are not surjections with wider and wider gaps can be generated by composing the as combinators:

```
*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]
```

3.4 Functional Binary Numbers

Church numerals are well known as a functional representation for Peano arithmetic. While benefiting from lazy evaluation, they implement a form of unary arithmetic that uses $O(n)$ space to represent n. This suggest devising a functional representation that mimics binary numbers. We will do this following the model described in subsection 3.3 to provide an isomorphism between Nat and the functional equivalent of the regular language $\{0,1\}^*$. We will view each bit as a $Nat \rightarrow Nat$ transformer:

```
b x = pred x -- begin
o x = 2*x+0  -- bit 0
i x = 2*x+1  -- bit 1
e = 1        -- end
```

As the following example shows, composition of functions $o$ and $i$ closely parallels the corresponding bitlists:

```
*ISO> b$i$o$o$i$i$o$i$i$i$i$e
2008
*ISO> as bits nat 2008
[1,0,0,1,1,0,1,1,1,1]
```

We can follow the same model with and abstract data type:

```
data D = E | O D | I D deriving (Eq,Ord,Show,Read)
data B = B D deriving (Eq,Ord,Show,Read)
```

from which we can generate functional bitstrings as an instance of a *fold* operation:

```
funbits2nat :: B → Nat
funbits2nat = bfold b o i e


bfold fb fo fi fe (B d) = fb (dfold d) where
  dfold E = fe
  dfold (O x) = fo (dfold x)
  dfold (I x) = fi (dfold x)
```

Dually, we can reverse the effect of the functions b, o, i, e as:

```
b' x = succ x
o' x | even x = x `div` 2
i' x | odd x = (x-1) `div` 2
e' = 1
```

and define a generator for our data type as an *unfold* operation:

```
nat2funbits :: Nat → B
nat2funbits = bunfold b' o' i' e'

bunfold fb fo fi fe x = B (dunfold (fb x)) where
  dunfold n | n==fe = E
  dunfold n | even n = O (dunfold (fo n))
  dunfold n | odd n = I (dunfold (fi n))
```

The two operations form an isomorphism:

```
*ISO> funbits2nat (B$I$O$O$I$I$O$I$I$I$I$E)
2008
*ISO> nat2funbits it
B (I (O (O (I (I (O (I (I (I (I E))))))))))
```

We can define our Encoder as follows:

```
funbits :: Encoder B
funbits = compose (Iso funbits2nat nat2funbits) nat
```

Arithmetic operations can now be performed directly on this representation. For instance, one can define a successor function as:

```
bsucc (B d) = B (dsucc d) where
  dsucc E = O E
  dsucc (O x) = I x
  dsucc (I x) = O (dsucc x)
```

Equivalently arithmetics can be borrowed from Nat:

```
*ISO> bsucc (B$I$O$O$I$I$O$I$I$I$I$E)
B (O (I (O (I (I (O (I (I (I (I E))))))))))
*ISO> as nat funbits it
2009
*ISO> borrow (with nat funbits)
            succ (B$I$O$O$I$I$O$I$I$I$I$E)
B (O (I (O (I (I (O (I (I (I (I E))))))))))
*ISO> as nat funbits it
2009
```

While Haskell's C-based arbitrary length integers are likely to be more efficient for most operations, this representation, like Church numerals, has the benefit of supporting partial or delayed computations through lazy evaluation.

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) (Hutton 1999; Meijer and Hutton 1995). 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)
```

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 (Ackermann 1937; Piazza and Policriti 2004) 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:

```
*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.

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`:

```
ulimit = 4
```

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

ranksU 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 $f : Nat \rightarrow [Nat]$ is a bijection with inverse g, such that $n \geq ulimit \wedge f(n) = [n_0, ...n_i, ...n_k] \Rightarrow n_i < n$, 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 UT, given that `map` preserves bijections and that adding/subtracting $ulimit$ ensures that encodings of atoms and sets never overlap. □

The resulting hylomorphisms are defined as previously:

```
hyloU (Iso f g) = Iso (rankU g) (unrankU f)
hylosU (Iso f g) = Iso (ranksU g) (unranksU f)
```

An Encoder for Hereditarily Finite Sets with Urelements is defined as:

```
uhfs :: Encoder (UT Nat)
uhfs = compose (hyloU nat_set) nat
```

Note that this encoder provides a generalization of Ackermann's mapping, to Hereditarily Finite Sets with Urelements in $[0..u-1]$ defined as:

$$f_u(x) = \texttt{if } x < u \texttt{ then } x \texttt{ else } u + \sum_{a \in x} 2^{f_u(a)}$$

A similar Encoder for Hereditarily Finite Functions with Urelements is defined as:

```
uhff :: Encoder (UT Nat)
uhff = compose (hyloU nat) nat
```

5. Permutations and Hereditarily Finite Permutations

We have seen that finite sets and their derivatives represent information in an order independent way, focusing exclusively on information content. We will now look at data representations that focus exclusively on order in a content independent way - finite permutations and their hereditarily finite derivatives.

To obtain an encoding for finite permutations we will first review a ranking/unranking mechanism for permutations that involves an unconventional numeric representation, *factoradics*.

5.1 The Factoradic Numeral System

The factoradic numeral system (Knuth 1997) replaces digits multiplied by a power of a base n with digits that multiply successive values of the factorial of n. In the increasing order variant `fr` the first digit d_0 is 0, the second is $d_1 \in \{0, 1\}$ and the n-th is $d_n \in [0..n]$. For instance, $42 = 0 * 0! + 0 * 1! + 0 * 2! + 3 * 3! + 1 * 4!$. The left-to-right, decreasing order variant `fl` is obtained by reversing the digits of `fr`.

```
fr 42
  [0,0,0,3,1]
rf [0,0,0,3,1]
  42
fl 42
  [1,3,0,0,0]
lf [1,3,0,0,0]
  42
```

The function `fr` generating the factoradics of n, right to left, handles the special case of 0 and calls a local function `f` which recurses and divides with increasing values of n while collecting digits with `mod`:

```
fr 0 = [0]
fr n = f 1 n where
   f _ 0 = []
   f j k = (k 'mod' j) :
           (f (j+1) (k 'div' j))
```

The function `fl`, with digits left to right is obtained as follows:

```
fl = reverse . fr
```

The function `lf` (inverse of `fl`) converts back to decimals by summing up results while computing the factorial progressively:

```
rf ns = sum (zipWith (*) ns factorials) where
  factorials=scanl (*) 1 [1..]
```

Finally, `lf`, the inverse of `fl` is obtained as:

```
lf = rf . reverse
```

5.2 Ranking and unranking permutations of given size with Lehmer codes and factoradics

The Lehmer code of a permutation f is defined as the sequence $l(f) = (l_1(f) \ldots l_i(f) \ldots l_n(f))$ where $l_i(f)$ is the number of elements of the set $\{j > i | f(j) < f(i)\}$ (Mantaci and Rakotondrajao 2001).

Proposition 3. *The Lehmer code of a permutation determines the permutation uniquely.*

The function `perm2nth` computes a `rank` for a permutation `ps` of `size>0`. It starts by first computing its Lehmer code `ls` with `perm2lehmer`. Then it associates a unique natural number `n` to `ls`, by converting it with the function `lf` from factoradics to decimals. Note that the Lehmer code `Ls` is used as the list of digits in the factoradic representation.

```
perm2nth ps = (l,lf ls) where
  ls=perm2lehmer ps
  l=genericLength ls

perm2lehmer [] = []
perm2lehmer (i:is) = l:(perm2lehmer is) where
  l=genericLength [j|j←is,j<i]
```

The function `nat2perm` provides the matching *unranking* operation associating a permutation `ps` to a given `size>0` and a natural number `n`. It generates the n-th permutation of given size.

```
nth2perm (size,n) =
  apply_lehmer2perm (zs++xs) [0..size-1] where
    xs=fl n
    l=genericLength xs
    k=size-l
    zs=genericReplicate k 0
```

The following function extracts a permutation from a “digit” list in factoradic representation.

```
apply_lehmer2perm [] [] = []
apply_lehmer2perm (n:ns) ps@(x:xs) =
   y : (apply_lehmer2perm ns ys) where
   (y,ys) = pick n ps

pick i xs = (x,ys++zs) where
  (ys,(x:zs)) = genericSplitAt i xs
```

Note also that `apply_lehmer2perm` is used this time to reconstruct the permutation `ps` from its Lehmer code, which in turn is computed from the permutation’s factoradic representation.

One can try out this bijective mapping as follows:

```
nth2perm (5,42)
  [1,4,0,2,3]
perm2nth [1,4,0,2,3]
  (5,42)
nth2perm (8,2008)
  [0,3,6,5,4,7,1,2]
perm2nth [0,3,6,5,4,7,1,2]
  (8,2008)
```

5.3 A bijective mapping from permutations to natural numbers

Like in the case of BDDs, one more step is needed to to extend the mapping between permutations of a given length to a bijective mapping from/to Nat: we will have to "shift towards infinity" the starting point of each new bloc of permutations in Nat as permutations of larger and larger sizes are enumerated.

First, we need to know by how much - so we compute the sum of all factorials up to $n!$.

```
sf n = rf (genericReplicate n 1)
```

This is done by noticing that the factoradic representation of [0,1,1,..] does just that.

What we are really interested into, is decomposing `n` into the distance to the last sum of factorials smaller than `n`, `n_m` and the its index in the sum, `k`.

```
to_sf n = (k,n-m) where
  k=pred (head [x|x←[0..],sf x>n])
  m=sf k
```

Unranking of an arbitrary permutation is now easy - the index `k` determines the size of the permutation and `n-m` determines the rank. Together they select the right permutation with `nth2perm`.

```
nat2perm 0 = []
nat2perm n = nth2perm (to_sf n)
```

Ranking of a permutation is even easier: we first compute its size and its rank, then we shift the rank by the sum of all factorials up to its size, enumerating the ranks previously assigned.

```
perm2nat ps = (sf l)+k where
  (l,k) = perm2nth ps
```

It works as follows:

```
nat2perm 2008
  [0,2,3,1,4]
perm2nat [0,2,3,1,4]
  42
nat2perm 2008
  [1,4,3,2,0,5,6]
perm2nat [1,4,3,2,0,5,6]
  2008
```

We can now define the Encoder as:

```
perm :: Encoder [Nat]
perm = compose (Iso perm2nat nat2perm) nat
```

The Encoder works as follows:

```
*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
```

5.4 Hereditarily Finite Permutations

By using the generic `unrank` and `rank` functions defined in section 4 we can extend the `nat2perm` and `perm2nat` to encodings of Hereditarily Finite Permutations (HFP).

```
nat2hfp = unrank nat2perm
hfp2nat = rank perm2nat
```

The encoding works as follows:

```
nat2hfp 42
  F [F [],F [F [],F [F []]],
    F [F [F []],F []],F [F []],
    F [F [],F [F []],F [F [],F [F []]]]]
hfp2nat it
  42
```

We can now define the Encoder as:

```
hfp :: Encoder T
hfp = compose (Iso hfp2nat nat2hfp) nat
```

The Encoder works as follows:

```
*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
```

6. Pairing/Unpairing

A *pairing* function is an isomorphism $f : Nat \times Nat \rightarrow Nat$. Its inverse is called *unpairing*.

We will introduce here an unusually simple pairing function (also mentioned in (Pigeon 2001), 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.

```
data Nat2 = P Nat Nat deriving (Eq,Ord,Read,Show)

bitpair ::  Nat2 → Nat
bitpair (P 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 = P (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
  (60,26)
```

```
-- 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
--   60:[      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
P 60 26
*ISO> as nat nat2 (P 60 26)
2008
```

7. Pairing Functions and Encodings of Binary Decision Diagrams

We will show in this section that a Binary Decision Diagram (BDD) representing the same logic function as an n-variable 2^n bit truth table can be obtained by applying bitunpair recursively to tt. More precisely, we will show that applying this *unfolding* operation results in a complete binary tree of depth n representing a BDD that returns tt when evaluated applying its boolean operations.

The binary tree type BT has the constants B0 and B1 as leaves representing the boolean values 0 and 1. Internal nodes (that will represent if-then-else decision points), will be marked with the constructor D. We will also add integers to represent logic variables, ordered identically in each branch, as first arguments of D. The two other arguments will be subtrees that represent THEN and ELSE branches:

```
data BT a = B0 | B1 | D a (BT a) (BT a)
             deriving (Eq,Ord,Read,Show)
```

The constructor BDD wraps together a tree of type BT and the number of logic variables occurring in it.

```
data BDD a = BDD a (BT a) deriving (Eq,Ord,Read,Show)
```

7.1 Unfolding natural numbers to binary trees with bitunpair

The following functions apply bitunpair recursively, on a Natural Number tt, seen as an n-variable 2^n bit truth table, to build a complete binary tree of depth n, that we will represent using the BDD data type.

```
unfold_bdd :: Nat2 → BDD Nat
unfold_bdd (P n tt) = BDD n bt where
  bt=if tt<max then shf bitunpair n tt
     else error
       ("unfold_bdd: last arg "++ (show tt)++
       " should be < " ++ (show max))
     where max = 2^2^n

  shf _ n 0 | n<1 =  B0
  shf _ n 1 | n<1 =  B1
  shf f n tt = D k (shf f k tt1) (shf f k tt2) where
    k=pred n
    P tt1 tt2=f tt
```

The following examples show results returned by unfold_bdd for the 2^{2^n} truth tables associated to n variables, for $n = 2$:

```
BDD 2 (D 1 (D 0 B0 B0) (D 0 B0 B0))
BDD 2 (D 1 (D 0 B1 B0) (D 0 B0 B0))
BDD 2 (D 1 (D 0 B0 B0) (D 0 B1 B0))
...
BDD 2 (D 1 (D 0 B1 B1) (D 0 B1 B1))
```

Note that no boolean operations have been performed so far and that we still have to prove that such trees actually represent BDDs associated to truth tables.

7.2 Folding binary trees to natural numbers with bitpair

One can "evaluate back" the binary tree of data type BDD, by using the pairing function bitpair. The inverse of unfold_bdd is implemented as follows:

```
fold_bdd :: BDD Nat → Nat2
fold_bdd (BDD n bt) =
  P n (rshf bitpair bt) where
    rshf rf B0 = 0
    rshf rf B1 = 1
    rshf rf (D _ l r) =
      rf (P (rshf rf l) (rshf rf r))
```

Note that this is a purely structural operation and that integers in first argument position of the constructor D are actually ignored.

The two bijections work as follows:

```
*ISO>unfold_bdd (P 3 42)
  BDD 3
    (D 2
      (D 1 (D 0 B0 B0)
           (D 0 B0 B0))
      (D 1 (D 0 B1 B1)
           (D 0 B1 B0)))

*ISO>fold_bdd it
  42
```

7.3 Boolean Evaluation of BDDs

Practical uses of BDDs involve reducing them by sharing nodes and eliminating identical branches (Bryant 1986). Note that in this case bdd2nat might give a different result as it computes different pairing operations. Fortunately, we can try to fold the binary tree back to a natural number by evaluating it as a boolean function.

The function eval_bdd describes the BDD evaluator:

```
eval_bdd (BDD n bt) = eval_with_mask (bigone n) n bt

eval_with_mask m _ B0 = 0
eval_with_mask m _ B1 = m
eval_with_mask m n (D x l r) =
  ite_ (var_mn m n x)
       (eval_with_mask m n l)
       (eval_with_mask m n r)

var_mn mask n k = mask `div` (2^(2^(n-k-1))+1)
bigone nvars = 2^2^nvars - 1
```

The *projection functions* var_mn can be combined with the usual bitwise integer operators, to obtain new bitstring truth tables, encoding all possible value combinations of their arguments. Note that the constant 0 evaluates to 0 while the constant 1 is evaluated as $2^{2^n}-1$ by the function bigone.

The function ite_ used in eval_with_mask implements the boolean function `if x then t else e` using arbitrary length bitvector operations:

```
ite_ x t e = ((t `xor` e).&.x) `xor` e
```

As the following example shows, it turns out that boolean evaluation eval_bdd *faithfully emulates* fold_bdd*!*

```
*ISO> unfold_bdd (P 3 42)
BDD 3 (D 2 (D 1 (D 0 B0 B0) (D 0 B0 B0))
           (D 1 (D 0 B1 B1) (D 0 B1 B0)))
*ISO> eval_bdd it
42
```

7.4 The Equivalence

We will now state the surprising (and new!) result that boolean evaluation and structural transformation with repeated application of *pairing* produce the same result:

Proposition 4. *The complete binary tree of depth n, obtained by recursive applications of* bitunpair *on a truth table tt computes an (unreduced) BDD, that, when evaluated, returns the truth table, i.e.*

$$fold\_bdd \circ unfold\_bdd \equiv id \tag{1}$$

$$eval\_bdd \circ unfold\_bdd \equiv id \tag{2}$$

Proof. The function unfold_bdd builds a binary tree by splitting the bitstring $tt \in [0..2^n-1]$ up to depth n. Observe that this corresponds to the Shannon expansion (Shannon 1993) of the formula associated to the truth table, using variable order $[n-1,...,0]$. Observe that the effect of bitunpair is the same as

- the effect of `var_mn m n (n-1)` acting as a mask selecting the left branch, and
- the effect of its complement, acting as a mask selecting the right branch.

Given that 2^n is the double of 2^{n-1}, the same invariant holds at each step, as the bitstring length of the truth table reduces to half. □

We can thus assume from now on, that the BDD data type defined in section 7 actually represents BDDs mapped one-to-one to truth tables given as natural numbers.

8. Ranking and Unranking of BDDs

One more step is needed to extend the mapping between $BDDs$ with n variables to a bijective mapping from/to Nat: we will have to "shift towards infinity" the starting point of each new block[2] of BDDs in Nat as BDDs of larger and larger sizes are enumerated.

First, we need to know by how much - so we will count the number of boolean functions with up to n variables.

```
bsum 0 = 0
bsum n | n>0 = bsum1 (n-1)

bsum1 0 = 2
bsum1 n | n>0 = bsum1 (n-1)+ 2^2^n
```

The stream of all such sums can now be generated as usual[3]:

```
bsums = map bsum [0..]

*ISO> genericTake 7 bsums
  [0,2,6,22,278,65814,4295033110]
```

What we are really interested into, is decomposing n into the distance n-m to the last bsum m smaller than n, and the index that generates the sum, k.

```
to_bsum n = (k,n-m) where
  k=pred (head [x|x←[0..],bsum x>n])
  m=bsum k
```

Unranking of an arbitrary BDD is now easy - the index k determines the number of variables and n-m determines the rank. Together they select the right BDD with unfold_bdd and bdd.

```
nat2bdd n = unfold_bdd (P k n_m)
  where (k,n_m)=to_bsum n
```

Ranking of a BDD is even easier: we shift its rank within the set of BDDs with nv variables, by the value (bsum nv) that counts the ranks previously assigned.

```
bdd2nat bdd@(BDD nv _) =  ((bsum nv)+tt) where
  P _ tt =fold_bdd bdd
```

As the following example shows bdd2nat implements the inverse of nat2bdd.

```
*ISO> nat2bdd 42
BDD 3 (D 2 (D 1 (D 0 B0 B1) (D 0 B1 B0))
           (D 1 (D 0 B0 B0) (D 0 B0 B0)))
*ISO> bdd2nat it
42
```

This provides the Encoder:

[2] defined by the same number of variables

[3] bsums is sequence A060803 in The On-Line Encyclopedia of Integer Sequences, http://www.research.att.com/~njas/sequences

```
pbdd :: Encoder (BDD Nat)
pbdd = compose (Iso bdd2nat nat2bdd) nat
```

working as follows:

```
*ISO> as pbdd nat 2008
BDD 4 (D 3 (D 2 B0 (D 1 (D 0 B0 B1) B1))
      (D 2 (D 1 (D 0 B1 B1) B0) (D 1 B0 B1)))
*ISO> as nat pbdd it
2008
```

We can now repeat the *ranking* function construction for eval_bdd:

```
ev_bdd2nat bdd@(BDD nv _) = (bsum nv)+(eval_bdd bdd)
```

We can confirm that ev_bdd2nat also acts as an inverse to nat2bdd:

```
*ISO> ev_bdd2nat (nat2bdd 2008)
2008
```

We obtain the Encoder:

```
bdd :: Encoder (BDD Nat)
bdd = compose (Iso ev_bdd2nat nat2bdd) nat
```

working as follows:

```
*ISO> as bdd nat 2008
BDD 4 (D 3 (D 2 (D 1 (D 0 B0 B0) (D 0 B0 B0))
                (D 1 (D 0 B0 B1) (D 0 B1 B0)))
           (D 2 (D 1 (D 0 B1 B1) (D 0 B0 B0))
                (D 1 (D 0 B0 B0) (D0 B1 B0))))
*ISO> as nat bdd it
2008
```

This result can be seen as an intriguing isomorphism between boolean, arithmetic and symbolic computations.

8.1 Reducing the $BDDs$

We will sketch here a simplified reduction mechanism for BDDs eliminating identical branches. Note that the general mechanism involves DAGs and provides also node sharing (Bryant 1986).

The function bdd_reduce reduces a BDD by collapsing identical left and right subtrees, and the function bdd associates this reduced form to $n \in Nat$.

```
bdd_reduce (BDD n bt) = BDD n (reduce bt) where
  reduce B0 = B0
  reduce B1 = B1
  reduce (D _ l r) | l == r = reduce l
  reduce (D v l r) = D v (reduce l) (reduce r)

unfold_rbdd = bdd_reduce . unfold_bdd
```

The results returned by unfold_rbdd for n=2 are:

```
  BDD 2 (C 0)
  BDD 2 (D 1 (D 0 (C 1) (C 0)) (C 0))
  BDD 2 (D 1 (C 0) (D 0 (C 1) (C 0)))
  BDD 2 (D 0 (C 1) (C 0))
  ...
  BDD 2 (D 1 (D 0 (C 0) (C 1)) (C 1))
  BDD 2 (C 1)
```

We can now define the *unranking* operation on reduced BDDs

```
nat2rbdd = bdd_reduce . nat2bdd
```

and obtain the Encoder

```
rbdd :: Encoder (BDD Nat)
rbdd = compose (Iso ev_bdd2nat nat2rbdd) nat
```

working as follows

```
*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
```

To be able to compare its space complexity with other representations we will define a size operation on a BDD as follows:

```
bdd_size (BDD _ t) = 1+(size t) where
  size B0 = 1
  size B1 = 1
  size (D _ l r) = 1+(size l)+(size r)
```

9. Generalizing BDD ranking/unrankig functions

9.1 Encoding BDDs with Arbitrary Variable Order

While the encoding built around the equivalence described in Prop. 4 between bitwise pairing/unpairing operations and boolean decomposition is arguably as simple and elegant as possible, it is useful to parametrize BDD generation with respect to an arbitrary variable order. This is of particular importance when using BDDs for circuit minimization, as different variable orders can make circuit sizes flip from linear to exponential in the number of variables (Bryant 1986).

Given a permutation of n variables represented as natural numbers in $[0..n-1]$ and a truth table $tt \in [0..2^{2^n}-1]$ we can define:

```
to_bdd vs tt | 0≤tt && tt ≤ m =
  BDD n (to_bdd_mn vs tt m n) where
    n=genericLength vs
    m=bigone n
to_bdd _ tt = error
   ("bad arg in to_bdd⇒" ++ (show tt))
```

where the function to_bdd_mn recurses over the list of variables vs and applies Shannon expansion (Shannon 1993), expressed as bitvector operations. This computes branches $f1$ and $f0$, to be used as then and else parts, when evaluating back the BDD to a truth table with if-the-else functions.

```
to_bdd_mn []       0 _ _ = B0
to_bdd_mn []       _ _ _ = B1
to_bdd_mn (v:vs) tt m n = D v l r where
  cond=var_mn m n v
  f0= (m ‘xor‘ cond) .&. tt
  f1= cond .&. tt
  l=to_bdd_mn vs f1 m n
  r=to_bdd_mn vs f0 m n
```

Proposition 5. *The function* `to_bdd` *computes an (unreduced) BDD corresponding having truth table encoded as a natural number* `tt` *for variable order* `vs`.

We can reduce the resulting BDDs, and convert back from BDDs and reduced BDDs to truth tables with boolean evaluation:

```
to_rbdd vs tt = bdd_reduce (to_bdd vs tt)
from_bdd bdd = eval_bdd bdd
```

We can obtain BDDs and reduced BDDs of various sizes as follows:

```
*ISO> as perm nat 5
[0,2,1]
*ISO> to_bdd (as perm nat 5) 42
BDD 3 (D 0 (D 2 (D 1 B0 B0) (D 1 B1 B1))
           (D 2 (D 1 B0 B0) (D 1 B1 B0)))
*ISO> to_rbdd (as perm nat 5) 42
BDD 3 (D 0 (D 2 B0 B1) (D 2 B0 (D 1 B1 B0)))
FISO> to_rbdd (as perm nat 8) 42
BDD 3 (D 2 B0 (D 0 B1 (D 1 B1 B0)))
ISO> from_bdd it
42
```

Finally, we can, obtain a minimal BDD expressing a logic function of n variables given as a truth table as follows:

```
to_min_bdd n tt = snd $ foldl1 min
 (map (sized_rbdd tt) (all_permutations n)) where
    sized_rbdd tt vs = (bdd_size b,b) where
      b=to_rbdd vs tt

all_permutations n = if n==0 then [[]] else
  [nth2perm (n,i)|i←[0..(factorial n)-1]] where
     factorial n=foldl1 (*) [1..n]
```

As the following examples shows, this can provide an effective multilevel boolean formula minimization up to functions with 6-7 arguments.

```
*ISO> to_min_bdd 3 42
BDD 3 (D 2 B0 (D 0 B1 (D 1 B1 B0)))
*ISO> to_min_bdd 4 2008
BDD 4 (D 0 (D 3 (D 1 B0 B1) (D 2 B0 B1))
           (D 3 (D 1 B1 B0) (D 1 (D 2 B1 B0) B0)))
*ISO> to_min_bdd 3 2008
BDD 7 (D 1 (D 2 (D 4 (D 3 (D 0 (D 5 ...
      ... (D 0 (D 5 B0 B1) B0)))))
*ISO> bdd_size it
110
```

9.2 Multi-Terminal Binary Decision Diagrams (MTBDD)

MTBDDs (Fujita et al. 1997; Ciesinski et al. 2008) are a natural generalization of BDDs allowing non-binary values as leaves. Such values are typically bitstrings representing the outputs of a multi-terminal boolean function, encoded as unsigned integers.

We shall now describe an encoding of $MTBDDs$ that can be extended to ranking/unranking functions, in a way similar to $BDDs$ as shown in section 8.

Our `MTBDD` data type is a binary tree like the one used for $BDDs$, parameterized by two integers `m` and `n`, indicating that an MTBDD represents a function from $[0..n-1]$ to $[0..m-1]$, or equivalently, an n-input/m-output boolean function.

```
data MT a = L a | M a (MT a) (MT a)
             deriving (Eq,Ord,Read,Show)
data MTBDD a = MTBDD a a (MT a) deriving (Show,Eq)
```

The function `to_mtbdd` creates, from a natural number tt representing a truth table, an MTBDD representing functions of type $N \rightarrow M$ with $M = [0..2^m-1], N = [0..2^n-1]$. Similarly to a BDD, it is represented as binary tree of n levels, except that its leaves are in $[0..2^m-1]$.

```
to_mtbdd m n tt = MTBDD m n r where
  mlimit=2^m
  nlimit=2^n
  ttlimit=mlimit^nlimit
  r=if tt<ttlimit
    then (to_mtbdd_ mlimit n tt)
    else error
      ("bt: last arg "++ (show tt)++
      " should be < " ++ (show ttlimit))
```

Given that correctness of the range of `tt` has been checked, the function `to_mtbdd_` applies `bitmerge_unpair` recursively up to depth n, where leaves in range $[0..mlimit-1]$ are created.

```
to_mtbdd_ mlimit n tt|(n<1)&&(tt<mlimit) = L tt
to_mtbdd_ mlimit n tt = (M k l r) where
   P x y=bitunpair tt
   k=pred n
   l=to_mtbdd_ mlimit k x
   r=to_mtbdd_ mlimit k y
```

Converting back from $MTBDDs$ to natural numbers is basically the same thing as for $BDDs$, except that assertions about the range of leaf data are enforced.

```
from_mtbdd (MTBDD m n b) = from_mtbdd_ (2^m) n b

from_mtbdd_ mlimit n (L tt)|(n<1)&&(tt<mlimit)=tt
from_mtbdd_ mlimit n (M _ l r) = tt where
   k=pred n
   x=from_mtbdd_ mlimit k l
   y=from_mtbdd_ mlimit k r
   tt=bitpair (P x y)
```

The following examples show that `to_mtbdd` and `from_mtbdd` are indeed inverses values in $[0..2^n-1] \times [0..2^m-1]$.

```
>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

>mprint (to_mtbdd 2 2) [0..3]
  MTBDD 2 2
    (M 1 (M 0 (L 0) (L 0)) (M 0 (L 0) (L 0)))
  MTBDD 2 2
    (M 1 (M 0 (L 1) (L 0)) (M 0 (L 0) (L 0)))
  MTBDD 2 2
    (M 1 (M 0 (L 0) (L 0)) (M 0 (L 1) (L 0)))
  MTBDD 2 2
    (M 1 (M 0 (L 1) (L 0)) (M 0 (L 1) (L 0)))
```

10. 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.

10.1 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
[P 1 1,P 2 0,P 2 1,P 3 1,P 0 2,P 1 2,P 0 3]
*ISO> as nat digraph it
2008
*ISO> as rbdd digraph
  [P 1 1,P 2 0,P 2 1,P 3 1,P 0 2,P 1 2,P 0 3]
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))))
```

10.2 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
```

11. A mapping to a dense set: Dyadic Rationals in $[0, 1)$

So far our isomorphisms have focused on natural numbers, finite sets and other discrete data types. Dyadic rationals are fractions with denominators restricted to be exponents of 2. They are a *dense* set in $\mathcal{R}$ i.e. they provide arbitrarily close approximations for any real number. An interesting isomorphism to such a set would allow borrowing things like distance or average functions that could have interesting interpretations in symbolic or boolean domains. It also makes sense to pick a bounded subdomain of the dyadic rationals that can be meaningful as the range of probabilistic boolean functions or fuzzy sets. We will build an Encoder for Dyadic Rationals in $[0, 1)$ by providing a bijection from finite sets of natural numbers seen this time as *negative* exponents of 2.

```
dyadic :: Encoder (Ratio Nat)
dyadic = compose (Iso dyadic2set set2dyadic) set
```

The function `set2dyadic` mimics `set2nat` defined in subsection 3.2, except for the use of negative exponents and computation on rationals.

```
set2dyadic :: [Nat] → Ratio Nat
set2dyadic ns = rsum (map nexp2 ns) where
  nexp2 0 = 1%2
  nexp2 n = (nexp2 (n-1))*(1%2)

  rsum [] = 0%1
  rsum (x:xs) = x+(rsum xs)
```

The function `dyadic2set` extracts negative exponents of two from a dyadic rational and it is modeled after `nat2set` defined in subsection 3.2.

```
dyadic2set :: Ratio Nat → [Nat]
dyadic2set n | good_dyadic n = dyadic2exps n 0 where
  dyadic2exps 0 _ = []
  dyadic2exps n x =
    if (d<1) then xs else (x:xs) where
      d = 2*n
      m = if d<1 then d else (pred d)
      xs=dyadic2exps m (succ x)
dyadic2set _ =  error
  "dyadic2set: argument not a dyadic rational"
```

As not all rational numbers are dyadics in $[0, 1)$, the predicate good_dyadic is needed validate the input of dyadic2set. This also ensures that dyadic2set always terminates returning a finite set.

```
good_dyadic kn = (k==0 && n==1)
  || ((kn>0%1) && (kn<1%1) && (is_exp2 n)) where
    k=numerator kn
    n=denominator kn

    is_exp2 1 = True
    is_exp2 n | even n = is_exp2 (n `div` 2)
    is_exp2 n = False
```

Some examples of borrow/lend operations are:

```
dyadic_dist x y = abs (x-y)

dist_for t x y =  as dyadic t
  (borrow2 (with dyadic t) dyadic_dist x y)
dsucc = borrow (with nat dyadic) succ
dplus = borrow2 (with nat dyadic) (+)

dconcat = lend2 dyadic (++)
```

```
*ISO> dist_for nat  6 7
1%2
*ISO> dist_for set [1,2,3] [3,4,5]
21%64
*ISO> dsucc (3%8)
7%8
```

12. Strings and Parenthesis Languages

12.1 Encoding Strings

As strings can be seen just as a notational equivalent of lists of natural numbers we obtain an Encoder immediately as:

```
string :: Encoder String
string = Iso string2fun fun2string

string2fun cs = map (fromIntegral . ord) cs

fun2string ns = map (chr . fromIntegral) ns
```

Note however that this is only an isomorphism within the chr/ord conversion range, therefore we shall assume this constraint as a *law* governing this Encoder.

```
*ISO> as set string "hello"
[104,206,315,424,536]
*ISO> as string set it
"hello"
```

12.2 Encoding a Parenthesis Language

An encoder for a parenthesis language is obtained by combining a parser and writer. As Hereditarily Finite Functions naturally map one-to-one to a parenthesis expression we will choose them as target of the transformers.

```
pars :: Encoder [Char]
pars = compose (Iso pars2hff hff2pars) hff
```

The parser recurses over a string and builds a HFF as follows:

```
pars2hff cs = parse_pars '(' ')' cs

parse_pars l r cs | newcs == [] = t where
  (t,newcs)=pars_expr l r cs

  pars_expr l r (c:cs) | c==l = ((H ts),newcs) where
     (ts,newcs) = pars_list l r cs

  pars_list l r (c:cs) | c==r = ([],cs)
  pars_list l r (c:cs) = ((t:ts),cs2) where
    (t,cs1)=pars_expr l r (c:cs)
    (ts,cs2)=pars_list l r cs1
```

The writer recurses over a HFF and collects matching parenthesis pairs:

```
hff2pars = collect_pars "(" ")" where
  collect_pars l r (H ns) =
    l++
      (concatMap (collect_pars l r) ns)
    ++r
```

The transformations of 42 look as follows:

```
*ISO> as pars nat 42
"((())(())(()))"
*ISO> as hff pars it
H [H [H []],H [H []],H [H []]]
*ISO> as nat hff it
42
```

13. Encoding DNA

We have covered so far encodings for “artificial entities” used in various fields. We will now add an encoding of “natural origin”, DNA bases and strands. While it is an (utterly) simplified model of the real thing, it captures some essential algebraic properties of DNA bases and strands.

We start with a DNA data type, following (Li 1999; Hinze and Sturm 2000):

```
data Base = Adenine | Cytosine | Guanine | Thymine
  deriving(Eq,Ord,Show,Read)

type DNA = [Base]
```

We will encode/decode the DNA base alphabet as follows:

```
alphabet2code Adenine = 0
alphabet2code Cytosine = 1
alphabet2code Guanine = 2
alphabet2code Thymine = 3

code2alphabet 0 = Adenine
code2alphabet 1 = Cytosine
code2alphabet 2 = Guanine
code2alphabet 3 = Thymine
```

The mapping is simply a symbolic variant of conversion to/from base 4:

```
dna2nat  = (from_base 4) . (map alphabet2code)

nat2dna = (map code2alphabet) . (to_base 4)
```

We can now define a decoder for base sequences as follows:

```
dna :: Encoder DNA
dna = compose (Iso dna2nat nat2dna)  nat
```

A first set of DNA operations act on base sequences. The transformation between complements looks as follows:

```
dna_complement :: DNA → DNA
dna_complement = map to_compl where
  to_compl Adenine = Thymine
  to_compl Cytosine = Guanine
  to_compl Guanine = Cytosine
  to_compl Thymine = Adenine
```

Reversing is just list reversal.

```
dna_reverse :: DNA → DNA
dna_reverse = reverse
```

As reversal and complement are independent operations their composition is commutative - we can pick reversing first and then complementing:

```
dna_comprev :: DNA → DNA
dna_comprev = dna_complement . dna_reverse
```

The following examples show interaction of DNA codes with other data types and their operations:

```
*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]
[1,1,1,0,1,0,0,0,0,1,1]
```

Note that each of these DNA operations induces a bijection $Nat \rightarrow Nat$.

Like signed integers, DNA strands have “polarity” - their direction matters:

```
data Polarity =  P3x5 | P5x3
  deriving(Eq,Ord,Show,Read)

data DNAstrand = DNAstrand Polarity DNA
  deriving(Eq,Ord,Show,Read)
```

Polarity can be easily encoded as parity even/odd:

```
strand2nat (DNAstrand polarity strand) =
  add_polarity polarity (dna2nat strand) where
    add_polarity P3x5 x = 2*x
    add_polarity P5x3 x = 2*x-1

nat2strand n =
  if even n
     then DNAstrand P3x5 (nat2dna (n ‘div‘ 2))
     else DNAstrand P5x3 (nat2dna ((n+1) ‘div‘ 2))
```

We can now define an Encoder for DNA strands:

```
dnaStrand :: Encoder DNAstrand
dnaStrand = compose (Iso strand2nat nat2strand) nat
```

Two additional operations lift DNA sequences to strands with polarities:

```
dna_down :: DNA → DNAstrand
dna_down = (DNAstrand P3x5) . dna_complement

dna_up :: DNA → DNAstrand
dna_up = DNAstrand P5x3
```

We can now lend or borrow operations as follows:

```
*ISO> as dnaStrand nat 1234
DNA P3x5 [Cytosine,Guanine,Guanine,Cytosine,Guanine]
*ISO> lend (with dnaStrand nat) succ
   (DNAstrand P5x3 [Adenine,Cytosine,Guanine,Thymine])
DNAstrand P5x3 [Cytosine,Cytosine,Guanine,Thymine]
```

The DoubleHelix is a stable combination of two complementary strands. This built-in redundance protects agains unwanted mutations.

```
data DoubleHelix = DoubleHelix DNAstrand DNAstrand
   deriving(Eq,Ord,Show,Read)

dna_double_helix :: DNA → DoubleHelix
dna_double_helix s =
  DoubleHelix (dna_up s) (dna_down s)
```

We can now generate a double helix from a natural number:

```
*ISO> dna_double_helix (nat2dna 33)
DoubleHelix
  (DNAstrand P5x3 [Cytosine,Adenine,Guanine])
  (DNAstrand P3x5 [Guanine,Thymine,Cytosine])
```

This can be used for generating random instances of double helixes by reusing a random generator for natural numbers.

14. Testing It All

We will now describe a random testing mechanism to validate our Encoders.

While QuickCheck (Claessen and Hughes 2002) provides an elegant general purpose random tester, it would require writing a specific adaptor for each isomorphism. We will describe here a shortcut through a few higher order combinators.

First, we build a simple random generator for *nat*

```
rannat = rand (2^50)

rand :: Nat→Nat→Nat
rand max seed = n where
  (n,g)=
    randomR (0,max) (mkStdGen (fromIntegral seed))
```

We can now design a generic random test for *any* Encoder as follows:

```
rantest :: Encoder t→Bool
rantest t = and (map (rantest1 t) [0..255])

rantest1 t n = x==(visit_as t x) where  x=rannat n

visit_as t = (to nat) . (from t) . (to t) . (from nat)
```

Note that in rantest1, visit_at starts with a random natural number from which it generates its test data of a given type. After testing the encoder, the result is brought back as a natural number that should be the same as the original random number.

We can now implement our tester isotest that in a few seconds goes over of thousands of test cases and aggregates the result with a final and:

```
isotest = and (map rt [0..19])

rt 0 = rantest nat
rt 1 = rantest fun
rt 2 = rantest set
rt 3 = rantest bits
rt 4 = rantest funbits
rt 5 = rantest hfs
rt 6 = rantest hff
rt 7 = rantest uhfs
rt 8 = rantest uhff
rt 9 = rantest nat2
rt 10 = rantest pbdd
rt 11 = rantest bdd
rt 12 = rantest rbdd
rt 13 = rantest digraph
rt 14 = rantest hypergraph
rt 15 = rantest dyadic
rt 16 = rantest string
rt 17 = rantest pars
rt 18 = rantest perm
rt 19 = rantest hfp
```

The empirical correctness test of the "whole enchilada" follows:

```
*ISO> isotest
True
```

suggesting that the probability of having errors in the code described so far is extremely small.

15. 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.

15.1 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> nth bits 42
[1,1,0,1,0]
*ISO> take 3 (stream_of hfs)
[H [],H [H []],H [H [H []]]]
*ISO> take 3 (stream_of bdd)
[BDD 0 B0,BDD 0 B1,BDD 1 (D 0 B0 B0)]
```

15.2 Random 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))

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 []]]]]
*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 (Claessen and Hughes 2002).

15.3 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
*ISO> as fun bits [0,1,0,0,0,0,0,0,0,0,0]
[0,10]
*ISO> as rbdd hfs (H [H [],H [H [],H [H []]],
                     H [H [H []],H [H [H []]]]])
BDD 3 (D 1 B1 B0)
*ISO> as hff bdd (BDD 3 (D 2
        (D 1 (D 0 B1 B0) (D 0 B0 B1))
        (D 1 (D 0 B1 B1) (D 0 B1 B1))))
H [H [],H [H [],H [],H []]]
```

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. Alternatively, lazy representations as provided by functional binary numbers or BDDs, for very large integers encapsulating results of some computations might turn out to be more effective spacewise or timewise.

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

15.4 Experimental Mathematics

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 permutations which are more compact than sets asymptotically

```
*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

One might observe that the same trend applies also
to their hereditarily finite derivatives:
*ISO> size_as hfs 123456789012345678901234567890
627
*ISO> size_as hfp 123456789012345678901234567890
276
*ISO> size_as hff 123456789012345678901234567890
91
```

While confirming or refuting this conjecture is beyond the scope of this paper, the affirmative case would imply, interestingly, that "order" (permutations) has asymptotically higher information density than "content" (sets), and explain why finite functions (that involve both) dominate data representations in various computing fields.

Based on the same experiment, reduced BDDs (especially if one would also implement sharing) also provide relatively compact representations:

```
*ISO> bdd_size $ as bdd
        nat 123456789012345678901234567890
256
*ISO> bdd_size $ as rbdd
        nat 123456789012345678901234567890
144
```

15.5 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 (surprising to us) "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. The cognoscenti might notice that such surprises are not totally unexpected in the world of functional programming. In a different context, they go back as early as Wadler's Free Theorems (Wadler 1989).

15.6 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 (Koza 1992) 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 particular, "biological proven" encodings like DNA strands are likely to provide interesting genotypes implementations. In the context of Software Transaction Memory implementations (like Haskell's STM (Harris et al. 2008)), 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.

16. Related work

The closest reference on encapsulating bijections as a Haskell data type is (Alimarine et al. 2005) and Connan Eliot's composable bijections module (Connan Eliot), where, in a more complex setting, Arrows (Hughes) are used as the underlying abstractions. While our Iso data type is similar to the *Bij* data type in (Connan Eliot) and BiArrow concept of (Alimarine et al. 2005), 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 (Gödel 1931; Hartmanis and Baker 1974) associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms (Martinez and Molinero 2003; Knuth 2006; Ruskey and Proskurowski 1990; Myrvold and Ruskey 2001). 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 (Takahashi 1976; Kaye and Wong 2007; Abian and Lamacchia 1978; Avigad 1997; Kirby 2007; Leontjev and Sazonov 2000). Computational and Data Representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in (Piazza and Policriti 2004; Paulson 1994).

Pairing functions have been used in work on decision problems as early as (Robinson 1950, 1968). A typical use in the foundations of mathematics is (Cégielski and Richard 2001). An extensive study of various pairing functions and their computational properties is presented in (Rosenberg 2002).

Various mappings from natural number encodings to Rational Numbers are described in (Gibbons et al. 2006), also in a functional programming framework.

17. 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. Using higher order combinators a simplified QuickCheck style random testing mechanism has been implemented as an empirical correctness test. 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. We plan to explore in depth in the near future, some of the results that are likely to be of interest in fields ranging from combinatorics and boolean logic to data compression and arbitrary precision numerical computations.

References

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.

Wilhelm Friedrich Ackermann. Die Widerspruchsfreiheit der allgemeinen Mengenlhere. *Mathematische Annalen*, (114):305–315, 1937.

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.

Jeremy Avigad. The Combinatorics of Propositional Provability. In *ASL Winter Meeting*, San Diego, January 1997.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

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.

F. Ciesinski, C. Baier, M. Groesser, and D. Parker. Generating compact MTBDD-representations from Probmela specifications. In *Proc. 15th International SPIN Workshop on Model Checking of Software (SPIN'08)*, 2008.

Koen Claessen and John Hughes. Testing monadic code with quickcheck. *SIGPLAN Notices*, 37(12):47–59, 2002.

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

Stephen Cook. Theories for complexity classes and their propositional translations. In *Complexity of computations and proofs*, pages 1–36, 2004.

Stephen Cook and Alasdair Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63:103–200, 1993.

Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

Jeremy Gibbons, David Lester, and Richard Bird. Enumerating the rationals. *Journal of Functional Programming*, 16(4), 2006. URL http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/rationals.pdf.

K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51 (8):91–100, 2008.

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. ISBN 3-540-06841-

4. URL http://dblp.uni-trier.de/db/conf/icalp/icalp74.html#HartmanisB74.

Thomas Hinze and Monika Sturm. A universal functional approach to dna computing and its experimental practicability. In *Proceedings 6th DIMACS Workshop on DNA Based Computers, held at the University of Leiden, Leiden, The Netherlands, 13 - 17*, pages 257–266, 2000.

John Hughes. Generalizing Monads to Arrows. Science of Computer Programming 37, pp. 67-111, May 2000.

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

Richard Kaye and Tin Lock Wong. On Interpretations of Arithmetic and Set Theory. *Notre Dame J. Formal Logic Volume*, 48 (4):497–510, 2007.

Laurence Kirby. Addition and multiplication of sets. *Math. Log. Q.*, 53(1):52–65, 2007.

Donald Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.

Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0201896842.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

George Lakoff and Mark Johnson. *Metaphors We Live By*. University of Chicago Press, Chicago, IL, USA, 1980.

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. ISBN 3-540-67100-5.

Zhuo Li. Algebraic properties of dna operations. *Biosystems*, 52: 55–61(7), October 1999.

Roberto Mantaci and Fanja Rakotondrajao. A permutations representation that knows what ”eulerian” means. *Discrete Mathematics & Theoretical Computer Science*, 4(2):101–108, 2001.

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.

Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *FPCA*, pages 324–333, 1995.

Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79: 281–284, 2001.

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. ISBN 3-540-60579-7.

Carla Piazza and Alberto Policriti. Ackermann Encoding, Bisimulations, and OBDDs. *TPLP*, 4(5-6):695–718, 2004.

Stephen Pigeon. *Contributions à la compression de données.* Ph.d. thesis, Université de Montréal, Montréal, 2001.

Julia Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950. ISSN 0002-9939.

Julia Robinson. Finite generation of recursively enumerable sets. *Proceedings of the American Mathematical Society*, 19(6): 1480–1486, dec 1968. ISSN 0002-9939.

Arnold L. Rosenberg. Efficient pairing functions - and why you should care. In *IPDPS*. IEEE Computer Society, 2002. ISBN 0-7695-1573-8.

Frank Ruskey and Andrzej Proskurowski. Generating binary trees by transpositions. *J. Algorithms*, 11:68–84, 1990.

Claude E. Shannon. *Claude Elwood Shannon: collected papers*. IEEE Press, Piscataway, NJ, USA, 1993. ISBN 0-7803-0434-9.

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

Philip Wadler. Theorems for free! In *FPCA ’89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, New York, NY, USA, 1989. ACM.