Isomorphic Data Encodings and their Generalization to Hylomorphisms on Hereditarily Finite Data Types

Paul Tarau

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

Abstract. This paper is an exploration in a functional programming framework of *isomorphisms* between elementary data types (natural numbers, sets, multisets, finite functions, permutations binary decision diagrams, graphs, hypergraphs, parenthesis languages, dyadic rationals, primes, DNA sequences 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.
Besides applications to experimental mathematics, 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 self-delimiting codes, succinct data representations and generation of random instances.
The paper covers 60 data types and, through the use of the embedded combinator language, provides 3660 distinct bijective transformations 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`.
A short, 5 page version of the paper, published as [1] describes the idea of organizing various data transformations as encodings to sequences of natural numbers and gives a few examples of hylomorphisms that lift the encodings to related hereditarily finite universes.
Keywords: *Haskell data representations, data type isomorphisms, declarative combinatorics, computational mathematics, Ackermann encoding, Gödel numberings, arithmetization, ranking/unranking, hereditarily finite sets, functions and permutations, encodings of binary decision diagrams, dyadic rationals, DNA encodings*

1 Introduction

Analogical/metaphorical thinking routinely shifts entities and operations from a field to another hoping to uncover similarities in representation or use [2].

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 [3, 4].

Mathematical theories often borrow proof patterns and reasoning techniques across 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 across 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 [5, 6] associated to formulae, but a wide diversity of common computer operations, ranging from data compression and serialization to wireless data transmissions and 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 groupoid of isomorphisms. We will then extended it with a set of higher order combinators mediating the composition of the encodings and the transfer of operations between data types.

2.1 The Groupoid 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 *groupoid* as follows:

$$X \underset{g = f^{-1}}{\overset{f = g^{-1}}{\rightleftarrows}} Y$$

```
data Iso a b = Iso (a→b) (b→a)

from (Iso f _) = f
to (Iso _ g) = g

compose :: Iso a b → Iso b c → Iso a c
compose (Iso f g) (Iso f' g') = Iso (f' . f) (g . g')
itself = Iso id id
invert (Iso f g) = Iso g f
```

Assuming that for any pair of type `Iso a b`, $f \circ g = id_a$ and $g \circ f = id_b$, we can now formulate *laws* about isomorphisms that can be used to test correctness of implementations with tools like QuickCheck [7].

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

In particular we can define a type for `automorphisms` (i.e. isomorphisms with the same source and target) as:

```
type Auto a = Iso a a
```

Such automorphisms can be seen as *modifiers* that can be used to "dress" isomorphisms with a different behaviour without changing their types (source and target).

```
dress :: Auto a → Iso a b → Iso a b
dress aa ab = compose aa ab
```

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, 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 in a connected groupoid

Within each connected groupoid, to avoid defining $n(n-1)/2$ isomorphisms between n objects, we can choose a *Root* object to/from which we will actually implement isomorphisms. Then we can extend our embedded combinator language using the groupoid 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 `finite functions` from an initial segment of $\mathbb{N}$, say $[0..n]$, to $\mathbb{N}$. This implies that a finite function can be seen as an array or a list of natural numbers except that we do not limit the size of the representation of its values. We will represent them as lists i.e. their Haskell type is $[\mathbb{N}]$.

```
type N = Integer
type Root = [N]
```

We can now define an *Encoder* as an isomorphism connecting an object to *Root*

```
type Encoder a = Iso a Root
```

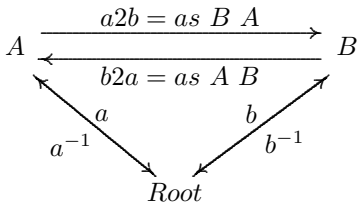
together with the combinators *with* and *as* providing an *embedded transformation language* for routing isomorphisms through two *Encoders*.

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



A particularly useful combinator that transports functions from an Encoder to another, `borrow_from`, can be defined as follows:

```
borrow_from :: Encoder b → (b → b) →
               Encoder a → a → a

borrow_from lender f borrower =
  (as borrower lender) . f . (as lender borrower)
```

Similarly, a two argument function `op` is transported between data types using the combinator `borrow_from2`:

```
borrow_from2 :: Encoder a → (a → a → a) →
                Encoder b → b → b → b

borrow_from2 lender op borrower x y =
    as borrower lender r where
       x'= as lender borrower x
       y'= as lender borrower y
       r = op x' y'
```

We will provide extensive use cases for these combinators as we populate our groupoid of isomorphisms. Given that $[\mathbb{N}]$ has been chosen as the root, we will define our finite function data type *fun* simply as the identity isomorphism on sequences in $[\mathbb{N}]$.

```
fun :: Encoder [N]
fun = itself
```

3 Extending the Groupoid of Isomorphisms

We will now populate our groupoid of isomorphisms with combinators based on a few primitive converters.

3.1 A ranking/unranking algorithm for finite sequences

A *ranking/unranking* function defined on a data type is a bijection to/from the set of natural numbers (denoted $\mathbb{N}$ through the paper). We start with an unusually simple but (at our best knowledge) novel ranking/unranking algorithm for finite sequences of arbitrary unbounded size natural numbers. Given the definitions

```
cons :: N→N→N
cons x y  = (2^x)*(2*y+1)

hd :: N→N
hd n | n>0 = if odd n then 0 else 1+hd  (n `div` 2)

tl :: N→N
tl n = n `div` 2^((hd n)+1)

nat2fun :: N→[N]
nat2fun 0 = []
nat2fun n = hd n : nat2fun (tl n)

fun2nat :: [N]→N
fun2nat [] = 0
fun2nat (x:xs) = cons x (fun2nat xs)
```

Proposition 2 `fun2nat` *is a bijection from finite sequences of natural numbers to natural numbers and* `nat2fun` *is its inverse.*

This follows from the fact that `cons` and the pair `(hd, tl)` define a bijection between $\mathbb{N}-\{0\}$ and $\mathbb{N}\times\mathbb{N}$ and that the value of `fun2nat` is uniquely determined by the sequence of applications of `tl` and the sequence of values returned by `hd`.

```
*ISO> hd 2008
3
*ISO> tl 2008
125
*ISO> cons 3 125
2008
```

We can define the `Encoder`

```
nat :: Encoder N
nat = Iso nat2fun fun2nat
```

working as follows

```
*ISO> as fun nat 2008
[3,0,1,0,0,0,0]
*ISO> as nat fun [3,0,1,0,0,0,0]
2008
```

Note also that this isomorphism preserves "list processing" operations i.e. if one defines:

```
app 0 ys = ys
app xs ys = cons (hd xs) (app (tl xs) ys)
```

then the isomorphism commutes with operations like list concatenation:

Proposition 3 *(as fun nat n) ++ (as fun nat m)* $\equiv$ *as fun nat (app n m)*
as nat fun (ns ++ ms) $\equiv$ *app (as nat fun ns) (as nat fun ms)*

Given the definitions:

```
unpair z = (hd (z+1),tl (z+1))
pair (x,y) = (cons x y)-1
```

shifting by 1 turns `hd` and `tl` in total functions on $\mathbb{N}$ such that $unpair\ 0 = (0, 0)$ i.e.

Proposition 4 $unpair : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ *is a bijection and* $pair = unpair^{-1}$.

Note that unlike `hd` and `tl`, `unpair` is defined for all natural numbers:

```
*ISO> map unpair [0..7]
[(0,0),(1,0),(0,1),(2,0),(0,2),(1,1),(0,3),(3,0)]
```

As the cognoscenti might notice, this is in fact a classic *pairing/unpairing function* that has been used, by Pepis, Kalmar and Robinson in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem in [8–12] and `hd,tl,cons,0` define on $\mathbb{N}$ an algebraic structure isomorphic to the one introduced by `CAR,CDR,CONS,NIL` in John McCarthy's classic LISP paper [13].

We will revisit pairing functions in section 10.

3.2 An Isomorphism between Finite Multisets and Finite Functions

Multisets [14] are unordered collections with repeated elements. Non-decreasing sequences provide a canonical representation for multisets of natural numbers. The isomorphism between finite multisets and finite functions is specified with two bijections `mset2fun` and `fun2mset`.

```
mset :: Encoder [N]
mset = Iso mset2fun fun2mset
```

While finite multisets and sequences representing finite functions share a common representation [$\mathbb{N}$], multisets are subject to the implicit constraint that

their order is immaterial [1]. This suggest that a multiset like $[4,4,1,3,3,3]$ could be represented by first ordering it as $[1,3,3,3,4,4]$ and then compute the differences between consecutive elements i.e. $[x_0 \ldots x_i, x_{i+1} \ldots] \rightarrow [x_0 \ldots x_{i+1} - x_i \ldots]$. This gives $[1,2,0,0,1,0]$, with the first element 1 followed by the increments $[2,0,0,1,0]$, as implemented by `mset2fun`:

```
mset2fun = to_diffs . sort . (map must_be_nat)

to_diffs xs = zipWith (-) (xs) (0:xs)

must_be_nat n | n≥0 = n
```

It can now be verified easily that prefix sums of the numbers in such a sequence return the original set in sorted form, as implemented by `fun2mset`:

```
fun2mset ns = tail (scanl (+) 0 (map must_be_nat ns))
```

The resulting isomorphism `mset` can be applied directly using its two components `mset2fun` and `fun2mset`. Equivalently, it can be expressed more "generically" by using the `as` combinator, as follows:

```
*ISO> mset2fun [1,3,3,3,4,4]
[1,2,0,0,1,0]
*ISO> fun2mset [1,2,0,0,1,0]
[1,3,3,3,4,4]
*ISO> as fun mset [1,3,3,3,4,4]
[1,2,0,0,1,0]
*ISO> as mset fun [1,2,0,0,1,0]
[1,3,3,3,4,4]
```

3.3 An Isomorphism to Finite Sets of Natural Numbers

While finite sets and sequences share a common representation $[\mathbb{N}]$, sets are subject to the implicit constraints that all their elements are distinct and order is immaterial. Like in the case of multisets, 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 xs | is_set xs = shift_tail pred (mset2fun xs)

shift_tail _ [] = []
shift_tail f (x:xs) = x:(map f xs)

is_set ns = ns==nub ns
```

[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.

It can now be verified easily that predecessors of the incremental sums of the successors of numbers in such a sequence, return the original set in sorted form, as implemented by `fun2set`:

```
fun2set = (map pred) . fun2mset . (map succ)
```

together with the dual definition, equivalent to `set2fun`:

```
set2fun' = (map pred) . mset2fun . (map succ)
```

The *Encoder* (an isomorphism with `fun`) can be specified with the two bijections `set2fun` and `fun2set`.

```
set :: Encoder [N]
set = Iso set2fun fun2set
```

The Encoder (`set`) is now ready to interoperate with another Encoder:

```
*ISO> as fun set [0,2,3,4,9]
[0,1,0,0,4]
*ISO> as set fun [0,1,0,0,4]
[0,2,3,4,9]
*ISO> as mset set [0,2,3,4,9]
[0,1,1,1,5]
*ISO> as set mset [0,1,1,1,5]
[0,2,3,4,9]
```

As the example shows,the Encoder `set` connects arbitrary lists of natural numbers representing finite functions to strictly increasing sequences of (distinct) natural numbers representing sets. Then, through the use of the combinator `as`, sets represented by `set` are connected to multisets represented by `mset`. This connection is (implicitly) routed through a connection to `fun`, as if

```
*ISO> as mset fun [0,1,0,0,4]
[0,1,1,1,5]
```

were executed.

3.4 On sharing the underlying data types

We have seen so far that finite functions, multisets and sets share the same representation $[\mathbb{N}]$. This hides injective applications between the corresponding mathematical abstractions. However, the overlap of representations is safe, provided that we think of multisets canonically represented as nondecreasing sequences and sets canonically represented as strictly increasing sequences. It is also safe to apply to unordered sequences seen as representations of sets or multisets operations that are order independent - for instance sums or products. With this warning in mind, we can use the injective mappings from overlapping set, multiset, sequence representations *implicitly* under the assumption that the order of operations is immaterial.

Alternatively, one can approximate a sequence as the set of its elements, ignoring order and multiplicity.

3.5 Folding Sets into Natural Numbers Directly

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 can standardize this pair of operations as an *Encoder* for a natural number using our Root as a mediator:

```
nat' :: Encoder N
nat' = compose nat_set set
```

The following holds:

Proposition 5 $nat \equiv nat'$

Given that `nat` is an isomorphism with the Root `fun`, one can use directly its `from` and `to` components:

```
*ISO> from nat' 2008
[3,0,1,0,0,0,0]
*ISO> to nat' it
2008
```

The resulting Encoder (`nat`) is now ready to interoperate with any Encoder, in a generic way:

```
*ISO> as fun nat' 2008
[3,0,1,0,0,0,0]
*ISO> as set nat' 2008
[3,4,6,7,8,9,10]
*ISO> as nat' set [3,4,6,7,8,9,10]
2008
*ISO> lend nat' reverse 2008
1135
*ISO> lend nat_set reverse 2008
2008
*ISO> borrow nat_set succ [1,2,3]
[0,1,2,3]
*ISO> as set nat' 42
[1,3,5]
*ISO> fit length nat' 42
3
```

```
*ISO> retrofit succ nat_set [1,3,5]
43
```

The reader might notice at this point that we have already made full circle - as finite sets can be seen as instances of finite sequences. Injective functions that are not surjections with wider and wider gaps can be generated using the fact that one of the representations is information theoretically "denser" than the other, for a given range:

```
*ISO> as set fun [0,1,2,3]
[0,2,5,9]
*ISO> as set fun $ as set fun [0,1,2,3]
[0,3,9,19]
*ISO> as set fun $ as set fun $ as set fun [0,1,2,3]
[0,4,14,34]
```

One can now define, for instance, a mapping from natural numbers to multi-sets simply as:

```
nat2mset = as mset nat
mset2nat = as nat mset
```

but we will not explicitly need such definitions as the the equivalent function is clearly provided by the combinator `as`. One can now borrow operations between `set` and nat as follows:

```
*ISO> borrow_from2 set union nat 42 2008
2042
*ISO> 42 .|. 2008 :: N
2042
*ISO> borrow_from2 set intersect nat 42 2008
8
*ISO> 42 .&. 2008 :: N
8
*ISO> borrow_from2 nat (*) set [1,2,3] [4,5]
[5,7,9]
*ISO> borrow_from2 nat (+) set [1,2,3] [3,4,5]
[1,2,6]
```

and notice that operations like union and intersection of sets map to boolean operations on numbers as expected, while other operations are not necessarily meaningful at first sight. We will show next a few cases where such "shapshiftings" of operations reveal more interesting analogies.

3.6 Encoding Finite Multisets with Primes

A factorization of a natural number is uniquely described as multi-set or primes. We will use the fact that each prime number is uniquely associated to its position in the infinite stream of primes to obtain a bijection from multisets of natural numbers to natural numbers. We assume defined a prime generator `primes` and a factoring function `to_factors` (see Appendix).

The function `nat2pmset` maps a natural number to the multiset of prime positions in its factoring. Note that we treat `0` as `[]` and shift `n` to `n+1` to accomodate `0` and `1`, to which prime factoring operations do not apply.

```
nat2pmset 0 = []
nat2pmset n = map (to_pos_in (h:ts)) (to_factors (n+1) h ts) where
  (h:ts)=genericTake (n+1) primes

to_pos_in xs x = fromIntegral i where
   Just i=elemIndex x xs
```

The function `pmset2nat` maps back a multiset of positions of primes to the result of the product of the corresponding primes. Again, we map `[]` to `0` and shift back by `1` the result.

```
pmset2nat [] = 0
pmset2nat ns = (product ks)-1 where
  ks=map (from_pos_in ps) ns
  ps=primes

from_pos_in xs n = xs !! (fromIntegral n)
```

We obtain the Encoder:

```
pmset :: Encoder [N]
pmset = compose (Iso pmset2nat nat2pmset) nat
```

working as follows:

```
*ISO> as pmset nat 2008
[3,3,12]
*ISO> as nat pmset it
2008
*ISO> map (as pmset nat) [0..7]
[[],[0],[1],[0,0],[2],[0,1],[3],[0,0,0]]
```

Note that the mappings from a set or sequence to a number work in time and space linear in the bitsize of the number. On the other hand, as prime number enumeration and factoring are involved in the mapping from numbers to multisets this encoding is intractable for all but small values.

We can also derive set and sequence encodings from this, by observing that the encoding between sets, multisets and sequences is independent of their mapping to natural numbers. We obtain the Encoders:

```
pnats :: Encoder [N]
pnats = compose (Iso (as mset fun) (as fun mset)) pmset

pset :: Encoder [N]
pset = compose (Iso (as fun set) (as set fun)) pnats
```

working as follows:

```
*Iso> as pnats nat 2009
[0,1,1,16]
```

```
*Iso> as pset pnats it
[0,2,4,21]
*Iso> as nat pset it
2009
```

We are now ready to "shapeshift" between data types while watching for interesting landscapes to show up.

3.7 Exploring the analogy between multiset decompositions and factoring

As natural numbers can be uniquely represented as a multiset of prime factors and, independently, they can also be represented as a multiset with the Encoder `mset` described in subsection 3.2, the following question arises naturally:

Can in any way the "easy to reverse" encoding `mset` *emulate or predict properties of the the difficult to reverse factoring operation?*

The first step is to define an analog of the multiplication operation in terms of the computationally easy multiset encoding `mset`. Clearly, it makes sense to take inspiration from the fact that factoring of an ordinary product of two numbers can be computed by concatenating the multisets of prime factors of its operands.

```
mprod = borrow_from2 mset (++) nat
```

Proposition 6 $< N, mprod, 0 >$ *is a commutative monoid i.e.* `mprod` *is defined for all pairs of natural numbers and it is associative, commutative and has 0 as an identity element.*

After rewriting the definition of `mprod` as the equivalent:

```
mprod_alt n m = as nat mset ((as mset nat n) ++ (as mset nat m))
```

the proposition follows immediately from the associativity of the concatenation operation and the order independence of the multiset encoding provided by `mset`.

We can derive an exponentiation operation as a repeated application of `mprod`:

```
mexp n 0 = 0
mexp n k = mprod n (mexp n (k-1))
```

Here are a few examples comparing `mprod` to ordinary multiplication and exponentiation:

```
*ISO> mprod 41 (mprod 33 88)
3539
*ISO> mprod (mprod 41 33)  88
3539
*ISO> mprod 33 46
605
*ISO> mprod 46 33
605
```

```
*ISO> mprod 0 712
712
*ISO> mprod 5513 0
5513

*ISO> (41*33)*88
119064
*ISO> 41*(33*88)
119064
*ISO> 33*46
1518
*ISO> 46*33
1518
*ISO> 1*712
712
*ISO> 5513*1
5513
*ISO> map (λx→mexp x 2) [0..15]
[0,3,6,15,12,27,30,63,24,51,54,111,60,123,126,255]
*ISO> map (λx→x^2) [0..15]
[0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225]
```

Note also that any multiset encoding of natural numbers can be used to define a similar commutative monoid structure. In the case of `pmset` we obtain:

```
pmprod n m = as nat pmset ((as pmset nat n) ++ (as pmset nat m))
```

If one defines:

```
pmprod' n m = (n+1)*(m+1)-1
```

it follows immediately from the definition of `mprod` that:

$$pmprod \equiv pmprod' \tag{1}$$

This is useful as computing `pmprod'` is easy while computing `mprod` is intractable for large values. This brings us back to observe that:

Proposition 7 $< N, pmprod, 0 >$ *is a commutative monoid i.e.* `pmprod` *is defined for all pairs of natural numbers and it is associative, commutative and has 0 as an identity element.*

Fig. 1 compares the shapes of `pmprod'` (virtually the same as ordinary multiplication) and mprod for operands in $[0..2^7 - 1]$. One can see the contrast between the regular shape of ordinary multiplication and the recursively "self-similar" landscape induced by `mprod`.

One can also bring `mprod` closer to ordinary multiplication by defining

```
mprod' 0 _ = 0
mprod' _ 0 = 0
mprod' m n = (mprod (n-1) (m-1)) + 1

mexp' n 0 = 1
mexp' n k = mprod' n (mexp' n (k-1))
```

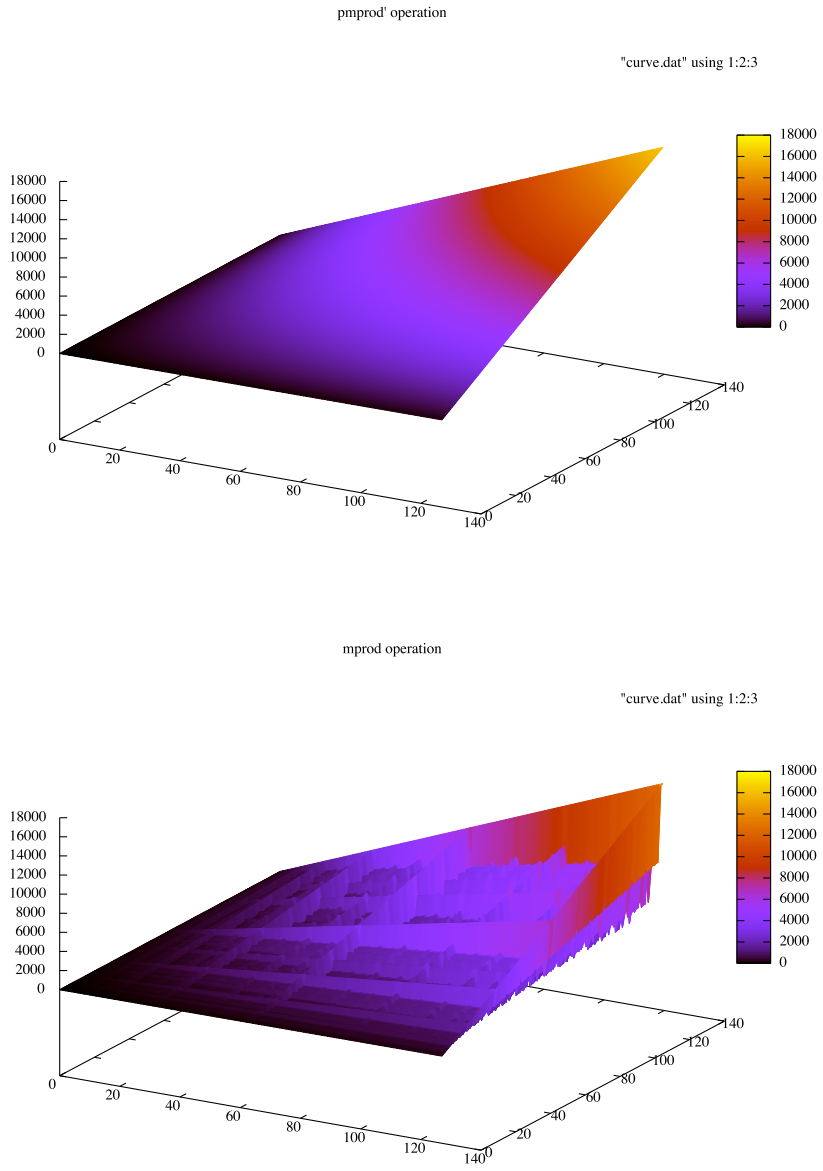


Fig. 1: multiplication vs mprod: *pmprod'* and *mprod*

and by observing that they correlate as follows:

```
*ISO> map (λx→mexp' x 2) [0..16]
[0,1,4,7,16,13,28,31,64,25,52,55,112,61,124,127,256]
*ISO> map (λx→x^2) [0..16]
[0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256]
[0,1,8,15,64,29,120,127,512,57,232,239,960,253,1016,1023,4096]
*ISO> map (λx→x^3) [0..16]
[0,1,8,27,64,125,216,343,512,729,1000,1331,1728,2197,2744,3375,4096]
```

Fig. 2 shows that values for `mexp'` follow from below those of the x^2 function and that equality only holds when x is a power of 2.

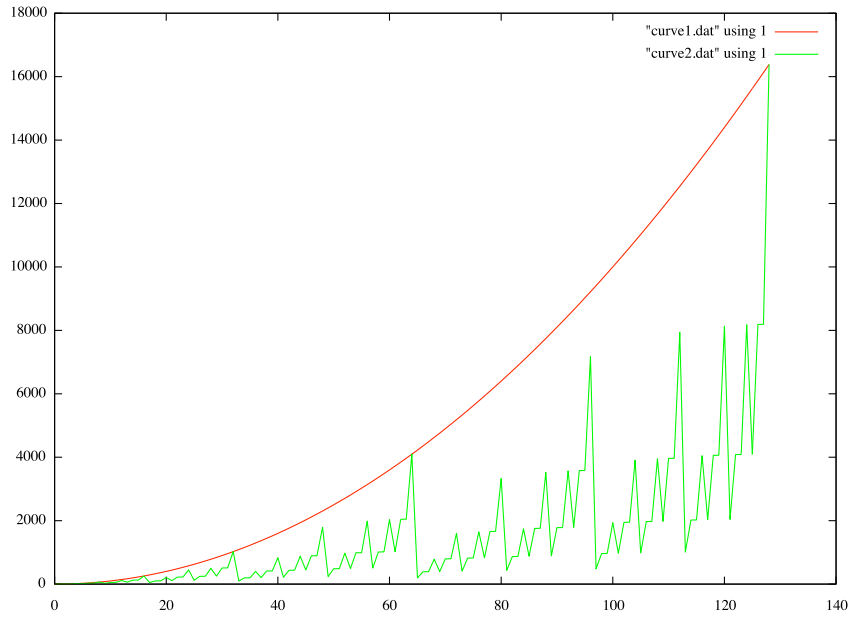


Fig. 2: Square vs. mexp' n 2

Note that the structure induced by `mprod'` is similar to ordinary multiplication:

Proposition 8 $< N, mprod', 1 >$ *is a commutative monoid i.e.* `mprod'` *is defined for all pairs of natural numbers and it is associative, commutative and has* `1` *as an identity element.*

Interestingly, `mprod'` coincides with ordinary multiplication if one of the operands is a power of 2. More precisely, the following holds:

Proposition 9 $mprod'\ x\ y = x * y$ *if and only if* $\exists n \geq 0$ *such that* $x = 2^n$ *or* $y = 2^n$. *Otherwise,* $mprod'\ x\ y < x * y$.

Fig. 3 shows the self-similar landscape generated by the [0..1]-valued function `(mprod' x y) / (x*y)` for values of x,y in [1..128].

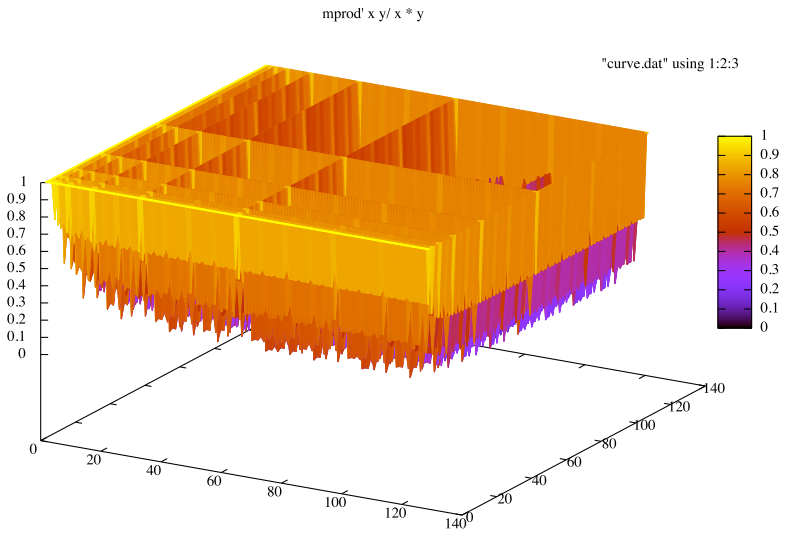


Fig. 3: Ratio between mprod' and product

Besides the connection with products, natural mappings worth investigating are the analogies between *multiset intersection* and `gcd` of the corresponding numbers or between *multiset union* and the `lcm` of the corresponding numbers. Assuming the definitions of multiset operations provided in the Appendix, one can define:

```
mgcd :: N → N → N
mgcd = borrow_from2 mset msetInter nat

mlcm :: N → N → N
mlcm = borrow_from2 mset msetUnion nat

mdiv :: N → N → N
mdiv = borrow_from2 mset msetDif nat
```

and note that properties similar to usual arithmetic operations hold:

$$mprod(mgcd\ x\ y)(mlcm\ x\ y) \equiv mprod\ x\ y \tag{2}$$

$$mdiv(mprod\ x\ y)\ y \equiv x \tag{3}$$

$$mdiv(mprod\ x\ y)\ x \equiv y \tag{4}$$

We are now ready to "emulate" primality in our multiset monoid by defining `is_mprime` as a recognizer for *multiset primes* and `mprimes` as a generator of their infinite stream:

```
is_mprime p = []==[n|n←[1..p-1],n `mdiv` p==0]

mprimes = filter is_mprime [1..]
```

Trying out `mprimes` gives:

```
*ISO> take 10 mprimes
[1,2,4,8,16,32,64,128,256,512]
```

suggesting the following proposition:

Proposition 10 *There's an infinite number of* multiset primes *and they are exactly the powers of 2.*

The proof follows immediately from observing that the first value of `as mset nat n` to contain k is $n = 2^k$ and the definition of `mdiv` as derived from multiset difference operation `msetDif`.

In a recent paper, Eric Rowland [15] proves that the successive differences `dgcd` of the recurence `rgcd`

```
rgcd 1 = 7
rgcd n | n>1 = n' + (gcd n n') where  n'=rgcd (pred n)

dgcd n = (rgcd (succ n')) - (rgcd n') where n'=succ n
```

generate only primes numbers and `1`s. Such relatively simple prime generating formulae are somewhat surprising as it is known for instance that no polynomials exist that have only primes as values. The paper [15] also mentions work by Benoit Cloitre on showing that a similar result holds for *predecessors of the ratios* of the "dual" recurrence `rlcm` implemented as `dlcm`:

```
rlcm 1 = 1
rlcm n | n>1 = n' + (lcm n n') where  n'=rlcm (pred n)

dlcm n = pred ((rlcm (succ n')) `div` (rlcm n')) where n'=succ n
```

After removing duplicates, `1`s and sorting the sequences `plcm` seems to generate all primes except `3` (our *conjecture*):

```
plcd n = (tail . sort . nub . (map dgcd)) [0..n]

plcm n = (tail . sort . nub . (map dlcm)) [0..n]
```

as the following examples show

```
*ISO> plcd 1000
[3,5,7,11,13,23,47,101,233,467,941]
```

```
*ISO> plcm 100
[2,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101]
*ISO> length it
25
*ISO> take 25 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

Let's construct the equivalent of the more productive `plcm` sequence.

```
rmlcm 1 = 1
rmlcm n | n>1 = n' + (mlcm n n') where  n'=rmlcm (pred n)

dmlcm n = pred ((rmlcm (succ n')) `div` (rmlcm n')) where n'=succ n
```

After removing duplicates, `1`s and sorting the sequence `dmlcm`

```
mplcm n = (tail . sort . nub . (map dmlcm)) [0..n]
```

The surprising outcome this time is that, with the exception of the single false positive `5`, all the "mprimes" (in this case powers of two) seem to be generated.

```
*ISO> mplcm 100
[2,4,5,8,16,32,64]
*ISO> mplcm 200
[2,4,5,8,16,32,64,128]
*ISO> mplcm 300
[2,4,5,8,16,32,64,128,256]
```

This leads to the very general "methodological" conjecture that *some fundamental properties of prime numbers are genuinely abstract i.e. they do not depend on the specifics of the multiplication operation but can be derived directly from alternative multiset representations of natural numbers.*

While `mprod,mprod',pmprod'` and `pmprod` are not distributive with ordinary addition, it looks like an interesting problem to find for each of them compatible additive operations.

3.8 Unfolding Natural Numbers into Bitstrings

The isomorphism between natural numbers and bitstrings 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 `0`s and `1`s. 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 transformation (a variant of the so called *bijective base n* representation), brings us an isomorphism between $\mathbb{N}$ and the regular language $\{0,1\}^*$.

```
bits :: Encoder [N]
bits = compose (Iso bits2nat nat2bits) nat

nat2bits = drop_last . (to_base 2) . succ

bits2nat bs = pred (from_base 2 (bs ++ [1]))
```

```
drop_last =
    reverse . tail . reverse

bits1 :: Encoder [N]
bits1 = compose (Iso (from_base 2) (to_base 2)) nat

to_base base n = d :
  (if q==0 then [] else (to_base base q)) where
     (q,d) = quotRem n base

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

In contrast with the conventional 1-terminated `bits1` encoder, the `bits` encoder achieves the information theoretical optimum. Note also that, strictly speaking, this is only an isomorphism when the digits in the bitlist are in $\{0, 1\}$, therefore we shall assume this constraint as a *law* governing this Encoder. Similarly, `bits1` is assumed constrained by working on 1-terminated bit sequences or `[0]` representing `0` as usual. 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> as bits1 nat 42
[0,1,0,1,0,1]
*ISO> as nat bits1 it
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]
```

One can introduce an automorphism[2] on bit sequences by defining:

```
mirror = dress (Iso dual dual)
dual bs = map f bs where
```

[2] An isomorphism from a data type to itself

```
  f 0=1
  f 1=0
```

working as follows:

```
*ISO> as (mirror bits) nat 42
[0,0,1,0,1]
*ISO> as nat (mirror bits) it
42
```

One can derive the Encoder

```
bits' :: Encoder [N]
bits' = mirror bits
```

Indeed, the action of `bits'` can be seen as the one of `bits` with `0`s and `1`s interchanged.

```
*ISO> as bits nat 42
[1,1,0,1,0]
*ISO> as bits' nat 42
[0,0,1,0,1]
*ISO> as nat bits' it
42
```

One can also "transport" the effect of this duality to natural numbers with

```
ndual = borrow_from bits dual nat
```

working as follows:

```
*ISO> map ndual [0..15]
[0,2,1,6,5,4,3,14,13,12,11,10,9,8,7,30]
```

which is a $\mathbb{N} \to \mathbb{N}$ automorphism equivalent to

```
ndual' = (as nat bits) . (as bits' nat)

as one can see from
*ISO> map ndual' [0..15]
[0,2,1,6,5,4,3,14,13,12,11,10,9,8,7,30]
```

3.9 Encoding Binary reflected Gray code

Binary reflected Gray code provides an encoding such that successive natural numbers differ by only one bit (except for steep transitions when cycles end).

```
nat2gray :: N→N
nat2gray n= n `xor` (shiftR n 1)

gray2nat = borrow_from bits1 (scanr1 xor) nat

grayer :: Auto N
grayer = Iso nat2gray gray2nat
```

```
grayed = dress grayer

gray :: Encoder N
gray = compose grayer nat
```

This encoder provides an automorphism on $\mathbb{N}$ with the remarkable property that it is also a permutation for each initial segment $[0..2^n - 1]$.

```
*ISO> map (as gray nat) [0..15]
[0,1,3,2,7,6,4,5,15,14,12,13,8,9,11,10]
*ISO> map (as nat gray) it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
*ISO> map (as nat gray) [0..15]
[0,1,3,2,6,7,5,4,12,13,15,14,10,11,9,8]
*ISO> map (as gray nat) it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

The two automorphisms are respectively sequences A003188 and A006068 in the *On-Line Encyclopedia of Integer Sequences*, `http://www.research.att.com/~njas/sequences`: Like in the case of `mirror`, one can also use the automorphism `grayed` as a modifier:

```
*ISO> as set (grayed nat) 42
[0,1,2,3,4,5]
*ISO> as (grayed nat) set it
42
```

3.10 Encoding numbers in bijective base-k

The conventional numbering system does not provide a bijection between arbitrary combinations of digits and natural numbers, given that leading 0s are ignored. An encoder generalizing `bits` for *numbers in bijective base-k* that provides such a bijection is implemented as follows:

```
bijnat :: N→Encoder [N]

bijnat a = compose (Iso (from_bbase a) (to_bbase a)) nat

from_bbase base xs = from_base' base (map succ xs)

from_base' base [] = 0
from_base' base (x:xs) | x>0 && x≤base =
   x+base*(from_base' base xs)

to_bbase base n = map pred (to_base' base n)

to_base' _ 0 = []
to_base' base n = d' : ds where
   (q,d) = quotRem n base
   d'=if d==0 then base else d
```

```
  q'=if d==0 then q-1 else q
  ds=if q'==0 then [] else to_base' base q'
```

Note that the encoder bijnat is parametrized by the base of numeration, i.e. the `as` combinator works as follows:

```
*ISO> as (bijnat 3) nat 2009
[1,2,2,0,2,0,1]
*ISO> as nat (bijnat 3) it
2009
*ISO> as (bijnat 10) nat 2009
[8,9,8,0]
*ISO> as nat (bijnat 10) it
2009
*ISO> map (as (bijnat 3) nat) [0..12]
[[],[0],[1],[2],[0,0],[1,0],[2,0],[0,1],
    [1,1],[2,1],[0,2],[1,2],[2,2]]
```

This encoding will turn out to be useful for instance in uniquely encoding symbols and strings of a finite alphabet.

3.11 Encoding Signed Integers

To encode signed integers one can map positive numbers to even numbers and strictly negative numbers to odd numbers. This gives the Encoder:

```
type Z = Integer
z:: Encoder Z
z = compose (Iso z2nat nat2z) nat

nat2z n = if even n then n `div` 2 else (-n-1) `div` 2
z2nat n = if n<0 then -2*n-1 else 2*n
```

working as follows:

```
*ISO> as set z (-42)
[0,1,4,6]
*ISO> as z set [0,1,4,6]
-42
```

As an interesting consequence, one can turn natural numbers into a group by borrowing subtraction from integers as follows:

```
0
*ISO> borrow_from2 z (-) nat 0 1
2
*ISO> borrow_from2 z (-) nat 0 2
1
*ISO> borrow_from2 z (-) nat 0 3
4
*ISO> borrow_from2 z (-) nat 0 4
3
```

```
*ISO> borrow_from2 z (-) nat 0 5
6
*ISO> borrow_from2 z (-) nat 10 20
9
```

More precisely one can borrow the additive group structure of `z` and induce an additive group structure on `nat` by defining:

```
nplus x y = borrow_from2 z (+) nat x y
nneg x = borrow_from z (λx→0-x) nat x
```

The following holds:

Proposition 11 $< \mathbb{N}, nplus, nneg, 0 >$ *is an additive group, with addition provided nplus and inverse by nneg.*

Similarly after defining a multiplication operation

```
nprod = borrow_from2 z (*) nat
```

one can obtain a ring structure on $\mathbb{N}$ with units `1 = as nat z (-1)` and `2=as nat z 1`.

3.12 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 3.8 to provide an isomorphism between $\mathbb{N}$ and the functional equivalent of the regular language $\{0,1\}^*$. We will view each bit as a $\mathbb{N} \to \mathbb{N}$ 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 an 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 → N
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 :: N → 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 $\mathbb{N}$:

```
*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) [16, 17]. 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 multi-way tree with a single leaf type `[]`.

```
data T = H [T] deriving (Eq,Ord,Read,Show)
```

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 $[T]$.

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 [T] [b]
hylos (Iso f g) = Iso (ranks g) (unranks f)
```

Hereditarily Finite Sets Hereditarily Finite Sets will be represented as an Encoder for the tree type `T`:

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

The `hfs` Encoder can now borrow operations from sets or natural numbers as follows:

```
hfs_union = borrow2 (with set hfs) union
hfs_succ = borrow (with nat hfs) succ
hfs_pred = borrow (with nat hfs) pred

*ISO> hfs_succ (H [])
H [H []]
*ISO> hfs_union (H [H []] ) (H [])
H [H []]
```

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

```
*ISO> as hfs nat 42
H [H [H []],H [H [],H [H []]],H [H [],H [H [H []]]]]
*ISO> as hfs nat 2008
H [H [H [],H [H []]],H [H [H [H []]]],H [H [H []],
   H [H [H []]]],H [H [],H [H []],H [H [H []]]],
   H [H [H [],H [H []]]],H [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 [18, 19] 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
```

One can represent the action of a hylomorphism unfolding a natural number into a hereditarily finite set as a directed graph with outgoing edges induced by by applying the `inverse_ackermann` function as shown in Fig. 4.

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
```

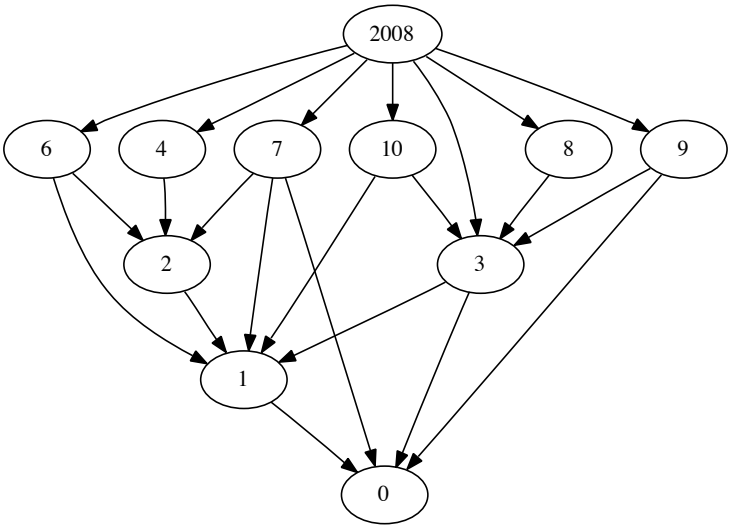


Fig. 4: 2008 as a HFS

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 []]]
∗ISO> as hff nat 2008
H [H [H [],H []],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 a sequence and is ignored in the case of a set. One can represent the action of a hylomorphism unfolding a natural number into a hereditarily finite function as a directed ordered multi-graph as shown in Fig. 5. Note that as the mapping `as fun nat` generates a sequence where the order of the edges matters, this order is indicated integers starting from `0` labeling the edges.

It is also interesting to connect sequences and HFF directly - in case one wants to represent giant "sparse numbers" that correspond to sequences that would overflow memory if represented as natural numbers but have a relatively simple structure as formulae used to compute them. We obtain the Encoder:

```
hffs :: Encoder [T]
hffs = Iso hffs2fun fun2hffs

fun2hffs ns =  (map (as hff nat) ns)
hffs2fun (hs) = map (as nat hff) hs
```

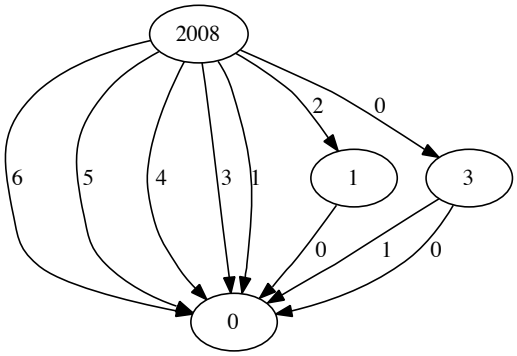


Fig. 5: 2008 as a HFF

which can be used to generate HFFs associated to very large numbers:

```
*ISO> as hffs fun [2^65,2^131]
[H [H [H [],H [H [],H [H []]]],H [H [H [],H [],H [H [],H [H []]]]]
```

4.2 Hereditarily Finite Multisets

In a similar way, one can derive an Encoder for Hereditarily Finite Multisets based on either the `mset` or the `pmset` isomorphisms:

```
nat_mset = Iso nat2mset mset2nat

hfm :: Encoder T
hfm = compose (hylo nat_mset) nat

nat_pmset = Iso nat2pmset pmset2nat

hfpm :: Encoder T
hfpm = compose (hylo nat_pmset) nat
```

working as follows:

```
*ISO> as hfm nat 2008
H [H [H [],H []],H [H [],H []],H [H [H [H []]],H [H [H [H []]],
   H [H [H [H []]],H [H [H [H []]],H [H [H [H []]]]
*ISO> as nat hfm it
2008

*ISO> as hfpm nat 2008
H [H [H [],H []],H [H [],H []],H [H [H [],H [H []]]]
*ISO> as nat hfpm it
2008
```

After implementing this encoding some Google search revealed that it is essentially the same as [20] where it appears as an encoding of *rooted trees*.

4.3 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 multi-way trees with a leaf type holding urelements/atoms:

```
data UT a = At a | UT [UT a] deriving (Eq,Ord,Read,Show)
```

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

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 $\mathbb{N}$ to UT, parameterized by the natural number `ulimit` and the transformer function `f`:

```
ulimit = 2^14

unrankU = unrankUL ulimit
unranksU = unranksUL ulimit

unrankUL l _ n | n≥0 && n<l = At n
unrankUL l f n = UT (unranksUL l f (f (n-l)))

unranksUL l f ns =  map (unrankUL l f) ns
```

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

```
rankU = rankUL ulimit
ranksU = ranksUL ulimit

rankUL l _ (At n) | n≥0 && n<l = n
rankUL l g (UT ts) = l+(g (ranksUL l g ts))

ranksUL l g ts = map (rankUL l 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 $[\mathbb{N}]$ and $\mathbb{N}$ to isomorphisms involving hereditarily finite structures:

Proposition 12 *If the transformer function $f : \mathbb{N} \to [\mathbb{N}]$ 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 $(unrankU\ f) : \mathbb{N} \to UT$ is a bijection with inverse $(rankU\ g) : UT \to \mathbb{N}$ 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 $\mathbb{N}$ 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 N)
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 N)
uhff = compose (hyloU nat) nat
```

4.4 Extending the encoding for the case of an infinite set of Atoms/Urelements

An adaptation of the previous construction for the case when an infinite supply of atoms/urelements is needed (i.e. when their number is not known in advance) follows.

Unranking As an adaptation of the *unfold* operation, natural numbers will be mapped to elements of UT with a generic higher order function `unrankIU f`, defined from $\mathbb{N}$ to UT, parameterized by the transformer function `f`:

```
unrankIU _ n | even n = At (n `div` 2)
unrankIU f n = UT (unranksIU f (f  ((n-1) `div` 2)))

unranksIU f ns =  map (unrankIU f) ns
```

Note that (an infinite supply of) even numbers provides codes for atoms, while odd numbers are used to encode the non-leaf structure of the trees in `UT`.

Ranking Similarly, as an adaptation of *fold*, a generic inverse mapping `rankIU g` is defined as:

```
rankIU _ (At n) = 2*n
rankIU g (UT ts) = 1+2*(g (ranksIU g ts))

ranksIU g ts = map (rankIU g) ts
```

where `rankIU g` maps trees to numbers and `ranksIU g` maps lists of trees to lists of numbers.

The resulting hylomorphisms are defined as previously:

```
hyloIU (Iso f g) = Iso (rankIU g) (unrankIU f)
hylosIU (Iso f g) = Iso (ranksIU g) (unranksIU f)
```

An Encoder for Hereditarily Finite Sets with an infinite supply of Urelements is defined as:

```
iuhfs :: Encoder (UT N)
iuhfs = compose (hyloIU nat_set) nat
```

A similar Encoder for Hereditarily Finite Functions with and infinite supply of Urelements is defined as:

```
iuhff :: Encoder (UT N)
iuhff = compose (hyloIU 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 [21] 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 of size n is defined as the sequence $l(f) = (l_1(f) \dots l_i(f) \dots l_n(f))$ where $l_i(f)$ is the number of elements of the set $\{j > i | f(j) < f(i)\}$ [22].

Proposition 13 *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 a 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 $\mathbb{N}$: we will have to "shift towards infinity" the starting point of each new bloc of permutations in $\mathbb{N}$ 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 [N]
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 isomorphism defined by `nat2perm` and `perm2nat` to encodings of Hereditarily Finite Permutations (HFP).

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

The encoding works as follows:

```
*ISO> nat2hfp 42
H [H [],H [H [],H [H []]],H [H [H []],H []],
   H [H []],H [H [],H [H []],H [H [],H [H []]]]]
*ISO> 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
*JFISO> as hfp nat 2008
H [H [H []],H [H [],H [H []],H [H [],H [H []]]],H [H [H []],H []],
  H [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
2008
```

As shown in Fig 6 an ordered digraph (with labels starting from 0 representing the order of outgoing edges) can be used to represent the unfolding of a natural number to the associated hereditarily finite permutation. An interesting prop-

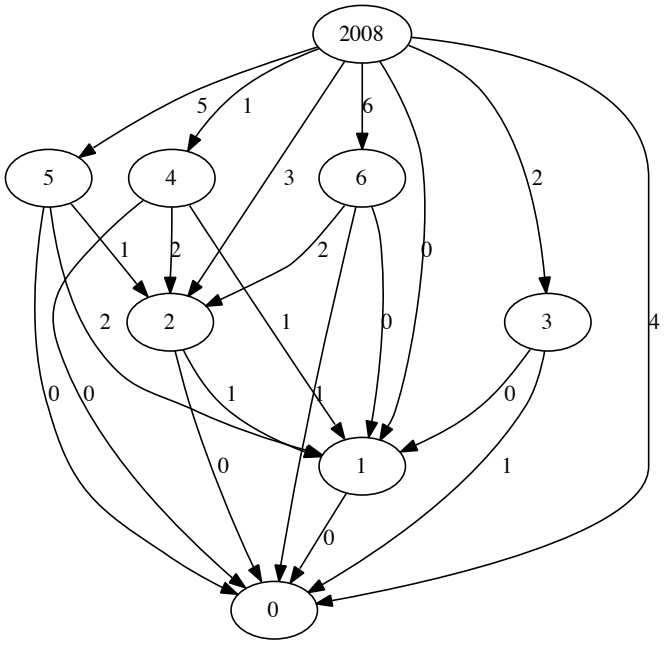


Fig. 6: 2008 as a HFP

erty of graphs associated to hereditarily finite permutations is that moving from a number n to its successor typically only induces a reordering of the labeled edges, as shown in Fig. 7.

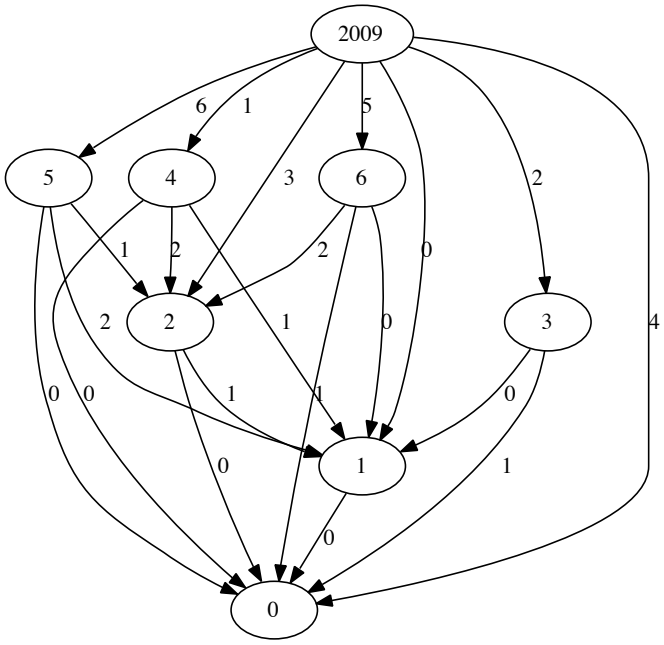


Fig. 7: 2009 as a HFP

6 Operations on hereditarily finite structures

Operations on trees describing hereditarily finite structures induce operations on natural numbers. Such and operation is replacing the (empty) root of B with A, `rerootWith`.

```
rerootWith a (H []) = a
rerootWith a (H bs) = H (map (rerootWith a) bs)

nreroot t = borrow_from2 t rerootWith nat
```

working as follows

```
*ISO> nreroot hff 5 0
5
*ISO> nreroot hff 0 5
5
*ISO> nreroot hff 2 3
36
*ISO> nreroot hff 3 2
256
*ISO> nreroot hff 3 3
136
*ISO> nreroot hff 3 4
1157920892373161954235709850086879078532699846656405640394575840079131296399360
```

An opposite operation `weedWith` can be defined as follows

```
weedWith _ (H []) = H []
weedWith w b | w == b = H []
weedWith w (H bs) = H (map (weedWith w) bs)

nweed t = borrow_from2 t weedWith nat

*ISO> map (nweed hff 1) [0..15]
[0,0,1,3,2,3,3,7,8,5,3,7,6,7,7,15]
*ISO> map (nweed hff 2) [0..15]
[0,1,0,3,1,5,6,7,8,3,10,11,3,13,14,15]
```

Note that the weedWith operation is not total on HFS. Note also that `reroot` and `weed` are not inverse operations as weed might also remove subtrees not originating from a previous reroot operation.

Another interesting operation is trimming of empty leaves.

```
trimEmpty (H []) = H []
trimEmpty (H xs) = H (trimEmpties xs)

trimEmpties [] = []
trimEmpties ((H []):xs) = trimEmpties xs
trimEmpties (x:xs)=(trimEmpty x):trimEmpties xs

ntrim t = borrow_from t trimEmpty nat

*ISO> map (ntrim hff) [0..31]
[0,0,1,0,2,1,1,0,1,2,3,1,2,1,1,0,4,1,5,2,6,3,3,1,1,2,3,1,2,1,1,0]
```

This can be seen as a hash key providing a lossy approximation of the tree structure with leaves removed. Note that this operation is not total on HFS but it is total on HFF,HFP,HFM,HFPM.

7 Hereditary base-k representations and Goodstein sequences

Definition 1 *Hereditary base-k representation of a number x is obtained by representing x as a sum of powers of k followed by expression of each of the exponents with nonzero coefficients as a sum of powers of k, recursively.*

First we express a single step of this transformation to/from a polynomial in base `k` as a pair of bijections:

```
nat2kpoly k n = filter (λp→0/=fst p) ps where
  ns=to_base k n
  l=genericLength ns
  is=[0..l-1]
  ps=zip ns is

kpoly2nat k ps = sum (map (λ(d,e)→d*k^e) ps)
```

The transformation works as follows:

```
*ISO> nat2kpoly 3 2009
[(2,0),(1,2),(2,3),(2,5),(2,6)]
*ISO> kpoly2nat 3 it
2009
```

The recursive process generates a tree, with coefficients of each expansion labeling nodes. We can host this expansion in the data type `HB`:

```
data HB a = HB a [HB a]  deriving (Eq,Ord,Show,Read)
```

We will define, for each base k, two isomorphisms `nat2hb k` and `hb2nat k` between natural numbers and polynomials:

```
nat2hb :: N→N→[HB N]

nat2hb _k 0 = []
nat2hb k n | n<k = [HB n []]
nat2hb k n = gs where
  ps'=nat2kpoly k n
  gs=map (nat2hb1 k) ps'
  nat2hb1 k (d,e) = HB d (nat2hb k e)

hb2nat :: N → [HB N] → N

hb2nat k [] = 0
hb2nat k ts = kpoly2nat k ps where
  ps=map (hb2nat1 k) ts
  hb2nat1 k (HB d ts) = (d,hb2nat k ts)
```

We can now define a family of `Encoders`, one for each base `k`, as follows:

```
hb :: N→Encoder [HB N]
hb k = compose (Iso (hb2nat k) (nat2hb k)) nat
```

The other new concept here is working with a parametric family of Encoders. With a small adaptation, the syntax of the `as` combinator scales up naturally:

```
*ISO> as  (hb 3) nat 42
[HB 2 [HB 1 []],HB 1 [HB 2 []],HB 1 [HB 1 [HB 1 []]]]
*ISO> as nat (hb 3) it
42
```

Fig. 8 shows a graph representation of the expansion of `2009` in hereditary base `3`, the nodes labeled with the exponents and the edges labeled with the coefficients in range [0..2].

Note that the base does not occur as such in the hereditary base-k expression obtained with the Encoder `hb`. This property can be used to obtain `Goodstein sequences` by *bumping the base* from `k` to `k+1` i.e. interpreting a `(hb k)` expression as a `(hb (k+1))` expression and then subtracting 1 from the result, i.e:

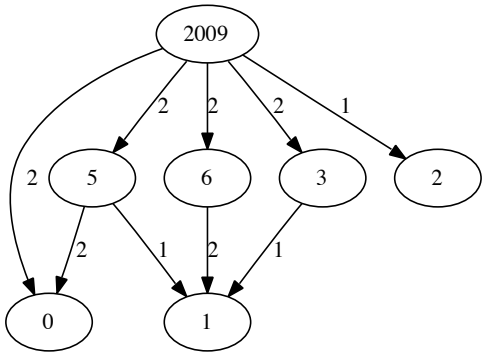


Fig. 8: Expansion of 2009 in hereditary base 3

```
goodsteinStep k n = (hb2nat (k+1) (nat2hb k n)) - 1

goodsteinSeq _ 0 = []
goodsteinSeq k n = n:(goodsteinSeq (k+1) m) where
  m=goodsteinStep k n

goodstein m = goodsteinSeq 2 m
```

Fig. 9 shows a graph representation of the expansion of `139793`, obtained by applying the function `goodsteinStep 3 2009`, in hereditary base `4=3+1`. As previously, the nodes are labeled with the exponents and the edges are labeled with the coefficients, this time in range [0..3]. As the following examples indicate,

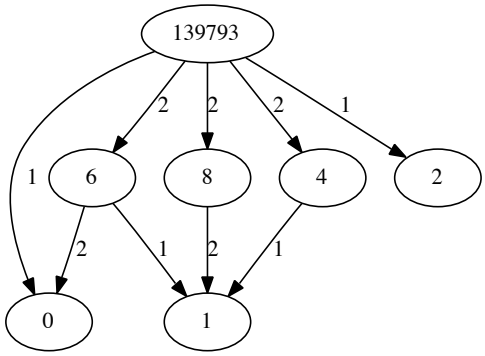


Fig. 9: Expansion of 139793 in hereditary base 4

Goodstein sequences might grow fast for most values, until eventually reverse course end descend to 0.

```
*ISO> goodsteinStep 3 2009
```

```
139793
*ISO> goodsteinStep 4 it
19693775
*ISO> goodstein 3
[3,3,3,2,1]
*ISO> take 12 (goodstein 4)
[4,26,41,60,83,109,139,173,211,253,299,348]
*ISO> take 2 (goodstein 19)
[19,7625597484990]
```

Goodstein's Theorem (only provable in second order arithmetics) states that this sequence always terminates at 0. The remarkable thing about it is that it is an example of an undecidable statement in first order Peano arithmetics, that in contrast to Gödel's theorem, involves only "conventional" numerical relations. Comparing Fig. 8 and Fig. 9 helps with the intuition for why the sequence eventually starts descending: the shape of the graph remains the same despite of the increase of the exponents - so in the resulting *Hydra game* the predecessor function "cutting one head of the Hydra" will eventually win over the bumping up of the base "growing new heads".

8 Generalizing Systems of Numeration

We have seen that factoradic numbers have helped ranking permutations and that Goodstein numbers have been obtained by applying a given construction principle recursively over exponents.

We will now introduce a generalization of numeration systems parameterized by a function f and a base b.

Converting n to base (f, b) starts by generating a sequence of iterates of function f starting with b, up to n, in decreasing order:

```
iterates f b n = reverse (takeWhile (≤n) (iterate f b))
```

For instance if f is the squaring operation and the base is 2 we obtain:

```
*ISO> iterates (^2) 2 100000
[65536,256,16,4,2]
```

Such a sequence will provide the sequence of divisors to be used as in the conventional conversion algorithm:

```
to_fbase f b n | b>0 = reverse (spread (iterates f b n) n)
spread [] n=[n]
spread (d:ds) n = q:spread ds r where (q,r)=quotRem n d
```

To convert back, one computes the value of the resulting representation with digits seen as multipliers:

```
from_fbase f b ns | b>0 = sum (zipWith (*) ns (1:iterate f b))
```

The two sides of the conversion work as follows:

```
*ISO> to_fbase (^2) 2 2009
[1,0,2,13,7]
*ISO> from_fbase (^2) 2 it
2009
```

as $2009 = 1 + 2 * 4 + 13 * 16 + 7 * 256$. We can now instantiate the conversion mechanism by fixing f or b or both. An interesting instance is:

```
to_sqbase b n | b>1 = to_fbase (^2) b n
from_sqbase b ns | b>1 = from_fbase (^2) b ns
```

that we will call "square-base" representation. The expansion of 2009, based on the equality $2009 = 5 + 4 * 6 + 19 * 6^2 + 1 * (6^2)^2$ gives:

```
*ISO> to_sqbase 6 2009
[5,4,19,1]
*ISO> from_sqbase 6 it
2009
```

The same in "square-base" 2:

```
*ISO> to_sqbase 2 2009
[1,0,2,13,7]
*ISO> from_sqbase 2 it
2009
```

Like with hereditary base-k, one can recurse over the expansions. Fig. 10 shows a graph obtained as a result of this process.

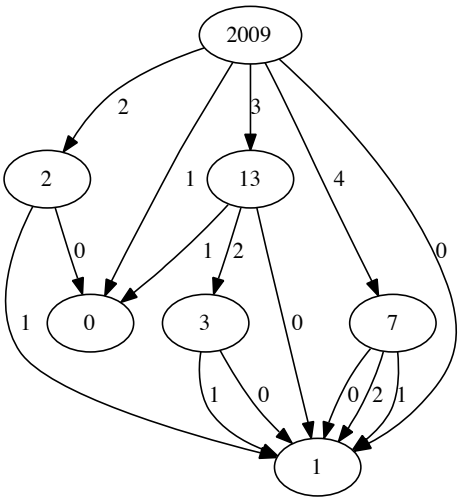


Fig. 10: Recursive expansion of 2009 in "square-base" 2

9 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 = compose (Iso string2nat nat2string) nat

base = 1+ord '~'- ord ' '
chr2ord c | c≥' ' && c≤'~' = ord c - ord ' '

ord2chr o | o≥0 && o<base = chr (ord ' '+o)

string2nat cs = from_bbase
  (fromIntegral base)
  (map (fromIntegral . chr2ord) cs)

nat2string n = map
  (ord2chr . fromIntegral)
  (to_bbase (fromIntegral base) n)
```

We have assumed here ASCII encoding - but changing `string_base` to appropriate values can accommodate a richer character set.

```
*ISO> as set string "hello"
[0,1,4,5,6,8,9,10,12,16,18,19,20,22,27,31,32]
*ISO> as string set it
"hello"
```

Note also that we are using `to_bbase` and `from_bbase` that provide bijective base-k encodings (see 3.10) and therefore ensure a unique representation of our strings as numbers in a given `base`.

10 Pairing/Unpairing

A *pairing* function is an isomorphism $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Its inverse is called *unpairing*.

10.1 Cantor's Pairing Function

Starting from Cantor's pairing function

$$z = \frac{(x+y)\cdot(x+y+1)}{2} + y \tag{5}$$

bijections from $Nat \times Nat$ to Nat have been used for various proofs and constructions of mathematical objects.

Cantor's geometrically inspired pairing function is defined as:

```
cantor_pair (x,y) = (x+y)*(x+y+1) `div` 2+y
```

Note that its range is quite compact

```
[nat_cpair i j|i<-[0..3],j<-[0..3]]
  [0,2,5,9,1,4,8,13,3,7,12,18,6,11,17,24]
```

and a 3D visualization shows a smooth surface (Fig. 11).

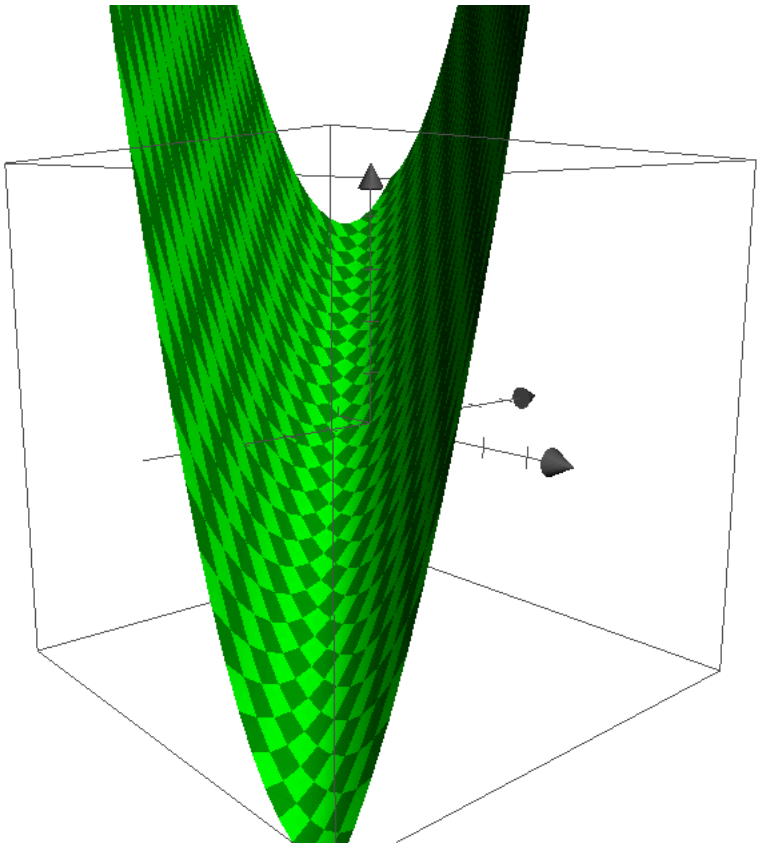


Fig. 11: Values of z=nat_cpair x y

Unfortunately, its inverse involves floating point operations that do not combine well with arbitrary length integers. However, if one implements an integer square root function using Newton's method, an inverse can be obtained using exclusively integer operations. We start with the integer square root function `isqrt`:

```
isqrt 0 = 0
isqrt n | n>0 = if k*k>n then k-1 else k where
  k=iter n
  iter x = if abs(r-x)  < 2
```

```
    then r
    else iter r where r = step x
  d x y = x `div` y
  step x = d (x+(d n x)) 2
```

The inverse of Cantor's pairing function can now be defined as:

```
cantor_unpair z = (x,y) where
  i=(isqrt (8*z+1)-1) `div` 2
  x=(i*(3+i) `div` 2)-z
  y=z-(i*(i+1) `div` 2)
```

After defining

```
type N2 = (N,N)
```

we obtain the encoder

```
cnat2 :: Encoder N2
cnat2 = compose (Iso cantor_pair cantor_unpair) nat
```

working as follows:

```
*ISO> as cnat2 nat 2009
(6,56)
*ISO> as nat cnat2 it
2009
```

10.2 The Pepis-Kalmar-Robinson Pairing Function

A classic pairing function is **pepis_J**, together with its left and right unpairing companions **pepis_K** and **pepis_L** that have been used, by Pepis, Kalmar and Robinson together with Cantor's functions, in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem in [8, 9, 23, 24, 10, 25, 11, 12, 26]. The function **pepis_J** combines two numbers reversibly by multiplying a power of 2 derived from the first and an odd number derived from the second:

$$f(x, y) = 2^x(2y + 1) - 1 \tag{6}$$

Its Haskell implementation, together with its inverse is:

```
pepis_J x y  = pred ((2^x)*(succ (2*y)))

pepis_K n = two_s (succ n)

pepis_L n = (pred (no_two_s (succ n))) `div` 2

two_s n | even n = succ (two_s (n `div` 2))
two_s _ = 0

no_two_s n = n `div` (2^(two_s n))
```

This pairing function (slower in the second argument) works as follows:

```
pepis_J 1 10
  41
pepis_J 10 1
  3071
[pepis_J i j|i←[0..3],j←[0..3]]
  [0,2,4,6,1,5,9,13,3,11,19,27,7,23,39,55]
```

As Haskell provides a built-in ordered pair, it is convenient to regroup the functions `J, K, L` (given in Julia Robinson's original notation) as mappings to/from built-in ordered pairs:

```
pepis_pair (x,y) = pepis_J x y
pepis_unpair n = (pepis_K n,pepis_L n)
```

Observing that the number of `0`s in front of the representation of a natural number `n` as a sequence equals `pepis_K n`, an alternative implementation could be:

```
pepis_pair' (x,y) = (fun2nat (x:(nat2fun y)))-1

pepis_unpair' n = (x,fun2nat ns) where
  (x:ns)=nat2fun (n+1)
```

Note also that `pepis_unpair` is "asymmetrical" in the sense that its first component grows much slower than the second, when applied to `[0..]`. Sometimes it is more useful to have the opposite behavior

```
rpepis_pair (x,y) = pepis_pair (y,x)
rpepis_unpair n = (y,x) where (x,y)=pepis_unpair n
```

We obtain the encoder

```
pnat2 :: Encoder N2
pnat2 = compose (Iso pepis_pair pepis_unpair) nat

rpnat2 :: Encoder N2
rpnat2 = compose (Iso rpepis_pair rpepis_unpair) nat
```

10.3 Deriving pairing/unpairing operations from hereditarily finite function trees

One can derive a pairing/unpairing function by combining/decomposing the tree representation provided by the Encoder `hff` as follows:

```
hpair (x,y) = z-1 where
  hx=as hff nat x
  H hs =as hff nat y
  hz= H (hx:hs)
  z=as nat hff hz

hunpair z = (x,y) where
  H (hx:hs) = as hff nat (z+1)
```

```
  x=as nat hff hx
  y=as nat hff (H hs)
```

Interestingly enough, `hpair/hunpair` turns out to be identical to `pepis_pair/pepis_unpair`, as well. One can notice that this is the case because it applies the same transformation as `pepis_pair'/pepis_unpair'`.

10.4 A Bitwise Pairing/Unpairing Function

We will now introduce an unusually simple pairing function (also mentioned in [27], 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.

```
bitpair ::  N2 → N
bitpair (i,j) =
  set2nat ((evens i) ++ (odds j)) where
    evens x = map (2*) (nat2set x)
    odds y = map succ (evens y)

bitunpair :: N→N2
bitunpair n = (f xs,f ys) where
  (xs,ys) = partition even (nat2set n)
  f = set2nat . (map (`div` 2))
```

The transformation of the bitlists is shown in the following example with bitstrings aligned:

```
*ISO> bitunpair 2008
  (60,26)

-- 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
--   60:[0,    0,    1,    1,    1,    1]
--   26:[   0,    1,    0,    1,    1   ]
```

We can derive the following Encoder:

```
nat2 :: Encoder N2
nat2 = compose (Iso bitpair bitunpair) nat
```

working as follows:

```
*ISO> as nat2 nat 2008
(60,26)
*ISO> as nat nat2 (60,26)
2008
```

In a way similar to hereditarily finite trees generated by unfoldings one can apply strictly decreasing[3] unpairing functions recursively. Figures 12 and 13 show the directed graphs describing recursive application of `bitunpair` and `pepis_unpair`.

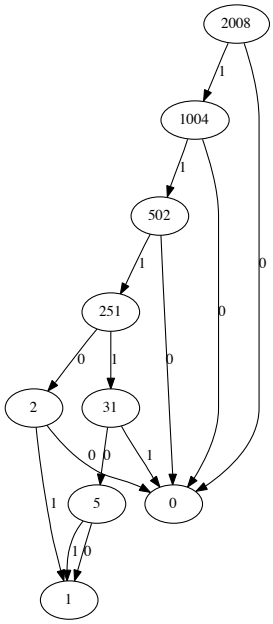


Fig. 12: Graph obtained by recursive application of `pepis_unpair` for 2008

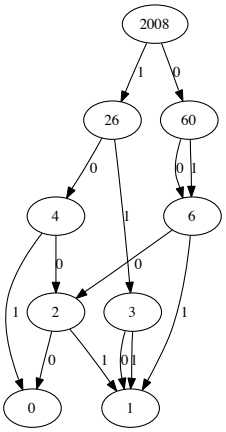


Fig. 13: Graph obtained by recursive application of `bitunpair` for 2008

Given that unpairing functions are bijections from $\mathbb{N}$ to $\mathbb{N} \times \mathbb{N}$ they will progressively cover all points having natural number coordinates in their range in the plane. Figures 14, 15 show the curves generated by `bitunpair` and `pepis_unpair`.

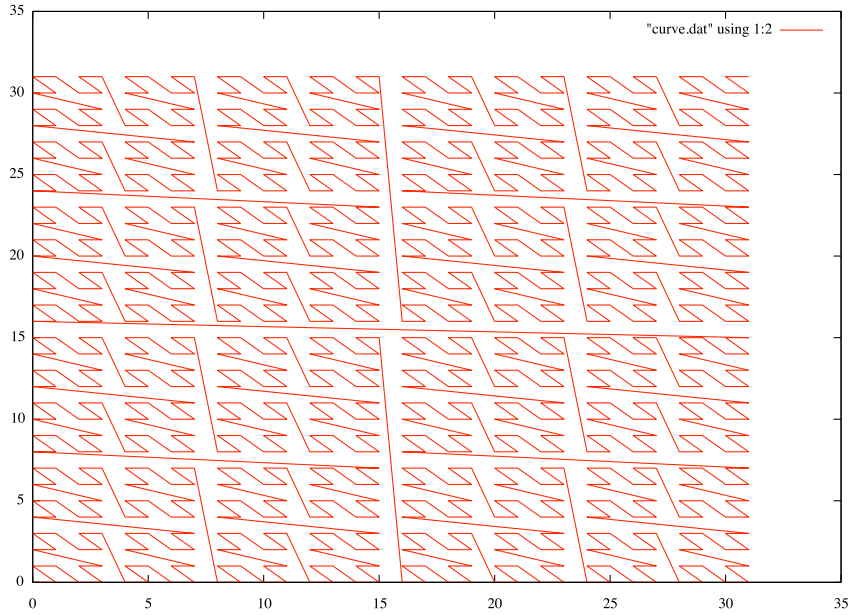


Fig. 14: 2D curve connecting values of `bitunpair n` for $n \in [0..2^{10} - 1]$

Fig. 16 shows the action of the pairing function `bitpair` on its two arguments arguments in [0..63].

10.5 Encoding Unordered Pairs

To derive an encoding of unordered pairs, i.e. 2 element sets, one can combine pairing/unpairing with conversion between sequences and sets:

```
pair2unord_pair (x,y) = fun2set [x,y]
unord_pair2pair [a,b] = (x,y) where
  [x,y]=set2fun [a,b]

unord_unpair = pair2unord_pair . bitunpair
unord_pair = bitpair . unord_pair2pair
```

We can derive the following equivalent Encoders:

[3] except for 0 and 1, typically

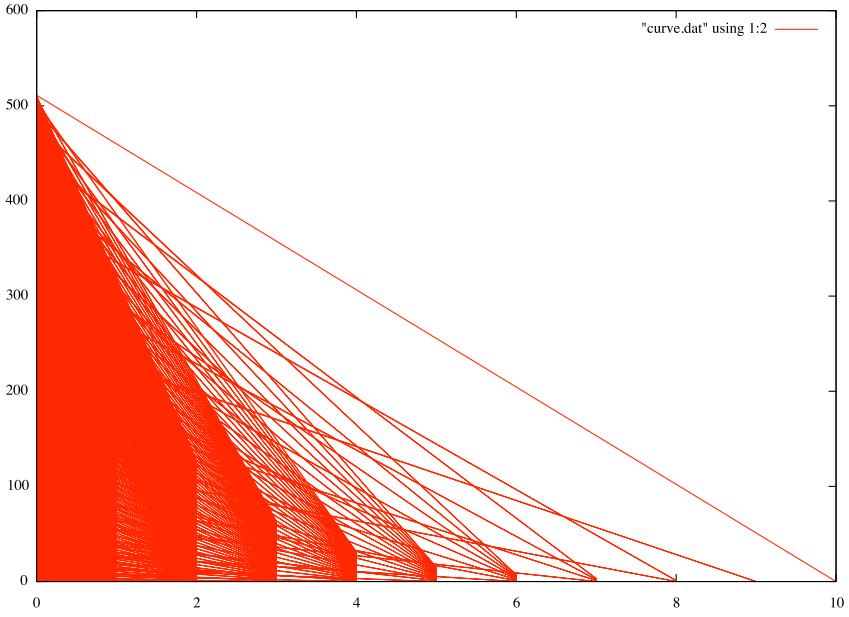


Fig. 15: 2D curve connecting values of `pepis_unpair n` for $n \in [0..2^{10} - 1]$

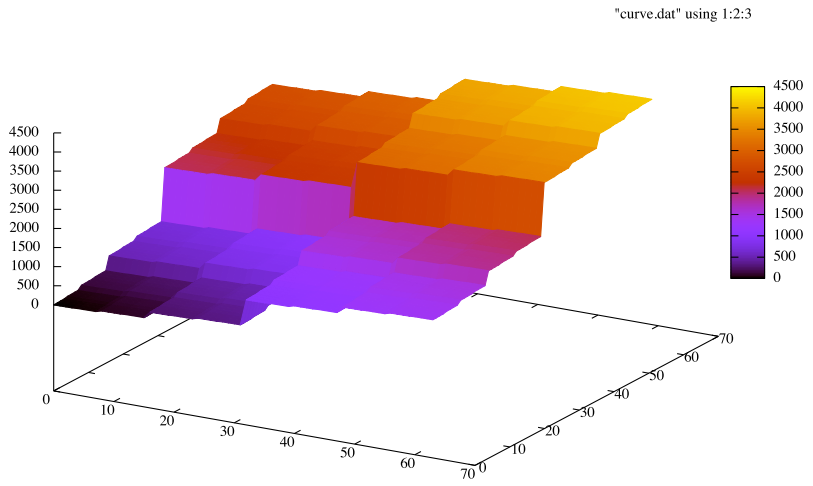


Fig. 16: Values of bitpair x y with x,y in [0..63]

```
set2 :: Encoder [N]
set2 = compose (Iso unord_pair2pair pair2unord_pair) nat2
```

that goes through `nat2`, working as follows:

```
*ISO> as set2 nat 2008
[60,87]
*ISO> as nat set2 it
2008
```

and

```
set2' :: Encoder [N]
set2' = compose (Iso unord_pair unord_unpair) nat
```

that goes through `nat`, working as follows:

```
*ISO> as set2' nat 2008
[60,87]
*ISO> as nat set2' [60,87]
2008
*ISO> as nat set2' [87,60]
2008
```

10.6 Encodings Multiset Pairs

To derive an encoding of 2 element multisets, one can combine pairing/unpairing with conversion between sequences and multisets:

```
pair2mset_pair (x,y) = (a,b) where [a,b]=fun2mset [x,y]
mset_unpair2pair (a,b) = (x,y) where [x,y]=mset2fun [a,b]

mset_unpair = pair2mset_pair . bitunpair
mset_pair = bitpair . mset_unpair2pair
```

We can derive the following Encoder:

```
mset2 :: Encoder N2
mset2 = compose (Iso mset_unpair2pair pair2mset_pair) nat2
```

working as follows:

```
*ISO> as mset2 nat 2008
(60,86)
*ISO> as nat mset2 it
2008
```

Figure 17 shows the curve generated by `mset_unpair` covering the lattice of points in its range.

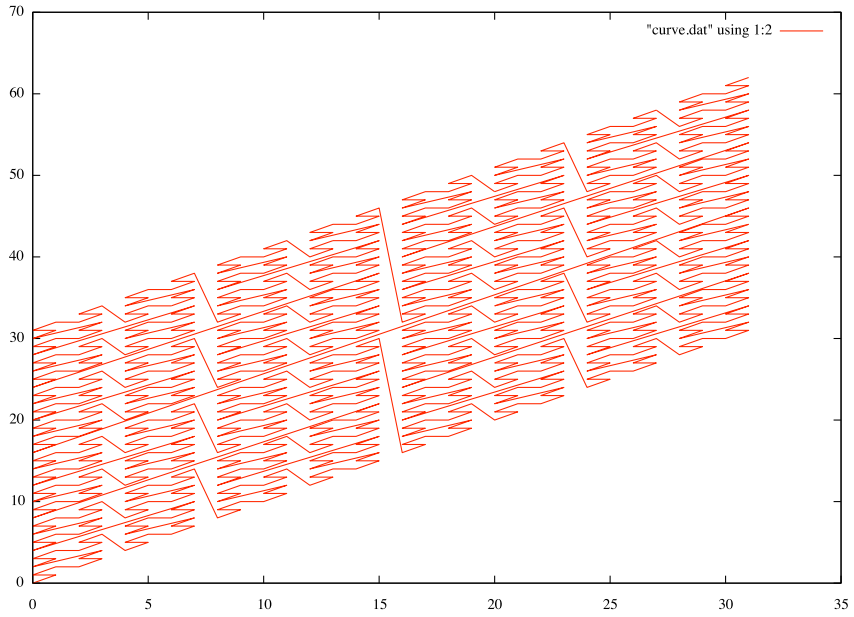


Fig. 17: 2D curve connecting values of `mset_unpair n` for $n \in [0..2^{10}-1]$

10.7 Extending Pairing/Unpairing to Signed Integers

Given the bijection from *nat* to z one can easily extend pairing/unpairing operations to signed integers. We obtain the Encoder:

```
type Z2 = (Z,Z)

z2 :: Encoder Z2
z2 = compose (Iso zpair zunpair) nat

zpair (x,y) = (nat2z . bitpair) (z2nat x,z2nat y)
zunpair z = (nat2z n,nat2z m) where (n,m)= (bitunpair . z2nat) z
```

working as follows:

```
*ISO> map zunpair [-5..5]
[(-1,1),(-2,-1),(-2,0),(-1,-1),(-1,0),(0,0),(0,-1),(1,0),(1,-1),(0,1),(0,-2)]
*ISO> map zpair it
[-5,-4,-3,-2,-1,0,1,2,3,4,5]

*ISO> as z2 z (-2008)
(63,-26)
*ISO> as z z2 it
-2008
```

Figure 18 shows the curve covering the lattice of integer coordinates generated by the function `zunpair`.

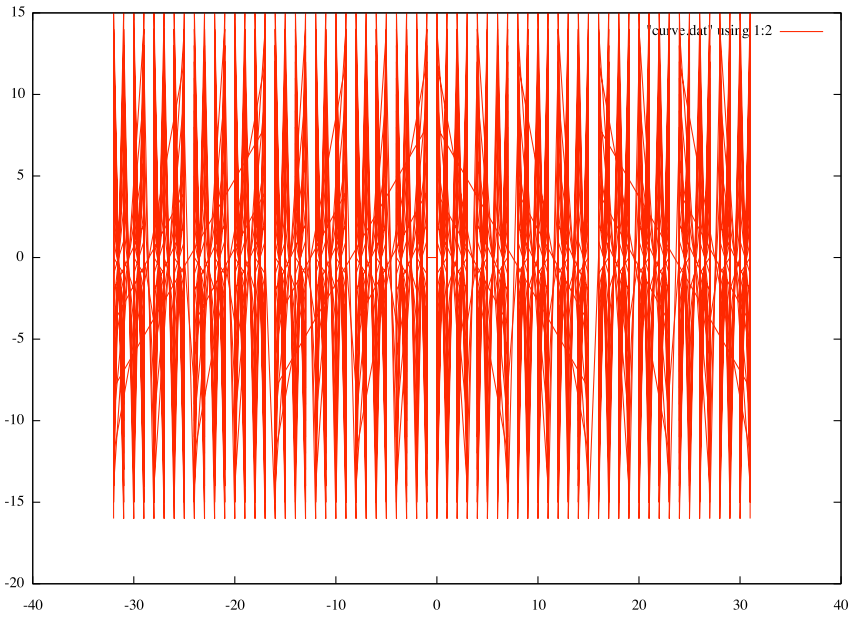


Fig. 18: Curve generated by unpairing function on signed integers

The same construction can be extended to multiset pairing functions:

```
mz2 :: Encoder Z2
mz2 = compose (Iso mzpair mzunpair) nat

mzpair (x,y) = (nat2z . mset_pair) (z2nat x,z2nat y)
mzunpair z = (nat2z n,nat2z m) where (n,m)= (mset_unpair . z2nat) z
```

working as follows:

```
*ISO> as mz2 z (-42)
(1,-8)
*ISO> as z mz2 it
-42
```

10.8 Gauss Integers and Pairing Functions

Visualizing complex variable functions requires 4 dimensions even for 1-variable functions. This is usually handled by associating a color/hue value to the *phase*

while representing the *modulus* along the z-axis. However, for 2-argument complex functions as simple as the sum, difference and the product 6 dimensions would be needed. Let us start shapeshifting operations on Gauss Integers (pairs of integers with a real and imaginary part) in combination with a mapping to ordinary integers using the (commutative!) multiset pairing/unpairing isomorphism provided by the Encoder `mz2`:

```
gauss_sum (ab,cd) = mzpair (a+b,c+d) where
  (a,b)=mzunpair ab
  (c,d)=mzunpair cd

gauss_dif (ab,cd) = mzpair (a-b,c-d) where
  (a,b)=mzunpair ab
  (c,d)=mzunpair cd

gauss_prod (ab,cd) = mzpair (a*c-b*d,b*c+a*d) where
  (a,b)=mzunpair ab
  (c,d)=mzunpair cd
```

Clearly one can now fit these operations in 3-dimensions as shown in Figures 19, 20, 21 visualizing sums, differences and products of Gauss Integers obtained by unpairing integers in $[-2^4..2^4-1]$.

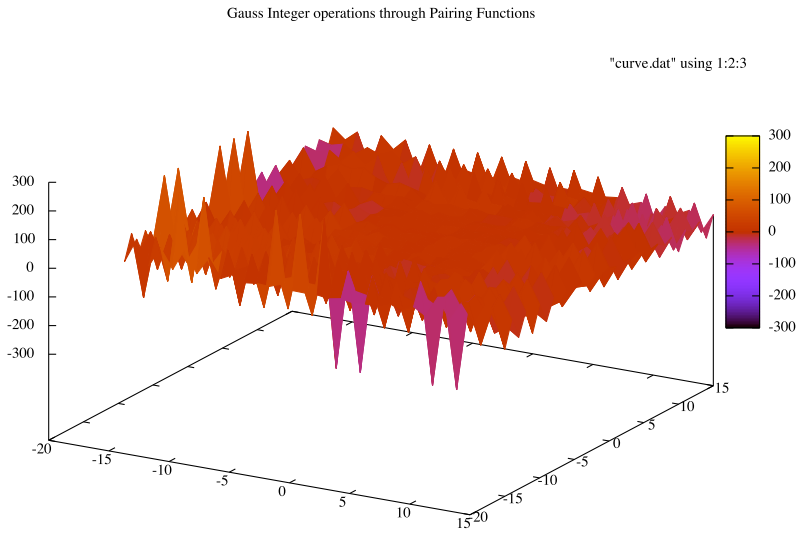


Fig. 19: Sums of Gauss Integers visualized with Pairing functions

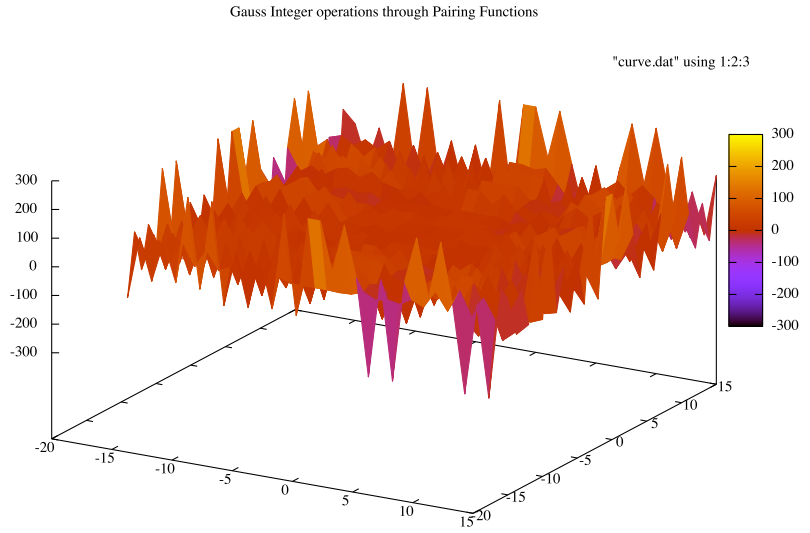


Fig. 20: Differences of Gauss Integers visualized with Pairing functions

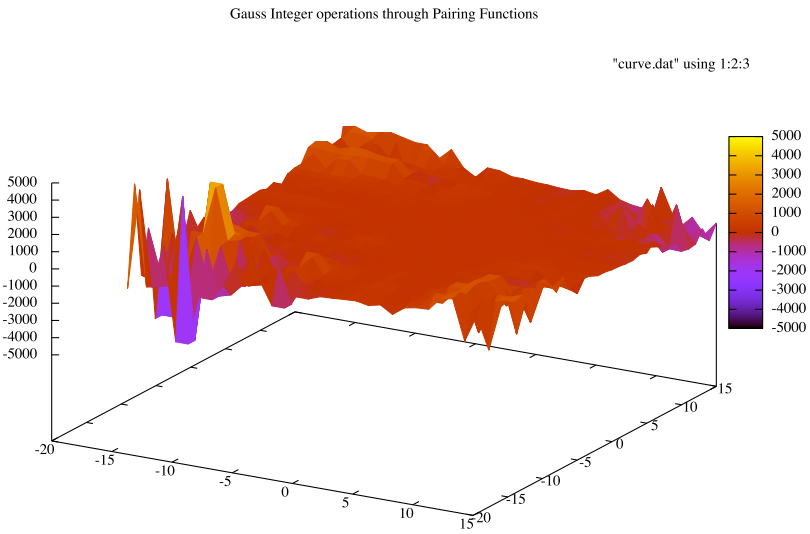


Fig. 21: Products of Gauss Integers visualized with Pairing functions

10.9 Some algebraic properties of pairing functions

The following propositions state some simple algebraic identities between pairing operations acting on ordered, unordered and multiset pairs.

Proposition 14 *Given the function definitions:*

```
bitlift x = bitpair (x,0)
bitlift' = (from_base 4) . (to_base 2)

bitclip = fst . bitunpair
bitclip' = (from_base 2) . (map (`div` 2)) . (to_base 4) . (*2)

bitpair' (x,y) = (bitpair (x,0))  +  (bitpair(0,y))
xbitpair (x,y) = (bitpair (x,0)) `xor` (bitpair (0,y))
obitpair (x,y) = (bitpair (x,0))  .|.  (bitpair (0,y))

pair_product (x,y) = a+b where
  x'=bitpair (x,0)
  y'=bitpair (0,y)
  ab=x'*y'
  (a,b)=bitunpair ab
```

the following identities hold:

$$bitlift \equiv bitlift' \tag{7}$$

$$bitclip \equiv bitclip' \tag{8}$$

$$bitclip \circ bitlift \equiv id \tag{9}$$

$$bitpair(0,n) \equiv 2 * bitpair(n,0) \tag{10}$$

$$bitpair(0,n) \equiv 2 * (bitlift\ n) \tag{11}$$

$$bitpair(n,n) \equiv 3 * (bitlift\ n) \tag{12}$$

$$bitpair(2*n, 2*m) \equiv 4 * (bitpair(n,m)) \tag{13}$$

$$bitpair(2^n, 0) \equiv (2^n)^2 \tag{14}$$

$$bitpair(2^{2^n}+1, 0) \equiv 2^{2^{n+1}}+1 \tag{15}$$

$$bitpair' \equiv bitpair \equiv xbitpair \equiv obitpair \tag{16}$$

$$bitpair(x,y) \equiv (bitlift\ x) + 2 * (bitlift\ y) \tag{17}$$

$$x < 15, y < 15 \Rightarrow pair\_productxy \equiv x * y \tag{18}$$

Proposition 15 *Given the function definitions*

```
bitpair'' (x,y) = mset_pair (min x y,x+y)

bitpair''' (x,y) = unord_pair [min x y,x+y+1]

mset_pair' (a,b) = bitpair (min a b, (max a b) - (min a b))

mset_pair'' (a,b) = unord_pair [min a b, (max a b)+1]

unord_pair' [a,b] = bitpair (min a b, (max a b) - (min a b) -1)

unord_pair'' [a,b] = mset_pair (min a b, (max a b)-1)
```

the following identities hold:

$$bitpair \equiv bitpair'' \equiv bitpair''' \tag{19}$$

$$mset\_pair \equiv mset\_pair' \equiv mset\_pair'' \tag{20}$$

$$unord\_pair \equiv unord\_pair' \equiv unord\_pair'' \tag{21}$$

11 Cons-Lists with Pairing/Unpairing

The simplest application of pairing/unpairing operations is encoding of cons-lists of natural numbers, defined as the data type:

```
data CList = Atom N | Cons CList CList
  deriving (Eq,Ord,Show,Read)
```

First, to provide an infinite supply of atoms, we encode them as even numbers:

```
to_atom n = 2*n
from_atom a | is_atom a = a `div` 2
is_atom n = even n  && n>=0
```

Next, as we want atoms and cons cells disjoint, we will encode the later as odd numbers:

```
is_cons n = odd n && n>0
decons z | is_cons z = pepis_unpair ((z-1) `div` 2)
conscell x y = 2*(pepis_pair (x,y))+1
```

We can deconstruct a natural number by recursing over applications of the unpairing-based `decons` combinator:

```
nat2cons n | is_atom n = Atom (from_atom n)
nat2cons n | is_cons n =
  Cons (nat2cons hd)
       (nat2cons tl) where
    (hd,tl) = decons n
```

We can reverse this process by recursing with the `conscell` combinator on the CList data type:

```
cons2nat (Atom a) =  to_atom a
cons2nat (Cons h t) = conscell (cons2nat h) (cons2nat t)
```

The following example shows both transformations as inverses.

```
*ISO> cons2nat (Cons (Atom 0) (Cons (Atom 1) (Cons (Atom 2) (Atom 3))))
26589
*ISO> nat2cons 26589
Cons (Atom 0) (Cons (Atom 1) (Cons (Atom 2) (Atom 3)))
```

We obtain the Encoder:

```
clist :: Encoder CList
clist = compose (Iso cons2nat nat2cons) nat
```

The Encoder works as follows:

```
*ISO> as clist nat 101
Cons (Atom 0) (Cons (Atom 0) (Atom 3))
```

and can be used to generate random LISP-like data and code skeletons from natural numbers.

12 Designing an efficient bijective Gödel numbering scheme

With all the building blocks in place, we can now proceed with the design of a compact bijective Gödel numbering algorithm.

12.1 Term Algebras

Term algebras are *free magmas* induced by a a set of variables and a set of function symbols of various arities (0 included), called `signature`, that are closed under the operation of inserting terms as an arguments of function symbols. In various logic formalisms a term algebra is called a Herbrand Universe.

We will represent function arguments as lists and assume their arity is implicitly given as the length of the the lists:

```
data Term var const =
   Var var |
   Fun const [Term var const]
   deriving (Eq,Ord,Show,Read)
```

12.2 Encoding in a term algebra with function symbols represented as natural numbers

Let's first instantiate the term algebra `Term var const` as:

```
type NTerm = Term N N
```

We will also instantiate our representation of finite sequences as:

```
nats = fun'
```

First, we will have to separate encodings of variable and function symbols. We can map them, respectively, to even and odd numbers. To deal with function arguments, we will use the bijective encoding of sequences recursively.

```
nterm2code :: Term N N → N

nterm2code (Var i) = 2*i
nterm2code (Fun cName args) = code where
  cs=map nterm2code args
  fc=as nat nats (cName:cs)
  code = 2*fc-1
```

The inverse is computed as follows:

```
code2nterm :: N → Term N N

code2nterm n | even n = Var (n `div` 2)
code2nterm n = Fun cName args where
  k = (n+1) `div` 2
  cName:cs = as nats nat k
  args = map code2nterm cs

*ISO> as nterm nat 55
Fun 1 [Fun 0 [],Var 0]
*ISO> as nat nterm it
55
```

We can encapsulate our transformers as the Encoder:

```
nterm :: Encoder NTerm
nterm = compose (Iso nterm2code code2nterm) nat
```

We shall extend this encoding for the case of more realistic term algebras where function symbols are encoded as strings.

12.3 Encoding in a term algebra with function symbols represented as strings

We can now instantiate our term algebra to have function symbols range over strings.

```
type STerm = Term N String
```

The only change from the `nterm` encoder is applying encoding/decoding to strings.

```
sterm2code :: Term N String → N

sterm2code (Var i) = 2*i
sterm2code (Fun name args) = code where
  cName=as nat string name
```

```
  cs=map sterm2code args
  fc=as nat nats (cName:cs)
  code=2*fc-1
```

The inverse is computed as follows:

```
code2sterm :: N → Term N String

code2sterm n | even n = Var (n `div` 2)
code2sterm n = Fun name args where
  k = (n+1) `div` 2
  cName:cs = as nats nat k
  name = as string nat cName
  args = map code2sterm cs
```

We can encapsulate our transformers as the Encoder:

```
sterm :: Encoder STerm
sterm = compose (Iso sterm2code code2sterm) nat

*ISO> as sterm nat 1234567
Fun "%" [Var 28,Var 7]
*ISO> as nat sterm it
1234567
*ISO> as nat sterm (Fun "forall" [Var 0, Fun "f" [Var 0]])
265918616110153395888011258947597984 7
*ISO> as sterm nat it
Fun "forall" [Var 0,Fun "f" [Var 0]]
```

12.4 Mapping terms to arbitrary bitstrings

Term algebras are free magmas generated through fairly complex substitution operations. Their underlying data representations involve ordered trees. Can we design a bijective mapping to, arguably, the simplest possible free magma - the set of strings on $\{0, 1\}$? The answer is affirmative, provided that we use a mapping from *arbitrary* bitstrings to natural numbers, as provided by the Encoder `bits`.

Using the `as` combinator we obtain:

```
nterm2bits = as bits nterm
bits2nterm = as nterm bits

sterm2bits = as bits sterm
bits2sterm = as sterm bits

*ISO> as nterm bits
  [0,0,0,1,0,1,0,0,1,1,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,
   0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Fun 0 [Fun 1 [Var 0,Fun 1 [Var 0]],Var 1]
*ISO> as bits nterm
```

```
    (Fun 0 [Fun 1 [Var 0,Fun 1 [Var 0]],Var 1])
[0,0,0,1,0,1,0,0,1,1,0,1,0,0,0,0,0,1,0,0,0,0,0,1,
 0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

The following is a consequence of the fact that each of the encoding steps are linear in the bitsize of their input and run in linear time and preserve syntactic validity by structural induction.

Proposition 16 *The Gödel numbering algorithms implemented by the encoders* `nterm` *and* `sterm` *are bijective and work in linear time and space linear in the bitsize of their input. Moreover, all natural numbers decode to syntactically valid terms.*

12.5 Deriving representations of finite sequences from one-solution Diophantic equations

Let's first observe that

Proposition 17 $\forall z \in \mathbb{N} - \{0\}$ *the diophantic equation*

$$2^x(2y+1) = z \tag{22}$$

has exactly one solution $x, y \in \mathbb{N}$.

This follows immediately from the unicity of the decomposition of a natural number as a multiset of prime factors.

This equation justifies the fact that the `cons, hd, tl` functions can be used to uncover a list structure inside natural numbers. The mechanism can be generalized to obtain a family of finite sequence encoders by choosing an arbitrary base `b` instead of 2. Care should be taken in this case to *rebase* i.e. switch from/to base `b-1` to ensure that we obtain a bijection. Such rebasing operations are similar to those used in generating the sequences used in Goodstein's theorem [28]. First we implement the head and tail operations `xhd` and `xtl` as follows:

```
xhd :: N→N→N
xhd b n | b>1 && n>0 =
  if n `mod` b > 0 then 0 else 1+xhd b (n `div` b)

xtl :: N→N→N
xtl b n = y-1 where
  y'= n `div` b^(xhd b n)
  y=rebase b (b-1) y'

rebase fromBase toBase n =  toBase*q + r where
  (q,r)=n `quotRem` fromBase
```

The `xcons` operation aggregates back the head and tail:

```
xcons :: N→N→N→N
xcons b x y | b>1 = (b^x)*(y'+1) where
  y'=rebase (b-1) b y
```

The following examples show the operations for base 3:

```
*ISO> xcons 3 10 20
1830519
*ISO> xhd 3 1830519
10
*ISO> xtl 3 1830519
20
```

We can extend the mechanism to derive *pairing/unpairing* functions as follows:

```
xunpair b n = (xhd b n',xtl b n') where n'=n+1

xpair b (x,y) = (xcons b x y)-1
```

One can see that we obtain a family of bijections $f_b : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ for every base b.

```
*ISO> map (xunpair 3) [0..7]
[(0,0),(0,1),(1,0),(0,2),(0,3),(1,1),(0,4),(0,5)]
*ISO> map (xpair 3) it
[0,1,2,3,4,5,6,7]
```

For each base `b>1` one obtains a pair of mappings:

```
xsplit _ 0 = []
xsplit b x = xhd b x : xsplit b (xtl b x)

xfuse _ [] = 0
xfuse b (x:xs) = xcons b x (xfuse b xs)
```

working as follows:

```
*ISO> xsplit 3 2009
[0,0,0,3,0,2]
*ISO> xfuse 3 it
2009
```

For each pair $l, k \in \mathbb{N}$ one can generate a family of bijections $\mathbb{N} \to \mathbb{N}$ by combining a `split` and a `fuse`

```
xbij k l n = xfuse l (xsplit k n)
```

working as follows:

```
*Main> map (xbij 2 3) [0..31]
[0,1,3,2,9,5,6,4,27,14,15,8,18,10,12,7,81,41,42,
 22,45,23,24,13,54,28,30,16,36,19,21,11]
*Main> map (xbij 3 2) [0..31]
[0,1,3,2,7,5,6,15,11,4,13,31,14,23,9,10,27,63,
 12,29,47,30,19,21,22,55,127,8,25,59,26,95]
```

It is easy th see that the following holds:

Proposition 18

$$(xbij\ k\ l) \circ (xbij\ l\ k) \equiv id \tag{23}$$

This might have applications to cryptography, provided that a method is devised to generate "interesting" pairs k,l defining the encoding.

More complex encodings can be obtained with:

```
ybij m | m>3 = (xbij 3 m) . (xbij m 2)

*ISO> map (ybij 4) [0..31]
[0,1,4,5,2,24,6,25,8,7,80,13,22,33,230,64,3,23,34,
 356,44,65,16,70,11,107,362,26,115,196,17,99]
```

13 Revisiting Multiset Encodings

We will now use pairing/unpairing functions, in combination with mappings to sequences and sets to design an efficient encoding of multisets.

The function `fmset2nat` starts by grouping the elements of a multiset. The lengths of the groups (decremented by 1), as well as an element of each are then collected in 2 lists. Then the second list is morphed from a set to a sequence, as this provides a more compact representation without changing the length of the list. The first list, seen as a sequence is then paired element by element with the second list. Finally, the resulting numbers, seen as a sequence, are then fused together.

```
fmset2nat pairingf ms = m where
  mss= group (sort ms)
  xs=map (pred . genericLength) mss
  zs=map head mss
  ys=set2fun zs
  ps=zip xs ys
  ns=map pairingf ps
  m=fun2nat ns
```

The function `fnat2mset` reverses the process step by step:

```
fnat2mset unpairingf m = rs where
   ns=nat2fun m
   ps=map unpairingf ns
   (xs,ys)=unzip ps
   xs'=map succ xs
   zs=fun2set ys
   f k x = genericTake k (repeat x)
   rs = concat (zipWith f xs' zs)
```

After instantiating these generic functions to interesting pairing/unpairing functions

```
bmset2nat = fmset2nat bitpair
nat2bmset = fnat2mset bitunpair

bmset2nat' = fmset2nat pepis_pair
nat2bmset' = fnat2mset pepis_unpair
```

We obtain the Encoders:

```
bmset :: Encoder [N]
bmset = compose (Iso bmset2nat nat2bmset) nat

bmset' :: Encoder [N]
bmset' = compose (Iso bmset2nat' nat2bmset') nat
```

working as follows:

```
ISO> as bmset nat 2008
[1,1,2,3,3,4,5,6,7]
*ISO> as nat bmset it
2008
*ISO> map (as bmset nat) [0..7]
[[],[0],[0,0],[0,1],[1],[0,1,1],[0,0,1],[0,1,2]]
*ISO> as bmset' nat 2008
[0,0,0,1,2,2,3,4,5,6]
```

Note that, in contrast to the intractable prime number based multiset encoding `pmset`, this time we obtain an encoding, linear in the bitsize of the natural numbers involved, as in the case of `mset`. Note also that the construction is generic in the sense that it works with any pairing / unpairing function. Like in the case of `mset` and `pmset` multiset encodings we can extend these encodings to a hylomorphism `hfbm`:

```
nat_bmset = Iso nat2bmset bmset2nat

hfbm :: Encoder T
hfbm = compose (hylo nat_bmset) nat

nat_bmset' = Iso nat2bmset' bmset2nat'

hfbm' :: Encoder T
hfbm' = compose (hylo nat_bmset') nat
```

working as follows:

```
*ISO> as hfbm nat 42
H [H [],H [],H [H []],H [H []],H [H [],H []],H [H [],H []]]
*ISO> as nat hfbm it
42
*ISO> as hfbm' nat 2008
H [H [],H [],H [],H [H []],H [H [],H []],
   H [H [],H []],H [H [],H [H []]],H [H [H []]],
   H [H [],H [H []],H [H []]],H [H [],H [],H [H []]]]
*ISO> as nat hfbm' it
2008
```

13.1 Revisiting Gödel's original encoding of finite sequences

Assuming that function symbols, variables and punctuation in a formula language have been mapped to consecutive natural numbers, Gödel's original prime number based encoding can be implemented as follows:

```
nats2goedel ns =  product xs where
  xs=zipWith (^) primes ns
```

i.e. by computing $2^{n_0} * 3^{n_1} * \ldots p_i^{n_i} * \ldots$ for $n_i \in ns$. We can reverse the process:

```
goedel2nats n = combine ds xs where
  pss=group (to_primes n)
  ps=map head pss
  xs=map genericLength pss
  ds=as fun set (map pi' ps)

  combine [] [] = []
  combine (b:bs) (x:xs) =
    replicate (fromIntegral b) 0 ++ x:(combine bs xs)

pi' p | is_prime p= to_pos_in primes p
```

The encoding/decoding so far works as follows:

```
*Iso> goedel2nats 2009
[0,0,0,2,0,0,0,0,0,0,0,0,1]
*Iso> nats2goedel [0,0,0,2,0,0,0,0,0,0,0,0,1]
2009
*Iso> nats2goedel [0,0,0,2,0,0,0,0,0,0,0,0,1,0,0,0]
2009
```

Reversing it requires factoring and, as seen from the previous example, needs a small fix: to avoid zeros after the last element in a sequence being ignored, we have to add how many of them are found at the end of the sequence, as part of the code. To accomodate 0 and 1, we treat 0 as a special case and shift by 1 by applying `succ` before calling `goedel2nats`.

```
goedel :: Encoder [N]
goedel = compose (Iso nats2g g2nats) nat

nats2g [] = 0
nats2g ns = pred (nats2goedel (z:ns)) where
  z=countZeros (reverse ns)

g2nats 0 = []
g2nats n = ns ++ (replicate (fromIntegral z) 0) where
  (z:ns)=goedel2nats (succ n)

countZeros (0:ns) = 1+countZeros ns
countZeros _ = 0
```

With the fix, Gödel's original encoding now works as a bijection $\mathbb{N} \to [\mathbb{N}]$:

```
*Iso> as goedel nat 2009
[1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]
*Iso> as nat goedel it
2009
```

A prime number based encoding similar to Gödel's original encoding can be derived from the encoder `pmset`, more directly, as follows:

```
goedel' :: Encoder [N]
goedel' = compose (Iso nats2gnat gnat2nats) nat

nats2gnat = pmset2nat . (as mset fun)

gnat2nats = (as fun mset) . nat2pmset

*Iso> as goedel' nat 2009
[0,1,1,16]
*Iso> as nat goedel' it
2009
```

14 Pairing Functions and Encodings of Binary Decision Diagrams

As a variation on the theme of pairing/unpairing functions, 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)
```

14.1 Unfolding natural numbers to binary trees

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 :: N2 → BDD N
unfold_bdd (n,tt) = BDD n bt where
  bt=if tt<max then split_with bitunpair n tt
     else error
       ("unfold_bdd: last arg "++ (show tt)++
       " should be < " ++ (show max))
     where max = 2^2^n

split_with _ 0 0 =  B0
split_with _ 0 1 =  B1
split_with f n tt | n>0 =
   D k (split_with f k tt1)
       (split_with f k tt2) where
    k=pred n
    (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.

14.2 Folding binary trees to natural numbers

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 N → N2
fold_bdd (BDD n bt) =
  (n,fuse_with bitpair bt) where
    fuse_with rf B0 = 0
    fuse_with rf B1 = 1
    fuse_with rf (D _ l r) =
      rf (fuse_with rf l,fuse_with 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 (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
```

14.3 Boolean Evaluation of BDDs

Practical uses of BDDs involve reducing them by sharing nodes and eliminating identical branches [29]. 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 _ _ 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, as shown in [30]. 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 (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
```

14.4 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 19 *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.*

$$fold\_bdd \circ unfold\_bdd \equiv id \tag{24}$$

$$eval\_bdd \circ unfold\_bdd \equiv id \tag{25}$$

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 [31] 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 14 actually represents BDDs mapped one-to-one to truth tables given as natural numbers. An interesting application of this result would be to investigate practical uses of `bitpair`/`bitunpair` operations in actual circuit design.

15 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 $\mathbb{N}$: we will have to "shift towards infinity" the starting point of each new block[4] of BDDs in $\mathbb{N}$ 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[5]:

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

[4] defined by the same number of variables

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

```
*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 (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
  (_,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:

```
pbdd :: Encoder (BDD N)
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 N)
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.

15.1 Reducing the *BDDs*

We will sketch here a simplified reduction mechanism for BDDs eliminating identical branches. As nodes of a BDD are mapped bijectively to unique natural numbers we will omit the (trivial) implementation of node sharing, with the implicit assumption that subtrees having the same encoding are shared.

The function `bdd_reduce` reduces a *BDD* by collapsing identical left and right subtrees, and the function `bdd` associates this reduced form to $n \in \mathbb{N}$.

```
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 N)
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)
```

This measures the size of the BDD or reduced BDD as an expression tree. To take into account sharing (as present in a standard ROBDD implementation) one can simply eliminate duplicated subtrees:

```
robdd_size (BDD _ t) = 1+(rsize t) where
  rsize = genericLength . nub . rbdd_nodes
  rbdd_nodes B0 = [B0]
  rbdd_nodes B1 = [B1]
  rbdd_nodes (D v l r) =
    [(D v l r)] ++ (rbdd_nodes l) ++ (rbdd_nodes r)
```

16 Generalizing BDD ranking/unranking functions

16.1 Encoding BDDs with Arbitrary Variable Order

While the encoding built around the equivalence described in Prop. 19 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 [29].

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 [31], 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
  (f1,f0)=unpair_mn v m n tt
  l=to_bdd_mn vs f1 m n
  r=to_bdd_mn vs f0 m n
```

```
unpair_mn v m n tt = (f1,f0) where
  cond=var_mn m n v
  f0= (m `xor` cond) .&. tt
  f1= cond .&. tt
```

Proposition 20 *The function* `to_bdd` *builds an (unreduced) BDD corresponding to a truth table* `tt` *for variable order* `vs` *that returns* `tt` *when evaluated as a boolean function.*

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)))
*ISO> to_rbdd (as perm nat 8) 42
BDD 3 (D 2 B0 (D 0 B1 (D 1 B1 B0)))
ISO> from_bdd it
42
```

Two interesting instances are worth pointing out. After defining

```
nat2bdd0 n = b where
  b=to_bdd (reverse [0..k-1]) m_n
  (k,m_n)=to_bsum n
```

the following holds:

Proposition 21 *nat2bdd0* $\equiv$ *nat2bdd*

After defining

```
nat2bddn n = b where
  b=to_bdd [0..k-1] m_n
  (k,m_n)=to_bsum n

bddn :: Encoder (BDD N)
bddn = compose (Iso ev_bdd2nat nat2bddn) nat
```

we observe that `bddn` works in reverse variable order compared to `bdd`.

```
*ISO> as bdd nat 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> as bddn nat 42
BDD 3 (D 0 (D 1 (D 2 B0 B0) (D 2 B1 B0)) (D 1 (D 2 B1 B0) (D 2 B0 B0)))
*ISO> as nat bddn 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 t = search_bdd min n t

search_bdd f n tt = snd (foldl1 f
 (map (sized_rbdd tt) (all_permutations n))) where
    sized_rbdd tt vs = (robdd_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 show, 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 0 (D 2 B0 B1) (D 1 (D 2 B0 B1) B0))
*ISO> to_min_bdd 4 2008
BDD 4 (D 3 (D 1 (D 0 B0 B1) (D 0 B1 B0))
      (D 2 (D 1 (D 0 B0 B1) B0) (D 0 B1 B0)))
*ISO> to_min_bdd 7 2008
BDD 7 (D 0 (D 1 (D 2 (D 6
                     (D 4 (D 3 B0 B1) (D 3 B1 B0))
      (D 5 (D 4 (D 3 B0 B1) B0)
                (D 3 B1 B0))) B0) B0) B0)
*ISO> robdd_size it
12
```

16.2 Multi-Terminal Binary Decision Diagrams (MTBDD)

MTBDDs [32, 33] 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 15.

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 = Lf 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 $N \to M$ 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) = Lf tt
to_mtbdd_ mlimit n tt = (M k l r) where
   (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 (Lf 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 (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 (Lf 2) (Lf 1))
         (M 0 (Lf 2) (Lf 1)))
      (M 1
         (M 0 (Lf 2) (Lf 0))
         (M 0 (Lf 1) (Lf 1))))

>from_mtbdd it
```

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

16.3 Encoding Powerlists

Powerlists [34] are lists of length 2^n on which a merge operation (called `zip`) and a concatenation operation (called `tie`) are defined. They are important for specifying/implementing parallel operations and efficient arithmetic operation hardware. In [35] a data type `Plist` is given that enforces the 2^n length using a Peano arithmetic successor to describe depth while recursing on pairs, but as we will see our BDD representation does the same without using successor arithmetics.

We will also describe here ranking/unranking operations on powerlists and their relation to our BDD types.

We start with a linear list specification of their merge and concatenation operations (assuming informally that their length is a power of two and that they are nonempty):

```
lTie xs ys = xs ++ ys

lZip [] [] = []
lZip (x:xs) (y:ys) = x:y:(lZip xs ys)
```

working as follows

```
*ISO> lZip (lTie [0] [1]) (lTie [2] [3])
[0,2,1,3]
```

We can convert a BDD to a linear representation of a powerlist with binary scalars enforcing the constraints on length, simply by collecting its leaves:

```
bdd2plist (BDD _ b) = bt2plist b where
  bt2plist B0 = [0]
  bt2plist B1 = [1]
  bt2plist (D _ l r) = (bt2plist l) ++ (bt2plist r)
```

Depending on the variable order used to build a BDD we obtain different powerlists. As we will see later, of particular interest are the cases when the Encoders `bdd` *and* `bddn` *are used.*

```
*ISO> bdd2plist (as bdd nat 42)
[0,1,1,0,0,0,0,0]
*ISO> bdd2plist (as bddn nat 42)
[0,0,1,0,1,0,0,0]
```

Let us define a tree representation that describes powerlists as a combination of `Tie` and `Zip` operations.

```
data PL a =  Sc a | Tie (PL a) (PL a) | Zip (PL a) (PL a)
  deriving (Eq,Show,Read)
```

We first design converters from BDDs:

```
bdd2pl f (BDD _ b) = bt2pl f b where
  bt2pl _ B0 = Sc 0
  bt2pl _ B1 = Sc 1
  bt2pl f (D _ l r) = f (bt2pl f l) (bt2pl f r)

bdd2zip b = bdd2pl Zip b

bdd2tie b = bdd2pl Tie b
```

We can flatten a powerlist to a linear list representation:

```
flattenPL (Sc a)= [a]
flattenPL (Tie x y) = lTie (flattenPL x) (flattenPL y)
flattenPL (Zip x y) = lZip (flattenPL x) (flattenPL y)
```

We can compute the depth of a powerlist, knowing that all leaves are at the same distance from the root.

```
depthPL (Sc _)= 0
depthPL (Tie x _) = 1+(depthPL x)
depthPL (Zip x _) = 1+(depthPL x)
```

We are now ready to define two encoders:

```
powerZip :: Encoder (PL N)
powerZip = compose (Iso pl2nat nat2zip) nat

powerTie :: Encoder (PL N)
powerTie = compose (Iso pl2nat nat2tie) nat

nat2zip n = bdd2pl Zip (as bdd nat n)
nat2tie n = bdd2pl Tie (as bddn nat n)

pl2nat pl = bsum (depthPL pl)+(as nat bits1 (flattenPL pl))
```

that provide Zip and Tie based representations of a natural number as a powerlist. Note that `pl2nat` is generically used to rank both `Zip` and `Tie` based powerlists while `nat2zip` and `nat2tie` respectively unrank a natural number to a `Zip` or `Tie` based powerlist. As in the case of BDDs note the use of `bsum` (see section 15)) computing the number of boolean functions up to n variables, needed to ensure that the encoding is a bijection. Also note, that, as it is known

from powerlist algebra [34], the same powerlist can be expressed indeed in two alternative forms in terms of `Zip` or `Tie`:

```
*ISO> as powerZip nat 42
Zip (Zip (Zip (Sc 0) (Sc 1)) (Zip (Sc 1) (Sc 0)))
    (Zip (Zip (Sc 0) (Sc 0)) (Zip (Sc 0) (Sc 0)))
*ISO> as powerTie powerZip it
Tie (Tie (Tie (Sc 0) (Sc 0)) (Tie (Sc 1) (Sc 0)))
    (Tie (Tie (Sc 1) (Sc 0)) (Tie (Sc 0) (Sc 0)))
*ISO> as nat powerTie it
42
```

As the example suggests, we obtain as "free algorithms" converters between Zip and Tie representation of powerlists as follows:

```
zip2tie = as powerTie powerZip
tie2zip = as powerZip powerTie
```

The crux of these equivalence results is the use of the Encoders `bdd` and `bddn` to obtain isomorphic BDD representations derived from two different variable orders, $[0..n-1]$ and $[n-1..0]$ for the same natural number in $[0..2^{2^n}-1]$ seen as truth table.

An interesting application is that automorphisms on powerlists induce automorphisms on natural numbers. As in [34]) one can define a reverse operation on powerlists:

```
rev (Sc x) = Sc x
rev (Tie x y) = Tie (rev y) (rev x)
rev (Zip x y) = Zip (rev y) (rev x)
```

we obtain

```
*ISO> borrow_from powerTie rev nat 42
62
*ISO> borrow_from powerTie rev nat 62
42
```

After defining

```
tierev = borrow_from powerTie rev nat
ziprev = borrow_from powerZip rev nat
```

we observe that

```
*ISO> map tierev [0..15]
[0,1,2,4,3,5,6,14,10,18,8,16,12,20,7,15]
*ISO> map tierev it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
*ISO> map ziprev [0..15]
[0,1,2,4,3,5,6,14,10,18,8,16,12,20,7,15]
*ISO> map ziprev it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
*ISO>
```

indicating that this is indeed an automorphism on the set of natural numbers.

Alternatively, operations borrowed from natural numbers can enrich the powerlist algebra with new operations

```
tieSucc = borrow_from nat succ powerTie
tiePred = borrow_from nat pred powerTie
tieSum = borrow_from2 nat (+) powerTie

zipSucc = borrow_from nat succ powerZip
zipPred = borrow_from nat pred powerZip
zipSum = borrow_from2 nat (+) powerZip
```

working as follows:

```
*ISO> tieSucc (Sc 0)
Sc 1
*ISO> tieSucc (tieSucc (Sc 0))
Tie (Sc 0) (Sc 0)
*ISO> tiePred (tiePred it)
Sc 0
*ISO> zipSum (Zip (Sc 0) (Sc 0)) (Sc 1)
Zip (Sc 1) (Sc 0)
*ISO> tieSum (Tie (Sc 0) (Sc 0)) (Sc 1)
Tie (Sc 1) (Sc 0)
```

As powerlist representation are well suited for parallel processing it looks like an interesting avenue to explore if they can actually provide hardware support for high performance parallel arithmetic operations on arbitrary size integers.

17 Revisiting Encodings of Finite Functions

We will now generalize the `bitpair` pairing function to k-tuples and then we will derive an alternative encoding for finite functions.

17.1 Tuple Encodings as Generalized Bitpair

The function `to_tuple`: $\mathbb{N} \to \mathbb{N}^k$ converts a natural number to a k-tuple by splitting its bit representation into k groups, from which the k members in the tuple are finally rebuilt. This operation can be seen as a transposition of a bit matrix obtained by expanding the number in base 2^k:

```
to_tuple k n | k>0 && n≥0 = map (from_base 2) (
    transpose (
      map (to_maxbits k) (
        to_base (2^k) n
      )
    )
  )
```

To convert a k-tuple back to a natural number we will merge their bits, k at a time. This operation uses the transposition of a bit matrix obtained from the tuple, seen as a number in base 2^k, with help from bit crunching functions given in APPENDIX:

```
from_tuple ns = from_base (2^k) (
    map (from_base 2) (
      transpose (
        map (to_maxbits l) ns
      )
    )
  ) where
      k=genericLength ns
      l=max_bitcount ns
```

The following example shows the decoding of `42`, its decomposition in bits (right to left), the formation of a 3-tuple and the encoding of the tuple back to `42`.

```
*ISO> to_base 2 42
[0,1,0,1,0,1]
*ISO> to_tuple 3 42
[2,1,2]
*ISO> to_base 2 2
[0,1]
*ISO> to_base 2 1
[1]
*ISO> from_tuple [2,1,2]
42
```

Fig. 22 shows multiple steps of the same decomposition, with shared nodes collected in a DAG.

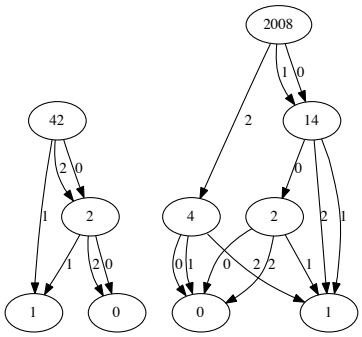


Fig. 22: Repeated 3-tuple expansions: *42* and *2008*

17.2 Encoding Finite Functions as Tuples

As finite sets can be put in a bijection with an initial segment of $\mathbb{N}$, a `finite function` can be seen as a function defined from an initial segment of $\mathbb{N}$ to $\mathbb{N}$. We can encode and decode a finite function from $[0..k-1]$ to $\mathbb{N}$ (seen as the list of its values), as a natural number:

```
ftuple2nat [] = 0
ftuple2nat ns = succ (pepis_pair (pred k,t)) where
  k=genericLength ns
  t=from_tuple ns

nat2ftuple 0 = []
nat2ftuple kf = to_tuple (succ k) f where
  (k,f)=pepis_unpair (pred kf)
```

As the length of the tuple, `k`, is usually smaller than the number obtained by merging the bits of the `k`-tuple, we have picked the Pepis pairing function, exponential in its first argument and linear in its second, to embed the length of the tuple needed for the decoding. This suggest the following alternative Encoder for finite functions:

```
fun' :: Encoder [N]
fun' = compose (Iso ftuple2nat nat2ftuple) nat
```

as well as the related alternative hylomorphism:

```
nat_fun' = Iso nat2ftuple ftuple2nat

hff' :: Encoder T
hff' = compose (hylo nat_fun') nat
```

The encoding/decoding and the hylomorphism work as follows:

```
*ISO> as fun' nat 2008
[3,2,3,1]
*ISO> as nat fun' it
2008

*ISO> as hff' nat 2008
H [H [H [H []]],H [H [],H []],H [H [H []]],H [H []]]
*ISO> as nat hff' it
2008
```

Given this alternative encoding for finite functions, one can derive from it new Encoders for multisets and sets as follows:

```
mset2nat' = ftuple2nat . mset2fun
nat2mset' = fun2mset . nat2ftuple

nat_mset' = Iso nat2mset' mset2nat'

mset' :: Encoder [N]
```

```
mset' = compose (invert nat_mset') nat

set2nat' = ftuple2nat . set2fun
nat2set' = fun2set . nat2ftuple

nat_set' = Iso nat2set' set2nat'

set' :: Encoder [N]
set' = compose (invert nat_set') nat
```

working as follows:

```
*ISO> set2nat [7,22,45]
35184376283264
*ISO> set2nat' [7,22,45]
143308
*ISO> nat2set' 143308
[7,22,45]
*ISO> mset2nat [7,22,22,45,45]
844424955297920
*ISO> mset2nat' [7,22,22,45,45]
270904688
*ISO> nat2mset' 270904688
[7,22,22,45,45]
```

We can define new Encoders lifting these operations to hereditarily finite structures:

```
hfs' :: Encoder T
hfs' = compose (hylo nat_set') nat

hfm' :: Encoder T
hfm' = compose (hylo nat_mset') nat
```

We can also define alternative `hd, tl, cons` and pairing/unpairing operations:

```
hd' = head . nat2ftuple
tl' = ftuple2nat . tail . nat2ftuple
cons' h t  = ftuple2nat (h:(nat2ftuple t))

pair' (x,y) = (cons' x y)-1
unpair' z = (hd' z', tl' z') where z'=z+1
```

Fig. 23 shows the values of `z=pair' (x,y)` for $x, y \in [0..2^6 - 1]$.

An unusual thing about this pairing function is that it reaches very large values strictly when its both arguments are powers of 2, for instance:

```
*ISO> pair' (4,4)
4103
*ISO> pair' (3,4)
279
```

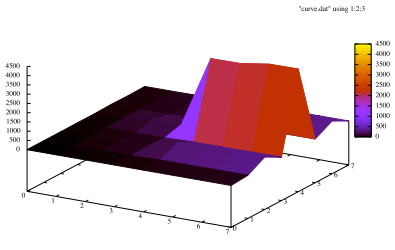


Fig. 23: Values of z=pair' (x,y)

```
*ISO> pair' (4,3)
73
```

Also, the bit splitting mechanism used in `bitpair/bitunpair` can be generalized using tuples, by allowing an arbitrary division in `k+l` groups with `k` aggregating into the first element and `l` aggregating as the second element of the pair as follows:

```
pairKL k l (x,y) = from_tuple (xs ++ ys) where
   xs = to_tuple k x
   ys = to_tuple l y

unpairKL k l z = (x,y) where
  zs=to_tuple (k+l) z
  xs=genericTake k zs
  ys=genericDrop k zs
  x=from_tuple xs
  y=from_tuple ys
```

Fig. 24 shows the values of `z=pairKL 3 2 n` for $n \in [0..2^6 - 1]$.

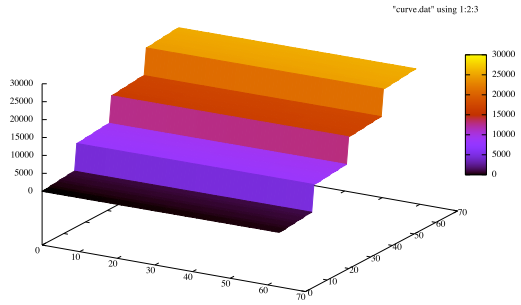


Fig. 24: Values of pairKL

Fig. 25 shows the path obtained by connecting pairs `(x,y) = unpairKL 2 3` for $n \in [0..15]$.

18 Directed Graphs, DAGs, Undirected graphs, Multigraphs and Hypergraphs

We will now show that more complex data types like digraphs, DAGs and hypergraphs have extremely simple encoders. This shows once more the importance of compositionality in the design of our embedded transformation language.

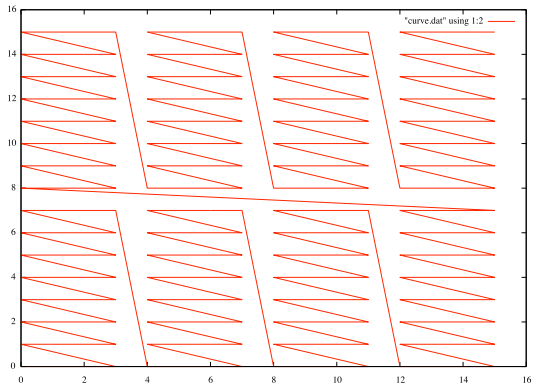


Fig. 25: Path connecting unpairKL

18.1 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 [N2]
digraph = compose (Iso digraph2set set2digraph) set
```

working as follows:

```
*ISO> as digraph nat 2008
[(1,1),(2,0),(2,1),(3,1),(0,2),(1,2),(0,3)]
*ISO> as nat digraph it
2008
*ISO> as digraph nat 17
[(0,0),(2,0)]
*ISO> as nat digraph it
17
```

Fig. 26 shows the digraph associated to 2008.

Fig. 27 shows the digraph associated to 17. Note that in this figure (and in the subsequent ones) isolated vertices are will not be drawn, but implicitly assumed present as shown in Fig. 28.

Note also that this encoding is parameterized by the pairing/unpairing functions used, i.e. an alternative encoding can be obtained as:

```
digraph2set' ps = map pepis_pair ps
set2digraph' ns = map pepis_unpair ns
```

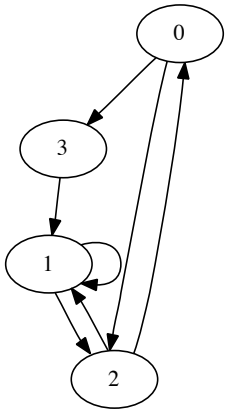


Fig. 26: 2008 as a digraph

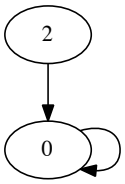


Fig. 27: 17 as a digraph with isolated vertex ignored

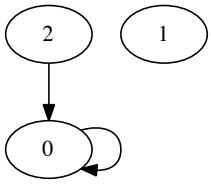


Fig. 28: 17 as a digraph with isolated vertex shown

The resulting Encoder is:

```
digraph' :: Encoder [N2]
digraph' = compose (Iso digraph2set' set2digraph') set
```

Fig. 29 shows this alternative digraph associated to 2008.

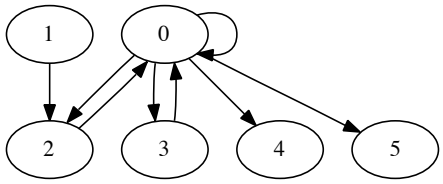


Fig. 29: 2008 as a different digraph

Discussion. For digraphs understood as subsets of $N \times N$, i.e. provided that a canonical mapping of vertices to an initial segment of N is assumed, our representations are clearly bijections. Once the mapping is fixed, a digraph is seen as a list of edges, each mapped to a distinct natural number using a pairing function. As all edges are distinct, the resulting list represents a set - which then is mapped to a unique natural number. Note also that digraphs can be disconnected and isolated vertices can be represented simply as vertices not occurring in the list of edges, assuming that the canonical mapping to vertices is such that the last vertex is connected to at least one other vertex. Under these assumptions, "digraphs" consisting only of isolated vertices collapse to the same encoding as the empty digraph. To avoid this problem, one can pair the representation of a digraph with a number indicating how many such vertices are considered part of the digraph after the last connected vertex occurring in an edge.

18.2 Encoding Undirected Graphs

We can find a bijection from undirected graphs to finite sets by fusing their list of unordered pair representation into finite sets with a pairing function on multiset pairs:

```
graph2mset ps = map mset_pair ps
mset2graph ns = map mset_unpair ns
```

The resulting Encoder is:

```
graph :: Encoder [N2]
graph = compose (Iso graph2mset mset2graph) set
```

working as follows:

```
*ISO> as graph nat 2008
[(1,2),(2,2),(2,3),(3,4),(0,2),(1,3),(0,3)]
*ISO> as nat graph it
2008
```

Note that, as expected, the result is invariant to changing the order of elements in pairs like `(1,2)` and `(3,4)` to `(2,1)` and `(4,3)`.

18.3 Encoding Directed Multigraphs

We can find a bijection from directed multigraphs (directed graphs with multiple edges between pairs of vertices, also called *multi-digraphs* or *quivers*) to finite sequences by fusing their list of ordered pair representation into finite sequences with a pairing function:

The resulting Encoder is:

```
mdigraph :: Encoder [N2]
mdigraph = compose (Iso digraph2set set2digraph) fun
```

working as follows:

```
*ISO> as mdigraph  nat 1234567890
[(1,0),(0,1),(1,0),(0,0),(1,0),(3,1),(0,0),(1,0),(0,1),(0,0),(0,1),(0,1)]
```

Note that the only change to the `digraph` Encoder is replacing the composition with `set` by a composition with `fun`.

Fig. 30 depicts the directed multi-digraph associated to 1234567890. Note that position in the sequence of pairs provides the (unique) label marking each edge.

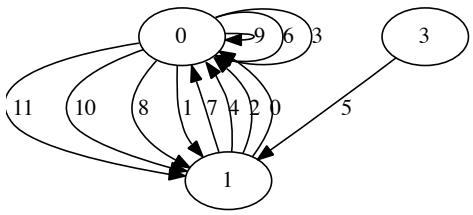


Fig. 30: 1234567890 as a directed multigraph

18.4 Encoding Undirected Multigraphs

We can find a bijection from undirected multigraphs (undirected graphs with multiple edges between unordered pairs of vertices) to finite sequences by fusing their list of pair representation into finite sequences with a pairing function on unordered pairs:

The resulting Encoder is:

```
mgraph :: Encoder [N2]
mgraph = compose (Iso graph2mset mset2graph) fun
```

working as follows:

```
*ISO> as mgraph nat 2008
[(1,2),(0,0),(1,1),(0,0),(0,0),(0,0),(0,0)]
*ISO> as nat mgraph it
2008
```

Note that the only change to the `graph` Encoder is replacing the composition with `set` by a composition with `fun`.

18.5 Encoding Hypergraphs

Definition 2 *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 . succ)
hypergraph2set = map (pred . set2nat)
```

The resulting Encoder is:

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

working as follows

```
*ISO> as hypergraph nat 2009
[[0],[2],[0,2],[0,1,2],[3],[0,3],[1,3],[0,1,3]]
*ISO> as nat hypergraph it
2009
```

Assuming that vertex numbers are lifted to even integers and hyperedge labels are odd integers derived from their position in the sequence, we can define the *bipartite graph* uniquely associated to a hypergraph as:

```
bipartite :: Encoder [N2]
bipartite = compose (Iso bipartite2hyper hyper2bipartite) hypergraph

hyper2bipartite xss = xs where
  l=genericLength xss
  xs=[(fromIntegral (2*i+1),fromIntegral (2*x))|i←[0..l-1],x←xss!!i]

bipartite2hyper xs = xss where
  pss = groupBy (λx y→fst x== fst y)  xs
  xss = map (map (hf . snd)) pss
  hf x = x `div` 2
```

working as follows:

```
as bipartite nat 2009
[(1,0),(3,4),(5,0),(5,4),(7,0),(7,2),(7,4),(9,6),(11,0),
 (11,6),(13,2),(13,6),(15,0),(15,2),(15,6)]
*ISO> as nat bipartite it
2009
```

Fig. 31 shows the bipartite graph derived from the hypergraph associated to 2009.

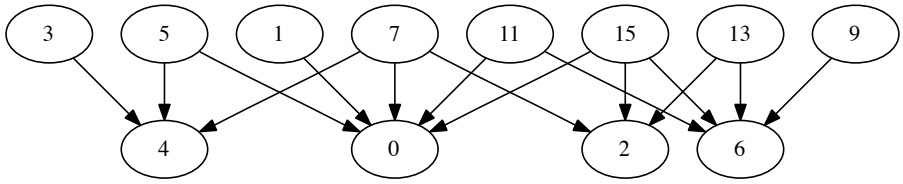


Fig. 31: 2009 as a bipartite graph

We can define the dual of a bipartite graph as follows:

```
bidual ps = sort (map (λ(x,y)→(succ y,pred x)) ps)
```

and derive involutions on natural numbers and hypergraphs with

```
borrow_bidual = borrow_from bipartite bidual
nbidual = borrow_bidual nat
hbidual = borrow_bidual hypergraph
```

An interesting graph is the *intersection graph* associated to the hypergraph, connecting distinct hyperedges with non-empty intersections:

```
to_igraph xss= gs where
  l=genericLength xss
  crosses xs ys = []/=intersect xs ys
  gs=[(fromIntegral i,fromIntegral j)|
     i←[0..l-1],j←[i+1..l-1],crosses (xss!!i) (xss!!j)]
```

working as follows:

```
*ISO> as hypergraph nat 2009
[[0],[2],[0,2],[0,1,2],[3],[0,3],[1,3],[0,1,3]]
*ISO> as nat hypergraph it
2009
*ISO> to_igraph it
[(0,2),(0,3),(0,5),(0,7),(1,2),(1,3),(2,3),(2,5),(2,7),
 (3,5),(3,6),(3,7),(4,5),(4,6),(4,7),(5,6),(5,7),(6,7)]
```

Note that a canonical representation of an intersection graph is obtained by relabeling hyperedges with integers given by their position in the list of hyperedges as shown in Fig. 32, where only edges smaller vertices to larger ones are drawn.

One can map this operation to an endomorphism $\mathbb{N} \to \mathbb{N}$

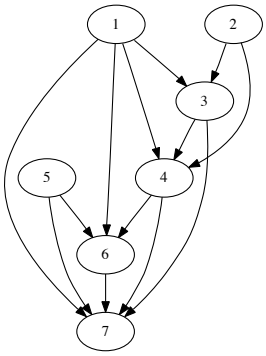


Fig. 32: Intersection graph derived from the hypergraph associated to 2009

```
to_ngraph n = as nat dag gs where
  xss=as hypergraph nat n
  gs=to_igraph xss
```

working as follows:

```
*ISO>  map to_ngraph [0..63]
[0,0,0,0,0,1,1,6,0,0,0,0,0,1,1,6,0,1,0,4,1,7,3,278,1,6,2,272,
 6,281,25,1126,0,0,1,2,1,3,7,30,1,2,6,24,6,25,281,614,1,3,6,
 28,7,31,283,886,7,30,282,880,286,889,1657,72934]
```

The k-iterate of `to_ngraph` can be defined as

```
to_kngraph n 0 = n
to_kngraph n k = to_ngraph (to_kngraph n (k-1))
```

working as follows:

```
*ISO> map (λx→to_kngraph x 2) [0..31]
[0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,6,0,27,0,1,0,1,1,278,6,1919]
*ISO> to_kngraph 2009 1
1172544926110
*ISO> to_kngraph 2009 2
1157920892373161954235710848732489010199483330488607977978525769528693070173 69
```

18.6 Encoding DAGs

One can derive an encoders for directed acyclic graphs (DAGs) from the encoding of digraphs under the assumption that they are canonically represented by pairs of edges such that the first element of the pair is strictly smaller.

After defining:

```
digraph2dag = map f where f (x,y) = (x,y+x+1)

dag2digraph = map f where f (x,y) | y>x = (x,y-x-1)
```

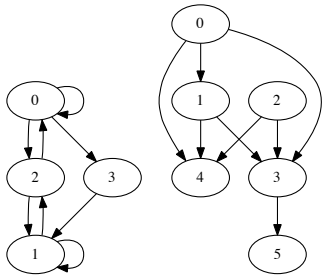
we obtain the Encoder:

```
dag :: Encoder [N2]
dag = compose (Iso dag2digraph digraph2dag ) digraph
```

working as follows:

```
*ISO> as nat dag [(0,1),(1,2),(0,2),(1,3),(2,3),(3,4),(4,5),(1,6)]
8590000191
*ISO> as dag nat it
[(0,1),(1,2),(0,2),(1,3),(2,3),(3,4),(4,5),(1,6)]
*ISO> as digraph nat 2009
[(0,0),(1,1),(2,0),(2,1),(3,1),(0,2),(1,2),(0,3)]
*ISO> as nat digraph it
2009
*ISO> as dag nat 2009
[(0,1),(1,3),(2,3),(2,4),(3,5),(0,3),(1,4),(0,4)]
*ISO> as nat dag it
2009
```

Fig. 33 shows the `bitpair` induced digraph and DAG associated to 2009. Note that they have the same number of edges. It looks interesting to explore in detail how properties of the two graphs are related. Clearly, we can state that:



(a) Digraph (b) DAG

Fig. 33: Digraph and DAG associated to 2009

Proposition 22 *There's a bijection between the set of finite digraphs and finite DAGs that preserves the number of edges.*

Note that there's an obvious mapping between DAGs and unordered graphs, without self-loops, obtained by always directing edges from vertices with lower indices to vertices with strictly higher indices.

As DAGs are digraphs, applying the digraph to DAG transformation to a DAG generates a new digraph. One can iterate this

```
kdag :: N→Encoder [N2]→Encoder [N2]
kdag k t = compose (Iso (dig2dag k) (dag2dig k)) t

dig2dag 0 g = g
dig2dag k g | k>0 = dag2digraph (dig2dag (k-1) g)

dag2dig 0 g = g
dag2dig k g | k>0 = digraph2dag (dag2dig (k-1) g)
```

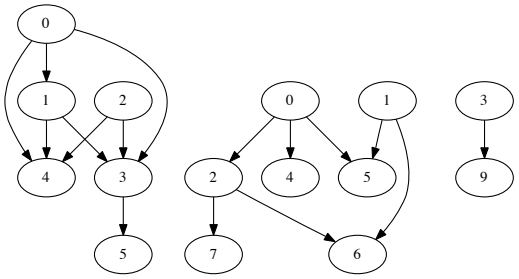
Using and alternative digraph mapping gives rise to a different DAG encoding. We can obtain an alternative Encoder simply as:

```
dag' :: Encoder [N2]
dag' = compose (Iso dag2digraph digraph2dag ) digraph'
```

Note also the use of the parameterized *Encoder transformer* `kdag`, working as follows:

```
*ISO> as (kdag 3 digraph) nat 2009
[(0,3),(1,7),(2,9),(2,10),(3,13),(0,5),(1,8),(0,6)]
*ISO> as (kdag 3 digraph') nat 2009
[(0,3),(2,9),(0,5),(0,6),(3,12),(0,7),(1,8),(0,8)]
```

Interestingly, after a few steps, while vertex codes increase, the resulting DAGs reach a fixpoint in the sense that they remain isomorphic to the DAGs obtain at the previous iteration up to a relabeling of their vertices. Figs. 34 and 35 show the DAGs associated to 2009 after 1 and 2 iterations, and 3 and 4 iterations, respectively.



(a) k=1 (b) k=2

Fig. 34: k-DAGs associated to 2009

18.7 Acyclic multi-digraphs

In a way similar to digraphs, we obtain the Encoder:

```
mdag :: Encoder [N2]
mdag = compose (Iso dag2digraph digraph2dag ) mdigraph
```

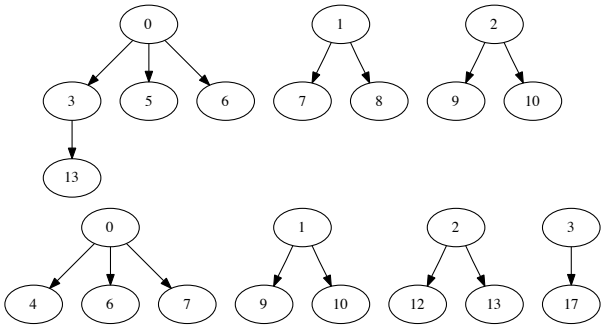


Fig. 35: k-DAGs associated to 2009: *k=3* and *k=4*

working as follows:

```
*ISO> as mdag nat 1234567890
[(1,2),(0,2),(1,2),(0,1),(1,2),(3,5),(0,1),(1,2),(0,2),(0,1),(0,2),(0,2)]
*ISO> as nat  mdag it
1234567890
```

Note that as in the case of the multi-digraph encoding, position of an edge defines it's unique label.

Fig. 36 depicts the acyclic multi-digraph associated to 1234567890.

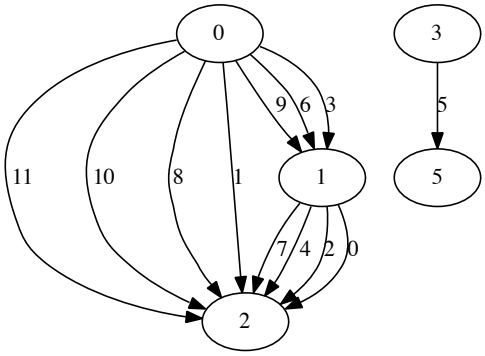


Fig. 36: 1234567890 as a directed multigraph

18.8 Encoding relations represented as boolean matrices

Assuming a binary relation defined on an initial segement of $\mathbb{N}$ is specified as (square) boolean matrix implmented as a lists of lists we first convert it to/from a digraph representation as follows:

```
binrel2digraph xss=[(fromIntegral x,fromIntegral y)
             |x←[0..l],y←[0..l],xss!!x!!y==1] where
  l=pred (genericLength xss)

digraph2binrel []= []
digraph2binrel ps= msplit (succ l) bs where
  (xs,ys)=unzip ps
  vs=sort (nub (xs++ys))
  l=maximum vs
  is=[0..l]
  f ps xy | xy `elem` ps = 1
  f _ _ = 0
  bs = map (f ps) [(x,y)|x←is,y←is]
  msplit _ []  = []
  msplit k xs  = h:(msplit k ts) where (h,ts)=genericSplitAt k xs
```

We obtain the Encoder:

```
binrel :: Encoder [[N]]
binrel = compose (Iso binrel2digraph digraph2binrel ) digraph
```

working as follows:

```
*ISO> as binrel nat 2010
[[0,0,1,1],[1,1,1,0],[1,1,0,0],[0,1,0,0]]
*ISO> as nat binrel it
2010
*ISO> as digraph nat it
[(1,0),(1,1),(2,0),(2,1),(3,1),(0,2),(1,2),(0,3)]
```

18.9 An alternative encoding scheme

Let us define first a mapping between pair and adjacency list representations of digraphs.

```
keygroup ps = xss where
  qss=groupBy (λp q→fst p==fst q) (sort ps)
  xss=map (λxs→(fst (head xs),map snd xs)) qss

keyungroup xys = [(x,y)|(x,ys)←xys,y←ys]
```

The two functions work as follows:

```
*ISO> keygroup [(0,10),(1,20),(1,30),(2,40),(2,50)]
[(0,[10]),(1,[20,30]),(2,[40,50])]
*ISO> keyungroup it
[(0,10),(1,20),(1,30),(2,40),(2,50)]
```

We can define the encoder `adigraph`

```
adigraph :: Encoder [N2]

adigraph = Iso adj2fun fun2adj
```

```
fun2adj ls = concat pss where
  lss=map ((as set nat) . succ) ls
  l=genericLength lss
  pss=zipWith f [0..l-1] lss
  f i js=[(i,j)|j←js]

adj2fun rs = ls where
   lss=map snd (keygroup rs)
   ls=map (pred . (as nat set)) lss
```

working as follows:

```
*ISO> as adigraph nat 2009
[(0,0),(1,0),(1,1),(2,0),(3,1),(4,0),(5,0),(6,0),(7,0)]
*ISO> as nat adigraph it
2009
```

Note that this alternative encoding is derived by viewing a digraph as a function from vertices to sets of neighbors.

18.10 Using graphs as data transformers

Let's first define a set of simple data transformation operations working on sequences:

```
ins xs at val = ys where
  fi (hs,ts) = hs ++ (val:ts)
  ys=fi (genericSplitAt at xs)

del xs at = ys where
  fd (hs,(_:ts)) = hs ++ ts
  ys=fd (genericSplitAt at xs)

subst xs at with = ins (del xs at) at with
```

Each edge in a graph can be seen as specifying a transposition `(i,j)` operating on a sequence `ms`:

```
gswap ms (i,j) = transp ms (2*i,2*j+1)

transp ms (i,j) = ms' where
  x=ms!!(fromIntegral i)
  y=ms!!(fromIntegral j)
  ms'=subst (subst ms i y) j x
```

We will use an infinite tape `newMem` with `0`s in even positions and `1`s in odd positions:

```
newMem = 0:1:newMem
```

A graph, seen as a list of pairs indicating transpositions can now operate on the tape `ms` and derive arbitrary boolean functions / sets represented as bitstrings.

```
applyPairs _ [] ms = ms
applyPairs sf (p:ps) ms = applyPairs sf ps (sf ms p)
```

One can extract the initial segment of the infinite tape after the computation:

```
*ISO> take 12 (applyPairs gswap [] newMem)
[0,1,0,1,0,1,0,1,0,1,0,1]
*ISO> take 12 (applyPairs gswap [(0,1)] newMem)
[1,1,0,0,0,1,0,1,0,1,0,1]
*ISO> take 12 (applyPairs gswap [(0,1),(1,2)] newMem)
[1,1,1,0,0,0,0,1,0,1,0,1]
```

An interesting property of this computation is *reversibility*. Applying the transposed graph restores the original tape.

An simple computation performed by a list of pairs can be defined as:

```
bitcompute graphType =  (as nat bits) . runPairs . (as graphType nat)

runPairs ps=rs where
  qs=applyPairs gswap ps newMem
  rs=genericTake (genericLength ps) qs
```

working as follows:

```
*ISO> bitcompute digraph 2009
420
*ISO> bitcompute dag 2009
498
```

An interesting operation is application of a digraph or dag to a bitstring representing a hereditarily finite structure.

```
encodeTransformer n = pepis_pair (l,as nat dag t) where
  (l,t) = induceTransformer n

induceTransformer n = ((hf l)-(genericLength ps)-1,ps) where
  ps=map f qs
  (l,qs)=induceTransps n
  hf n = n `div` 2
  f (x,y) = (hf (x-1),hf y)

induceTransps n = (l,qs) where
   bs= as hff_pars nat n
   l=genericLength bs
   ps=zip [0..l-1] bs
   ls=filter (λ(i,b)→(odd i) && (0==b))  ps
   rs=filter (λ(i,b)→(even i) && (1==b)) ps
   qs=zipWith f ls rs where
     f (i,_) (j,_) = (i,j)
```

working as follows:

```
*ISO> induceTransformer 2009
(2,[(0,1),(1,3),(2,4),(4,5),(5,7),(7,8),(8,9),(9,10),(10,11)])
```

```
*ISO> encodeTransformer 2009
2803905099203873342027
```

18.11 Encoding Finite Transducers

One can derive encoders for finite transducers (consuming and generating bits) from the encoding of a digraph, by borrowing a bit from each end of an edge.

After defining:

```
digraph2transducer = map f where
  f (x,y) = ((x',y'),(bx,by)) where
     ht z=quotRem z 2
     (x',bx)=ht x
     (y',by)=ht y

transducer2digraph = map g where
  g ((x',y'),(bx,by))=(x,y) where
    c b z = b+2*z
    x = c bx x'
    y = c by y'
```

we obtain the Encoder:

```
transducer :: Encoder [(N2,N2)]
transducer = compose (Iso transducer2digraph digraph2transducer) digraph
```

working as follows:

```
*ISO> as transducer nat 123456789
[((0,0),(0,0)),((0,0),(0,1)),((1,0),(0,0)),((0,1),(0,0)),((0,1),(0,1)),
 ((0,1),(1,1)),((1,1),(0,1)),((1,1),(1,1)),((2,0),(0,0)),((2,0),(1,0)),
 ((2,0),(1,1)),((3,0),(0,0)),((3,0),(0,1)),((2,1),(0,0)),((2,1),(1,0)),
 ((2,1),(0,1))]
*ISO> as nat transducer it
123456789
```

19 Encoding SAT problems

Boolean Satisfiability (SAT) problems are encoded as lists of lists representing conjunctions of disjunctions of positive or negative propositional symbols.

After defining:

```
set2sat = map (set2disj . nat2set) where
  shift0 z = if (z<0) then z else z+1
  set2disj = map (shift0. nat2z)

sat2set = map (set2nat . disj2set) where
  shiftback0 z = if(z<0) then z else z-1
  disj2set = map (z2nat . shiftback0)
```

we obtain the Encoder

```
sat :: Encoder [[Z]]
sat = compose (Iso sat2set set2sat) set
```

working as follows:

```
*ISO> as sat nat 2008
[[1,-1],[2],[-1,2],[1,-1,2],[-2],[1,-2],[-1,-2]]
*ISO> as nat sat it
2008
```

Clearly this encoding can be used to generate random SAT problems out of easier to generate random natural numbers.

20 Solving the SAT problems

One can actually try out the generated SAT problems by interfacing them with a SAT solver like David Bueno's `Funsat.Solver`.

First, we map our encoding to the CNF form required by the solver:

```
toCNF :: [[Z]] → CNF
toCNF xss = CNF nvars ncls (fromList cls) where
  xs = concat xss
  nvars = toInt (foldl max 0 (map abs xs))
  ncls = length xss
  cls = map toCls xss
  toCls = map (L . toInt)

toInt :: Z→Int
toInt x = fromIntegral x
```

The we define the function ssolve that works directly on our `sat` type:

```
ssolve :: [[Z]] → Solution
ssolve xss = s where
  (s,_,_) = solve1 (toCNF xss)
```

Finally, using our encoder we can define a solver working on a natural number `n` - and similarly by using an isomorphism from any other data representation.

```
nsolve k  = (xss,s) where
  xss=(as sat nat k)
  s=ssolve xss
```

The following examples show the associated SAT problem as well as a model found by the solver, when the problem is satisfiable.

```
*ISO> nsolve 123456
([[-1,2],[1,-2],[1,2,-2],[-1,2,-2],[1,-1,2,-2],[3]],satisfiable: 1 2 3)
*ISO> nsolve 1234567
([[],[1],[-1],[1,-1,2],[1,-2],[-1,-2],[2,-2],[-1,2,-2],[1,-1,2,-2],[1,3],[2,3]],unsatisfiable)
```

The predicates isSAT and isUNSAT can be used to test satisfiability directly on a natural number encoding of a SAT problem:

```
isSAT = found . snd . nsolve

isUNSAT = not . isSAT

found (Sat _) = True
found (Unsat _) = False
```

They can be used to collect statistics on solvability of SAT problems in various ranges:

```
sats xs = [x|x←xs, isSAT x]

unsats xs = [x|x←xs, isUNSAT x]

satStats xs = (s,u) where
  s=genericLength (sats xs)
  u=genericLength (unsats xs)
```

On can try to sample sat/unsat statistics as follows:

```
*ISO> satStats [2000..3000]
(440,561)
*ISO> satStats [10000..11000]
(433,568)
*ISO> satStats [100000..101000]
(399,602)
```

21 An Encoder for Graph Models

Graph models [36, 37] provide a semantics of λ-calculus (Y-combinator included) in terms of sets of finite sets of natural numbers. Following [36] a *graph model* is a pair (D, p) where D is an infinite set and $p : D^* \times D \to D$ is an injective total function. We will strengthen this to be a bijection, for the case $D = \mathbb{N}$ as follows.

```
gmodel2nat (set,m) = pred (fun2nat (m : (set2fun set)))
nat2gmodel n = (fun2set xs,m) where (m:xs) = nat2fun (succ n)
```

This provides the Encoder:

```
type Gdomain= ([N],N)
gmodel :: Encoder Gdomain
gmodel = compose (Iso gmodel2nat nat2gmodel) nat
```

working as follows:

```
*ISO> as gmodel nat 42
([0,2,4],0)
*ISO> as nat gmodel it
42
```

The interests of such models is that they provide an accurate set theoretic semantics for untyped lambda calculus describing key computational mechanisms like β-conversion and fixpoint combinators.

22 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 N)
dyadic = compose (Iso dyadic2set set2dyadic) set
```

The function `set2dyadic` mimics `set2nat` defined in subsection 3.5, except for the use of negative exponents and computation on rationals.

```
set2dyadic :: [N] → Ratio N
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 3.5.

```
dyadic2set :: Ratio N → [N]
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
```

Fig. 37 shows the dyadic rationals associated to natural numbers in [0..255].

23 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
```

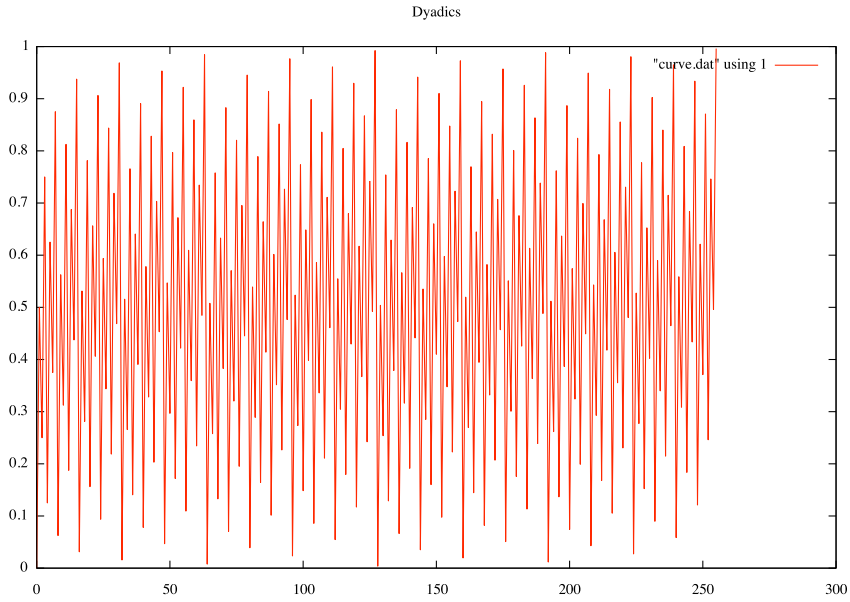


Fig. 37: Dyadic rationals associated to n in [0..255]

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

```
hff2pars = collect_pars '(' ')'

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
```

An interesting way to visualize the balanced bitstring expressions associated to the HFF, HFM, HFS and HFP trees is to represent them as "hills":

```
to_hill =scanr (λx y→y+(if 0==x then -1 else 1)) 0

hill t n = to_hill (as t nat n)

*ISO> hill hff_pars 42
```

```
[0,1,2,3,2,1,2,3,2,1,2,3,2,1,0]
*ISO> hill hfm_pars 42
[0,1,2,3,2,1,2,3,4,3,2,1,2,3,2,3,2,1,0]
*ISO> hill hfs_pars 42
[0,1,2,3,2,1,2,3,2,3,4,3,2,1,2,3,2,3,4,5,4,3,2,1,0]
*ISO> hill hfp_pars 42
[0,1,2,1,2,3,2,3,4,3,2,1,2,3,4,3,2,3,2,1,2,
 3,2,1,2,3,2,3,4,3,2,3,4,3,4,5,4,3,2,1,0]
```

Figs. 38 and 39 show the hills associated to 2009.

Alternatively, by using a `0` and `1` as left and right parenthesis we can define:

```
bitpars2hff cs = parse_pars 0 1 cs
hff2bitpars = collect_pars 0 1

hff_pars :: Encoder [N]
hff_pars = compose (Iso bitpars2hff hff2bitpars) hff
```

working as follows:

```
*ISO> as hff_pars nat 2008
[0,0,0,1,0,1,1,0,1,0,0,1,1,0,1,0,1,0,1,0,1,1]
*ISO> as nat hff_pars it
2008
*ISO> as nat bits (as hff_pars nat 2008)
7690599
```

As the last example shows, the information density of a parenthesis representation is lower. This is expected, given that order is constrained by balancing and content is constrained by having the same number of `0`s and `1`s. The following example

```
*ISO> map ((as nat bits) .  (as hff_pars nat)) [0..7]
[5,27,119,115,495,483,471,467]
```

shows that this application is injective only. Therefore a succinct representation of an abstract tree structure can be obtained by encoding it as a natural number as in:

```
*ISO> as nat pars "((()())()(()))()()())"
2008
```

Note however, that

```
*ISO> as nat bits (as hff_pars nat (2^2^16))
32639
```

while the conventional representation of the same number would have a few thousand digits. This suggest defining:

```
nat2parnat n = as nat bits (as hff_pars nat n)

parnat2nat n = as nat hff_pars (as bits nat n)
```

and find out that

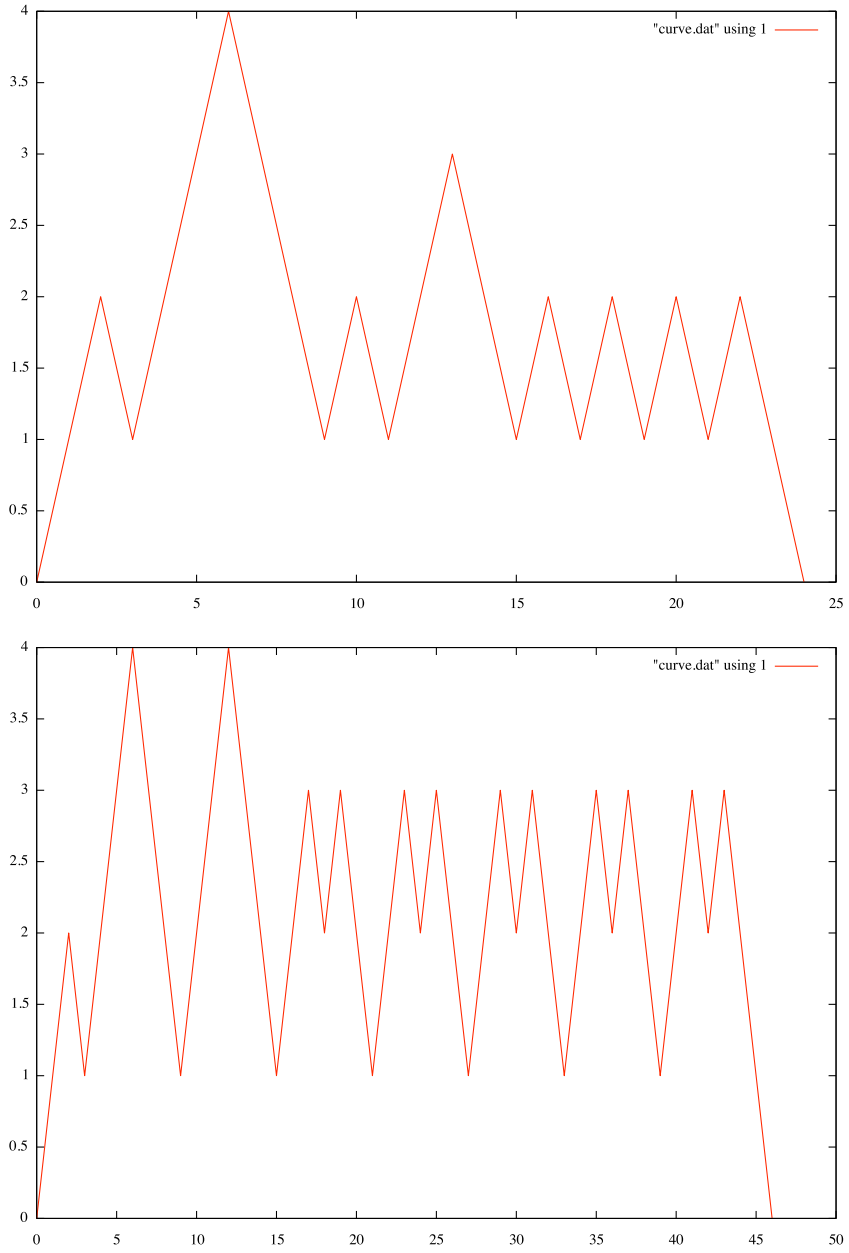


Fig. 38: Hills associated to 2009: *HFF hill* and *HFM hill*

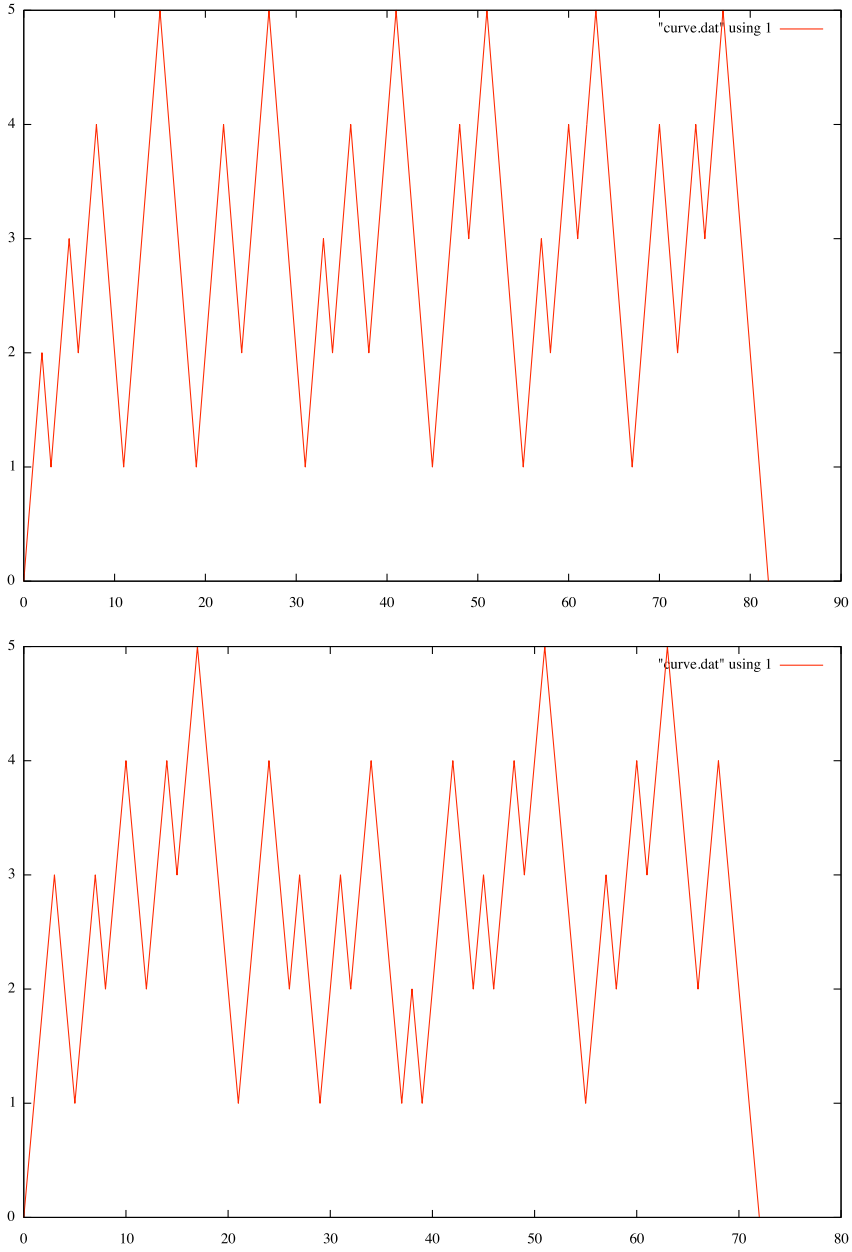


Fig. 39: Hills associated to 2009: *HFS hill* and *HFP hill*

```
*ISO> [x|x←[0..2^16],nat2parnat x<x]
[8192,16384,32768,32769,49152,65536]
```

One can see that more compact representations only happen for a few numbers that are powers of two or "sparse" sums of powers of two. A good way to evaluate "information density" for an arbitrary data type that is isomorphic to `N` through one of our encoders is to compute the total bitsize of its actual encoding over an interval like $[0..2^{n-1}]$. For instance,

```
hff_bitsize n= sum (map hff_bsize [0..2^n-1])

hff_bsize k=genericLength (as bits nat (nat2parnat k))
```

Knowing that the optimal bit representation of all numbers in $[0..2^{n-1}]$ totals $n * 2^n$ (2^n of them, n bits each), we can define a measure of information density for a bit-encoded parenthesis language seen as a representation for `HFF` as:

```
info_density_hff n = (n*2^n)%(hff_bitsize n)
```

One can see that information density progressively increases to converge to a value above half of the "perfect" value of `1`:

```
*ISO> map info_density_hff [0..12]
[0%1,1%3,4%9,12%25,1%2,80%157,16%31,112%215,
    32%61,48%91,1024%1933,2816%5297,2048%3841]
*ISO> map fromRational it
[0.0,0.3333333333333333,0.4444444444444444,0.48,0.5,
 0.5095541401273885,0.5161290322580645,0.5209302325581395,
 0.5245901639344263,0.5274725274725275,0.5297465080186239,
 0.5316216726448934,0.5331944806040094]
```

To compare this with the information density of hereditarily finite sets, multisets and permutations, we can also map their structure to a bit-represented parenthesis language by defining the encoder:

```
pars_hf=Iso bitpars2hff hff2bitpars

hff_pars' :: Encoder [N]
hff_pars' = compose pars_hf hff'

hfs_pars :: Encoder [N]
hfs_pars = compose pars_hf hfs

hfpm_pars :: Encoder [N]
hfpm_pars = compose pars_hf hfpm

hfm_pars :: Encoder [N]
hfm_pars = compose pars_hf hfm

bhfm_pars :: Encoder [N]
bhfm_pars = compose pars_hf hfbm
```

```
bhfm_pars' :: Encoder [N]
bhfm_pars' = compose pars_hf hfbm'

hfp_pars :: Encoder [N]
hfp_pars = compose pars_hf hfp
```

and then defining:

```
parsize_as t n = genericLength (hff2bitpars (as t nat n))

parsizes_as t m = map (parsize_as t) [0..2^m-1]

nat2hfsnat n = as nat bits (as hfs_pars nat n)

hfs_bitsize n= sum (map hfs_bsize [0..2^n-1])

hfs_bsize k=genericLength (as bits nat (nat2hfsnat k))

info_density_hfs n = (n*2^n)%(hfs_bitsize n)
```

The intuition that hereditarily finite functions have higher information density than hereditarily finite sets can now be conjectured:

```
*ISO> map info_density_hfs [0..12]
[0%1,1%3,2%5,3%8,1%3,5%16,2%7,7%27,4%17,3%13,2%9,11%52,1%5]
*ISO> map fromRational it
[0.0,0.3333333333333333,0.4,0.375,0.3333333333333333,
 0.3125,0.2857142857142857,0.25925925925925924,0.23529411764705882,
 0.23076923076923078,0.2222222222222222,0.21153846153846154,0.2]
```

Contrary to the case of bit-encoded HFFs, in this case information density is decreasing for larger values - an observation that can help with finding a simple proof for the conjecture. More generally, such techniques suggest applications to experimental mathematics.

As all our representations are fully parenthesized, they implement *uniquely decodable*, *self-delimiting* codes. Moreover, each of them is also a *prefix code*, i.e. there's no way to add a string made of any combination of left or right parenthesis at the end of a code and obtain another code. Therefore, the *Kraft inequality* holds for all of them. Let us first define the function computing the left side of the *Kraft* inequality, and the corresponding test:

```
kraft t n=sum xs where
  f x = 1/2^x
  xs=map (f . (parsize_as t)) [0..n-1]

kraft_check t n = kraft t n ≤1
```

working as follows:

```
*ISO> kraft hff' 2010
0.3995348811149597
*ISO> kraft hff' 2010
```

```
0.3995348811149597
*ISO> kraft hff 2010
0.39161574840545654
*ISO> kraft hfm 2010
0.37343781457623665
*ISO> kraft hfs 2010
0.34073090080675555
*ISO> kraft hfp 2010
0.320405483354989
```

Fig. 40 plots the values of the `kraft hff` function for $n \in [0..63]$.

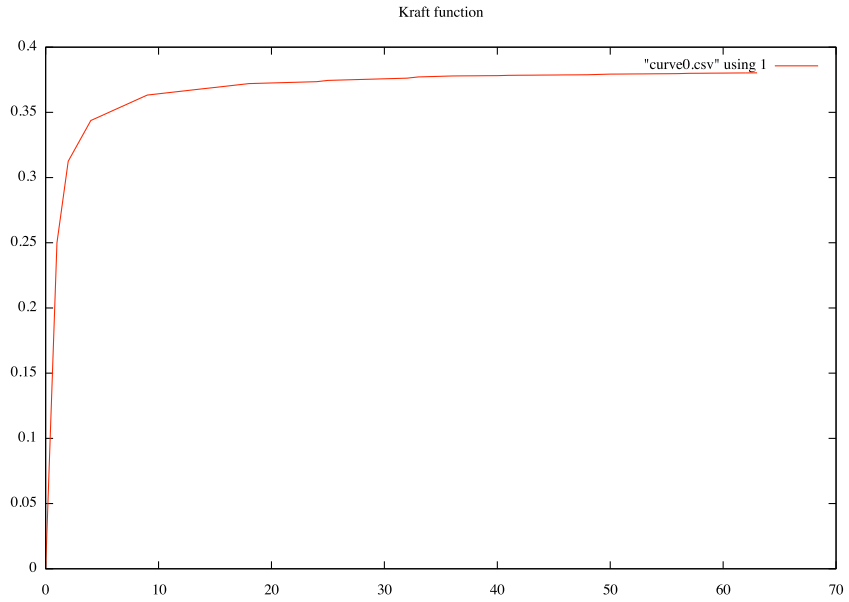


Fig. 40: Kraft function for parenthesis encoding of HFFs

In fact, as the code is decodable either from the beginning or from the end it is a "fix-free" code. It is conjectured that the highesy `Kraft sum` for such codes is `3/4`.

24 Self-delimiting codes

A more precise estimate of the actual size of various bitstring representations requires also counting the overhead for "delimiting" their components. An asymptotically optimal mechanism for this is the use of a *universal self-delimiting code* for instance, the *Elias omega code*. To implement it, the encoder proceeds by

recursively encoding length of the string, the length of the length of the strings etc.

```
to_elias :: N → [N]
to_elias n = (to_eliasx (succ n))++[0]

to_eliasx 1 = []
to_eliasx n = xs where
  bs=to_lbits n
  l=(genericLength bs)-1
  xs = if l<2 then bs else (to_eliasx l)++bs
```

The decoder first rebuilds recursively the sequence of lengths and then the actual bitstring. It makes sense to design the decoder to extract the number represented by the self-delimiting code from a sequence/stream of bits and also return what is left after the extraction.

```
from_elias :: [N] → (N, [N])
from_elias bs = (pred n,cs) where (n,cs)=from_eliasx 1 bs

from_eliasx n (0:bs) = (n,bs)
from_eliasx n (1:bs) = r where
  hs=genericTake n bs
  ts=genericDrop n bs
  n'=from_lbits (1:hs)
  r=from_eliasx n' ts

to_lbits = reverse . (to_base 2)

from_lbits = (from_base 2) . reverse
```

We obtain the Encoder:

```
elias :: Encoder [N]
elias = compose (Iso (fst . from_elias) to_elias) nat
```

working as follows:

```
*ISO> as elias nat 42
[1,0,1,0,1,1,0,1,0,1,1,0]
*ISO> as nat elias it
42
*ISO> as elias nat 2008
[1,1,1,0,1,0,1,1,1,1,1,0,1,1,0,0,1,0]
*ISO> as nat elias it
2008
```

Note that self-delimiting codes are not *onto* the regular language $\{0,1\}^*$, therefore this Encoder cannot be used to map arbitrary bitstrings to numbers. Similarly to our parenthesis representations, one can define a function computing the left side of the Kraft inequality for Elias encodings:

```
kraft_elias n = sum (map f [0..n-1]) where
  f k=1/2^(genericLength (as elias nat k))
```

working as follows:

```
*ISO> kraft_elias 2010
0.9178276062011719
```

Fig. 41 plots the values of the `kraft_elias` function for $n \in [0..63]$.

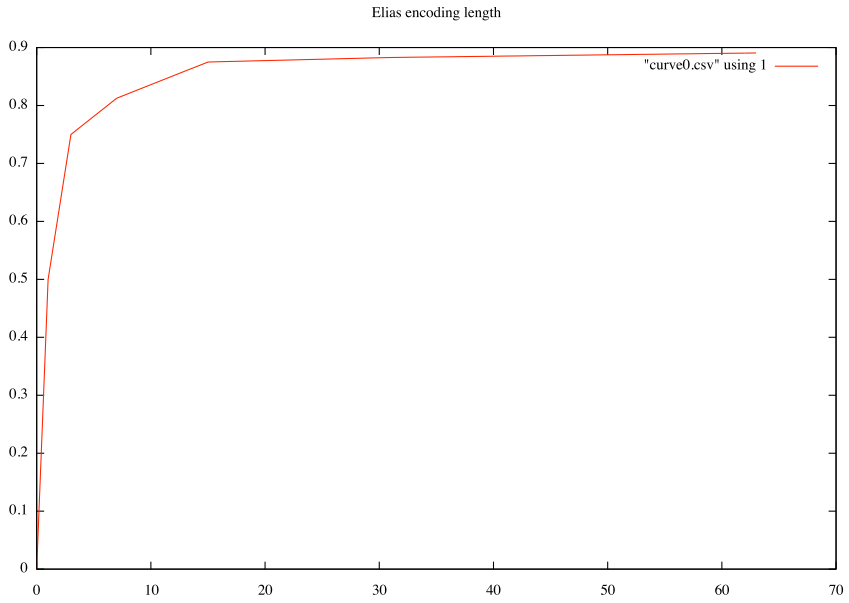


Fig. 41: Kraft function for Elias encodings

25 Automorphisms as Encoder transformers

An *automorphism* is an isomorphism for which the source and target are the same. Clearly, by applying an automorphism before or after an isomorphism we obtain a similarly typed new isomorphism i.e. we can define:

```
after2 i j t = compose (Iso i j) t
after i t = after2 i i t
```

working as follows:

```
*ISO> as (after id bits) nat 2009
[0,1,0,1,1,0,1,1,1,1]
*ISO> as (after dual bits) nat 2009
[1,0,1,0,0,1,0,0,0,0]
*ISO> as (after reverse bits) nat 2009
```

```
[1,1,1,1,0,1,1,0,1,0]
*ISO> as perm nat 2009
[1,4,3,2,0,6,5]
*ISO> as (after invertPerm perm) nat 2009
[4,0,3,2,1,6,5]
```

Dually, automorphisms can be borrowed (as one argument functions) form a data type and used as automorphisms on another data type:

```
*ISO> borrow_from perm invertPerm nat 2009
3809
*ISO> borrow_from perm invertPerm nat 3809
2009
*ISO> borrow_from nat succ perm [0,1,2,3]
[0,1,3,2]
*ISO> borrow_from nat pred perm [0,1,3,2]
[0,1,2,3]
*ISO> borrow_from bits dual nat 2009
1060
*ISO> borrow_from bits dual nat 1060
2009
*ISO> borrow_from bits dual set [1,3,4,7]
[0,1,5,6,7]
*ISO> borrow_from bits dual set [0,1,5,6,7]
[1,3,4,7]
```

Of particular interest are automorphism of $\mathbb{N}$ obtained by borrowing from other data types, with potential applications to cryptography and data compression. An interesting case is when such automorphisms are involutions i.e. such the $f \circ f = id$.

Fig. 42 depicts the involution on $\mathbb{N}$ obtained by borrowing `invertPerm` from `perm` i.e.

```
natInvert = borrow_from perm invertPerm nat
```

Fig. 43 depicts the involution on $\mathbb{N}$ obtained by borrowing `dual` from `bits` i.e.

```
natDual = borrow_from bits dual nat
```

Note that automorphisms can be borrowed from any connected data type and fairly complex transformations likely to have good *diffusion* and *confusion* properties can be generated by their iterated compositions.

Interesting automorphisms can also be generated by combining alternative encodings that share the same data representation. We can define

```
auto t s = (as nat s) . (as t nat)

autoS = auto fun fun'

autoS' = auto fun' fun
```

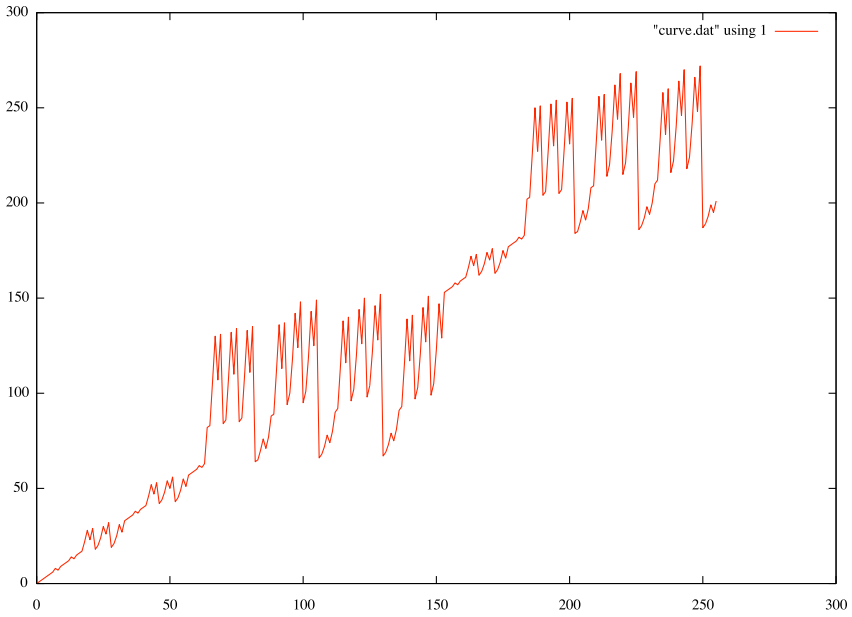


Fig. 42: Automorphism on $\mathbb{N}$ borrowd from inverted permutations

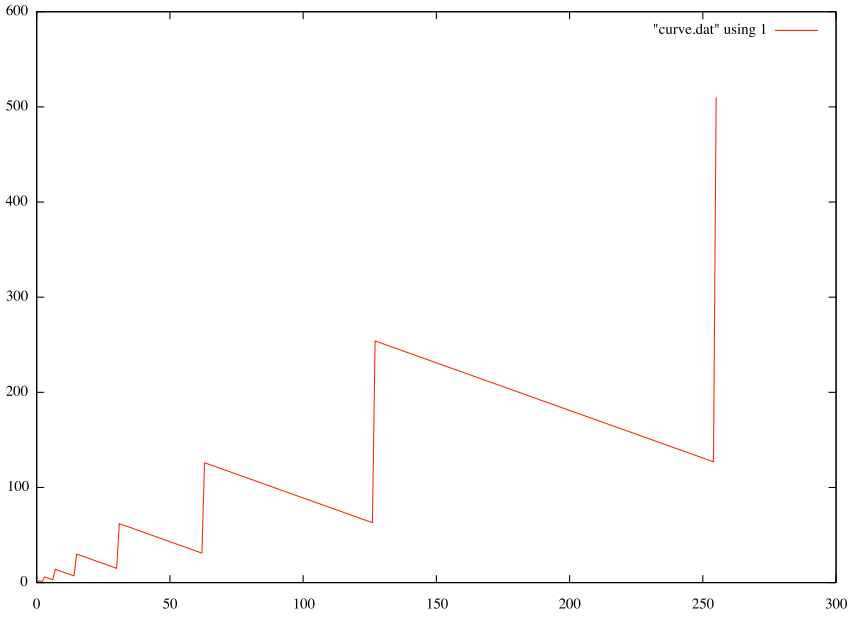


Fig. 43: Automorphism on $\mathbb{N}$ borrowd from 0-1 flipped bitstrings

```
autoP = auto mset pmset

autoP' = auto pmset mset

autoH = auto hff hff'

autoH'= auto hff' hff

autoM = auto hfm hfpm

autoM' = auto hfpm hfm
```

working as follows

```
*ISO> map autoS [0..31]
[0,1,3,2,5,10,6,4,7,34,14,36,18,20,12,8,9,42,38,260,
 26,52,44,136,22,132,28,72,68,40,24,16]
*ISO> map autoS' [0..31]
[0,1,3,2,7,4,6,8,15,16,5,32,14,64,10,128,31,256,12,
 512,13,1024,24,2048,30,4096,20,8192,26,16384,40,32768]
*ISO> (autoS . autoS') 2009
2009
*ISO> (autoS' . autoS) 2009
2009
*ISO> map autoH [0..31]
[0,1,3,2,7,10,6,4,5,42,14,36,22,20,12,8,15,34,46,292,30,52,
 44,136,18,148,28,72,76,40,24,16]
*ISO> map autoH' [0..31]
[0,1,3,2,7,8,6,4,15,128,5,256,14,64,10,16,31,32768,24,
 340282366920938463463374607431768211456,13,32,12,
 115792089237316195423570985008687907853269984665640564039457584007913129639936,
 30,16384,40,18446744073709551616,26,1024,20,65536]
 *ISO> (autoH . autoH') 2009
2009
*ISO> (autoH' . autoH) 2009
2009
*ISO> map autoM [0..31]
[0,1,2,3,4,5,8,7,6,9,14,11,24,17,26,15,10,
 13,20,19,34,29,44,23,48,49,74,35,124,53,80,31]
*ISO> map autoM' [0..31]
[0,1,2,3,4,5,8,7,6,9,16,11,32,17,10,15,256,13,128,19,
 18,33,64,23,12,65,14,35,512,21,65536,31]
 *ISO> (autoM . autoM') 2009
2009
*ISO> (autoM' . autoM) 2009
2009
*ISO> map autoP [0..15]
[0,1,2,3,4,5,8,7,6,9,14,11,24,17,26,15]
*ISO> map autoP' [0..15]
[0,1,2,3,4,5,8,7,6,9,16,11,32,17,10,15]
*ISO> (autoP . autoP') 2009
```

```
2009
*ISO> (autoP' . autoP) 2009
2009
```

Given that these are automorphisms, one can expect that some values are below and some are above the identity function.

```
autoLess t s m = filter (λi→auto t s i<i) [0..2^m-1]
```

This is indeed the case

```
*ISO> autoLess fun' fun 6
[3,5,10,14,18,20,26,28,34,36,38,40,42,44,46,50,52,60]
*ISO> autoLess fun fun' 6
[3,7,8,14,15,16,24,30,31,32,40,58,62,63]
*ISO> autoLess hff hff' 5
[3,7,8,14,15,16,24,30,31]
*ISO> autoLess hff' hff 5
[3,7,10,14,20,22,28,30]
```

25.1 Using permutations as encoders

One can define a simple encoder/decoder using a permutation associated to a natural number as follows:

```
pencode = pcode id
pdecode = pcode invertPerm

pcode f k n = n' where
  ps=as perm nat k
  ps'=f ps
  l=genericLength ps'
  ns=to_tuple l n
  ns'=applyPerm ps' ns
  n'=from_tuple ns'

applyPerm ps xs = [xs!!(fromIntegral p)|p←ps]

invertPerm ps = snd (unzip (sort (zip ps [0..])))
```

working as follows:

```
*ISO> pencode 1234 567890
599366
*ISO> pdecode 1234 it
567890
```

Note also that the decoder and encoder can trade roles:

```
*ISO> pdecode 1234 567890
795992
*ISO> pencode 1234 it
567890
```

and that they can be cascaded as in:

```
*ISO> ((pencode 123) . (pencode 345))  10203040
19455458
*ISO> ((pdecode 345) . (pdecode 123))  it
10203040
```

It would be interesting to see if these operations can be used as building blocks for applications to cryptography.

Another similar mechanism, using gray code representations is:

```
pencode' = pcode' id
pdecode' = pcode' invertPerm

pcode' f k n = n' where
  ps=as perm nat k
  ps'=f ps
  l=genericLength ps'
  ns=to_tuple l (as gray nat n)
  ns'=applyPerm ps' ns
  n'=as nat gray (from_tuple ns')
```

Injective endomorphisms from $\mathbb{N} \to \mathbb{N}$ are common inhabitants of the mathematics ontology. An interesting fast growing injective endomorphism is obtained from the permutations induced by gray codes.

```
grayPerm n = p where
  ps=map (as gray nat) [0..2^n-1]
  p=as nat perm ps

*ISO> map gray2perm [0..5]
[1,2,11,6056,1407956705291,850612470802100037944974763973536]
```

25.2 Deriving automorphisms on hereditarily finite structures

Given an automorphism $\mathbb{N} \to \mathbb{N}$ one can derive hylomorphisms similar to `hff, hfm etc.` We obtain the encoders:

```
ghff :: Encoder T
ghff = compose (hylo gray) gray

gray2fun = nat2fun . gray2nat
fun2gray = nat2gray . fun2nat

gray_fun = Iso gray2fun fun2gray

ghff' :: Encoder T
ghff' = compose (hylo gray_fun) gray

gray2set = nat2set . gray2nat
set2gray = nat2gray . set2nat
```

```
gray_set = Iso gray2set set2gray

ghfs :: Encoder T
ghfs = compose (hylo gray_set) gray
```

working as follows

```
*ISO> as ghff nat 2009
H [H [],H [H [],H []],H [],H [H []],H [],H [],H [],H []]
*ISO> as ghff' nat 2009
H [H [],H [],H [H []],H [H [],H []],H [H [],H []],H []]
*ISO> as ghfs nat 2009
H [H [],H [H []],H [H [H []]],H [H [H [],
   H [H []]]],H [H [H []],H [H [],H [H []]],H [H [H []]]],
   H [H [H [],H [H []]],H [H [H []]]]]
*ISO> as nat ghfs it
2009
```

A simple $\mathbb{N} \to \mathbb{N}$ endomorphism is obtained by reversing and flipping the left and right perentheses in the parenthesis representation of hereditarily finite functions.

```
parmorph = as nat pars . dualpars . as pars nat

dualpars  = flip_pars '(' ')' . reverse

flip_pars _ _ [] = []
flip_pars l r (x:xs) | x==l = r: (flip_pars l r xs)
flip_pars l r (x:xs) | x==r = l: (flip_pars l r xs)
```

Clearly, this is also an involution:

```
*ISO> map parmorph [0..15]
[0,1,2,3,4,6,5,7,8,12,10,14,9,13,11,15]
*ISO> map parmorph it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Fig. 44 show a plot of this involution.

25.3 Passkey Induced Automorphisms

Automorphisms on `string` can be used as "boxes" for cryptographic algorithms, provided that they exhibit good diffusion and confusion properties.

```
pkencode auto passkey plaintext = cyphertext where
  k=as nat string passkey
  p=as nat string plaintext
  c=auto k p
  cyphertext=as string nat c

simpleEncode k p = k `xor` (as gray nat p)
simpleDecode k q = as nat gray (k `xor` q)
```

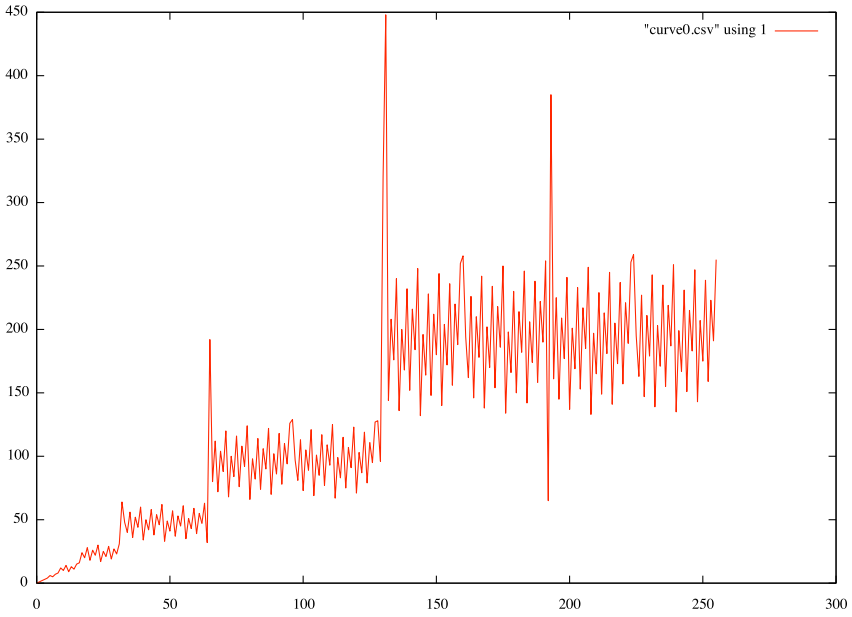


Fig. 44: Values of `parmorph` in [0..255]

```
*ISO> pkencode simpleEncode "Alice" "Bob likes beer"
"Mc%&WlFx|QN^yV"
*ISO> pkencode simpleDecode "Alice" it
"Bob likes beer"
```

26 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 [38, 39]:

```
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 $\mathbb{N} \to \mathbb{N}$.

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 redundancy protects against 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.

27 Testing It All

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

While QuickCheck [7] 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 :: N→N→N
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..25])

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 perm
```

```
rt 10 = rantest hfp
rt 11 = rantest nat2
rt 12 = rantest set2
rt 13 = rantest clist
rt 14 = rantest pbdd
rt 15 = rantest bdd
rt 16 = rantest rbdd
rt 17 = rantest digraph
rt 18 = rantest graph
rt 19 = rantest mdigraph
rt 20 = rantest mgraph
rt 21 = rantest hypergraph
rt 22 = rantest dyadic
rt 23 = rantest string
rt 24 = rantest pars
rt 25 = rantest dna
```

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.

28 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.

28.1 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)]
```

28.2 Random Generation

Combining `nth` with a random generator for *nat* provides free algorithms for random generation of complex objects of customizable size:

```
ran thing seed largest = head (random_gen thing seed largest 1)

random_gen thing seed largest n = genericTake n
  (nths thing (rans seed largest))

rans seed largest =
    randomRs (0,largest) (mkStdGen seed)
```

For instance

```
*ISO> random_gen set 11 999 3
[[0,2,5],[0,5,9],[0,1,5,6]]
```

generates a list of 3 random sets.

For instance

```
*ISO>ran digraph 5 (2^31)
[(1,0),(0,1),(2,1),(1,3),(2,2),(3,2),(4,0),(4,1),
 (5,1),(6,0),(6,1),(7,1),(5,3),(6,2),(6,3)]

*ISO> ran hfs 7 30
H [H [],H [H [],H [H []]],H [H [H [H []]]]]
*ISO> ran dnaStrand 1 123456789

DNAstrand P5x3 [Guanine,Thymine,Guanine,Cytosine,
  Cytosine,Thymine,Thymine,Thymine,Thymine,
  Adenine,Thymine,Cytosine,Cytosine]
```

generate a random digraph, a hereditarily finite set and a DNA strand.

Random generator for various data types are useful for further automating test generators in tools like QuickCheck [7] by generating customized random tests.

An interesting other application is generating random problems or programs of a given type and size. For instance

```
*ISO> ran sat 8 (2^31)
[[-1],[1,-1],[-1,2],[1,-1,2],[-2],[1,-2],[-1,-2],[1,-1,-2],
[2,-2],[1,2,-2],[-1,2,-2],[3],[1,-1,3],[1,-1,2,3],[1,-2,3],
[-1,-2,3],[2,-2,3],[1,2,-2,3],[-1,2,-2,3]]

*ISO> ran clist 8 12345
Cons (Atom 0) (Cons (Cons (Atom 0) (Atom 0)) (Atom 100))
```

generate, respectively, a random SAT-problem and a random `Cons`-list.

28.3 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 [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 space-wise or time-wise.

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

28.4 Experimental Mathematics

Comparing compactness of representations 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 12345678901234567890123456789O
54
*ISO> length_as perm 12345678901234567890123456789O
28
*ISO> length_as fun 12345678901234567890123456789O
54
*ISO> sum_as set 12345678901234567890123456789O
2690
*ISO> sum_as perm 12345678901234567890123456789O
378
*ISO> sum_as fun 12345678901234567890123456789O
43
```

One might observe that the same trend applies also to their hereditarily finite derivatives:

```
*ISO> size_as hfs 12345678901234567890123456789O
627
*ISO> size_as hfp 12345678901234567890123456789O
276
*ISO> size_as hff 12345678901234567890123456789O
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 implements sharing, as computed by `robdd_size`) also provide relatively compact representations:

```
*ISO> bdd_size $ as bdd
        nat 12345678901234567890123456789O
256
*ISO> bdd_size $ as rbdd
        nat 12345678901234567890123456789O
144
*ISO> robdd_size $ as rbdd
         nat 12345678901234567890123456789O
39
```

Figures 45, 46, 47 compare the sizes of bitstring, BDD, HFF, HFS, HFP representations, first with the most succinct ones (bitstring, BDDs, HFF) grouped together in Fig. 45, then the less succinct ones (HFS and HFP) in Fig. 46 and finally all representations together for `n` in the larger interval $[0..2^{16}-1]$.

It is also interesting to observe the ability of some representations to express huge numbers that normally overflow computer memory but which are genuinely "low complexity" as a result of a small numbers of simple computational steps that generate them.

For instance,

```
 *ISO> map (as nat pars)
     [ "()","(())","((()))","(((())))", "((((()))))","((((((()))))))"]
 [0,1,2,4,16,65536]
*ISO> as hff pars "((()))"
H [H [H []]]
```

shows that parenthesis sequences (structurally isomorphic to hereditarily finite functions) can represent succinctly the fast growing but low complexity series $a_n = 2^{2^n}$. Clearly, terms of the series would exhaust computer memory quite quickly using a conventional bitvector based arbitrary size integer representation! This suggest the usefulness of a *universal* possibly lazy "shapeshifting"

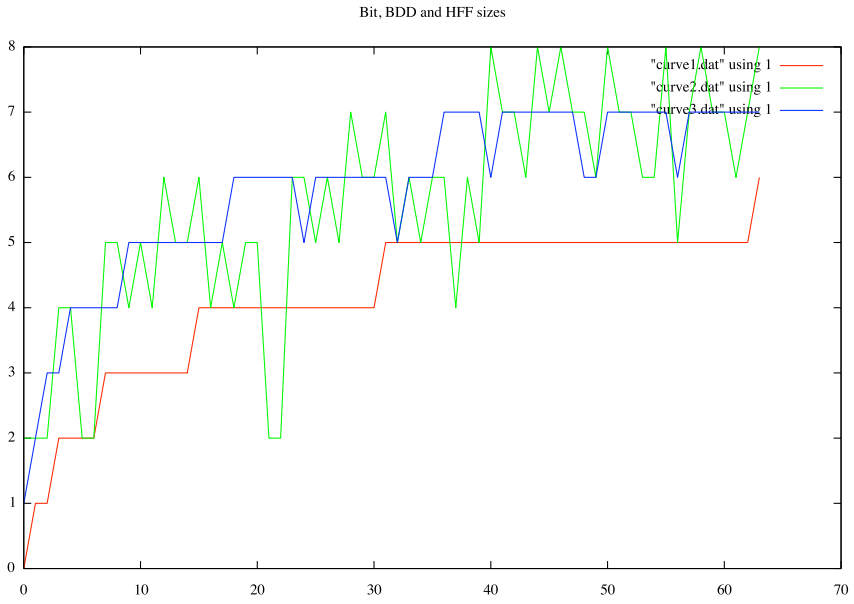


Fig. 45: Comparison of curve1=Bit, curve2=BDD and curve3=HFF sizes

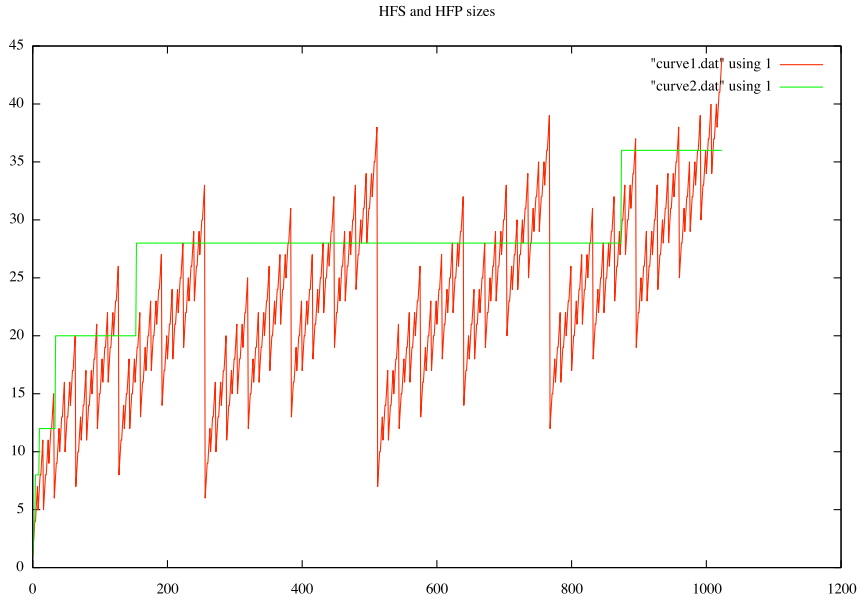


Fig. 46: Comparison of curve1=HFS and curve1=HFP sizes

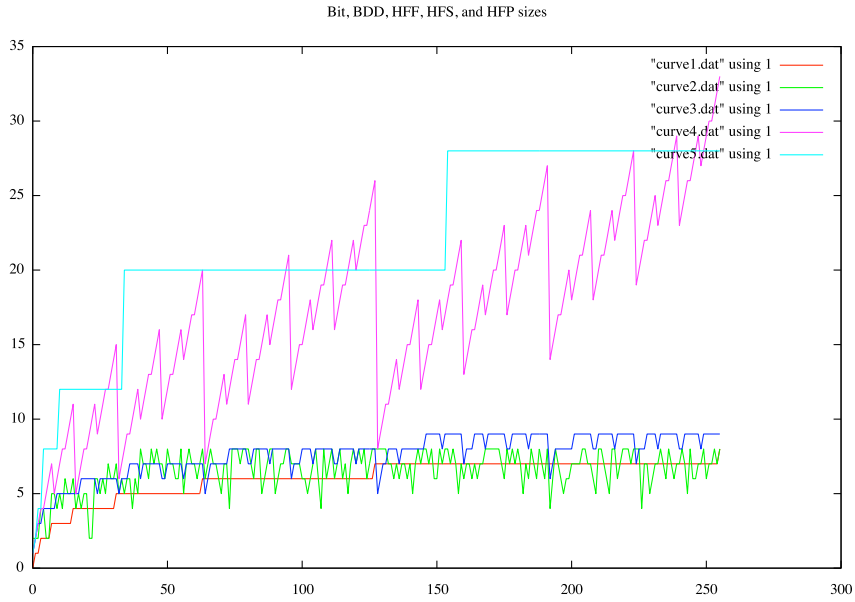


Fig. 47: Comparison of all representation sizes at a larger scale

algorithm, that can decide on the most efficient data representation automatically, using size estimates, at the time when data is actually constructed.

Sparseness criteria As a first step, one can introduce a "sparseness criteria" by comparing the size of a representation `f` with the size of the self-delimiting Elias omega code.

One can obtain an encoding of a sequence by encoding its length and then encoding each term, parametrized by a function $f : \mathbb{N} \to [\mathbb{N}]$:

```
nat2self f n = (to_elias l) ++ concatMap to_elias ns where
  ns = f n
  l=genericLength ns

nat2sfun n = nat2self (as fun nat) n
```

This function is injective (but not onto!) and its action can be reversed by first decoding the length l and then extracting self delimited sequences l times.

```
self2nat g ts = (g xs,ts') where
  (l,ns) = from_elias ts
  (xs,ts')=take_from_elias l ns

  take_from_elias 0 ns = ([],ns)
  take_from_elias k ns = ((x:xs),ns'') where
```

```
    (x,ns')=from_elias ns
    (xs,ns'')=take_from_elias (k-1) ns'

sfun2nat ns = xs where
  (xs,[])=self2nat (as nat fun) ns
```

We obtain the Encoder:

```
sfun :: Encoder [N]
sfun = compose (Iso sfun2nat nat2sfun) nat
```

working as follows:

```
*ISO> as sfun nat 42
[1,0,1,0,0,0,1,0,0,1,0,0,1,0,0]
*ISO> as nat sfun it
42
```

A simple concept of sparseness is derived by comparing the size of a self-delimiting code for a number `n` vs. the size of its self-delimiting representation as a finite sequence, finite set or finite permutation as shown in Fig. 48, computed as follows:

```
linear_sparseness_pair t n =
  (genericLength (to_elias n),genericLength (nat2self (as t nat) n))

linear_sparseness f n = x/y where (x,y)=linear_sparseness_pair f n
```

We can also extend this comparison the hereditarily finite representations, which, as a pleasant surprise, turn out to provide self-delimiting codes.

```
sparseness_pair f n =
  (genericLength (to_elias n),genericLength (as f nat n))

sparseness f n = x/y where (x,y)=sparseness_pair f n
```

One can then compare (self-delimiting) parenthesis language representations for hereditarily finite encoders provided by HFF, HFS, HFP and discover the "peaks" of sparseness as shown in Fig. 49 and 50.

A new self-delimiting code While the HFF representation is generally less compact than Elias omega code, its simplicity suggest it as a possibly useful self-delimiting code, especially interesting for streams of "sparse" values, as shown in Fig. 51.

One can collect values that have smaller HFF codes than Elias omega codes i.e. "sparse numbers" with:

```
sparses_to m = [n|n←[0..m-1],
  (genericLength (as hff_pars nat n))
  <
  (genericLength (as elias nat n))]
```

working as follows

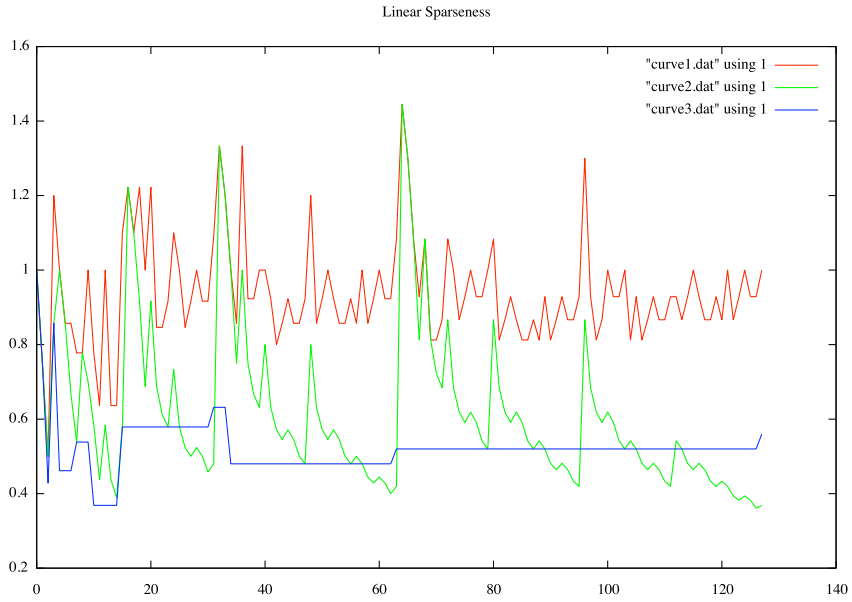


Fig. 48: Sparseness measures with curve1=fun, curve2=set, curve3=perm up to 2^7

```
ISO> sparses_to (2^11)
[15,16,17,24,32,64,65,96,128,129,192,256,257,258,259,320,384,
 385,448,512,513,514,515,516,517,518,519,520,544,576,640,641,704,768,
 769,770,771,832,896,897,960,1024,1025,1026,1027,1028,1029,1030,1031,
 1032,1088,1152,1280,1281,1408,1536,1537,1538,1539,1664,1792,1793,1920]
```

and notice that the list collects an unusually large number of various popular memory chip and computer screen sizes. Figure 52 shows distribution of "sparse numbers" in $[0..2^{18}]$.

Primes and Pairing Functions Products of two prime numbers have the interesting property that they are special a case where no information is lost by multiplication in the sense of [40]. Indeed, in this case multiplication is reversible, i.e. the two factors can be recovered given the product. As the product is comparatively easy to compute, while in case of large primes factoring is believed intractable, this property has well-known uses in cryptography. Given the isomorphism between natural numbers and primes mapping a prime to its position in the sequence of primes, one can transport pairing/unpairing operations to prime numbers

```
ppair pairingf (p1,p2) | is_prime p1 && is_prime p2 =
  from_pos_in ps (pairingf (to_pos_in ps p1,to_pos_in ps p2)) where
```

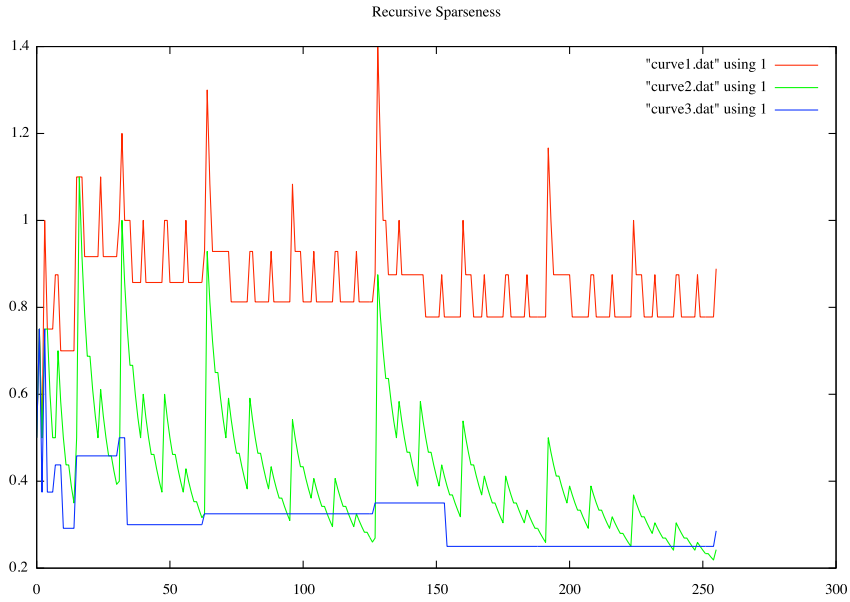


Fig. 49: Sparseness measures with curve1=HFF, curve2=HFS, curve3=HFP up to 2^8

```
    ps = primes

punpair unpairingf p | is_prime p = (from_pos_in ps n1,from_pos_in ps n2) where
  ps=primes
  (n1,n2)=unpairingf (to_pos_in ps p)
```

working as follows:

```
*ISO> ppair bitpair (11,17)
269
*ISO> punpair bitunpair it
(11,17)
```

Clearly, this defines a bijection $f : Primes \times Primes \rightarrow Primes$ that is tempting to compare with the product of two primes. Figs. 53 and 54 shows the surfaces generated by products and multiset pairings of primes. While both commutative operations are reversible and likely to be asymptotically equivalent in terms of information density, one can notice the much smoother transition in the case of lossless multiplication.

We have seen that recursive application of the unpairing function `bitunpair` provided an isomorphism between natural numbers and BDDs. Given an *unpairing function* $u : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ and a predicate `p(n)` over the set of natural numbers, it makes sense to investigate subsets of $\mathbb{N}$ such that if `p` holds for `n` then it also

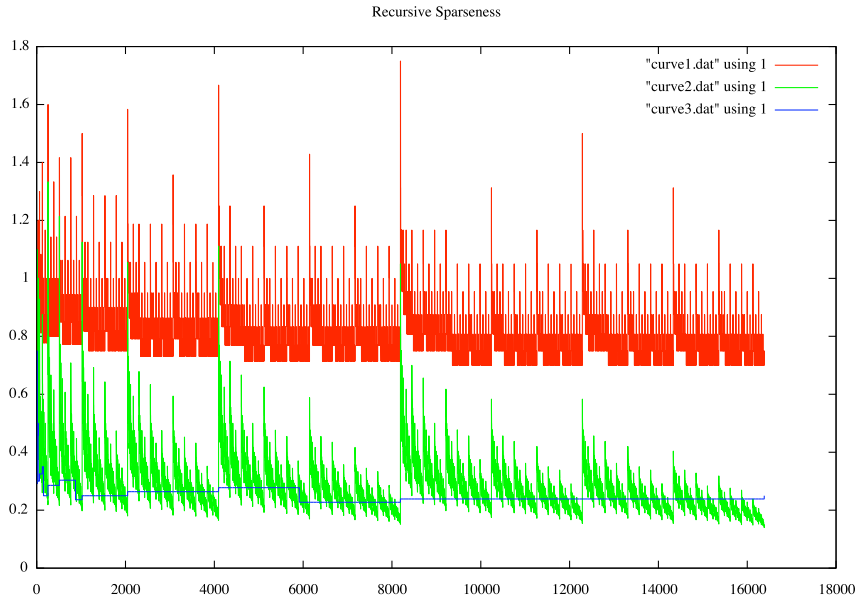


Fig. 50: Sparseness measures with curve1=HFF curve2=HFS, curve3=HFP up to 2^{14}

holds after applying the unpairing function `u` to `n`. More interestingly, one can look at subsets for which this property holds recursively.

Assuming a prime recognizer `is_prime` and a generator `primes` for the stream of prime numbers (see Appendix), we can define:

```
hyper_primes u = [n|n←primes, all_are_primes (uparts u n)] where
  all_are_primes ns = and (map is_prime ns)

uparts u = sort . nub . tail . (split_with u) where
    split_with _ 0 = []
    split_with _ 1 = []
    split_with u n = n:(split_with u n0)++(split_with u n1) where
      (n0,n1)=u n
```

working as follows:

```
*ISO> take 20 (hyper_primes bitunpair)
[2,3,5,7,11,13,17,19,23,29,31,43,47,59,71,79,83,89,103,139]
*ISO> take 20 (hyper_primes pepis_unpair)
[2,3,5,7,11,13,19,23,29,31,43,53,59,107,127,173,223,251,311,347]
```

This leads to the following conjectures, in increasing order of generality:

Conjecture 1 *The sets generated by (hyper_primes bitpair) and (hyper_primes pepis_unpair) are infinite.*

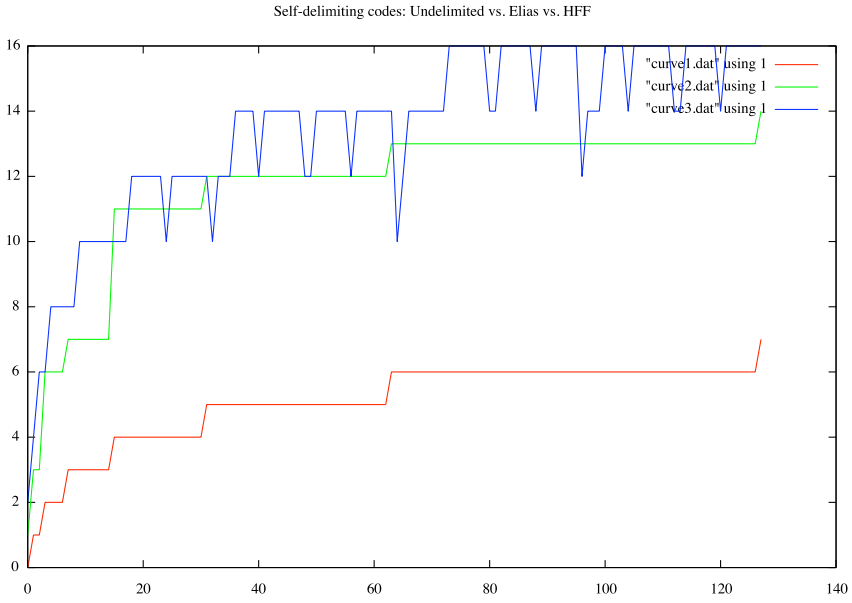


Fig. 51: Comparison of codes: curve1=Undelimited curve2=Elias, curve3=HFF up to 2^7

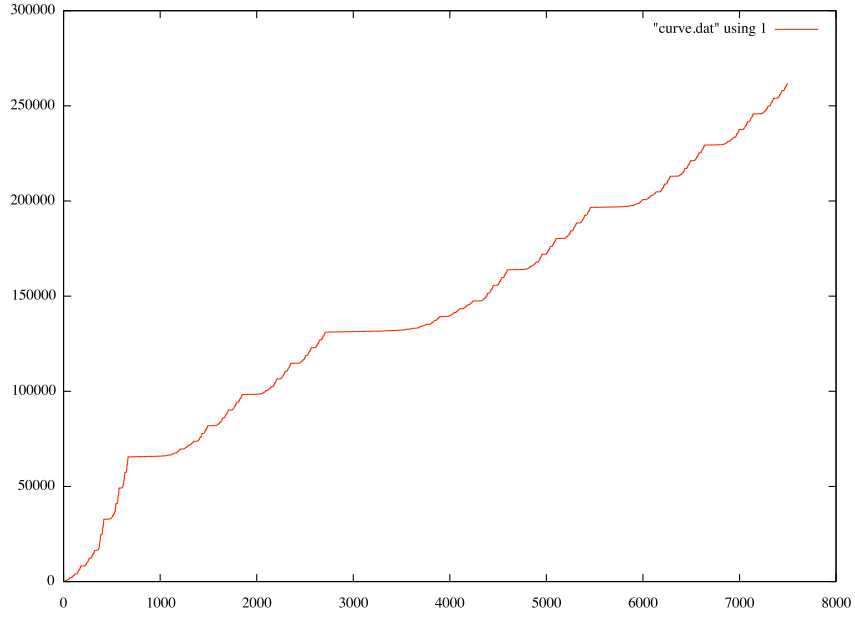


Fig. 52: Sparse numbers in $[0..2^{18}]$, x=nth sparse number, y=its value

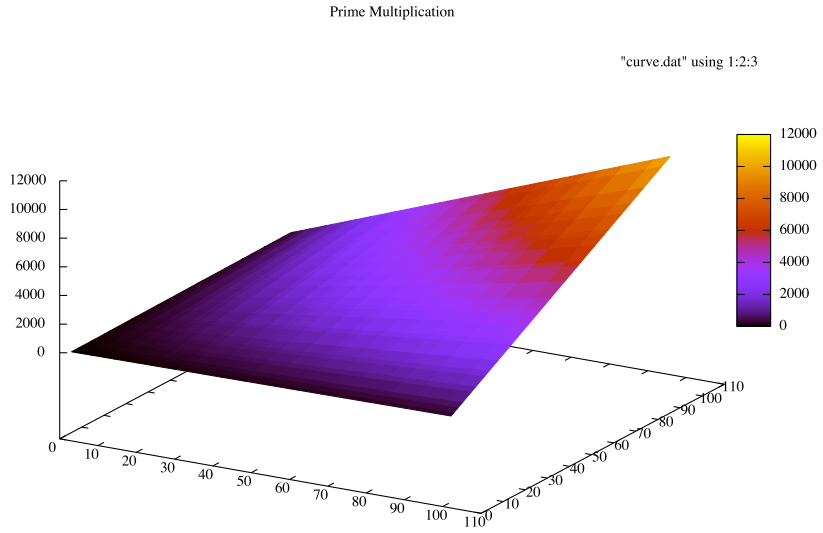


Fig. 53: Lossless multiplication of primes

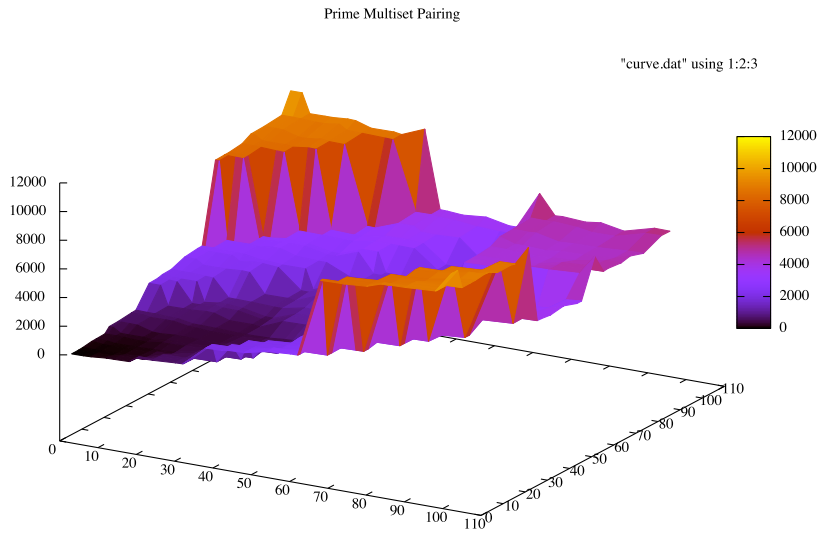


Fig. 54: Lossless multiset pairing of primes

Conjecture 2 *If* `u` *is a bijection from* $u : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ *such that:*

1. *if* $n > 1$ *and* $u\ n = (n_0, n_1)$ *then* $n_0 < n$ *and* $n_1 < n$
2. `p` *is a predicate on* $\mathbb{N}$ *such that* $P = \{n : p(n)\}$ *is infinite*

then the set $P \cap \{n : uparts\ u\ n\}$ *is also infinite.*

Figure 55 shows the complete unpairing graph for two hyper-primes obtained with `bitunpair`.

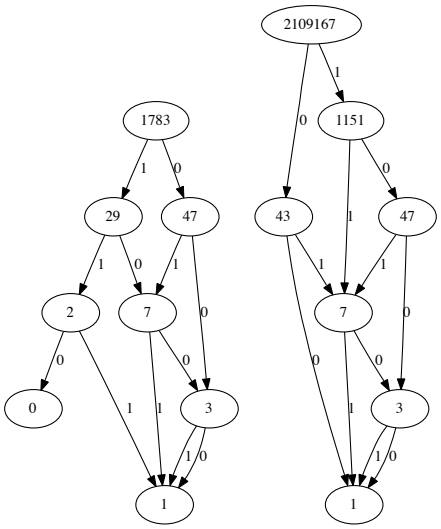


Fig. 55: `mset_unpair` hyper-primes: *1783* and *2109167*

It is interesting to compare the action of a pairing function on natural numbers with its action on primes and hyper-primes with products. Clearly products are not reversible, except when numbers are primes, while pairing functions are always reversible. To factor in the fact that products commute while pairing functions do not, we have considered $2xy$ instead of xy.

Figures 56 and 57 show this comparison.

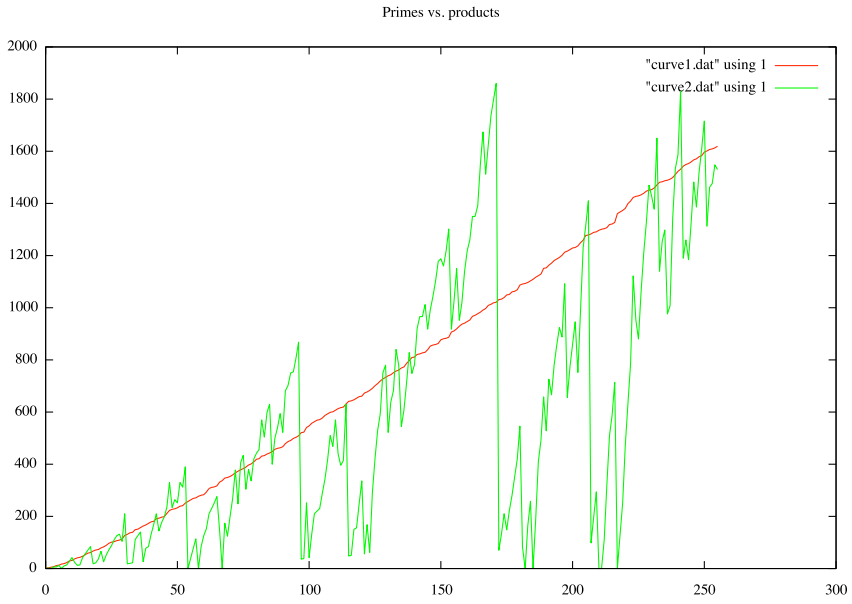


Fig. 56: Pairing of primes vs. 2xy

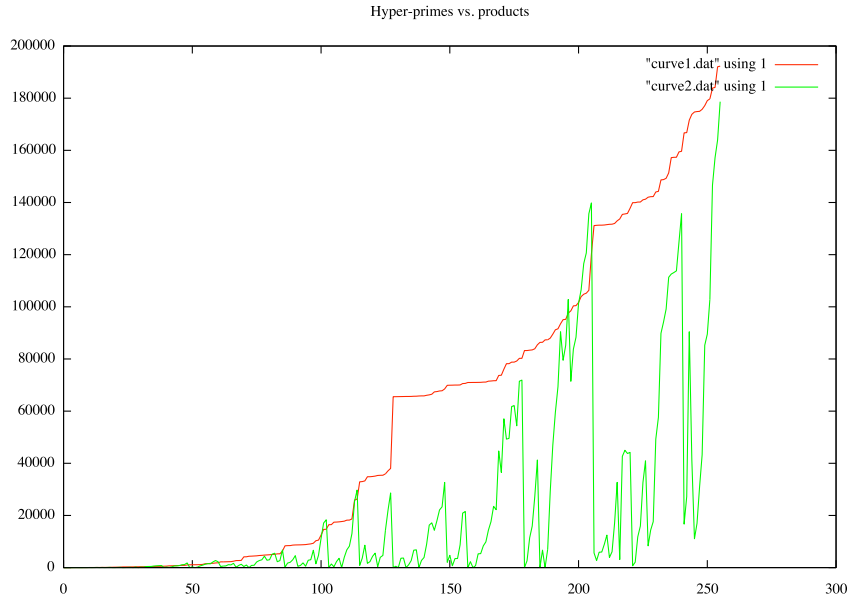


Fig. 57: Pairing of hyper-primes vs. 2xy

Hyper-primes and Fermat primes One could expect to model more closely the behavior of primes and products by focusing on commutative functions like the multiset pairing function `mset_pair`:

```
*ISO> take 16 (hyper_primes mset_unpair)
[2,3,5,13,17,113,173,257,10753,17489,34897,34961,43633,43777,65537,142781101]
```

We remind that:

Definition 3 *A Fermat-prime is a prime of the form* $2^{2^n}+1$ *with* $n > 0$.

Fig. 58 shows a hyper-prime that is also a Fermat prime and a hyper-prime that is not a Fermat prime.

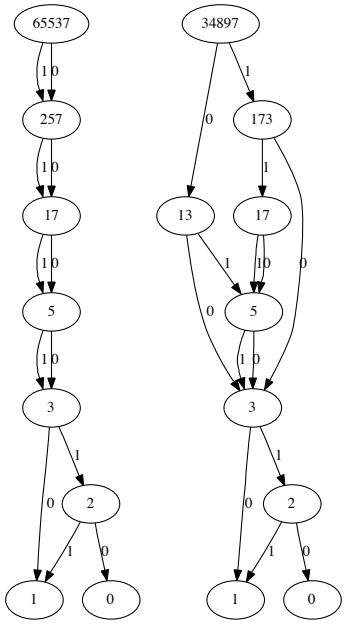


Fig. 58: `mset_unpair` hyper-primes: *Fermat prime* and *Non-Fermat prime*

This time a more interesting conjecture emerges. We can now state that:

Conjecture 3 *All Fermat primes are* `mset_unpair` *induced hyper-primes.*

We will just observe that this would follow from the widely believed conjecture that there the only Fermat primes are [3,5,17,257,65537] as these 5 primes are indeed on our list of `mset_unpair` hyperprimes.

In the event of the alternative, we will now state:

Proposition 23 *If there are Fermat primes other than [3,5,17,257,65537] then there are Fermat primes that are not* `mset_unpair` *hyper-primes.*

To prove Prop. 23 we need a few additional results. First, the following known fact, implying that we only need to prove that there are primes of the form $2^{2^n}+1$ that are not hyper-primes.

Lemma 1 *If $n > 0$ and $2^n + 1$ is prime then n is a power of 2.*

It is easy to prove, from the definition of `mset_pair` that:

Lemma 2

$$mset\_pair\ (2^{2^n}+1, 2^{2^n}+1) \equiv 2^{2^{n+1}}+1 \tag{26}$$

Indeed, from the identity 20 we obtain

$$mset\_pair(a,a) \equiv bitpair(a,0) \tag{27}$$

and then observe that from 15 it follows that

$$bitpair(2^{2^n}+1, 0) \equiv 2^{2^{n+1}}+1 \tag{28}$$

We can now prove Prop. 58. If $2^{2^{n+1}}+1$ is a Fermat prime that is also a hyper-prime, then $2^{2^n}+1$ would be also a Fermat prime that is hyper-prime. This would form a descending sequence of consecutive Fermat primes - a contradiction, given that it has been proven (by Leonhard Euler in 1732) that for instance, $2^{32}+1 = 641 * 6,700,417$ is not a prime.

28.5 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 [41]. In a similar way, to sort sequences with repeated elements one can write

```
strange_sort' = (to mset) . (from mset)
strange_sort'' = (as mset nat) . (as nat mset)

*ISO> strange_sort' [2,4,1,1,0,3,17,1.4]
[0,1,1,1,2,3,4,4,17]
*ISO> strange_sort'' [2,4,1,1,0,3,17,1,4]
[0,1,1,1,2,3,4,4,17]
```

28.6 Circuit Minimization

Let us consider the classic problem of synthesizing a half adder, composed of an XOR (`^`) and an AND (`*`) function. We can combine the two functions with an if-then-else with selector variable A to obtain: `ITE(A,B^C,B*C)` with the following truth table:

```
[0,0,0]:0
[0,0,1]:0
[0,1,0]:0
[0,1,1]:1
[1,0,0]:0
[1,0,1]:1
[1,1,0]:1
[1,1,1]:0
```

Note that this `3` argument single output function (encoded as the natural number `22` by reading its value column in binary), fuses the two operations with the upper half of the truth table representing the `AND` and the lower half representing the `XOR`. When running `to_min_bdd` on this function we obtain:

```
ISO> from_base 2 [0,1,1,0, 1,0,0,0]
22
*ISO> to_min_bdd 3 22
BDD 3 (D 0
  (D 1 (D 2 B0 B1) (D 2 B1 B0))
               (D 1 (D 2 B1 B0) B0))
```

28.7 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 [42] 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 [43]), 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.

29 Related work

The closest reference on encapsulating bijections as a Haskell data type is [44] and Conal Elliott's composable bijections module [45], where, in a more complex setting, Arrows [46] are used as the underlying abstractions. While our `Iso` data type is similar to the *Bij* data type in [45] and BiArrow concept of [44], the techniques for using such isomorphisms as building blocks of an embedded composition language centered around encodings as Natural Numbers are new.

As the domains between which we define our isomorphisms can be organized as categories, it is likely that some of our constructs would benefit from *natural transformation* [47] and *n-category* formulations [48].

Ranking functions can be traced back to Gödel numberings [5, 6] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [49–55]. 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 [56–61]. Computational and Data Representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in [19, 62].

Pairing functions have been used in work on decision problems as early as [10, 12]. A typical use in the foundations of mathematics is [63]. An extensive study of various pairing functions and their computational properties is presented in [64].

Various mappings from natural numbers to rational numbers are described in [65], also in a functional programming framework.

We have learned from Knuth's recent work on combinatorial algorithms [30] the techniques related to bitvector encodings of projection functions and boolean operations and about BDDs and reduced ordered BDDs from Bryant's seminal paper on the topic [29]. However, the connection with pairing/unpairing functions and the equivalence results of subsection 14.4 are new.

The concepts of hereditarily finite functions and permutations as well as their encodings, are likely to be new, given that our sustained search efforts have not lead so far to anything similar.

Some other techniques, ranging from factoradics to cons-lists and functional binary numbers to DNA encodings and dyadic rationals are for sure part of the scientific commons. In that case our focus was to express them as elegantly as possible in a uniform framework. In these cases as well, most of the time it was faster to "just do it", by implementing them from scratch in a functional programming framework, rather than adapting procedural algorithms found elsewhere.

30 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 groupoid 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

1. Tarau, P.: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In: Proceedings of ACM SAC'09, Honolulu, Hawaii, ACM (March 2009) 1898–1903
2. Lakoff, G., Johnson, M.: Metaphors We Live By. University of Chicago Press, Chicago, IL, USA (1980)
3. Cook, S.: Theories for complexity classes and their propositional translations. In: Complexity of computations and proofs. (2004) 1–36
4. Cook, S., Urquhart, A.: Functional interpretations of feasibly constructive arithmetic. Annals of Pure and Applied Logic **63** (1993) 103–200
5. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
6. Hartmanis, J., Baker, T.P.: On Simple Goedel Numberings and Translations. In Loeckx, J., ed.: ICALP. Volume 14 of Lecture Notes in Computer Science., Springer (1974) 301–316
7. Claessen, K., Hughes, J.: Testing monadic code with quickcheck. SIGPLAN Notices **37**(12) (2002) 47–59
8. Pepis, J.: Ein verfahren der mathematischen logik. The Journal of Symbolic Logic **3**(2) (jun 1938) 61–76
9. Kalmar, L.: On the Reduction of the Decision Problem. First Paper. Ackermann Prefix, A Single Binary Predicate. The Journal of Symbolic Logic **4**(1) (mar 1939) 1–9
10. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
11. Robinson, J.: Recursive functions of one variable. Proceedings of the American Mathematical Society **19**(4) (aug 1968) 815–820
12. Robinson, J.: Finite generation of recursively enumerable sets. Proceedings of the American Mathematical Society **19**(6) (dec 1968) 1480–1486

13. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM **3**(4) (1960) 184–195
14. Singh, D., Ibrahim, A.M., Yohanna, T., Singh, J.N.: An overview of the applications of multisets. Novi Sad J. Math **52**(2) (2007) 73–92
15. Rowland, E.S.: A natural prime-generating recurrence. Journal of Integer Sequences **11**(1) (2008)
16. Hutton, G.: A Tutorial on the Universality and Expressiveness of Fold. J. Funct. Program. **9**(4) (1999) 355–372
17. Meijer, E., Hutton, G.: Bananas in Space: Extending Fold and Unfold to Exponential Types. In: FPCA. (1995) 324–333
18. Ackermann, W.F.: Die Widerspruchsfreiheit der allgemeinen Mengenlhere. Mathematische Annalen (114) (1937) 305–315
19. Piazza, C., Policriti, A.: Ackermann Encoding, Bisimulations, and OBDDs. TPLP **4**(5-6) (2004) 695–718
20. Göbel, F.: On a 1-1-correspondence between rooted trees and natural numbers. J. Comb. Theory, Ser. B **29**(1) (1980) 141–143
21. Knuth, D.E.: The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
22. Mantaci, R., Rakotondrajao, F.: A permutations representation that knows what "eulerian" means. Discrete Mathematics & Theoretical Computer Science **4**(2) (2001) 101–108
23. Kalmar, Laszlo, Suranyi, Janos: On the Reduction of the Decision Problem. The Journal of Symbolic Logic **12**(3) (sep 1947) 65–73
24. Kalmar, Laszlo, Suranyi, Janos: On the Reduction of the Decision Problem: Third Paper. Pepis Prefix, a Single Binary Predicate. The Journal of Symbolic Logic **15**(3) (sep 1950) 161–173
25. Robinson, J.: A note on primitive recursive functions. Proceedings of the American Mathematical Society **6**(4) (aug 1955) 667–670
26. Robinson, J.: An introduction to hyperarithmetical functions. The Journal of Symbolic Logic **32**(3) (sep 1967) 325–342
27. Pigeon, S.: Contributions á la compression de données. Ph.d. thesis, Université de Montréal, Montréal (2001)
28. Goodstein, R.: On the restricted ordinal theorem. Journal of Symbolic Logic (9) (1944) 33–41
29. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691
30. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
31. Shannon, C.E.: Claude Elwood Shannon: collected papers. IEEE Press, Piscataway, NJ, USA (1993)
32. Fujita, M., McGeer, P.C., Yang, J.C.Y.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in System Design **10**(2/3) (1997) 149–169
33. Ciesinski, F., Baier, C., Groesser, M., Parker, D.: Generating compact MTBDD-representations from Probmela specifications. In: Proc. 15th International SPIN Workshop on Model Checking of Software (SPIN'08). (2008)
34. Misra, J.: Powerlist: a structure for parallel recursion. ACM Transactions on Programming Languages and Systems **16** (1994) 1737–1767
35. Bird, R., Gibbons, J., Jones, G.: Program optimisation, naturally. In: Millenial Perspectives in Computer Science. Palgrave. (1999)

36. Bucciarelli, A., Salibra, A.: The sensible graph theories of lambda calculus. In: LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, IEEE Computer Society (2004) 276–285
37. Berline, C.: Graph models of λ-calculus at work, and variations. Mathematical. Structures in Comp. Sci. **16**(2) (2006) 185–221
38. Li, Z.: Algebraic properties of dna operations. Biosystems **52** (October 1999) 55–61(7)
39. Hinze, T., Sturm, M.: 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. (2000) 257–266
40. Pippenger, N.: The average amount of information lost in multiplication. IEEE Transactions on Information Theory **51**(2) (2005) 684–687
41. Wadler, P.: Theorems for free! In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture, New York, NY, USA, ACM (1989) 347–359
42. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
43. Harris, T., Marlow, S., Jones, S.L.P., Herlihy, M.: Composable memory transactions. Commun. ACM **51**(8) (2008) 91–100
44. Alimarine, A., Smetsers, S., van Weelden, A., van Eekelen, M., Plasmeijer, R.: There and back again: arrows for invertible programming. In: Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM Press (2005) 86–97
45. Conal Elliott: Module: Data.Bijections Haskell source code library at: http://haskell.org/haskellwiki/TypeCompose.
46. Hughes, J.: Generalizing Monads to Arrows Science of Computer Programming 37, pp. 67-111, May 2000.
47. Mac Lane, S.: Categories for the Working Mathematician. Springer-Verlag, New York, NY, USA (1998)
48. Baez, J.C.: An introduction to n-categories. In: In 7th Conference on Category Theory and Computer Science, Springer-Verlag (1997) 1–33
49. Martinez, C., Molinero, X.: Generic algorithms for the generation of combinatorial objects. In Rovan, B., Vojtas, P., eds.: MFCS. Volume 2747 of Lecture Notes in Computer Science., Springer (2003) 572–581
50. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional (2009)
51. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming). Addison-Wesley Professional (2005)
52. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions. Addison-Wesley Professional (2005)
53. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming). Addison-Wesley Professional (2006)
54. Ruskey, F., Proskurowski, A.: Generating binary trees by transpositions. J. Algorithms **11** (1990) 68–84
55. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters **79** (2001) 281–284

56. Takahashi, M.o.: A Foundation of Finite Mathematics. Publ. Res. Inst. Math. Sci. **12**(3) (1976) 577–708
57. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume **48**(4) (2007) 497–510
58. Abian, A., Lamacchia, S.: On the consistency and independence of some set-theoretical constructs. Notre Dame Journal of Formal Logic **X1X**(1) (1978) 155–158
59. Avigad, J.: The Combinatorics of Propositional Provability. In: ASL Winter Meeting, San Diego (January 1997)
60. Kirby, L.: Addition and multiplication of sets. Math. Log. Q. **53**(1) (2007) 52–65
61. Leontjev, A., Sazonov, V.Y.: Capturing LOGSPACE over Hereditarily-Finite Sets. In Schewe, K.D., Thalheim, B., eds.: FoIKS. Volume 1762 of Lecture Notes in Computer Science., Springer (2000) 156–175
62. Paulson, L.C.: A Concrete Final Coalgebra Theorem for ZF Set Theory. In Dybjer, P., Nordström, B., Smith, J.M., eds.: TYPES. Volume 996 of Lecture Notes in Computer Science., Springer (1994) 120–139
63. Cégielski, P., Richard, D.: Decidability of the theory of the natural integers with the cantor pairing function and the successor. Theor. Comput. Sci. **257**(1-2) (2001) 51–77
64. Rosenberg, A.L.: Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science **14**(1) (2003) 3–17
65. Gibbons, J., Lester, D., Bird, R.: Enumerating the rationals. Journal of Functional Programming **16**(4) (2006)

Appendix

The code in the paper is organized in a module with the following dependencies:

```
module ISO where
import Data.List
import Data.Bits
import Data.Graph
import Data.Graph.Inductive
import Graphics.Gnuplot.Simple
import Data.Char
import Ratio
import Random
```

Bit crunching functions

The function bitcount computes the number of bits needed to represent an integer and max_bitcount computes the maximum bitcount for a list of integers.

```
bitcount n = head [x|x←[1..],(2^x)>n]
max_bitcount ns = foldl max 0 (map bitcount ns)
```

The following function convert a number to to binary, padded with 0s, up to maxbits.

```
to_maxbits maxbits n =
  bs ++ (genericTake (maxbits-l)) (repeat 0) where
    bs=to_base 2 n
    l=genericLength bs
```

Primes

The following code implements factoring function `to_primes` a primality test (`is_prime`) and a generator for the infinite stream of prime numbers `primes`.

```
primes = 2 : filter is_prime [3,5..]

is_prime p = [p]==to_primes p

to_primes n | n>1 = to_factors n p ps where
  (p:ps) = primes

to_factors n p ps | p*p > n = [n]
to_factors n p ps | 0==n `mod` p = p : to_factors (n `div` p)  p ps
to_factors n p ps@(hd:tl) = to_factors n hd tl
```

We will briefly describe here the functions used to visualize various data types with the help of Haskell libraries providing interfaces to `graphviz` and `gnuplot`.

Multiset Operations

The following functions provide multiset analogues of the usual set operations, under the assumption that multisets are represented as non-decreasing sequences.

```
msetInter xs ys = sort (msetInter' xs ys)

msetInter' [] _ = []
msetInter' _ [] = []
msetInter' (x:xs) (y:ys) | x==y =
  (x:zs) where zs=msetInter' xs ys
msetInter' (x:xs) (y:ys) | x<y = msetInter' xs (y:ys)
msetInter' (x:xs) (y:ys) | x>y = msetInter' (x:xs) ys

msetDif xs ys = sort (msetDif' xs ys)

msetDif' [] _ = []
msetDif' xs [] = xs
msetDif' (x:xs) (y:ys) | x==y = zs where
  zs=msetDif' xs ys
msetDif' (x:xs) (y:ys) | x<y = (x:zs) where
  zs=msetDif' xs (y:ys)
msetDif' (x:xs) (y:ys) | x>y = zs where
  zs=msetDif' (x:xs) ys
```

```
msetSymDif xs ys =
  sort ((msetDif xs ys) ++ (msetDif ys xs))

msetUnion xs ys = sort ((msetDif xs ys) ++
  (msetInter xs ys) ++ (msetDif ys xs))

msetIncl xs ys = xs==msetInter xs ys
```

Building a multigraph from a natural number using a function associating to each natural number a sequence or set of natural numbers.

```
fun2g ns = nat2fgs nat2fun ns
set2g ns = nat2sgs nat2set ns
perm2g ns = nat2fgs nat2perm ns
pmset2g ns = nat2fgs nat2pmset ns
bmset2g ns = nat2fgs nat2bmset ns

nat2fg f n = nat2gx fun_edge f nat2pftree n :: Gr N Int

nat2fgs f ns = nat2gsx fun_edge f nat2pftree ns :: Gr N Int

nat2sg f n = nat2gx set_edge f nat2pftree n :: Gr N ()

nat2sgs f ns = nat2gsx set_edge f nat2pftree ns :: Gr N ()

set_edge xs (a,b,i) = (lookUp a xs,lookUp b xs,())

fun_edge xs (a,b,i) = (lookUp a xs,lookUp b xs,i)

nat2gx e f g n = mkGraph vs  (map (e xs) es) where
  es=g f n
  (xs,vs)=labeledVertices es

nat2gsx e f g ns = mkGraph vs  (map (e xs) es)  where
  es=nub (concatMap (g f) ns)
  (xs,vs)=labeledVertices es

labeledVertices es= (xs,vs) where
  xs=fvertices es
  is=[0..(length xs)-1]
  vs = zip is xs

nat2pftree f n = nub (nat2pftreex f (n,n,0))

nat2pftreex f (_,n,_) = ps ++ (concatMap (nat2pftreex f) ps) where
  ps = nat2pfun f (n,n,0)
```

```
nat2pfun _ (_,0,_) = []
nat2pfun f (_,n,_) | n> 0 = ps where
  ps = zipWith (λx i→(n,x,i)) (f n) [0..]

fvertices ps = (sort . nub) (concatMap f ps) where
  f (a,b,_) = [a,b]

lookUp n ns = i where Just i=elemIndex n ns
```

Building Inductive Graphs from Lists of Pairs

We can build a graph directly from edges represented as pairs of small integers as follows:

```
edges2gr ::  [(N,N)] → Gr Int ()

edges2gr es = mkGraph lvs les where
  vs=[0..foldl max 0 (concatMap g es)]
  lvs=zip vs vs
  les=map f es
  f (x,y) = (fromIntegral x,fromIntegral y,())
  g (x,y) = [fromIntegral x,fromIntegral y]
```

When the set of vertices is sparse, vertex numbers are used as labels while actual vertices become integers in [0..numberOf Vertices-1]

```
pairs2gr ::  [(N,N)] → Gr N ()

pairs2gr ps = mkGraph lvs les where
  vs=to_vertices ps
  lvs=zip [0..] vs
  es=to_edges vs ps
  les=map f es
  f (x,y) = (x,y,())

to_vertices es = sort $ nub $ concat [[fst p,snd p]|p←es]

to_edges vs ps = map (f vs) ps where
  f vs (x,y) = (lookUp x vs,lookUp y vs)
```

Generating labeled edge triplets by recursing over unpairing functions

The following function represents a number as a set of triplets expressing branches of decomposition with an unpairing function `f`, for instance, in the case of BDDs with function `bitunpair`.

```
unpairing_edges f tt = nub  (h f tt) where
  h _ tt | tt<2 = []
  h f n  = ys where
     (n0,n1)=f n
```

```
    ys= (n,n0,0):(n,n1,1):
          (h f n0) ++
          (h f n1)
```

The function works as follows:

```
*ISO> unpairing_edges bitunpair 42
[(42,0,0),(42,7,1),(7,3,0),(7,1,1),(3,1,0),(3,1,1)]
*JFISO> unpairing_edges pepis_unpair 42
[(42,0,0),(42,21,1),(21,1,0),(21,5,1),(5,1,0),(5,1,1)]
*ISO>
```

Generating labeled edge triplets by recursing over untupling functions

The following function represents a number as a set of triplets expressing branches of decomposition with an untupling function `fk`, for instance `to_tuple k`.

```
untupling_edges f k l tt = nub  (h f k tt) where
  h _ _ tt | tt<l = []
  h f k n  = ys where
     ns = f k n
     ys = (zip3 (repeat n) ns [0..]) ++
          (concatMap (h f k) ns)
```

The function works as follows:

```
*ISO> untupling_edges to_tuple 3 2 2008
[(2008,14,0),(2008,14,1),(2008,4,2),(14,2,0),(14,1,1),(14,1,2),
 (2,0,0),(2,1,1),(2,0,2),(4,0,0),(4,0,1),(4,1,2)]
```

Building Inductive Graphs from hereditary base-k trees

We will first build a set of labeled edges, recursing over expansion to polynomials in base k.

```
nat2edges k n = xs ++ (concatMap (expandEdge k) xs) where
  p2e (c,e) = (n,e,c)
  xs= map p2e (nat2kpoly k n)

expandEdge k (_,e,_) | e < k = []
expandEdge k (_,e,_) = nat2edges k e
```

working as follows:

```
*ISO> nat2edges 3 42
[(42,1,2),(42,2,1),(42,3,1),(3,1,1)]
*ISO> nat2edges 3 2009
[(2009,0,2),(2009,2,1),(2009,3,2),(2009,5,2),(2009,6,2),(3,1,1),(5,0,2),(5,1,1),(6,1,2)]
```

Building Inductive Graphs from Unpairing, Untupling and Hereditary base-k Trees

We can now turn a BDD as well as any other unpairing/untupling/hereditary base k function generated tree into an inductive graph, as follows:

```
to_unpair_graph f tt = nat2fun_graph (unpairing_edges f) tt

to_untuple_graph f k l tt = nat2fun_graph (untupling_edges f k l) tt

nat2fun_graph f n = nat2graph f n :: Gr N Int

to_hb_graph k n = nat2hb_graph (nat2edges k) n

nat2hb_graph f n = nat2graph f n :: Gr N N

nat2graph f n = mkGraph vs fs where
  es=f n
  (xs,vs)=labeledVertices es
  fs=map (fun_edge xs) es
```

The functions work as follows:

```
*ISO> to_unpair_graph bitunpair 42
0:0→[]
1:1→[]
2:3→[(1,1),(0,1)]
3:7→[(0,2),(1,1)]
4:42→[(1,3),(0,0)]

*ISO> to_unpair_graph pepis_unpair 42
0:0→[]
1:1→[]
2:5→[(1,1),(0,1)]
3:21→[(1,2),(0,1)]
4:42→[(1,3),(0,0)]

*ISO> to_untuple_graph to_tuple 3 2 2008
0:0→[]
1:1→[]
2:2→[(2,0),(1,1),(0,0)]
3:4→[(2,1),(1,0),(0,0)]
4:14→[(0,2),(2,1),(1,1)]
5:2008→[(2,3),(1,4),(0,4)]

*ISO> to_hbase_graph 4 123456789

0:1→[]
1:2→[]
2:3→[]
3:4→[(1,0)]
```

```
4:5→[(1,0),(0,0)]
5:7→[(1,0),(0,2)]
6:8→[(1,1)]
7:9→[(1,1),(0,0)]
8:10→[(1,1),(0,1)]
9:11→[(1,1),(0,2)]
10:12→[(1,2)]
11:13→[(1,2),(0,0)]
12:123456789→[(13,0),(12,2),(11,0),(10,0),(9,1),(8,2),
               (7,2),(5,2),(4,0),(2,0),(1,0),(0,0)]
```

Visualization with Graphviz

```
gviz g = writeFile "iso.gv"
  ((graphviz g "" (0.0,0.0) (2,2) Portrait)++"λn")

funviz f n = gviz (nat2fg f n)

setviz f n = gviz (nat2sg f n)

eviz t n = gviz (edges2gr (as t nat n)) -- dag, digraph

pviz t n = gviz (pairs2gr (as t nat n))

psviz ps = gviz (pairs2gr ps)

uviz f tt = gviz (to_unpair_graph f tt) -- unpairings

tviz f k tt = gviz (to_untuple_graph f k 2 tt)

pbviz k tt = gviz (to_untuple_graph to_sqbase k k tt)

v3x1 k tt = gviz (to_untuple_graph to_3x1 k k tt)

hbviz k n = gviz (to_hb_graph k n)

fviz t n = gviz (es2g (as t nat n))

es2g :: [(N, N)] → Gr () Int

es2g es = mkGraph lvs les where (lvs,les)=es2ls es

es2ls es = (lvs,les) where
  es' = map (λ(x,y)→(fromIntegral x,fromIntegral y)) es
  g ((x,y),l)=(x,y,l)
  les=map g (zip es' [0..length es-1])
  vs = [0..foldr max 0 (concatMap f es')]
  lvs=map (λx→(x,())) vs
  f (from,to)=[from,to]
```

Plotting with gnuplot

```
plot3d f xs ys = plotFunc3d [Title ""] [] xs ys f

plotop op m= plot3d op xs xs where xs=[0..2^m-1]

cplot3d f = plot3d (curry f)

pair3d pf m | m≤8 = cplot3d pf ls ls where ls=[0..2^m-1]

plotpairs m | m≤2^8 = cplot3d bitpair ls ls where ls=[0..m-1]

plotdyadics m = plotList
  [Title "Dyadics"]
  (map (fromRational . (as dyadic nat)) [0..m-1])

plotkraft t m = plotList
  [Title "Kraft function"]
  (map (kraft t) [0..2^m-1])

plotelias m = plotList
  [Title "Elias encoding length"]
  (map kraft_elias [0..2^m-1])

sizes_to m t = map (size_as t) [0..m-1]

plot_hf m = plotLists [Title "Bit, BDD, HFF, HFS, and HFP sizes"]
  (
    [bits_to m,bsizes_to m] ++
    (map (sizes_to m) [hff,hfs,hfm,hfp])
  )

plot_hf1 m = plotLists [Title "Bit, HFF, HFS, HFM and HFP sizes"]
  (
    [bits_to m] ++
    (map (sizes_to m) [hff,hfs,hfm,hfp])
  )

plot_good m = plotLists [Title "Bit and HFF sizes"]
  (
    [bits_to m] ++
    (map (sizes_to m) [hff])
  )

plot_best m = plotLists [Title "Bit, BDD and HFF and HFF' sizes"]
  (
    [bits_to m,bsizes_to m] ++
    (map (sizes_to m) [hff,hff'])
  )
```

```
plot_worse m = plotLists [Title "HFM, HFS and HFP sizes"]
  (
    (map (sizes_to m) [hfm,hfs,hfp])
  )

plot_hfs_hfp m = plotLists [Title "HFS and HFP sizes"]
  (
    (map (sizes_to m) [hfs,hfp])
  )

plot hf m = plotx [hf] m

plotx hfx m = plotLists [Title "HF tree size"]
  (
    (map (sizes_to (2^m-1)) hfx)
  )

-- plots pairs
pplot f m = plotPath [] (map (to_ints . f) [0..2^m-1])

priplot f m = plotPath [] (map (to_ints . f) (genericTake m primes))

zplot f m = plotPath [] (map (to_ints . f) [-(2^m)..2^m-1])

to_ints (i,j)=(fromIntegral i,fromIntegral j)

diplot n = plotPath [] (map to_ints (as digraph nat n))

bsize_of n = robdd_size (as rbdd nat n)

bsizes_to m = map bsize_of [0..m-1]

bits_to m = map s [0..m-1] where s n = genericLength (as bits nat n)

plot_linear_sparseness m = plotLists [Title "Linear Sparseness"]
  [(map (linear_sparseness fun) [0..m-1]),
   (map (linear_sparseness pmset) [0..m-1]),
   (map (linear_sparseness mset) [0..m-1]),
   (map (linear_sparseness set) [0..m-1]),
   (map (linear_sparseness perm) [0..m-1])]


plot_sparseness m = plotLists [Title "Recursive Sparseness"]
  [(map (sparseness hff_pars) [0..m-1]),
   (map (sparseness hfpm_pars) [0..m-1]),
   (map (sparseness hfm_pars) [0..m-1]),
   (map (sparseness hfs_pars) [0..m-1]),
   (map (sparseness hfp_pars) [0..m-1])]

plot_sparseness1 m = plotLists
```

```
  [Title "Recursive Sequence vs. Multiset Sparseness"]
  [
   (map (sparseness hff_pars) [0..m-1]),
   (map (sparseness hfpm_pars) [0..m-1])
  ]

plot_sparseness2 m = plotLists [Title "Recursive Multiset Sparseness"]
  [
   (map (sparseness bhfm_pars) [0..m-1]),
   (map (sparseness hfm_pars) [0..m-1])
  ]

plot_sparseness3 m = plotLists [Title "Recursive Multiset Sparseness"]
  [
   (map (sparseness hff_pars) [0..m-1]),
   (map (sparseness hff_pars') [0..m-1])
  ]


plot_sparseness4 m = plotLists
  [Title "Recursive Multiset vs Multiset with Primes Sparseness"] [
   (map (sparseness hfm_pars) [0..m-1]),
   (map (sparseness hfpm_pars) [0..m-1])
  ]

plot_sparseness5 m = plotLists
  [Title "Recursive Multisets vs. Sequences"] [
   (map (sparseness hff_pars) [0..m-1]),
   (map (sparseness hfm_pars) [0..m-1])
  ]

plot_selfdels m = plotLists
   [Title "Self-delimiting codes: Undelimited vs. Elias vs. HFF"]
   [(map (genericLength . (as bits nat)) [0..m-1]),
    (map (genericLength . (as elias nat)) [0..m-1]),
    (map (genericLength . (as hff_pars nat)) [0..m-1])]

plot_pairs_prods unpF m = plotLists [Title "Pairs vs. products"]
   [ms,prods] where
     ms=[1..m]
     pairs=map unpF ms
     prods=map prod pairs where prod (x,y)=2*x*y

plot_lifted_pairs m =
   plotLists [Title "Lifted pairs"]  [us0,us1] where
     ms=[0..m-1]
     pairs=map bitunpair ms
     us0=map fst pairs
     us1=map snd pairs
```

```
plot_lifted_pairs1 m =
   plotLists [Title "Lifted pairs and products"]  [ps,s0,s1,xys] where
     ms=[0..m-1]
     pairs=map bitunpair ms
     us0=map fst pairs
     us1=map snd pairs
     ps=zipWith (*) us0 us1
     s0=map (^2) us0
     s1=map (^2) us1
     xys=map f pairs where
       f (x,y) = x*y

plot_primes_prods m = plotLists [Title "Primes vs. products"]
  [ps,prods] where
     ms=[0..m]
     ps=genericTake m primes
     pairs=map bitunpair ps
     prods=map prod pairs where prod (x,y)=2*x*y

plot_hypers_prods m = plotLists [Title "Hyper-primes vs. products"]
   [ps,prods] where
     ms=[0..m]
     ps=genericTake m (hyper_primes bitunpair)
     pairs=map bitunpair ps
     prods=map prod pairs where prod (x,y)=2*x*y
```

Generated Figures

```
hset n=gviz  (nat2sg nat2set n)
hfun n=gviz (nat2fg nat2fun n)

hmset n=gviz (nat2fg nat2mset n)
hperm n=gviz (nat2fg nat2perm n)
dig n =pviz digraph n
dig' n =pviz digraph' n
plotdag n = pviz dag n

hset' n=gviz  (nat2sg nat2set' n)
hfun' n=gviz (nat2fg nat2ftuple n)
hmset' n=gviz (nat2fg nat2mset' n)
hpset' n=gviz (nat2sg ((as set mset) . nat2pmset) n)

unp' tt= uviz unpair' tt
unpp' n=pplot unpair' n
unpp n=pplot bitunpair n

---cunp tt= uviz cunpair tt
cunpp n=pplot cunpair n

plotcantor n =pplot cantor_unpair n
```

```
p3d' m = pair3d pair' m

p3d pf m = pair3d pf m

ukl k l n = uviz (unpairKL k l) n

uklp k l n = pplot (unpairKL k l) n

klp k l m = pair3d (pairKL k l) m

f4=gviz (nat2fg nat2perm 2009)

f6=plotpairs 64
f7=plotdyadics 256

f8=plot_best (2^6)
f8a=plot_good (2^10)

f9=plot_worse (2^10)
f9a=plot_hfs_hfp (2^16)
f9b=plot_hfs_hfp (2^17)
f10=plot_hf (2^8)
f10a=plot_hf1 (2^8)

f11a=plot_linear_sparseness (2^7)
f11=plot_sparseness (2^8)
f11b=plot_sparseness1 (2^8)
f11c=plot_sparseness2 (2^10)

f12=plot_sparseness (2^14)

f13=plot_sparseness (2^17)

f14=plot_selfdels (2^7)

fs1 m=plotLists
  [Title "Representation Sizes"]
  [bs,es,hffs,hfms,hfss,hfps] where
       bsize n=genericLength (as bits nat n)
       bs=map bsize [0..2^m-1]
       esize n=genericLength (as elias nat n)
       es=map esize [0..2^m-1]
       hffs=parsizes_as hff m
       hfms=parsizes_as hfm m
       hfss=parsizes_as hfs m
       hfps=parsizes_as hfp m

fs1a m=plotLists
  [Title "Representation Sizes"]
```

```
  [bs,es,hffs] where
       bsize n=genericLength (as bits nat n)
       bs=map bsize [0..2^m-1]
       esize n=genericLength (as elias nat n)
       es=map esize [0..2^m-1]
       hffs=parsizes_as hff m

fs1b m=plotLists
  [Title "Representation Sizes"]
  [hfms,hfss,hfps] where
       hfms=parsizes_as hfm m
       hfss=parsizes_as hfs m
       hfps=parsizes_as hfp m


f15=plotList [] (sparses_to (2^18))

dip m = plotf di m

dup m = plotf ndual m

-- see also plot3d f m
plotf f m = plotList [] (map f [0..2^m-1])
plotf1 f k m = plotList [] (map f [2^k..2^m-1])

penc0 = plotf (pencode 12239) 8

plotss xss = plotFunc3d [] [] ks ys f where
  ks=[0..length xss-1]
  ls=map length xss
  len=foldl max 0 ls
  ys=[0..len-1]
  zss= map (λxs→take len (xs++repeat 0)) xss
  f x y =   (zss!!(fromIntegral x))!!(fromIntegral y)


f16=gviz (nat2fgs nat2fun [0..7])

arp24 i =468395662504823 + 205619*23*i

arps24 = map arp24 [0..23]

arp25 i = 6171054912832631 + 366384*23*i

arps25 = map arp25 [0..24]

f17 = gviz (fun2g arps24)
f17a = gviz (fun2g arps25)

f18 = gviz (fun2g [2^65+1,2^131+3])
```

```
f18a = gviz (set2g [2^65+1,2^131+3])


f19 = gviz (fun2g [0..7])

f20 = gviz (pmset2g [0..7])

f20a = gviz (bmset2g [0..7])

f21 = gviz (set2g [0..7])

f22 = gviz (perm2g [0..7])

isoigraph n = psviz (to_igraph (as hypergraph nat n))
isobi n = psviz (as bipartite nat n)

g1 tt= uviz bitunpair tt
g2 tt= uviz pepis_unpair tt
g2' tt= uviz pepis_unpair' tt
g3 tt= uviz rpepis_unpair tt

ug1 tt = uviz (altunpair bitunpair) tt
ug2 tt = uviz (altunpair pepis_unpair) tt

up1 n=pplot (altunpair bitunpair) n
up2 n=pplot (altunpair pepis_unpair) n


isofermat=uviz mset_unpair 65537
isofermat1=uviz mset_unpair 142781101
isonfermat=uviz mset_unpair 34897
mhyper n=uviz mset_unpair (2^n+1)
punpairs n=uviz mset_unpair (2^n+1)


prifraq n=priplot (punpair bitunpair) n -- 1000

isopairs = plot_pairs_prods bitunpair 256
misopairs = plot_pairs_prods mset_unpair 256

isoprimes = plot_primes_prods 256
isohypers = plot_hypers_prods 256

ip1 n=pplot bitunpair n
ip2 n=pplot pepis_unpair n
isounpair3=pplot mset_unpair 10
isounpair4=pplot xorunpair 10
isoyunpair n =pplot yunpair n
```

```
isozunpair n=zplot zunpair n
isorepair i1 i2 m = plotList [] (map (repair i1 i2) [0..2^m-1])

isobij f m = plotList [] (map f [0..2^m-1])

-- xorbij is the diff between id and f
isoxbij f m = plotList [] (map g [0..2^m-1]) where g=xorbij f

-- bijection N→N using multisets and factorings
ms2pms n = as nat pmset (as mset nat n)

pms2ms n = as nat mset (as pmset nat n)

-- same but iterating k times
kms2pms 0 n = n
kms2pms k n = ms2pms (kms2pms (k-1) n)

kpms2ms 0 n = n
kpms2ms k n = pms2ms (kpms2ms (k-1) n)

lms k m = [x|x←[0..2^m-1], kms2pms k x < kpms2ms k x]

xms k m = [x|x←[0..2^m-1],kms2pms k x < x]

eqms k m = [x|x←[0..2^m-1],kms2pms k x == x]

xpms k m = [x|x←[0..2^m-1],kpms2ms k x < x]

eqpms k m = [x|x←[0..2^m-1],kpms2ms k x == x]

qms k m =
 [(toRational (kpms2ms k x)) - (toRational (kms2pms k x))|x←[1..2^m-1]]

q1 k m = plotList []  (qms k m)

q2 k m = plotLists []
  [map (kms2pms k) xs,map (kpms2ms k) xs] where
    xs = [0..2^m-1]

mult_vs_pairing p1 p2 = fromRational ( (p1*p2) % (ppair bitpair (p1,p2)))
mult_vs_mset_pairing p1 p2 = fromRational ((p1*p2) % (ppair mset_pair (p1,p2)))


q3 n = plotFunc3d
        [Title "Prime Multiplication vs. Prime Pairing"] []
          ps ps  mult_vs_pairing where
            ps=genericTake n primes

q4 n = plotFunc3d
  [Title "Prime Multiplication vs. Prime Multiset Pairing"] []
```

```
        ps ps mult_vs_mset_pairing where
        ps=genericTake n primes

n4a n = plotFunc3d [Title "Multiplication"] []
        ps ps (*) where
          ps=[0..2^n-1]

n4b n = plotFunc3d [Title "Multiset Pairing"] []
        ps ps (curry mset_pair) where
          ps=[0..2^n-1]


n4c n = plotFunc3d [Title "mprod operation"] []
        ps ps (mprod) where
          ps=[0..2^n-1]

n4d n = plotFunc3d [Title "pmprod' operation"] []
        ps ps (pmprod') where
          ps=[0..2^n-1]

n4e n = plotFunc3d [Title "mprod' operation"] []
        ps ps (mprod') where
          ps=[0..2^n-1]


n4f n = plotFunc3d [Title "mprod' x y/ x * y"] []
        ps ps (λx y→fromRational ((mprod' x y) % (x*y))) where
          ps=[1..2^n]

mplot f ps = plotFunc3d [Title "product operation"] [] ps ps f

dplot f g ps = plotFunc3d [Title "division comparison of operations"] []
        ps ps (λx y→fromRational ((f x y) % (g x y))) where

nplot f n = mplot f [1..2^n]
rplot f n = mplot f (genericTake n primes)

ndplot f g n = dplot f g [1..2^n]
rdplot f g n = dplot f g  (genericTake n primes)

expMexp k m = plotLists []
   [map (λx→x^k) xs, map (λx→mexp' x k) xs] where
   xs = [0..2^m]

p4a n = plotFunc3d [Title "Prime Multiplication"] []
        ps ps (*) where
          ps=genericTake n primes

p4b n = plotFunc3d [Title "Prime Multiset Pairing"] []
        xs ys (curry mset_pair) where
```

```
        ps=genericTake n primes
        xs=ps
        ys=ps

p4c n = plotFunc3d [Title "mprod on primes"] []
        xs ys (mprod) where
        ps=genericTake n primes
        xs=ps
        ys=ps

p4d n = plotFunc3d [Title "pmprod on primes"] []
        xs ys (pmprod) where
        ps=genericTake n primes
        xs=ps
        ys=ps


p4f n = plotFunc3d [Title "mprod' x y/ x * y"] []
        ps ps (λx y→(mprod' x y) % (x*y)) where
          ps=genericTake n primes

q4c n = plotFunc3d [Title "Prime Pairing"] []
        ps ps (curry bitpair) where
          ps=genericTake n primes

q5 n = plotLists
  [Title "Prime Multiplication vs. Prime Pairing curves"]
  [prods,pairs] where
    us= map bitunpair [0..2^n-1]
    (xs,ys) = unzip us
    ps=primes
    xs'=map (from_pos_in ps) xs
    ys'=map (from_pos_in ps) ys
    prods = zipWith (*) xs' ys'
    us'=zip xs' ys'
    pairs= map (ppair mset_pair) us'

plot_gauss_op f m = plotFunc3d title [] zs zs (curry f) where
  title=[Title "Gauss Integer operations through Pairing Functions"]
  zs=[-2^m..2^m-1]

gsum m = plot_gauss_op gauss_sum m
gdif m = plot_gauss_op gauss_dif m

gprod m = plot_gauss_op gauss_prod m
```

Experiments with $\mathbb{N} \to \mathbb{N}$ endomorphisms

```
compose_nperm l nf nx = ps where
   fs=nth2perm (l,nf)
```

```
   xs=nth2perm (l,nx)
   ys=applyPerm fs xs
   (_,ps)=perm2nth ys

ip n = as nat perm cs where
  bs=as perm nat n
  cs=invertPerm bs

rp n = as nat perm cs where
  bs=as perm nat n
  cs=reverse bs

ipb = ip . ib
ibp = ib . ip

--permPow l p k =

-- complement borrowed from bits
ib n = as nat bits cs where
  bs=as bits nat n
  cs=invertBits bs

-- complement in {0,1}^*
invertBits bs = map (λx→if 0==x then 1 else 0) bs

pc1 l = plotFunc3d [Title "Product as permutations"]
  [] ns ns (λx y→compose_nperm l (fromIntegral x) (fromIntegral y)) where
    lim=product [1..l]
    ns=[0..lim-1]


-- nat to nat bijections
pflip u p n = p (y,x) where (x,y) = u n

pf = pflip pepis_unpair pepis_pair
bf = pflip bitunpair bitpair

xorunpair n = (x `xor` y,y) where (x,y)=bitunpair n

xorpair (x,y) = bitpair (x `xor` y,y)

yunpair n = (x `xor` y,y) where (x,y)=pepis_unpair' n

ypair (x,y) = pepis_pair' (x `xor` y,y)

xb = bitpair . xorunpair

nr = (as nat fun) . (fit reverse nat)

di n = as nat digraph (map rev ps) where
```

```
  ps=as digraph nat n
  rev (x,y)=(y,x)

xorbij f n = n `xor` (f n)

enctest w = isoDecode bs pwd (isoEncode bs pwd w) where
  pwd="eureka"
  bs=bijlist

isoDecode bs pwd txt =  encodeWith bs reverse pwd txt

isoEncode bs pwd txt = encodeWith bs id pwd txt

bijlist = [di,bf,ip,xb,ib,rp,nr,(xbij 2 3),(xbij 3 4)]

encodeWith bs r pwd txt = as string nat ctxt where
  ntxt = as nat string txt
  npwd = as nat string pwd
  b x = npwd `xor` x
  ctxt = foldr (.) id (r (pwd2bs npwd (b:bs))) ntxt

pwd2bs npwd bs = newbs where
  l=genericLength bs
  lfact=product [1..l]
  mpwd=npwd `mod` lfact
  wperm = nth2perm (l,mpwd)
  newbs=applyPerm wperm bs

combWith f xs ys = [f x y|x←xs,y←ys]

comb f m = sort (combWith f xs xs) where xs=[0..2^m-1]
```