Under consideration for publication in J. Functional Programming

FUNCTIONAL PEARLS

On Primes and Pairs: Experimental Mathematics in Haskell

PAUL TARAU
Department of Computer Science and Engineering
University of North Texas
(*e-mail:* tarau@cs.unt.edu)

Abstract

This paper uses the functional programming language Haskell as a catalyst for exploring analogies in experimental mathematics, derived from bijective mappings between dataypes.

In the process, we emulate properties of *prime numbers* through isomorphisms connecting them to computationally simpler representations involving *multisets* and explore interactions between prime numbers and *pairing/unpairing functions*, resulting in a number of interesting facts and conjectures about various classes of primes.

The paper is organized as a self-contained *literate Haskell program* inviting the reader to explore its content independently. The code file is also directly available at `http://logic.cse.unt.edu/tarau/research/2009/jfpPRIMES.hs`

Keywords: *experimental mathematics and functional programming, bijective datatype transformations, multiset encodings, pairing functions and prime numbers, prime number sequences, Fermat primes, hyperprimes*

1 Introduction

Paul Erdös's statement, shortly before he died, that *"It will be another million years at least, before we understand the primes"* is indicative of the difficulty of the field as perceived by number theorists. The growing number of conjectures (Cgielski *et al.*, 2007) and the large number of still unsolved problems involving prime numbers (Crandall & Pomerance, 2005) shows that the field is still open to surprises, after thousands of years of efforts by some of the brightest human minds.

Interestingly, some significant progress on prime numbers correlates with unexpected paradigm shifts, the prototypical example being Riemann's paper (Riemann, 1859) connecting primality and complex analysis, all evolving around the still unsolved *Riemann Hypothesis* (van de Lune *et al.*, 1986; Miller, 1975; Chaitin, 2004). The genuine difficulty of the problems and the seemingly deeper and deeper connections with fields ranging from cryptography to quantum physics suggest that unusual venues might be worth trying out.

A significant number of recent results on prime numbers involve extensive computer experiments (Borwein *et al.*, 2000) as their most interesting properties quickly take us beyond the edges of human intuition. In particular, despite fascination with *Fermat primes*

some basic conjectures about them remains unanswered - for instance it is still not known if there are any other Fermat primes besides 3,5,17,257 and 65537.

A number of breakthroughs in various sciences involve small scale emulation of complex phenomena. Common sense analogies thrive on our ability to extrapolate from simpler (or, at least, more frequently occurring and better understood) mechanisms to infer surprising properties in a more distant ontology.

Prime numbers exhibit a number of fundamental properties of natural phenomena and human artifacts in an unusually pure form. For instance, *reversibility* is present as the ability to recover the operands of a product of distinct primes. This relates to the information theoretical view of multiplication (Pippenger, 2005) and it suggests investigating connections between combinatorial properties and operations on multisets and multiplicative number theory.

With such methodological hints in mind, this paper will explore mappings between multiset encodings and prime numbers, that have emerged from our effort to provide a compositional and extensible *data type transformation framework* connecting most of the fundamental data types used in computer science with a *groupoid of isomorphisms* (Tarau, 2009a; Tarau, 2009d; Tarau, 2009b). Another analogy, facilitated by the same framework, is between *reversible operations* like the product of two primes and its factoring on one hand, and *pairing/unpairing functions* on the other. As a result, more tractable concepts that "emulate" properties of primes emerge, like *multiset primes* and *hyperprimes*.

The paper (also a *literate Haskell program!*) is organized as follows. Section 2 revisits the well-known connection between multisets and primes using a variant of Gödel's encoding (Gödel, 1931). Section 3 describes our computationally efficient multiset encoding. Based on these encodings, section 4 explores the analogy between multiset decompositions and factoring and describes a multiplicative monoid structure on multisets that "emulates" properties of the monoid induced by ordinary multiplication. Section 5 describes pairing/unpairing functions and studies some of their algebraic properties that are used in section 6 to introduce *hyperprimes* and study their connection to Fermat primes and related conjectures in section 7. Section 8 overviews some related work and section 9 concludes the paper. We organize our literate programming code as a Haskell module, relying only on the a few library modules.

```
module MPrimes where
import Data.List
import Data.Bits
```

2 Encoding finite multisets with primes

Multisets (Singh *et al.*, 2007) are unordered collections with repeated elements. Non-decreasing sequences provide a canonical representation for multisets of natural numbers.

The mapping between finite multisets and primes described in this section goes back to Gödel's arithmetic encoding of formulae (Gödel, 1931; Hartmanis & Baker, 1974). A factorization of a natural number is uniquely described as multiset of primes. We can 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. This

mapping is the same as the *prime counting function* traditionally denoted $\pi(n)$, which associates to n the number of primes smaller or equal to n, restricted to primes. It is provided by the function `to_prime_positions` defined in Appendix. The function `nat2pmset` maps a natural number to the multiset of prime positions in its factoring[1].

```
nat2pmset 1 = []
nat2pmset n = to_prime_positions n
```

Proposition 1
p is prime if and only if its decomposition in a multiset given by `nat2pmset` is a singleton.

The function `pmset2nat` (relying on `from_pos` and `primes` defined in Appendix) maps back a multiset of positions of primes to the result of the product of the corresponding primes.

```
pmset2nat [] = 1
pmset2nat ns =
  product (map (from_pos_in primes . pred) ns)
```

The operations `nat2pmset` and `pmset2nat` form an isomorphism that, using the combinator language defined in (Tarau, 2009a) (and summarized in the Appendix to ensure that this paper is fully self-contained) provides *any-to-any encodings* between various data types. This gives the Encoder `pmset` for prime encoded multisets as follows:

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

working as follows:

```
*MPrimes> as pmset nat 2010
[1,2,3,19]
*MPrimes> as nat pmset [1,2,3,19]
2010
```

For instance, as the factoring of 2010 is $2*3*5*67$, the list [1,2,3,19] contains the positions of the factors, starting from 1, in the sequence of primes.

3 A bijection between finite multisets and natural numbers

We will now define *ranking/unranking* functions for multisets i.e. bijective mappings to/from natural numbers. While finite multisets and sequences representing finite functions share a common representation $[N]$, multisets are subject to the implicit constraint that their ordering is immaterial. This suggest that a multiset like $[4,4,1,3,3,3]$ could be represented canonically as sequence by first ordering it as $[1,3,3,3,4,4]$ and then computing the differences between consecutive elements i.e. $[x_0,x_1 \ldots x_i,x_{i+1} \ldots] \rightarrow [x_0,x_1-x_0,\ldots x_{i+1}-x_i \ldots]$. 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 `mset2list`:

[1] In contrast to (Tarau, 2009a), we will assume that our mappings are defined on $\mathbb{N}^+ = \mathbb{N} - 0$ rather than $\mathbb{N}$.

```
mset2list = to_diffs . sort
to_diffs xs = zipWith (-) (xs) (0:xs)
```

It is now clear that incremental sums of the numbers in such a sequence return the original set in sorted form, as implemented by `list2mset`:

```
list2mset ns = tail (scanl (+) 0 ns)
```

The isomorphism between finite multisets and finite functions (seen as finite sequences in $\mathbb{N}$) is specified with two bijections `mset2list` and `list2mset`.

```
mset0 :: Encoder [N]
mset0 = Iso mset2list list2mset
```

The resulting isomorphism `mset` can be applied by using its two components `mset2list` and `list2mset` directly.

```
*MPrimes> mset2list [1,3,3,3,4,4]
[1,2,0,0,1,0]
*MPrimes> list2mset [1,2,0,0,1,0]
[1,3,3,3,4,4]
```

Equivalently, following (Tarau, 2009a) (see also summary in Appendix), it can be expressed generically by using the "as" combinator:

```
*MPrimes> as list mset0 [1,3,3,3,4,4]
[1,2,0,0,1,0]
*MPrimes> as mset0 list [1,2,0,0,1,0]
[1,3,3,3,4,4]
```

The combinator "as" derives automatically *any-to-any* encodings, by routing through the appropriate one-to-one transformations (see (Tarau, 2009a) and the `Appendix`). As a result, we obtain "for free" a bijection from *N* to finite multisets of elements of *N* :

```
*MPrimes> as mset0 nat 2009
[0,2,2,3,3,3,3,3]
*MPrimes> as  nat mset0 it
2009
```

We will need one small change to convert this into a mapping on $\mathbb{N}^+$.

```
nat2mset1 n=map succ (as mset0 nat (pred n))
mset2nat1 ns=succ (as nat mset0 (map pred ns))

mset :: Encoder [N]
mset = compose (Iso mset2nat1 nat2mset1) nat
```

The resulting mapping, like `pmset` now works on $\mathbb{N}^+$.

```
*MPrimes> as mset nat 2010
[1,3,3,4,4,4,4,4]
*MPrimes> as nat mset it
2010
*MPrimes> map (as mset nat) [1..7]
[[],[1],[2],[1,1],[3],[1,2],[2,2]]
```

Note that these mappings work in time and space linear in the bitsize of the numbers. On the other hand, as prime number enumeration and factoring are involved in the mapping from numbers to multisets, the encoding described in section 2 is intractable for all but small values.

4 Exploring the analogy between multiset decompositions and factoring

As natural numbers can be uniquely represented as multisets of prime factors and, independently, they can also be represented as a multiset with the Encoder `mset` (described in section 3), the following question arises naturally:

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

4.1 A multiset analog to multiplication

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. We use the combinator `borrow_from` (see Appendix) to express this:

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

Proposition 2
$< N^+, mprod, 1 >$ 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.

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`.

Here are a few examples showing that `mprod` has properties similar to ordinary multiplication and exponentiation:

```
*MPrimes> mprod 41 (mprod 33 38) == mprod (mprod 41 33) 38
True
*MPrimes> mprod 33 46 == mprod 46 33
True
mprod 1 712==712
True
```

Given the associativity of `mprod`, it makes sense to define the product of a list of numbers as

```
mproduct ns = foldl mprod 1 ns
```

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))
```

This brings us back to observe that:

Proposition 3
$< N, pmprod, 1 >$ is a commutative monoid i.e. `pmprod` is defined for all pairs of natural numbers and it is associative, commutative and has 1 as an identity element.

Unsurprisingly, this is the case indeed as one can deduce immediately from the definition of `mprod` that:

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

As obvious as this equivalence is, note that computing `*` is easy, while computing `mprod` involves factoring which is intractable for large values.

Interestingly, `mprod` coincides with `pmprod` i.e. ordinary multiplication for some values. More precisely, the following holds:

Proposition 4
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. 1 shows the self-similar landscape generated by the function `(mprod x y) / (x*y)` for values of x,y in $[1..128]$.

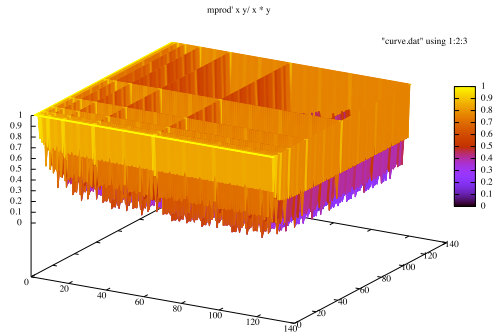

Fig. 1: Ratio between mprod and product

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

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

Let us first observe that ordinary exponent and our emulated variant correlate as follows:

```
*MPrimes> map (λx→mexp 2 x) [1..8]
[2,4,8,16,32,64,128,256]
*MPrimes> map (λx→2^x) [1..8]
[2,4,8,16,32,64,128,256]
```

```
*MPrimes> map (λx→mexp x 2) [1..16]
[1,4,7,16,13,28,31,64,25,52,55,112,61,124,127,256]
*MPrimes> map (λx→x^2) [1..16]
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256]
```

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.

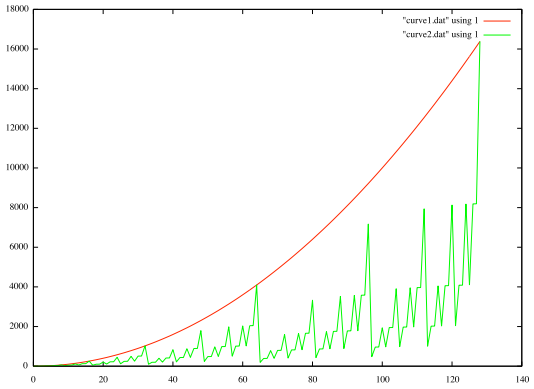

Fig. 2: Square vs. mexp n 2

4.2 Multiset analogues for divisibility, gcd and lcd

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_from mset msetInter nat

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

mdivisible :: N→N→Bool
mdivisible n m = mgcd n m==m

mdiv :: N → N → N
mdiv = borrow_from mset msetDif nat

mexactdiv :: N → N → N
mexactdiv n m | mdivisible n m = mdiv n m
```

and note that properties similar to usual arithmetic operations hold:

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

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

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

4.3 Multiset primes

Definition 1
We say that $p > 1$ is a multiset-prime (or *mprime* shortly), if its decomposition as a multiset is a singleton.

The following holds

Proposition 5
$p > 1$ is a multiset prime if and only if it is not mdivisible by any number in [2..p-1].

This follows immediately by observing that singleton multisets are the first to contain a given number as the multiset [a,b] corresponds to a number strictly larger than the numbers corresponding to multisets [a] and [b].

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

```
is_mprime p | p >1 = 1==length (as mset nat p)

alt_is_mprime p | p>1 =
  []==[n|n←[2..p-1],p `mdivisible` n]

mprimes = filter is_mprime [2..]
```

Trying out `mprimes` gives:

```
*MPrimes> take 10 mprimes
[2,3,5,9,17,33,65,129,257,513]
```

suggesting the following proposition:

Proposition 6
There's an infinite number of *multiset primes* and they are exactly the numbers of the form $2^n + 1$.

The proof follows immediately by observing that the first value of `as mset nat n` that contains k, is $n = 2^k + 1$ and that numbers of that form are exactly the numbers resulting in singleton multisets:

```
*MPrimes> map (as mset nat) [1..9]
[[],[1],[2],[1,1],[3],[1,2],[2,2],[1,1,1],[4]]
        ^^^       ^^^                     ^^^
        2+1       4+1                     8+1
```

We can now implement faster versions of `mprimes` and `is_mprime`:

```
mprimes' = map (λx→2^x+1) [0..]

is_mprime' p | p>1 = p==
  last (takeWhile (λx→x≤p) mprimes')
```

4.4 An analog to the `rad` *function*

Definition 2
n is square-free if each prime on its list of factors occurs exactly once.

The `rad(n)` function (`A007947` in (Sloane, 2006)) is defined as follows:

Definition 3
rad(n) is the largest square-free number that divides n

Clearly, `rad` can be computed by factoring, then trimming multiple occurrences with the `nub` library function and finally by multiplying the resulting primes with `product`.

```
rad n = product (nub (to_primes n))
```

Note that rad can also be computed by trimming multiplicities in a multiset representation of n i.e. after defining respectively

```
pfactors n = nub (as pmset nat n)
mfactors n = nub (as mset nat n)
```

we can define `prad` ≡ `rad` and its multiset equivalent `mrad`:

```
prad n =  as nat pmset (pfactors n)

mrad n =  as nat mset (mfactors n)
```

```
*MPrimes> map rad [2..16]
[2,3,2,5,6,7,2,3,10,11,6,13,14,15,2]
*MPrimes> map prad [2..16]
[2,3,2,5,6,7,2,3,10,11,6,13,14,15,2]
*MPrimes> map mrad [2..16]
[2,3,2,5,6,3,2,9,10,11,6,5,6,3,2]
```

A comparison of the plots of the two functions (Figs. 3 and 4) shows that `rad`'s chaotic behavior corresponds to a more regular, self-similar behavior in the case of `mrad`.

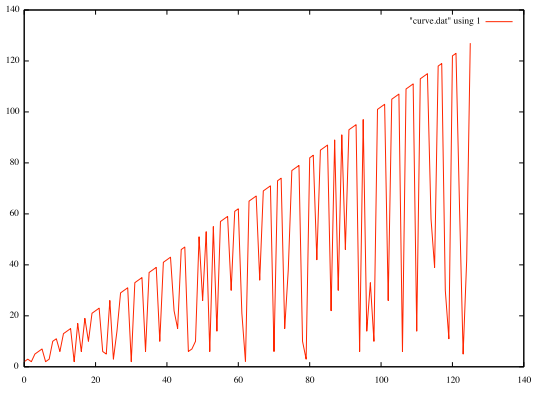

Fig. 3: rad(n) on $[2..2^7 - 1]$

One can further explore if this "emulation" of the `rad` function can bring some light on the well known connections between the `rad` function and the famous *abc conjecture* (Granville, 1998).

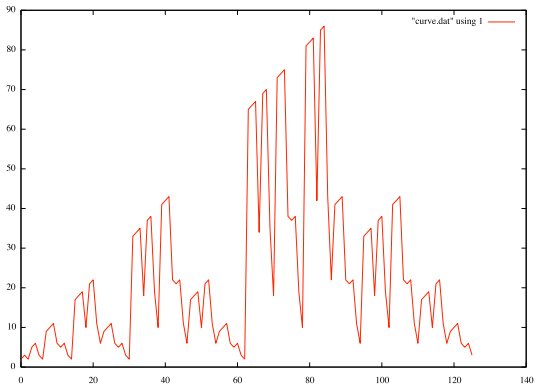

Fig. 4: mrad(n) on $[2..2^7-1]$

4.5 Deriving automorphisms of $\mathbb{N}$

Definition 4
An *automorphism* is an isomorphism for which the source and target are the same.

A nice property of automorphisms is that, given the isomorphisms provided by the data transformation framework (Tarau, 2009a), they propagate from one data type to another. In our case, the multiset representations provided by `pmset` and `mset` induce two automorphisms on $\mathbb{N}$

```
auto_m2p 0 = 0
auto_m2p n = as nat pmset (as mset nat n)

auto_p2m 0 = 0
auto_p2m n = as nat mset (as pmset nat n)
```

working as follows:

```
*MPrimes> map auto_m2p [0..31]
[0,1,2,3,4,5,6,9,8,7,10,15,12,25,18,27,16,11,14,
 21,20,35,30,45,24,49,50,75,36,125,54,81]
*MPrimes> map auto_p2m [0..31]
[0,1,2,3,4,5,6,9,8,7,10,17,12,33,18,11,16,65,14,
 129,20,19,34,257,24,13,66,15,36,513,22,1025]
```

Figs. 5 and 6 show the quickly amplifying reshuffling of the sequence [0..] generated by the functions `auto_m2p` and `auto_p2m` for values in in $[0..2^6-1]$.

5 Pairing/Unpairing

Definition 5
A *pairing* function is an isomorphism $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Its inverse is called *unpairing*.

Definition 6
We call an unpairing function *well-founded* if, with the exception of 0 and 1, it splits any number z into strictly smaller ones.

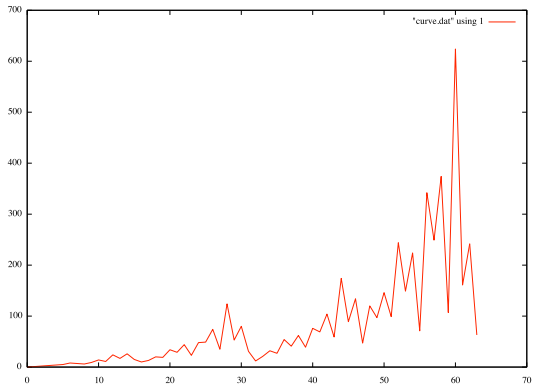

Fig. 5: The automorphism auto_m2p

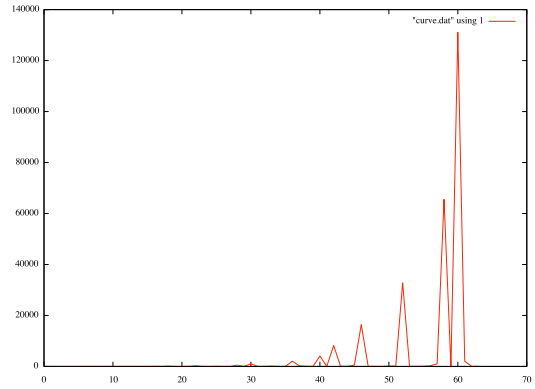

Fig. 6: The automorphism auto_p2m

5.1 A Bitwise Pairing/Unpairing Function

We will now introduce an unusually simple pairing function (also mentioned in (Pigeon, 2001), p.142 and similar to Misra's *zip* function (Misra, 1994) in the *powerlist* algebra).

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. Pairs of natural numbers will be hosted as a special type, `NxN`:

```
type NxN = (N,N)
```

we define:

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

bitunpair :: N→NxN
bitunpair n = (f xs,f ys) where
```

```
  (xs,ys) = partition even (nat2set n)
  f = set2nat . (map (`div` 2))
```

It is easy to see that the following holds

Proposition 7
The unpairing function `bitunpair` is well-founded.

We can derive the following Encoder:

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

working as follows:

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

Given that unpairing functions are bijections from $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ they will progressively cover all points having natural number coordinates in their range in the plane. Figure 7 shows the curves generated by `bitunpair`.

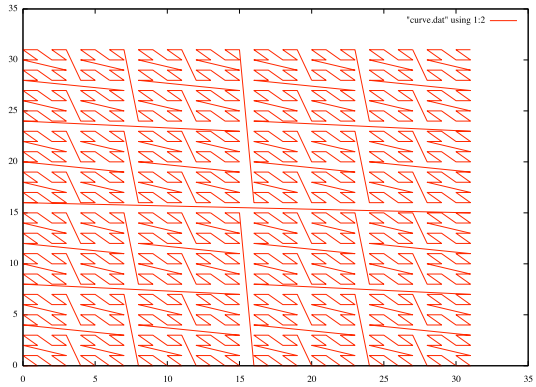

Fig. 7: 2D curve connecting values of `bitunpair n` for $n \in [0..2^{10}-1]$

5.2 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) = list2set [x,y]
unord_pair2pair [a,b] = (x,y) where
  [x,y]=set2list [a,b]

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

We can derive the Encoder:

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

working as follows:

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

5.3 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]=list2mset [x,y]
mset_unpair2pair (a,b) = (x,y) where
  [x,y]=mset2list [a,b]

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

We can derive the following Encoder:

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

working as follows:

```
*MPrimes> as mset2 nat 2008
(60,86)
*MPrimes> as nat mset2 it
2008
*MPrimes> as mset2 nat 1092
(42,42)
```

The following holds:

Proposition 8
The unpairing functions and `mset_unpair` is well-founded.

Figure 8 shows the curve generated by `mset_unpair` covering the lattice of points in its range.

5.4 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 9
Given the function definitions:

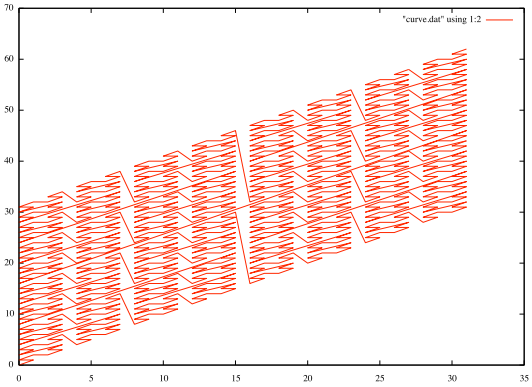

Fig. 8: 2D curve connecting values of `mset_unpair n` for $n \in [0..2^{10}-1]$

```
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:

$$bitlift \equiv bitlift' \tag{5}$$

$$bitclip \equiv bitclip' \tag{6}$$

$$bitclip \circ bitlift \equiv id \tag{7}$$

$$bitpair(0,n) \equiv 2 * bitpair(n,0) \tag{8}$$

$$bitpair(0,n) \equiv 2 * (bitlift\ n) \tag{9}$$

$$bitpair(n,n) \equiv 3 * (bitlift\ n) \tag{10}$$

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

$$bitpair(2^n+1,0) \equiv (2^n)^2+1 \tag{12}$$

$$bitpair(2^{2^n}+1,0) \equiv 2^{2^{n+1}}+1 \tag{13}$$

$$bitpair' \equiv bitpair \equiv xbitpair \equiv obitpair \tag{14}$$

$$bitpair(x,y) \equiv (bitlift\ x) + 2 * (bitlift\ y) \tag{15}$$

$$pair\_product \equiv * \tag{16}$$

Proposition 10
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:

$$bitpair \equiv bitpair'' \equiv bitpair''' \tag{17}$$

$$mset\_pair \equiv mset\_pair' \equiv mset\_pair'' \tag{18}$$

$$unord\_pair \equiv unord\_pair' \equiv unord\_pair'' \tag{19}$$

6 Primes and Pairing Functions

Products of two prime numbers have the interesting property that they provide the special case when no information is lost by multiplication, in the sense of (Pippenger, 2005). 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 with the combinators `ppair` and `punpair`.

```
ppair :: ((N, N) → N) → (N, N) → N
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
      ps = primes

punpair :: (N → (N, N)) → N → (N, N)
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:

```
*MPrimes> ppair bitpair (11,17)
269
*MPrimes> 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.

Fig. 9 shows the self similar plot connecting points with prime coordinates generated by `(punpair bitunpair)`.

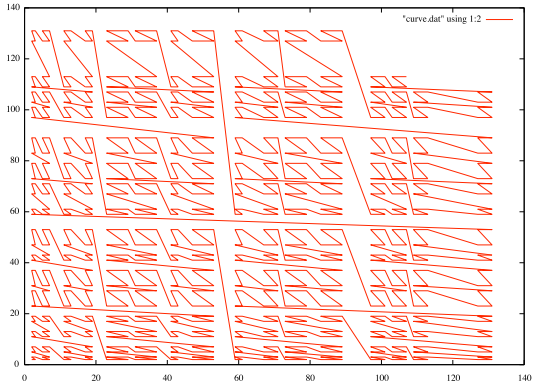

Fig. 9: Points with prime coordinates connected with an unpairing function

Figs. 10 and 11 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.

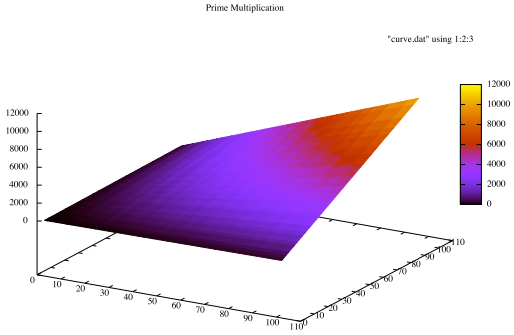

Fig. 10: Lossless multiplication of primes

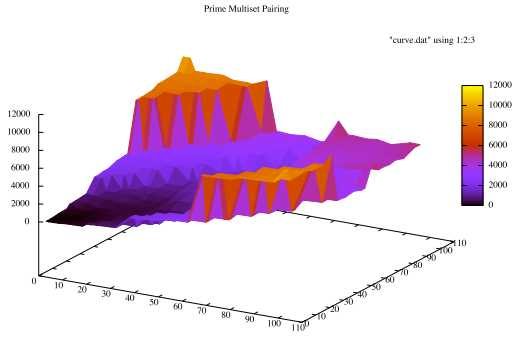

Fig. 11: Lossless multiset pairing of primes

Given an *unpairing function* `u` 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 holds after applying the unpairing function `u` to `n`. More interestingly, one can look at subsets for which this property holds recursively.

Definition 7
Let u be a well-founded unpairing function. A prime number is called u-hyperprime if its recursive decomposition with u contains only primes (besides 0 and 1).

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
```

where the function `uparts` extracts recursively all the unique parts obtained by splitting a number with the unpairing function `u`. The function `hyper_primes` generates the stream of hyperprimes induced by an unpairing function:

```
*MPrimes> 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]
```

Experiments suggest the following conjecture:

Conjecture 1
The set generated by (hyper_primes bitunpair) is infinite.

with possible generalization to sets induced by arbitrary well-founded unpairing functions.

Figure 12 shows the complete unpairing graph for two hyperprimes obtained with `bitunpair`.

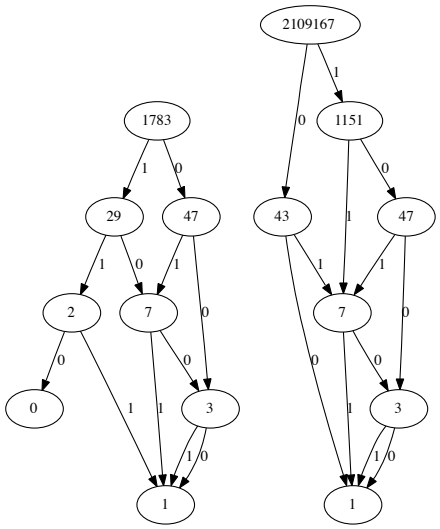

Fig. 12: `bitunpair` hyperprimes: *1783* and *2109167*

7 Hyper-primes, multiset 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` i.e. by studying subsets of primes invariant under applications of `mset_unpair`.

7.1 Emulating hyperprimes with multiset primes

Definition 8
Let u be a well-founded unpairing function. A multiset prime is called a u-hyper-mprime if its recursive decomposition with u contains only multiset primes (besides 0 and 1).

First we define `hyper_mprimes` using the faster recognizer `is_mprime'`

```
hyper_mprimes u =
  [n|n←mprimes', all_are_mprimes (uparts u n)] where
     all_are_mprimes ns = and (map is_mprime' ns)
```

Surprisingly, we obtain:

```
*MPrimes> take 15 (hyper_mprimes mset_unpair)
[2,3,5,9,17,33,65,129,257,513,1025,2049,4097,8193,16385]
```

suggesting the proposition:

Proposition 11
All multiset primes are hyper_mprimes.

The proof is immediate, noting that if $n = 2^k + 1$ and $(x,y) = mset\_unpair\ n$, then x and y are either `1` or of the form $2^k + 1$, given that the bitstring representation of $2^k + 1$ is $100\ldots001$ and the unpairing function either keeps both `1`s at the two ends or splits them.

Figure 13 shows the unpairing graph containing the hyper-mprimes obtained with `mset_unpair`.

7.2 Hyper-primes and Fermat primes

Let us look at the "real thing" now, hyperprimes derived using `mset_unpair` and the primality tester `is_prime`.

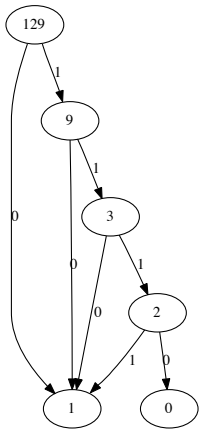

Fig. 13: The mhyper_prime tree rooted in `129`

```
*MPrimes> 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 9
A Fermat number is a number of the form $2^{2^n}+1$.

Given that multiset primes are of the form 2^n+1 we have:

Proposition 12
All Fermat numbers are multiset primes.

Definition 10
A Fermat-prime (Riesel, 1963; Keller, 1983) is a prime of the form $2^{2^n}+1$ with $n \geq 0$.

Obviously, given that that mset hyperprimes are of the form 2^n+1 the following holds:

Proposition 13
All Fermat primes are `mset_unpair` induced *hyper-mprimes*.

Assuming the widely believed conjecture that the only Fermat primes are [3,5,17,257,65537] as these 5 primes are indeed on our list of `mset_unpair` hyperprimes, the following would also hold:

Conjecture 2
All Fermat primes are `mset_unpair` induced *hyperprimes*.

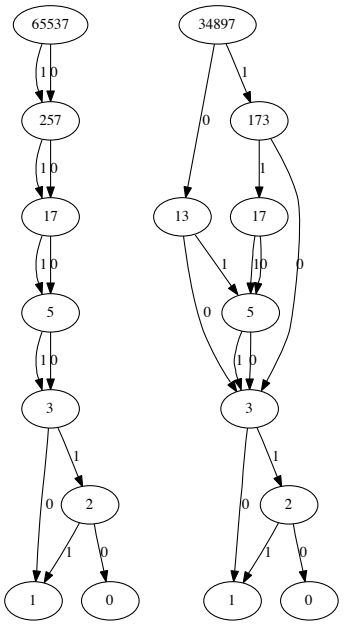

Fig. 14: `mset_unpair` hyperprimes: *A Fermat prime* and *A Non-Fermat prime*

Fig. 14 shows a hyperprime that is also a Fermat prime and a hyperprime that is not a Fermat prime.

In the (unlikely) event that there would be other Fermat primes, we will now state:

Proposition 14
If there are Fermat primes other than [3, 5, 17, 257, 65537] then there are Fermat primes that are not `mset_unpair` hyperprimes.

To prove Prop. 14 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 hyperprimes.

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

$$mset\_pair\ (2^{2^n}+1, 2^{2^n}+1) \equiv 2^{2^{n+1}}+1 \tag{20}$$

Indeed, from the identity 18 we obtain

$$mset\_pair(a,a) \equiv bitpair(a,0) \tag{21}$$

and then observe that from 12 it follows that

$$bitpair(2^{2^n}+1,0) \equiv 2^{2^{n+1}}+1 \tag{22}$$

We can now prove Prop. 14. If $2^{2^{n+1}}+1$ is a Fermat prime that is also a hyperprime, then $2^{2^n}+1$ would also be a Fermat prime that is hyperprime. This would form a descending sequence of consecutive Fermat primes - a contradiction, given that for instance, $2^{32}+1 = 641 * 6,700,417$ is not prime[2].

Discussion We have seen that, assuming Conj. 14, the Fermat primes are included in the intersection of the emulated and "real" primes, but also that one should not look for Fermat primes among Fermat numbers larger than `65537` that would lead to primes when decomposed with `mset_unpair`.

We can conclude that our "weak" multiset based prime emulation has been useful in detecting some interesting classes and properties of primes.

More generally, this suggests that various other experiments using the bijective mappings introduced in (Tarau, 2009c; Tarau, 2009a) can bring interesting insights to open problems in computational mathematics.

8 Related work

There's a huge amount of work on prime numbers and related aspects of multiplicative and additive number theory. Studies of prime number distribution and various probabilistic and information theoretic aspects also abound.

While we have not made use of any significantly advanced facts about prime numbers, the following references circumscribe the main topics to which our experiments can be connected (Pippenger, 2005; Crandall & Pomerance, 2005; Young, 1998; Riesel, 1985; Keller, 1983; Cgielski *et al.*, 2007).

Pairing functions are have been used in work on decision problems as early as (Robinson, 1950; Robinson, 1968). A typical use in the foundations of mathematics is (Cégielski & Richard, 2001). An extensive study of various pairing functions and their computational properties is presented in (Rosenberg, 2003). Natural Number encodings of various set-theoretic constructs have triggered the interest of researchers in fields ranging from Axiomatic Set Theory and Foundations of Logic to Complexity Theory and Combinatorics (Takahashi, 1976; Kaye & Wong, 2007; Abian & Lamacchia, 1978; Avigad, 1997; Kirby, 2007). In combinatorics they show up as *Ranking* functions that can be traced back to Gödel numberings (Gödel, 1931; Hartmanis & Baker, 1974) associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms for various data types (Knuth, 2006; Ruskey & Proskurowski, 1990).

The closest reference on encapsulating bijections as a Haskell data type is (Alimarine *et al.*, 2005) and Conal Elliott's composable bijections module (Conal Elliott, n.d.), where, in a more complex setting, Arrows (Hughes, n.d.) are used as the underlying abstractions. While our `Iso` data type is similar to the *Bij* data type in (Conal Elliott, n.d.) and BiArrow concept of (Alimarine *et al.*, 2005), the techniques for using such isomorphisms as building blocks of an embedded composition language centered around encodings as natural numbers, in particular our multiset and prime number encodings, are new.

[2] As pointed out by L. Euler in 1732.

This paper relies on the compositional and extensible data transformation framework (summarized in the Appendix) that connects most of the fundamental data types used in computer science with a *groupoid of isomorphisms*. A large (100+ pages) unpublished draft (Tarau, 2009c), provides encodings between more than 60 different data types. The basic idea of the framework is described in (Tarau, 2009d) and some of its applications to computational mathematics in (Tarau, 2009a). A compact Prolog implementation of the framework with focus on mapping between complex data structures is described in (Tarau, 2009b).

Some other techniques 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.

9 Conclusion

We have explored some computational analogies between multisets, natural number encodings, pairing functions and prime numbers in a framework for experimental mathematics implemented as a literate Haskell program.

This has resulted in some interesting conjectures on prime generating formulae and a methodology for emulating more difficult number theoretic phenomena through simpler isomorphic representations. In a way this parallels abstract interpretation by using a simpler domain to approximate interesting properties in a more complex one.

Another interesting concept, *hyperprimes* has emerged that might be useful to number theorists working on related problems involving Fermat primes.

Future work will focus on finding a matching additive operation for the multiset induced commutative monoid and an exploration of some possible practical applications to representing arbitrary length integer operations using pairing functions and multiset representations.

References

Abian, Alexander, & Lamacchia, Samuel. (1978). On the consistency and independence of some set-theoretical constructs. *Notre dame journal of formal logic*, **X1X**(1), 155–158.

Alimarine, Artem, Smetsers, Sjaak, van Weelden, Arjen, van Eekelen, Marko, & Plasmeijer, Rinus. (2005). There and back again: arrows for invertible programming. *Pages 86–97 of: Haskell '05: Proceedings of the 2005 acm sigplan workshop on haskell*. New York, NY, USA: ACM Press.

Avigad, Jeremy. 1997 (Jan.). The Combinatorics of Propositional Provability. *Asl winter meeting*.

Borwein, J. M., Bradley, D. M., & Crandall, R. E. (2000). Computational strategies for the Riemann zeta function. *J. comput. appl. math.*, **121**(1–2), 247–296. Numerical analysis in the 20th century, Vol. I, Approximation.

Cégielski, Patrick, & Richard, Denis. (2001). Decidability of the theory of the natural integers with the cantor pairing function and the successor. *Theor. comput. sci.*, **257**(1-2), 51–77.

Chaitin, G.J. (2004). Thoughts on the riemann hypothesis. *Math. intelligencer*, **26**(1), 4–7.

Conal Elliott. *Module: Data.Bijections*. Haskell source code library at: http://haskell.org/haskellwiki/TypeCompose.

Crandall, R., & Pomerance, C. (2005). *Prime numbers–a computational approach*. Second edn. New York: Springer.

Cgielski, Patrick, Richard, Denis, & Vsemirnov, Maxim. (2007). On the additive theory of prime numbers. *Fundam. inform.*, **81**(1-3), 83–96.

Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für mathematik und physik*, **38**, 173–198.

Granville, A. (1998). ABC allows us to count squarefrees. *International mathematics research notices*, **1998**(19), 991.

Hartmanis, Juris, & Baker, Theodore P. (1974). On simple goedel numberings and translations. *Pages 301–316 of:* Loeckx, Jacques (ed), *Icalp*. Lecture Notes in Computer Science, vol. 14. Springer.

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

Kaye, Richard, & Wong, Tin Lock. (2007). On Interpretations of Arithmetic and Set Theory. *Notre dame j. formal logic volume*, **48**(4), 497–510.

Keller, W. (1983). Factors of Fermat numbers and large primes of the form $k \cdot 2^n + 1$. *Math. comp.*, **41**, 661–673.

Kirby, Laurence. (2007). Addition and multiplication of sets. *Math. log. q.*, **53**(1), 52–65.

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

Miller, Gary L. (1975). Riemann's hypothesis and tests for primality. *Pages 234–239 of: Stoc*. ACM.

Misra, Jayadev. (1994). Powerlist: a structure for parallel recursion. *Acm transactions on programming languages and systems*, **16**, 1737–1767.

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

Pippenger, Nicholas. (2005). The average amount of information lost in multiplication. *Ieee transactions on information theory*, **51**(2), 684–687.

Riemann, Bernhard. (1859). Ueber die anzahl der primzahlen unter einer gegebenen grösse. *Monatsberichte der berliner akademie*, Nov.

Riesel, H. (1963). A factor of the Fermat number F_{19}. *Math. comp.*, **17**, 458.

Riesel, H. (1985). *Prime numbers and computer methods for factorization*. Progress in Mathematics, vol. 57.

Robinson, Julia. (1950). General recursive functions. *Proceedings of the american mathematical society*, **1**(6), 703–718.

Robinson, Julia. (1968). Finite generation of recursively enumerable sets. *Proceedings of the american mathematical society*, **19**(6), 1480–1486.

Rosenberg, Arnold L. (2003). Efficient pairing functions - and why you should care. *International journal of foundations of computer science*, **14**(1), 3–17.

Ruskey, Frank, & Proskurowski, Andrzej. (1990). Generating binary trees by transpositions. *J. algorithms*, **11**, 68–84.

Singh, D., Ibrahim, A. M., Yohanna, T., & Singh, J. N. (2007). An overview of the applications of multisets. *Novi sad j. math*, **52**(2), 73–92.

Sloane, N. J. A. (2006). The On-Line Encyclopedia of Integer Sequences. published electronically at www.research.att.com/∼njas/sequences.

Takahashi, Moto-o. (1976). A Foundation of Finite Mathematics. *Publ. res. inst. math. sci.*, **12**(3), 577–708.

Tarau, Paul. (2009a). A Groupoid of Isomorphic Data Transformations. et al., J. Carette (ed), *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009* . Grand Bend, Canada: Springer, LNAI 5625.

Tarau, Paul. (2009b). An Embedded Declarative Data Transformation Language. *Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009*. Coimbra, Portugal: ACM.

Tarau, Paul. 2009c (Jan.). *Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell*. http://arXiv.org/abs/0808.2953, unpublished draft, 104 pages.

Tarau, Paul. (2009d). Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. *Pages 1898–1903 of: Proceedings of ACM SAC'09*. Honolulu, Hawaii: ACM.

van de Lune, J., te Riele, H. J. J., & Winter, D. T. (1986). On the zeros of the Riemann zeta function in the critical strip, iv. *Math. comp.*, **46**(174), 667–681.

Young, J. (1998). Large primes and Fermat factors. *Math. comp.*, **67**(244), 1735–1738.

Appendix

An Embedded Data Transformation Language

We will describe briefly the embedded data transformation language used in this paper 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.

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.

```
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 these isomorphisms.

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.*

Choosing a Root: Finite Sequences of Natural Numbers

To avoid defining $\frac{n(n-1)}{2}$ isomorphisms between n objects, we choose a *Root* object to/from which we will actually implement isomorphisms. We will extend our embedded combinator language using the 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 and 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 $\mathbb{N}$. We will represent them as lists i.e. their Haskell type is `[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*.

```
as :: Encoder a → Encoder b → b → a
as that this x = g x where
   Iso _ g = compose that (invert this)
```

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 binary operations from an Encoder to another, `borrow_from`, can be defined as follows:

```
borrow_from :: Encoder a → (a → a → a) →
               Encoder b → (b → b → b)
borrow_from lender op borrower x y = as borrower lender
   (op (as lender borrower x) (as lender borrower y))
```

Given that `[N]` has been chosen as the root, we will define our finite function data type *list* simply as the identity isomorphism on sequences in `[N]`.

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

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

to_prime_positions n=
  map (succ . (to_pos_in (h:ps))) qs where
    (h:ps)=genericTake n primes
    qs=to_factors n h ps

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

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

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))
```

Ranking/unranking of sets and finite sequences

First, an isomorphism between sets and finite sequences (the Root of the groupoid of isomorphisms) is defined, resulting in the Encoder `set`:

```
set2list xs = shift_tail pred (mset2list xs) where
  shift_tail _ [] = []
  shift_tail f (x:xs) = x:(map f xs)

list2set = (map pred) . list2mset . (map succ)

set :: Encoder [N]
set = Iso set2list list2set
```

We can *rank/unrank* a set represented as a list of distinct natural numbers 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 = sum (map (2^) ns)
```

The resulting Encoder is:

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

Conversions to/from a given base

The following two functions (given here to ensure that the paper is a self contained Haskell literate program) convert a number to/from a given base. Numbers in a given base are represented as lists of coefficients of the respective polynomials, in ascending order.

```
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)
```