On primes and pairing/unpairing functions

Paul Tarau

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

Abstract. The paper explores in the form of a *literate Haskell program* interactions between prime numbers and *pairing/unpairing functions* deriving from the simple intuition that, like the product of two primes, unpairing operations are also reversible. An interesting concept, *hyper-primes* emerges, which turns out to be a superset of Fermat primes.
Keywords: *pairing/unpairing functions, multiset encodings, prime number sequences, Fermat primes, computational mathematics*

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 [1] and the large number of still unsolved problems involving prime numbers [2] 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 [3] connecting primality and complex analysis, all evolving around the still unsolved *Riemann Hypothesis* [4–6]. 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 [7] as their most interesting properties quickly take us beyond the edges of computability. 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.

This paper will explore in the form of a *literate Haskell program* some unusual interactions between prime numbers and *pairing/unpairing functions* deriving from the simple intuition that, like the product of two primes, unpairing operations are also reversible. An interesting concept, *hyper-primes* emerges which turns out to be a superset of Fermat primes.

The paper is organized as follows: section 2 describes pairing/unpairing functions and studies some of their algebraic properties that are used in section 3 to introduce *hyper-primes* and study their connection to Fermat primes and related

conjectures. Section 5 overviews some related work and section 6 concludes the paper.

To make the paper self-contained as a literate Haskell program, we will provide in the `Appendix` the relevant code borrowed from [8] to be able use the embedded data transformation language specified in [8] as a set of operations on a groupoid of isomorphisms connecting various data types.

2 Pairing/Unpairing

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

2.1 A Bitwise Pairing/Unpairing Function

We will now introduce an unusually simple pairing function (also mentioned in [9], 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. Pairs of natural numbers will be hosted as a special type, `Nat2`:

```
type Nat2 = (Nat,Nat)
```

we define:

```
bitpair ::  Nat2 → Nat
bitpair (i,j) =
  set2nat ((evens i) ++ (odds j)) where
    evens x = map (2*) (nat2set x)
    odds y = map succ (evens y)

bitunpair :: Nat→Nat2
bitunpair n = (f xs,f ys) where
  (xs,ys) = partition even (nat2set n)
  f = set2nat . (map (`div` 2))
```

We can derive the following Encoder:

```
nat2 :: Encoder Nat2
nat2 = compose (Iso bitpair bitunpair) nat
```

working as follows:

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

Figure 1 shows the directed graph describing recursive application of `bitunpair`.

Given that unpairing functions are bijections from Nat to $Nat \times Nat$ they will progressively cover all points having natural number coordinates in their range in the plane. Figure 2 shows the curves generated by `bitunpair`.

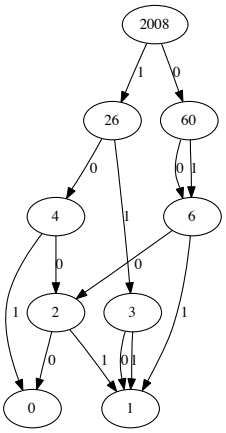


Fig. 1: Graph obtained by recursive application of `bitunpair` for 2008

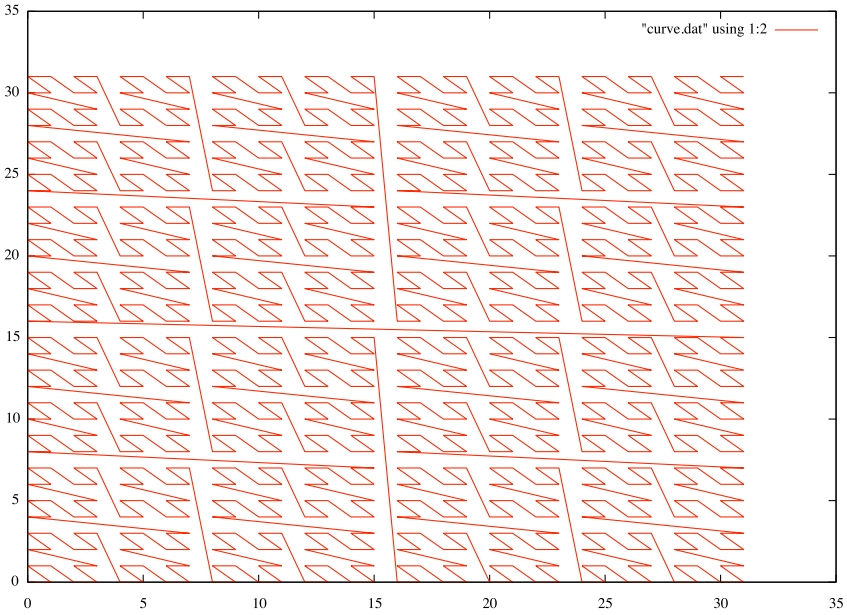


Fig. 2: 2D curve connecting values of `bitunpair n` for $n \in [0..2^{10} - 1]$

2.2 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) = 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 Encoder:

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

working as follows:

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

2.3 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]=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 Nat2
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 3 shows the curve generated by `mset_unpair` covering the lattice of points in its range.

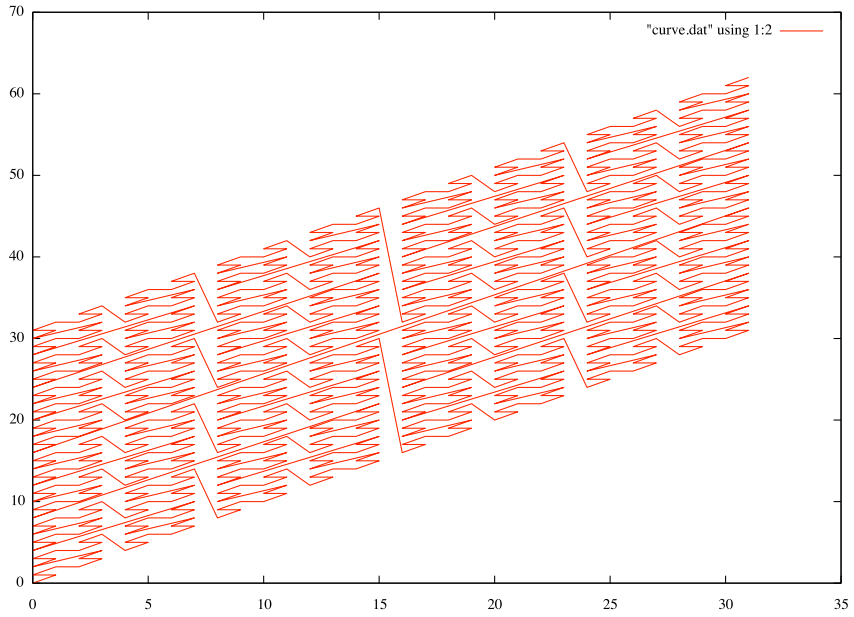


Fig. 3: 2D curve connecting values of `mset_unpair n` for $n \in [0..2^{10}-1]$

2.4 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 1 *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:

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

$$bitclip \equiv bitclip' \tag{2}$$

$$bitclip \circ bitlift \equiv id \tag{3}$$

$$bitpair(0, n) \equiv 2 * bitpair(n, 0) \tag{4}$$

$$bitpair(0, n) \equiv 2 * (bitlift\ n) \tag{5}$$

$$bitpair(n, n) \equiv 3 * (bitlift\ n) \tag{6}$$

$$bitpair(2^n, 0) \equiv (2^n)^2 \tag{7}$$

$$bitpair(2^{2^n} + 1, 0) \equiv 2^{2^{n+1}} + 1 \tag{8}$$

$$bitpair' \equiv bitpair \equiv xbitpair \equiv obitpair \tag{9}$$

$$bitpair(x, y) \equiv (bitlift\ x) + 2 * (bitlift\ y) \tag{10}$$

$$pair\_product \equiv * \tag{11}$$

Proposition 2 *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:

$$bitpair \equiv bitpair'' \equiv bitpair''' \tag{12}$$

$$mset\_pair \equiv mset\_pair' \equiv mset\_pair'' \tag{13}$$

$$unord\_pair \equiv unord\_pair' \equiv unord\_pair'' \tag{14}$$

3 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 [10]. 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
    ps = primes

to_pos_in xs x = fromIntegral i where Just i=elemIndex x xs
from_pos_in xs n = xs !! (fromIntegral 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:

```
*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. 7 (given in Appendix) and 4 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.

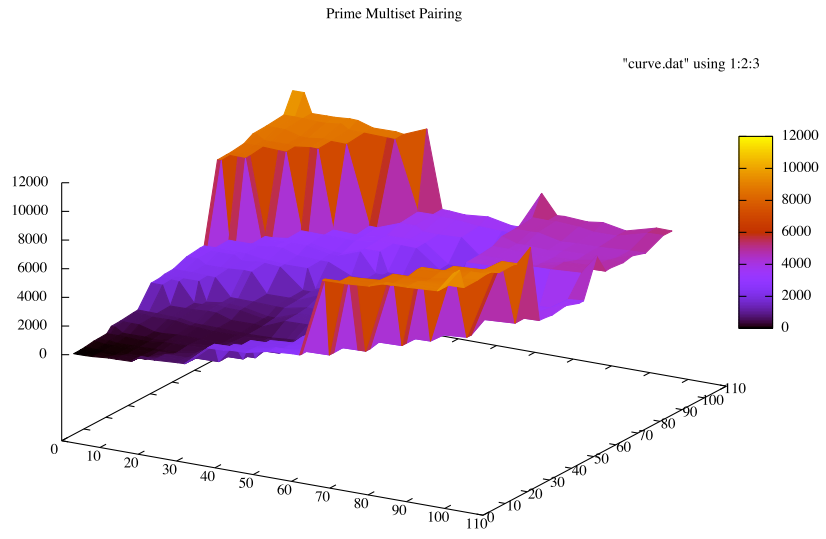


Fig. 4: Lossless multiset pairing of primes

Given an *unpairing function* $u : Nat \to Nat \times Nat$ and a predicate `p(n)` over the set of natural numbers, it makes sense to investigate subsets of Nat 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.

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]
```

This leads to the following conjecture:

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

Figure 5 shows the complete unpairing graph for two hyper-primes obtained with `bitunpair`.

4 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 1 *A Fermat-prime is [11, 12] a prime of the form* $2^{2^n}+1$ *with* $n > 0$.

Fig. 6 shows a hyper-prime that is also a Fermat prime and a hyper-prime that is not a Fermat prime.

We can now state that:

Conjecture 2 *All Fermat primes are* `mset_unpair` *induced hyper-primes.*

We can observe that this would follow from 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.

In the (unlikely) event of the alternative, we will now state:

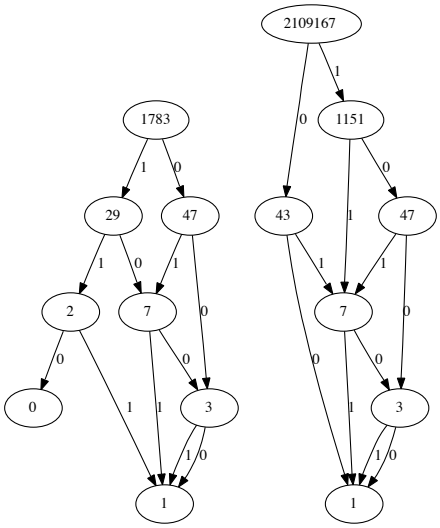


Fig. 5: `bitunpair` hyper-primes: *1783* and *2109167*

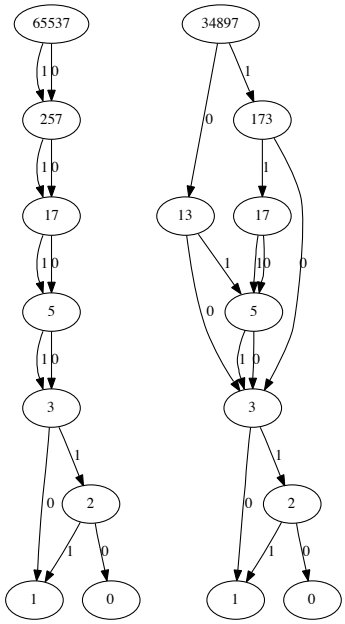


Fig. 6: `mset_unpair` hyper-primes: *A Fermat prime* and *A Non-Fermat prime*

Proposition 3 *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. 3 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

$$mset\_pair\ (2^{2^n}+1, 2^{2^n}+1) \equiv 2^{2^{n+1}}+1 \tag{15}$$

Indeed, from the identity 13 we obtain

$$mset\_pair(a,a) \equiv bitpair(a,0) \tag{16}$$

and then observe that from 8 it follows that

$$bitpair(2^{2^n}+1, 0) \equiv 2^{2^{n+1}}+1 \tag{17}$$

We can now prove Prop. 3. 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 for instance, $2^{32}+1 = 641 * 6,700,417$ is not prime[1].

5 Related work

The paper relies on the compositional and extensible data transformation framework connecting most of the fundamental data types used in computer science with a *groupoid of isomorphisms* described in [13].

Pairing functions have been used in work on decision problems as early as [14, 15]. A typical use in the foundations of mathematics is [16]. An extensive study of various pairing functions and their computational properties is presented in [17].

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 [10, 2, 18, 19, 12, 1].

6 Conclusion

We have explored some computational analogies between pairing functions and prime numbers in a framework for experimental mathematics implemented as a literate Haskell program. An interesting concept - hyper-primes has emerged that might be useful to number theorists working on related problems involving Fermat primes.

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

References

1. Cgielski, P., Richard, D., Vsemirnov, M.: On the additive theory of prime numbers. Fundam. Inform. **81**(1-3) (2007) 83–96
2. Crandall, R., Pomerance, C.: Prime Numbers–a Computational Approach. Second edn. Springer, New York (2005)
3. Riemann, B.: Ueber die anzahl der primzahlen unter einer gegebenen grösse. Monatsberichte der Berliner Akademie (November 1859)
4. van de Lune, J., te Riele, H.J.J., Winter, D.T.: On the zeros of the Riemann zeta function in the critical strip, iv. Math. Comp. **46**(174) (1986) 667–681
5. Miller, G.L.: Riemann's hypothesis and tests for primality. In: STOC, ACM (1975) 234–239
6. Chaitin, G.: Thoughts on the riemann hypothesis. Math. Intelligencer **26**(1) (2004) 4–7
7. Borwein, J.M., Bradley, D.M., Crandall, R.E.: Computational strategies for the Riemann zeta function. J. Comput. Appl. Math. **121**(1–2) (2000) 247–296 Numerical analysis in the 20th century, Vol. I, Approximation.
8. Tarau, P.: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In: Proceedings of ACM SAC'09, Honolulu, Hawaii, ACM (March 2009) 1898–1903
9. Pigeon, S.: Contributions à la compression de données. Ph.d. thesis, Université de Montréal, Montréal (2001)
10. Pippenger, N.: The average amount of information lost in multiplication. IEEE Transactions on Information Theory **51**(2) (2005) 684–687
11. Riesel, H.: A factor of the Fermat number F_{19}. Math. Comp. **17** (1963) 458
12. Keller, W.: Factors of Fermat numbers and large primes of the form $k \cdot 2^n + 1$. Math. Comp. **41** (1983) 661–673
13. Tarau, P.: Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell (January 2009) http://arXiv.org/abs/0808.2953, unpublished draft, 104 pages.
14. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
15. Robinson, J.: Finite generation of recursively enumerable sets. Proceedings of the American Mathematical Society **19**(6) (dec 1968) 1480–1486
16. 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
17. Rosenberg, A.L.: Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science **14**(1) (2003) 3–17
18. Young, J.: Large primes and Fermat factors. Math. Comp. **67**(244) (1998) 1735–1738
19. Riesel, H.: Prime Numbers and Computer Methods for Factorization. Volume 57 of Progress in Mathematics., Boston, MA (1985) Current edition is [**?**].

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. 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 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 $n(n-1)/2$ 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 Nat, say $[0..n]$, to Nat. We will represent them as lists i.e. their Haskell type is $[Nat]$.

```
type Nat = Integer
type Root = [Nat]
```

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

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

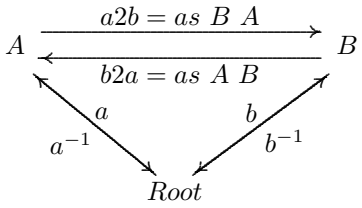
together with the combinators *with* and *as* providing an *embedded transformation language* for routing isomorphisms through two *Encoders*.

```
with :: Encoder a→Encoder b→Iso a b
with this that = compose this (invert that)

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

The combinator `with` turns two Encoders into an arbitrary isomorphism, i.e. acts as a connection hub between their domains. The combinator `as` adds a more convenient syntax such that converters between A and B can be designed as:

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



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 other op this x y = borrow2 (with other this) op x y
```

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

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

Ranking/unranking of sets, multisets 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`:

```
set2fun xs = shift_tail pred (mset2fun xs) where
  shift_tail _ [] = []
  shift_tail f (x:xs) = x:(map f xs)

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

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

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 Nat
nat = compose nat_set set
```

We obtain ranking/unranking function for multisets, indirectly, through the Encoder `mset`:

```
mset2fun = sort (zipWith (-) (xs) (0:xs))

fun2mset ns = tail (scanl (+) 0 ns)

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

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`.

Conversions to/from a given base

The following two functions 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)
```

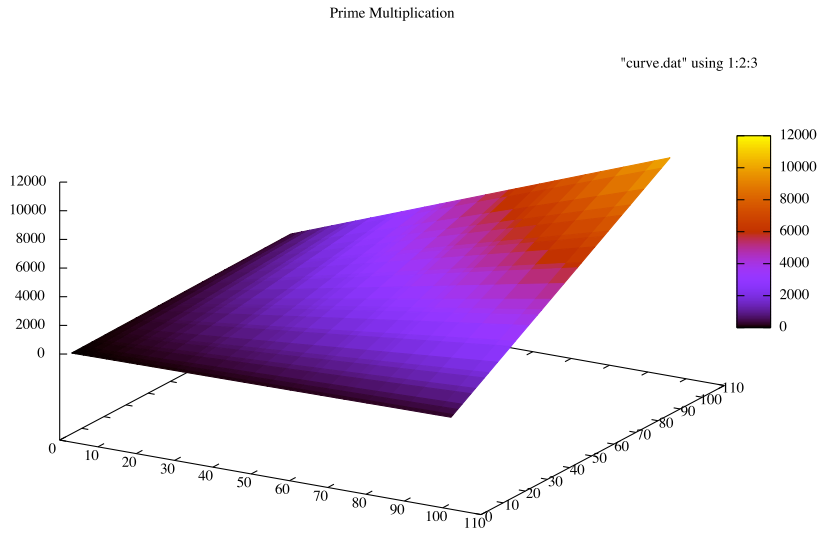


Fig. 7: Lossless multiplication of primes