Emulating Primality: Experimental Combinatorics in Haskell

Paul Tarau

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

Abstract. We emulate interesting properties of prime numbers through a groupoid of isomorphisms connecting them to computationally simpler representations involving multisets. The paper is organized as a self-contained literate Haskell program inviting the reader to explore its content independently. Interactive use is facilitated by an embedded combinator language providing any-to-any isomorphisms between heterogenous types.

Keywords: *multiset encodings and prime numbers, ranking/unranking bijections, experimental combinatorics, computational mathematics*

1 Introduction

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 well 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 [1] and it suggests exploring connections between combinatorial properties and operations on multisets and multiplicative number theory.

With such methodological hints in mind, this paper will explore in the form of a literate Haskell program some unusual analogies involving prime numbers, that have emerged from our effort to provide a compositional and extensible data transformation framework connecting most of the fundamental data types used in computer science with a *groupoid of isomorphisms* [2].

The paper uses a pure subset of Haskell as a formal executable specification language. Given that it faithfully embodies typed λ-calculus we hope the reader will forgive our assumption that it is a safe replacement for conventional mathematical notation. While our effort is sometimes driven by basic category theoretical concepts, we have limited unfamiliar terminology to a bare minimum and provided self-contained intuitions through an extensive set of examples and executable specifications.

The paper is organized as follows: section 2 introduces an isomorphism between finite multisets and sequences which is then extended to sets in subsection 2.1 and natural numbers in subsection 2.2. Section 3 connects multiset encodings and primes. Section 4 explores the analogy between multiset decompositions and factoring and describes a multiplicative monoid structure on multisets that "emulates" interesting properties of its natural number counterpart. Section 5 overviews some related work and section 6 concludes the paper by pointing out its main contributions.

In [3] an embedded data transformation language is introduced, specified as a set of operations on a groupoid of isomorphisms connecting various data types. To make this paper self-contained as a literate Haskell program, we will provide in the `Appendix` the relevant code borrowed from [3].

2 An Isomorphism between Finite Multisets and Finite Functions

Multisets [4] 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 (seen as finite sequences of natural numbers is specified with two bijections `mset2fun` and `fun2mset`.

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

While finite multisets and sequences representing finite functions share a common representation $[Nat]$, 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. $[x_0 \ldots x_i, x_{i+1} \ldots] \rightarrow [x_0 \ldots x_{i+1} - x_i \ldots]$. 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
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 `fun2mset`:

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

2.1 Extending the isomorphism to finite sets

While finite sets and sequences share a common representation $[Nat]$, sets are subject to the implicit constraints that all their elements are distinct and ordering 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 = shift_tail pred (mset2fun xs) where
  shift_tail _ [] = []
  shift_tail f (x:xs) = x:(map f xs)
```

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)
```

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

```
set :: Encoder [Nat]
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 natural numbers representing sets. Then, through the use of the combinator `as`, sets represented by `set` are (bijectively) 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.

2.2 Ranking/unranking: from sets into natural numbers and back

We can *rank* a set represented as a list of distinct natural numbers by mapping it into a single natural number, and, 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 = sum (map (2^) ns)
```

Using the groupoid of isomorphisms given in [3] (also described in the Appendix), we will standardize our *ranking* and *unranking* operations as an *Encoder* for a natural number using (implicitly) finite sequences as a mediator:

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

Given that `nat` is an isomorphism with the Root object `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
```

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

```
*ISO> as set nat 2008
[3,4,6,7,8,9,10]
*ISO> as nat set [3,4,6,7,8,9,10]
2008
```

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_from set union nat 42 2008
2042
*ISO> 42 .|. 2008 :: Nat
2042
*ISO> borrow_from set intersect nat 42 2008
8
```

```
*ISO> 42 .&. 2008 :: Nat
8
```

and notice that operations like union and intersection of sets map to boolean operations on numbers.

3 Encoding finite multisets with primes

A factorization of a natural number is uniquely described as multiset 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. Note that 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. 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 [Nat]
pmset = compose (Iso pmset2nat nat2pmset) nat
```

working as follows:

```
*ISO> as pmset nat 2008
[3,3,12]
*ISO> as nat pmset it
2008
```

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.

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 subsection 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_from mset (++) nat
```

Proposition 1 $< N, mprod, 0 >$ *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 showing that `mprod` has properties similar 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> 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` involves factoring which is intractable for large values. This brings us back to observe that:

Proposition 2 $< N, pmprod, 0 >$ *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.*

One can 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))
```

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. 1 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.

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

Proposition 3 $< N, mprod', 1 >$ *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 4 *$mprod'\ x\ y = x * y$ if and only if $\exists n \geq 0$ such that $x = 2^n$ or $y = 2^n$. Otherwise, $mprod'\ x\ y < x * y$.*

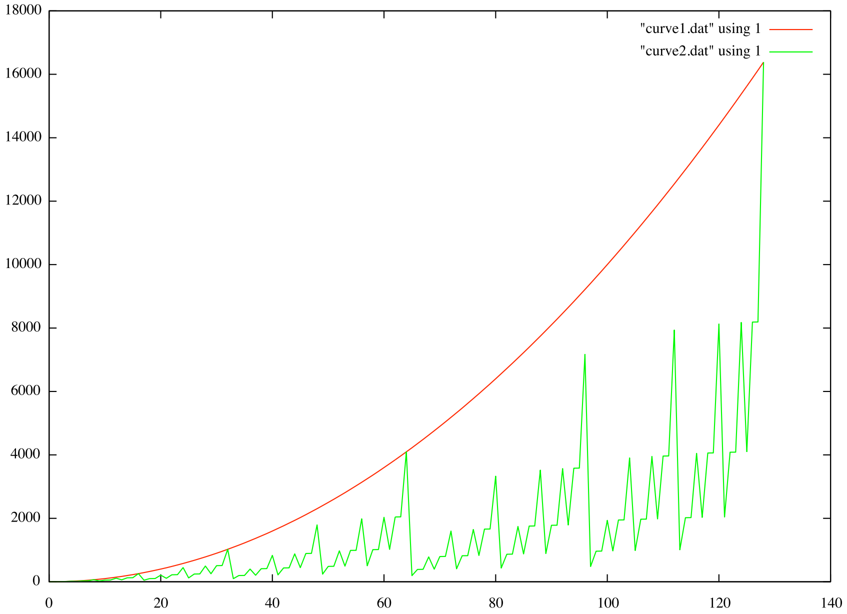

Fig. 1: Square vs. mexp' n 2

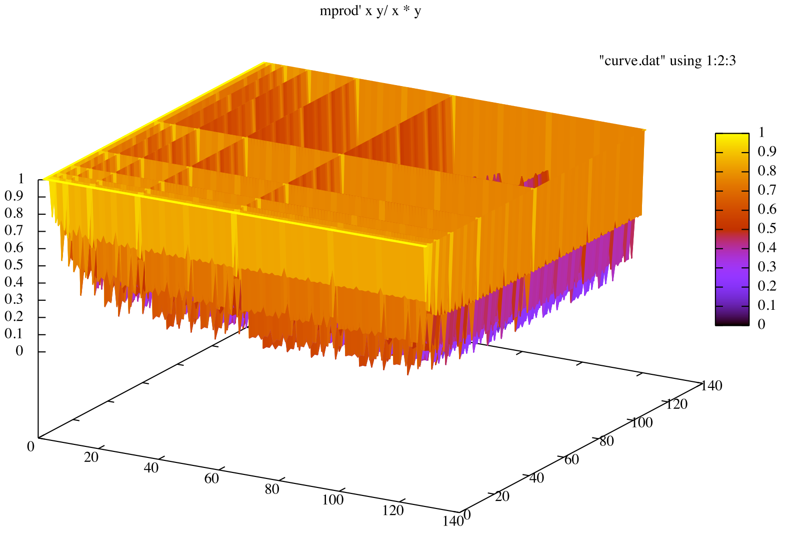

Fig. 2: Ratio between mprod' and product

Fig. 2 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].

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 :: Nat → Nat → Nat
mgcd = borrow_from mset msetInter nat

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

mdiv :: Nat → Nat → Nat
mdiv = borrow_from 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 5 *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` that contains k, is $n = 2^k$, and the definition of `mdiv`, as derived from the multiset difference operation `msetDif`.

The key difference between our "emulated" multiplicative arithmetics and the conventional one is that we do not have an obvious equivalent of addition. In its simplest form, this would mean defining a successor and predecessor function. Also, given that `mprod,mprod',pmprod'` and `pmprod` are not distributive with ordinary addition, it looks like an interesting *open problem* to find for each of them compatible additive operations.

5 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 used of any significantly advanced facts about prime numbers, the following references can be used to circumscribe the main topics to which our experiments can be connected [5–9].

Ranking functions can be traced back to Gödel numberings [10, 11] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms for various data types [12–15].

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 [16–20].

The closest reference on encapsulating bijections as a Haskell data type is [21] and Conal Elliott's composable bijections module [22], where, in a more complex setting, Arrows [23] are used as the underlying abstractions. While our `Iso` data type is similar to the *Bij* data type in [22] and BiArrow concept of [21], 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.

As the domains between which we define our groupoid of isomorphisms can be organized as categories, it is likely that some of our constructs would benefit from *natural transformation* [24] formulations.

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.

6 Conclusion

We have explored some computational analogies between multisets, natural number encodings and prime numbers in a framework for experimental mathematics implemented as a literate Haskell program. We will conclude by pointing out the new contributions of the paper:

- the isomorphisms between multisets/sets/sequences/natural numbers
- the encoding of multisets with primes and the π prime-counting function
- the commutative monoid structure on multisets emulating factoring

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 arbitrary length integer operations based on multiset representations.

References

1. Pippenger, N.: The average amount of information lost in multiplication. IEEE Transactions on Information Theory **51**(2) (2005) 684–687
2. 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.
3. Tarau, P.: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In: Proceedings of ACM SAC'09, Honolulu, Hawaii, ACM (March 2009) 1898–1903
4. 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
5. Crandall, R., Pomerance, C.: Prime Numbers–a Computational Approach. Second edn. Springer, New York (2005)
6. Young, J.: Large primes and Fermat factors. Math. Comp. **67**(244) (1998) 1735–1738
7. Riesel, H.: Prime Numbers and Computer Methods for Factorization. Volume 57 of Progress in Mathematics., Boston, MA (1985) Current edition is [**?**].
8. Keller, W.: Factors of Fermat numbers and large primes of the form $k \cdot 2^n + 1$. Math. Comp. **41** (1983) 661–673
9. Cgielski, P., Richard, D., Vsemirnov, M.: On the additive theory of prime numbers. Fundam. Inform. **81**(1-3) (2007) 83–96
10. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
11. 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
12. 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
13. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
14. Ruskey, F., Proskurowski, A.: Generating binary trees by transpositions. J. Algorithms **11** (1990) 68–84
15. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters **79** (2001) 281–284
16. Takahashi, M.o.: A Foundation of Finite Mathematics. Publ. Res. Inst. Math. Sci. **12**(3) (1976) 577–708
17. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume **48**(4) (2007) 497–510
18. 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
19. Avigad, J.: The Combinatorics of Propositional Provability. In: ASL Winter Meeting, San Diego (January 1997)
20. Kirby, L.: Addition and multiplication of sets. Math. Log. Q. **53**(1) (2007) 52–65
21. 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

22. Conal Elliott: Data.Bijections Haskell Module. http://haskell.org/haskellwiki/TypeCompose.
23. Hughes, J.: Generalizing Monads to Arrows Science of Computer Programming 37, pp. 67-111, May 2000.
24. Mac Lane, S.: Categories for the Working Mathematician. Springer-Verlag, New York, NY, USA (1998)

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:

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

```
data Iso a b = Iso (a→b) (b→a)

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

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

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
```

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

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
```

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
```

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 [] _ = []
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 [] _ = []
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))
```