Two Mechanisms for Generating Infinite Families of Pairing Bijections

Paul Tarau
University of North Texas
tarau@cs.unt.edu

ABSTRACT

We explore two general mechanisms for producing pairing bijections (bijective functions defined from $\mathbb{N}^2 \to \mathbb{N}$). The first mechanism, using n-adic valuations results in parameterized algorithms generating a countable family of distinct pairing bijections. The second mechanism, using characteristic functions of subsets of $\mathbb{N}$ provides $2^{\mathbb{N}}$ distinct pairing bijections. Mechanisms to combine such pairing bijections and their application to generate families of permutations of $\mathbb{N}$ are also described. The paper uses a small subset of the functional language Haskell to provide executable specifications of various the functions defined in a *literate programming* style.

Categories and Subject Descriptors

D.3.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and features—*Data types and structures* [Recursion]

General Terms

Algorithms, Languages, Theory

Keywords

pairing / unpairing functions, data type isomorphisms, infinite data objects, lazy evaluation, functional programming.

1. INTRODUCTION

DEFINITION 1. *A* pairing bijection *is a bijection* $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. *Its inverse* f^{-1} *is called an* unpairing *bijection.*

We are emphasizing here the fact that these functions are bijections as the name *pairing function* is sometime used in the literature to indicate injective-only functions from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$.

Pairing bijections have been used in the first half of 19-th century by Cauchy as a mechanism to express double summations as simple summations in series. They have been made famous by their uses in the second half of the 19-th century by Cantor's work on foundations of set theory. Their most well known application is to show that infinite sets like $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$ have the same cardinality. A classic use in the theory of recursive functions is to reduce functions on multiple arguments to single argument functions. Reasons on why they are an interesting object of study in terms of practical applications range from multi-dimensional dynamic arrays to proximity search using space filling curves are described in [9, 3, 4, 6].

Like in the case of Cantor's original function $f(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y$, pairing bijections have been usually hand-crafted by putting to work geometric or arithmetic intuitions.

This paper introduces two general mechanisms for building infinite families of pairing functions, using n-adic valuations (section 2) and characteristic functions of subsets of $\mathbb{N}$ (section 3), followed by a discussion of related work (section 4) and our conclusions (section 5).

It is easy to prove that there is an uncountable family of distinct pairing bijections and it is even easier to generate a countable family of pairing functions simply by modifying its result of a fixed pairing function with a reversible operation (e.g XOR with a natural number, seen as the index of the family). Therefore, we will point out the main reasons for why our more complex pairing bijections are *interesting*.

The n-adic valuation based pairing functions will provide a general mechanism for designing strongly asymmetric pairing functions, where changes in one of the arguments have an exponential impact on the result.

The characteristic-function mechanism, while intuitively obvious, opens the doors, in combination with a framework providing bijections between them and arbitrary data-types [13], to custom-build arbitrarily intricate pairing functions associated, for instance, to "interesting" sequences of natural numbers [10] or binary expansions of real numbers.

We will use a subset of the non-strict functional language Haskell to provide executable definitions of mathematical functions on $\mathbb{N}$, pairs in $\mathbb{N} \times \mathbb{N}$, subsets of $\mathbb{N}$, and sequences of natural numbers.

2. DERIVING PAIRING BIJECTIONS FROM N-ADIC VALUATIONS

We first overview a mechanism for deriving pairing bijections from one-solution Diophantine equations. Let us observe that

PROPOSITION 1. $\forall z \in \mathbb{N}^+ = \mathbb{N} - \{0\}$ *the Diophantine*

RACS'13 October 1-4, 2013, Montreal, QC, Canada.

equation

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

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

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

Note that a slight modification of equation 1 results in the pairing bijection originally introduced in [7, 2], seen as a mapping between the pair (x, y) and z.

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

We will generalize this mechanism to obtain a family of bijections between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}^+$ (and the corresponding pairing bijections between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$) by choosing an arbitrary base b instead of 2.

DEFINITION 2. *Given a number* $n \in \mathbb{N}$, $n > 1$, *the* n*-adic valuation of a natural number* m *is the largest exponent* k *of* n, *such that* n^k *divides* m. *It is denoted* $\nu_n(m)$.

Note that the solution x of the equation (1) is actually $\nu_2(z)$.

Our generalization proceeds as follows. We implement, for an arbitrary $b \in \mathbb{N}$, the functions `nAdicCons b` and `nAdicDeCons b`, forming a bijection between $\mathbb{N} \times \mathbb{N}$ and N^+.

```
nAdicCons :: N→(N,N)→N
nAdicCons b (x,y') | b>1 = (b^x)*y where
  q = y' `div` (b-1)
  y = y'+q+1
```

```
nAdicDeCons :: N→N→(N,N)
nAdicDeCons b z | b>1 && z>0 = (x,y') where
  hd n = if n `mod` b > 0 then 0 else 1+hd (n `div` b)
  x = hd z
  y = z `div` (b^x)
  q = y `div` b
  y' = y-q-1
```

Using `nAdicDeCons` we define the head and tail projection functions `nAdicHead` and `nAdicTail`:

```
nAdicHead, nAdicTail :: N→N→N
nAdicHead b = fst . nAdicDeCons b
nAdicTail b = snd . nAdicDeCons b
```

Intuitively, their correctness follows from the fact that the quotient y, after dividing z with the largest possible power b^x is not divisible by b. We then "rebase" it to base $b-1$, and make z correspond to a unique pair of natural numbers.

More precisely, non-divisibility of y by b, stated as $y = bq+m, b > m > 0$ can be rewritten as $y-q-1 = bq-q+m-1, b > m > 0$, or equivalently $y-q-1 = (b-1)q+(m-1), b > m > 0$ from where it follows that by setting $y' = y-q-1$ and $m' = m-1$ we map z to a pair (y', m') such that $y' = (b-1)q+m'$ and $b-1 > m' \geq 0$. So we can transform (y, m) such that $y = bq + m$ with $b > m > 0$, into a pair (y', m'), such that $y' = q(b-1) + m'$ with $b-1 > m' \geq 0$. Note that the transformation works also in the opposite direction with $y' = y - q - 1$ giving $y = y' + q + 1$, and with $m' = m - 1$ giving $m = m' + 1$.

The following examples illustrate the operations for base `3`:

```
*InfPair> nAdicCons 3 (10,20)
1830519
*InfPair> nAdicHead 3 1830519
10
*InfPair> nAdicTail 3 1830519
20
```

Note that `nAdicHead n x` computes the n-adic valuation of x, $\nu_n(x)$ while the tail corresponds to the "information content" extracted from the quotient, after division by $\nu_n(x)$.

DEFINITION 3. *We call the natural number computed by* `nAdicHead n x` *the* n**-adic head** *of* $x \in \mathbb{N}^+$, *by* `nAdicTail n x` *the* n**-adic tail** *of* $x \in \mathbb{N}^+$ *and the natural number in* $\mathbb{N}^+$ *computed by* `nAdicCons n (x,y)` *the* n**-adic cons** *of* $x, y \in \mathbb{N}$.

By generalizing the mechanism shown for the equations 1 and 2, we derive from `nAdicDeCons` and `nAdicCons` the corresponding *pairing* and *unpairing* bijections `nAdicPair` and `nAdicUnPair`:

```
nAdicUnPair :: N→N→(N,N)
nAdicUnPair b n = nAdicDeCons b (n+1)

nAdicPair :: N→(N,N)→N
nAdicPair b xy = (nAdicCons b xy)-1
```

One can see that we obtain a countable family of bijections $f_b : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ indexed by $b \in \mathbb{N}$, $b > 1$.

The following examples illustrate the work of these bijections for $b = 3$.

```
*InfPair> map (nAdicUnPair 3) [0..7]
[(0,0),(0,1),(1,0),(0,2),(0,3),(1,1),(0,4),(0,5)]
*InfPair> map (nAdicPair 3) it
[0,1,2,3,4,5,6,7]
```

Note the use of Haskell's higher-order function "`map`", that applies the function `nAdicUnPair 3` to a list of elements and collects the results to a list, and the special value "`it`", standing for the previously computed result.

2.1 Deriving bijections between $\mathbb{N}$ and $[\mathbb{N}]$

For each base `b>1`, we can also obtain a pair of bijections between natural numbers and lists of natural numbers in terms of `nAdicHead`, `nAdicTail` and `nAdicCons`:

```
nat2nats :: N→N→[N]
nat2nats _ 0 = []
nat2nats b n | n>0 =
   nAdicHead b n : nat2nats b (nAdicTail b n)
```

```
nats2nat :: N→[N]→N
nats2nat _ [] = 0
nats2nat b (x:xs) = nAdicCons b (x,nats2nat b xs)
```

The following examples illustrate how they work:

```
*InfPair> nat2nats 3 2012
[0,2,2,0,0,0,0]
*InfPair> nats2nat 3 it
2012
```

Using the framework introduced in [11, 13] and summarized in the **Appendix**, we can "reify" these bijections as `Encoders` between natural numbers and sequences of natural numbers (parameterized by the first argument of `nAdicHead` and `nAdicTail`). Such Encoders can now be "morphed", by using the bijections provided by the framework, into various data types sharing the same "information content" (e.g. lists, sets, multisets).

```
nAdicNat :: N→Encoder N
nAdicNat k = Iso (nat2nats k) (nats2nat k)
```

In particular, for $k = 2$, we obtain the `Encoder` corresponding to the Diophantine equation (1)

```
nat :: Encoder N
nat = nAdicNat 2
```

The following examples illustrate these operations, lifted through the framework defining bijections between datatypes, given in **Appendix**.

```
*InfPair> as (nAdicNat 3) list [2,0,1,2]
873
*InfPair> as (nAdicNat 7) list [2,0,1,2]
27146
*InfPair> as nat list [2,0,1,2]
300
*InfPair> as list nat it
[2,0,1,2]
```

2.2 Deriving new families of Encoders and Permutations of $\mathbb{N}$

For each $l, k \in \mathbb{N}$ one can generate a family of permutations (bijections $f : \mathbb{N} \to \mathbb{N}$), parameterized by the pair `(l,k)`, by composing `nat2nats l` and `nats2nat k`.

```
nAdicBij :: N → N → N → N
nAdicBij k l = (nats2nat l) . (nat2nats k)
```

The following example illustrates their work on the initial segment `[0..31]` of $\mathbb{N}$:

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

It is easy to see that the following holds:

PROPOSITION 2.

$$(nAdicBij\ k\ l) \circ (nAdicBij\ l\ k) \equiv id \tag{3}$$

We can derive `Encoders` representing functions between $\mathbb{N}$ and sequences of natural numbers, parameterized by a (possibly infinite) list of `nAdicHead / nAdicTail` bases, by repeatedly applying the n-adic head, tail and cons operation parameterized by the (assumed infinite) sequence `ks`:

```
nAdicNats :: [N]→Encoder N
nAdicNats ks = Iso (nat2nAdicNats ks) (nAdicNats2nat ks)

nat2nAdicNats :: [N]→N→[N]
nat2nAdicNats _ 0 = []
nat2nAdicNats (k:ks) n | n>0 =
  nAdicHead k n : nat2nAdicNats ks (nAdicTail k n)

nAdicNats2nat :: [N]→[N]→N
nAdicNats2nat _ [] = 0
nAdicNats2nat (k:ks) (x:xs) =
  nAdicCons k (x,nAdicNats2nat ks xs)
```

For instance, the Encoder `nat'` corresponds to `ks` defined as the infinite sequence starting at `2`.

```
nat' :: Encoder N
nat' = nAdicNats [2..]
```

The following examples illustrate the mechanism:

```
*InfPair> as nat' list [2,0,1,2]
1644
*InfPair> as list nat' it
[2,0,1,2]
*InfPair> map (as nat' nat) [0..15]
[0,1,2,3,4,7,6,5,8,19,14,15,12,13,10,9]
*InfPair> map (as nat' nat) [0..15]
[0,1,2,3,4,7,6,5,8,19,14,15,12,13,10,9]
```

Note that functions like `as nat' nat` illustrate another general mechanism for defining permutations of $\mathbb{N}$.

3. PAIRING BIJECTIONS DERIVED FROM CHARACTERISTIC FUNCTIONS OF SUBSETS OF $\mathbb{N}$

We start by connecting the bitstring representation of characteristic functions to our bijective data transformation framework (overviewed in the **Appendix**).

3.1 The bijection between lists and characteristic functions of sets

The function `list2bins` converts a sequence of natural numbers into a characteristic function of a subset of $\mathbb{N}$ represented as a string of binary digits. The algorithm interprets each element of the list as the number of `0` digits before the next `1` digit. Note that infinite sequences are handled as well, resulting in infinite bitstrings.

```
list2bins :: [N]→[N]
list2bins [] = [0]
list2bins ns = f ns where
  f [] = []
  f (x:xs) = (repl x 0) ++ (1:f xs) where
    repl n a | n ≤ 0 = []
    repl n a = a:repl (pred n) a
```

The function `bin2list` converts a characteristic function represented as bitstrings back to a list of natural numbers.

```
bins2list :: [N] → [N]
bins2list xs = f xs 0 where
  f [] _ = []
  f (0:xs) k = f xs (k+1)
  f (1:xs) k = k : f xs 0
```

Together they provide the Encoder `bins`, that we will use to connect characteristic functions to various data types.

```
bins :: Encoder [N]
bins = Iso bins2list list2bins
```

The following examples (where the Haskell library function `take` is used to restrict execution to an initial segment of an infinite list) illustrate their use:

```
*InfPair> list2bins [2,0,1,2]
[0,0,1,1,0,1,0,0,1]
*InfPair> bins2list it
[2,0,1,2]

*InfPair> take 20 (list2bins [0,2..])
[1,0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0]
*InfPair> bins2list it
[0,2,4,6]
```

The following holds:

PROPOSITION 3. *If M is a subset of $\mathbb{N}$, the bijection* `as bins set` *returns the bitstring associated to M and its inverse is the bijection* `as set bins`.

PROOF. Observe that the transformations are the composition of bijections between bitstrings and lists and bijections between lists and sets. □

The following example illustrates this correspondence:

```
*InfPair> as bins set [0,2,4,5,7,8,9]
[1,0,1,0,1,1,0,1,1,1]
*InfPair> as set bins it
[0,2,4,5,7,8,9]
```

Note that, for convenient use on finite sets, the functions do not add the infinite stream of `0` digits indicating its infinite stream of non-members, but we will add it as needed

when the semantics of the code requires it for representing accurately operations on infinite sequences. We will use the same convention through the paper.

3.2 Splitting and merging bitstrings with a characteristic function

Guided by the characteristic function of a subset of $\mathbb{N}$, represented as a bitstring, the function `bsplit` separates a (possibly infinite) sequence of numbers into two lists: members and non-members.

```
bsplit :: [N] → [N] → ([N], [N])
bsplit _ [] = ([],[])
bsplit [] (n:ns) =
  error ("bspilt provides no guidance at: "++(show n))
bsplit (0:bs) (n:ns) = (xs,n:ys) where
  (xs,ys) = bsplit bs ns
bsplit (1:bs) (n:ns) = (n:xs,ys) where
  (xs,ys) = bsplit bs ns
```

Guided by the characteristic function of a subset of $\mathbb{N}$, represented as a bitstring, the function `bmerge` merges two lists of natural numbers into one, by interpreting each `1` in the characteristic function as a request to extract an element of the first list and each `0` as a request to extract an element of the second list.

```
bmerge :: [N] → ([N], [N]) → [N]
bmerge _ ([],[]) = []
bmerge bs ([],[y]) = [y]
bmerge bs ([x],[]) = [x]
bmerge bs ([],ys) = bmerge bs ([0],ys)
bmerge bs (xs,[]) = bmerge bs (xs,[0])
bmerge (0:bs) (xs,y:ys) = y : bmerge bs (xs,ys)
bmerge (1:bs) (x:xs,ys) = x : bmerge bs (xs,ys)
```

The following examples (trimmed to finite lists) illustrate their use:

```
*InfPair> bsplit [0,1,0,1,0,1] [10,20,30,40,50,60]
([20,40,60],[10,30,50])
*InfPair> bmerge [0,1,0,1,0,1] it
[10,20,30,40,50,60]
```

3.3 Defining pairing bijections, generically

We design a generic mechanism to derive pairing functions by combining the data type transformation operation `as` with the `bsplit` and `bmerge` functions that apply a characteristic function encoded as a list of bits.

```
genericUnpair :: Encoder t → t →   N → (N, N)
genericUnpair xEncoder xs n = (l,r) where
  bs = as bins xEncoder xs
  ns = as bins nat n
  (ls,rs) = bsplit bs ns
  l = as nat bins ls
  r = as nat bins rs
```

```
genericPair :: Encoder t → t →   (N, N) → N
genericPair xEncoder xs (l,r) = n where
  bs = as bins xEncoder xs
  ls = as bins nat l
  rs = as bins nat r
  ns = bmerge bs (ls,rs)
  n = as nat bins ns
```

Let us observe first that termination of this function depends on termination of the calls to `bsplit` and `bmerge`, as illustrated by the following examples:

```
*InfPair> genericPair bins (cycle [0]) (10,20)
^CInterrupted.
*InfPair> genericUnpair bins (cycle [1]) 42
(^CInterrupted.
```

In this case, the characteristic functions given by `cycle [0]` or `cycle [1]` would trigger an infinite search for a non-existing first `1` or `0` in `bsplit` and `bmerge`.

Clearly, this suggests restrictions on the acceptable characteristic functions.

We will now give sufficient conditions ensuring that the functions `genericUnpair` and `genericPair` terminate for any values of their last arguments. Such restrictions, will enable them to define families of pairing functions parameterized by characteristic functions derived from various data types.

Definition 4. *We call* block *of digits occurring in a characteristic function any (finite or infinite) contiguous sequence of digits.*

Note that an infinite block made entirely of 0 (or 1) digits can only occur at the end of the sequence defining the characteristic function, i.e. only if it exists a number n such that the index of each member of the block is larger than n.

Proposition 4. *If $\{a_n\}_{n\in\mathbb{N}}$ is an infinite sequence of bits containing only finite blocks of* `0` *and* `1` *digits,* `genericPair bins` *and* `genericUnPair bins` *define a family of pairing bijections parameterized by $\{a_n\}_{n\in\mathbb{N}}$.*

Proof. Having an alternation of finite blocks of 1s and 0s, ensures that, when called from `genericPair` and `genericUnPair`, the functions `bmerge` and `bsplit` terminate. □

For instance, *Morton* codes [4] are derived by using a stream of alternating `1` and `0` digits (provided by the Haskell library function `cycle`)

```
bunpair2 = genericUnpair bins (cycle [1,0])
bpair2 = genericPair bins (cycle [1,0])
```

and working as follows:

```
*InfPair> map bunpair2 [0..10]
[(0,0),(1,0),(0,1),(1,1),(2,0),
 (3,0),(2,1),(3,1),(0,2),(1,2),(0,3)]
*InfPair> map bpair2 it
[0,1,2,3,4,5,6,7,8,9,10]
```

Proposition 5. *If $\{a_n\}_{n\in\mathbb{N}}$ is an infinite sequence of non-decreasing natural numbers only containing finite blocks of equal values, the functions* `genericPair set` *and* `genericUnPair set` *define a family of pairing bijections parameterized by $\{a_n\}_{n\in\mathbb{N}}$.*

Proof. Observe that the sequence represents canonically an infinite set such that its complement is also an infinite non-decreasing sequence. Therefore, the associated characteristic function will have an alternation of finite blocks of `1` and `0` digits, inducing a pairing/unpairing bijection. □

The bijection `bpair k` and its inverse `bunpair k` are derived from a `set` representation (implicitly morphed into a characteristic function).

```
bpair k = genericPair set [0,k..]
bunpair k = genericUnpair set [0,k..]
```

Note that for `k = 2` we obtain exactly the bijections `bpair2` and `bunpair2` derived previously, as illustrated by the following example:

```
*InfPair> map (bunpair 2) [0..10]
[(0,0),(1,0),(0,1),(1,1),(2,0),(3,0),
 (2,1),(3,1),(0,2),(1,2),(0,3)]
*InfPair> map (bpair 2) it
[0,1,2,3,4,5,6,7,8,9,10]
```

We conclude with a similar result for lists:

PROPOSITION 6. *If $\{a_n\}_{n\in\mathbb{N}}$ is an infinite sequence of natural numbers only containing finite blocks of 0s, the functions* `genericPair list` *and* `genericUnPair list` *define a family of pairing bijections parameterized by $\{a_n\}_{n\in\mathbb{N}}$.*

PROOF. It follows from Prop. 5 by observing that such sequences are transformed into infinite sets represented as non-decreasing sequences only containing finite blocks of equal values. □

PROPOSITION 7. *There are $2^{\mathbb{N}}$ pairing functions defined using characteristic functions of sets of $\mathbb{N}$.*

PROOF. Observe that a characteristic function corresponding to a subset of $\mathbb{N}$ containing an infinite block of `0` or `1` digits necessarily ends with the block. Therefore, by erasing the block we can put such functions in a bijection with a finite subset of $\mathbb{N}$. Given that there are only a countable number of finite subsets of $\mathbb{N}$, the cardinality of the set of the remaining subsets' characteristic functions is $2^{\mathbb{N}}$. □

PROPOSITION 8. *The infinite sequence of binary digits representing any irrational number corresponds to a unique pairing function.*

PROOF. Observe that no periodic sequence can be present in the binary representation of an irrational number, and in particular the representation cannot end with an infinite block of 0s or 1s. □

The following proposition answers the converse question: can we associate infinite subsets of$\mathbb{N}$ to an arbitrary unpairing bijection?

PROPOSITION 9. *Given an unpairing bijection $f : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ one can construct two disjoint infinite subsets of $\mathbb{N}$ uniquely determined by f.*

PROOF. Observe that, based on the parity of $f(n)$ we can use any of the 14 two-argument boolean functions (except constant functions 0 and 1) to place n in one of the two infinite sets. The Haskell function `unpair2subsets` describes the construction, based on the `xor` function. □

```
unpair2subsets :: (N → (N,N)) → ([N],[N])
unpair2subsets f = unpair_split f 0

unpair_split f n = xor_split (odd a) (odd b) where
  (a,b) = f n
  (xs,ys) = unpair_split f (n+1)
  xor_split False False = (n:xs,ys)
  xor_split False True = (xs,n:ys)
  xor_split True False = (xs,n:ys)
  xor_split True True = (n:xs,ys)
```

The following example illustrates the construction. Note that Haskell's call-by-need mechanism allows building the two infinite sets step-by-step.

```
*InfPair> take 10 (fst (unpair2subsets (bunpair 3)))
[0,3,4,7,8,11,12,15,16,19]
*InfPair> take 10 (snd (unpair2subsets (bunpair 3)))
[1,2,5,6,9,10,13,14,17,18]
```

4. RELATED WORK

Pairing functions have been used in work on decision problems as early as [7, 2, 8]. There are a large number of Google documents referring to the original "Cantor pairing function". An extensive study of various pairing functions and their computational properties is presented in [1, 9]. They are also related to 2D-space filling curves (Z-order, Gray-code and Hilbert curves) [3, 4, 5, 6]. Such curves are obtained by connecting pairs of coordinates corresponding to successive natural numbers (obtained by applying unpairing operations). They have applications to spatial and multi-dimensional database indexing [3, 4, 5, 6] and symbolic arbitrary length arithmetic computations [14]. Note also that `bpair 2` and `bunpair 2` are also known as Morton-codes, with uses in indexing of spatial databases [3].

5. CONCLUSION

We have described two mechanisms for generating countable and uncountable families of pairing / unpairing bijections. The mechanism involving n-adic valuations is definitely novel, and we have high confidence (despite of their obviousness) that the characteristic function-based mechanisms are novel as well, at least in terms of their connections to list, set or multiset representations provided by the implicit use of our bijective data transformation framework [11].

Given the space constraints, we have not explored the natural extensions to more general tupling / untupling bijections (defined between $\mathbb{N}^k$ and $\mathbb{N}$) as well as bijections between finite lists, sets and multisets that can be derived quite easily, using the data transformation framework given in the **Appendix**. For the same reasons we have not discussed specific applications of these families of pairing functions, but we foresee interesting connections with possible cryptographic uses (e.g "one time pads" generated through intricate combinations of members of these families).

The ability to associate such pairing functions to arbitrary characteristic functions as well as to their equivalent set, multiset, list representations provides convenient tools for inventing and customizing pairing / unpairing bijections, as well as the related tupling / untupling bijections and those defined between natural numbers and sequences, sets and multisets of natural numbers.

6. REFERENCES

[1] P. Cegielski and D. Richard. On arithmetical first-order theories allowing encoding and decoding of lists. *Theoretical Computer Science*, 222(1-2):55–75, 1999.

[2] L. Kalmar. On the Reduction of the Decision Problem. First Paper. Ackermann Prefix, A Single Binary Predicate. *The Journal of Symbolic Logic*, 4(1):1–9, mar 1939.

[3] J. Lawder. The application of space-filling curves to the storage and retrieval of multi-dimensional data, 2000. PhD Thesis, University of London, UK.

[4] J. Lawder and P. King. Using space-filling curves for multi-dimensional indexing. In B. Lings and K. Jeffery, editors, *Advances in Databases*, volume 1832 of *Lecture Notes in Computer Science*, pages 20–35. Springer Berlin / Heidelberg, 2000.

[5] J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Rec.*, 30:19–24, March 2001.

[6] B. Moon, H. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13:124–141, 2001.

[7] J. Pepis. Ein verfahren der mathematischen logik. *The Journal of Symbolic Logic*, 3(2):61–76, jun 1938.

[8] J. Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950.

[9] A. L. Rosenberg. Efficient pairing functions - and why you should care. *International Journal of Foundations of Computer Science*, 14(1):3–17, 2003.

[10] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. 2010. published electronically at www.research.att.com/∼njas/sequences.

[11] P. Tarau. A Groupoid of Isomorphic Data Transformations. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009* , pages 170–185, Grand Bend, Canada, July 2009. Springer, LNAI 5625.

[12] P. Tarau. Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell, Jan. 2009. Unpublished draft, http://arXiv.org/abs/0808.2953, updated version at http://logic.cse.unt.edu/tarau/research/2010/ISO.pdf, 150 pages.

[13] P. Tarau. "Everything Is Everything" Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms. *Complex Systems*, (18):475–493, 2010.

[14] P. Tarau and D. Haraburda. On Computing with Types. In *Proceedings of SAC'12, ACM Symposium on Applied Computing, PL track*, pages 1889–1896, Riva del Garda (Trento), Italy, Mar. 2012.

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 refer to ([11, 12]) for details.

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.

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

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. The isomorphisms are naturally organized as a *groupoid*.

```
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`, $f \circ g = id_b$ and $g \circ f = id_a$, 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.*

The Hub: Sequences of Natural Numbers

To avoid defining $\frac{n(n-1)}{2}$ isomorphisms between n objects, we choose a *Hub* object to/from which we will actually implement isomorphisms.

Choosing a *Hub* 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 `Hub` object *sequences of natural numbers*. We will represent them as lists i.e. their Haskell type is `[N]`.

```
type N = Integer
type Hub = [N]
```

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

```
type Encoder a = Iso a Hub
```

together with the combinator "`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 "`as`" adds a 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
```

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

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

The `Encoder mset` for multisets of natural numbers is defined as:

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

mset2list xs = zipWith (-) (xs) (0:xs)
list2mset ns = tail (scanl (+) 0 ns)
```

The `Encoder set` for sets of natural numbers is defined as:

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

list2set = (map pred) . list2mset . (map succ)
set2list = (map pred) . mset2list . (map succ)
```

Note that these converters between lists, multisets and sets make no assumption about finiteness of their arguments and therefore they can used in a non-strict language like Haskell on infinite objects as well.