On Two Infinite Families of Pairing Bijections

Paul Tarau
Department of Computer Science and Engineering
University of North Texas
Denton, Texas
Email: tarau@cs.unt.edu

Abstract—We describe two general mechanisms for producing pairing bijections (bijective functions defined from \( N^2 \to N \)). The first mechanism, using \( n \)-adic functions defined from \( N \), results in parameterized algorithms that provide a countable family of distinct pairing bijections. The second mechanism, using characteristic functions of subsets of \( N \), provides \( 2^n \) distinct pairing bijections. Mechanisms to combine such pairing functions and their application to generate families of permutations of \( N \) are also described. The paper uses a small subset of the functional language Haskell to provide type checked executable specifications of all the functions defined in a literate programming style. The self-contained Haskell code extracted from the paper is available at \( \text{http://logic.cse.unt.edu/tarau/research/2012/infpair.hs} \).

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

I. INTRODUCTION

Definition 1: A pairing bijection is a bijection \( f : N \times N \to N \). 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 functions from \( N \times N \) to \( N \).

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. 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 \( N \) and \( N \times N \) have the same cardinality. A classic use in the theory of recursive functions is to reduce functions on multiple arguments to single arguments functions. Reasons on why they are an interesting object of study in terms of practical applications ranging from multi-dimensional dynamic arrays to proximity search using space filling curves are described in [4], [6], [7], [14].

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.

While it is easy to prove (non-constructively) that there is an uncountable family of distinct pairing bijections, we have not seen in the literature general mechanisms for building families of pairing bijections indexed by \( N \) or \( 2^N \). It is even easier to generate (constructively) 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).

This paper introduces two general mechanisms for generating such families, using \( n \)-adic valuations (section II) and characteristic functions of subsets of \( N \) (section III), followed by a discussion of related work (section IV) and our conclusions (section V).

We will give here a glimpse of why our arguably 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 [15], to custom-build arbitrarily intricate pairing functions associated to for instance to “interesting” sequences of natural numbers or binary expansions of real numbers.

We will use a subset of the non-strict functional language Haskell (see an equational notation for typed \( \lambda \)-calculus) to provide executable definitions of mathematical functions on \( N \), pairs in \( N \times N \), subsets of \( N \), and sequences of natural numbers. We mention, for the benefit of the reader unfamiliar with the language, that a notation like \( f \times y \) stands for \( f(x, y) \), \([\_\_] \) represents sequences of type \( t \) and a type declaration like \( f : s \to t \to u \) stands for a function \( f : s \times t \to u \) (modulo Haskell’s “currying” operation, given the isomorphism between the function spaces \( s \times t \to u \) and \( s \to t \to u \)). Our Haskell functions are always represented as sets of recursive equations guided by pattern matching, conditional to constraints (simple arithmetic relations following \( \mid \) and before the = symbol). Locally scoped helper functions are defined in Haskell after the where keyword, using the same equational style. The composition of functions \( f \) and \( g \) is denoted \( f \cdot g \). It is also customary in Haskell, when defining functions in an equational style (using =) to write \( f = g \) instead of \( f x = g x \) (“point-free” notation). The use of Haskell’s “call-by-need” evaluation allows us to work with infinite sequences, like the \([0..]\) infinite list notation, corresponding to the set \( N \) itself.

II. 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

\[ 2^2(2y + 1) = z \]

has exactly one solution \( x, y \in N \).

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 \( I \) results in the pairing bijection originally introduced in [3], [8], seen as a mapping between the pair \( (x, y) \) and \( z \).

\[ 2^2(2y + 1) − 1 = z \]

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

Definition 2: Given a number \( n \in N \), \( n > 1 \), 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 \( I \) is actually \( \nu_2(z) \). This suggest deriving similar Diophantine equations for an arbitrary \( n \)-adic valuation. We start by observing that the following holds:

\[ \forall b \in N, b > 1, \forall y \in N \text{ if } \exists q, m \text{ such that } b > m > 0, y = bq + m, \text{ then there’s exactly one pair } (y', m') \]

The characteristic-function mechanism, while intuitively obvious, opens the doors, in combination with a framework providing bijections between them and arbitrary data-types [15], to custom-build arbitrarily intricate pairing functions associated to for instance to \( n \)-adic binary expansions of real numbers.

We will use a subset of the non-strict functional language Haskell (see an equational notation for typed \( \lambda \)-calculus) to provide executable definitions of mathematical functions on \( N \), pairs in \( N \times N \), subsets of \( N \), and sequences of natural numbers. We mention, for the benefit of the reader unfamiliar with the language, that a notation like \( f \times y \) stands for \( f(x, y) \), \([\_\_] \) represents sequences of type \( t \) and a type declaration like \( f : s \to t \to u \) stands for a function \( f : s \times t \to u \) (modulo Haskell’s “currying” operation, given the isomorphism between the function spaces \( s \times t \to u \) and \( s \to t \to u \)). Our Haskell functions are always represented as sets of recursive equations guided by pattern matching, conditional to constraints (simple arithmetic relations following \( \mid \) and before the = symbol). Locally scoped helper functions are defined in Haskell after the where keyword, using the same equational style. The composition of functions \( f \) and \( g \) is denoted \( f \cdot g \). It is also customary in Haskell, when defining functions in an equational style (using =) to write \( f = g \) instead of \( f x = g x \) (“point-free” notation). The use of Haskell’s “call-by-need” evaluation allows us to work with infinite sequences, like the \([0..]\) infinite list notation, corresponding to the set \( N \) itself.

II. 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

\[ 2^2(2y + 1) = z \]

has exactly one solution \( x, y \in N \).

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 \( I \) results in the pairing bijection originally introduced in [3], [8], seen as a mapping between the pair \( (x, y) \) and \( z \).

\[ 2^2(2y + 1) − 1 = z \]

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

Definition 2: Given a number \( n \in N \), \( n > 1 \), 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 \( I \) is actually \( \nu_2(z) \). This suggest deriving similar Diophantine equations for an arbitrary \( n \)-adic valuation. We start by observing that the following holds:

\[ \forall b \in N, b > 1, \forall y \in N \text{ if } \exists q, m \text{ such that } b > m > 0, y = bq + m, \text{ then there’s exactly one pair } (y', m') \]

The characteristic-function mechanism, while intuitively obvious, opens the doors, in combination with a framework providing bijections between them and arbitrary data-types [15], to custom-build arbitrarily intricate pairing functions associated to for instance to \( n \)-adic binary expansions of real numbers.
m' ≥ 0 such that y' = (b−1)q + m' and the function associating (y', m') to a bijection.  

Proof: Let f be the bijection defined by f(x, y, z) = (bq + m, b > m > 0). Then f is a bijection between (N, N) and (N, N), since it is a bijection between (N, N) and (N, N).  

Proposition 3: ∀b ∈ N, b > 1, ∀z ∈ N, z > 0 the system of Diophantine equations and inequalities  

\[ b^\ast (y' + q + 1) = z \]  
\[ y' = (b−1)q + m' \]  
\[ b - 1 > m' ≥ 0 \]  

has exactly one solution x, y' ∈ N.  

Proof: Let \( f^{-1} \) be the inverse of the bijection f defined in Proposition 2. Then \( f^{-1} \) provides the desired unique mapping, which gives y = y' + q + 1 and m = m' = 1. Therefore y ≡ m (mod b) with m > 0. And as y is not divisible by b, we can determine uniquely x as the largest power of b dividing z, x = \( \nu_b(z) \).  

We implement, for and arbitrary b ∈ N, the Haskell code corresponding to these bijections as the functions nAdicCons b and nAdicDeCons b, defined between N × N and N∗.  

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

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

nAdicHead, nAdicTail :: N → (N,N)  
nAdicHead b = fst . nAdicCons b  
nAdicTail b = snd . nAdicCons b  

The following examples illustrate the operations for base 3:  

\*InfPair> nAdicCons 3 10 20 \n1830519  
\*InfPair> nAdicHead 3 1830519 \n10  
\*InfPair> nAdicTail 3 1830519 \n20  

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 remainder, after division by \( \nu_n(x) \).  

Definition 3: We call the natural number computed by nAdicHead n x the n-adic head of x ∈ N∗, by nAdicTail n x the n-adic tail of x ∈ N∗ and the natural number in N∗ computed by nAdicCons n (x, y) the n-adic cons of x, y ∈ 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 \( f_b : N \times N \rightarrow N \) indexed by \( b \in N, b > 1 \).  

The following examples illustrate the work of these bijections for \( b = 3 \). 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.  

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

1) Deriving bijections between \( N \) and \( [N] \): For each base \( b \), we can also obtain a pair of bijections between natural numbers and lists of natural numbers in terms of nAdicHead, nAdicTail and nAdicCons:  

\( \text{nat2nats} :: N \rightarrow [N] \text{ } \)  
\( \text{nat2nats} 0 = [] \text{ } \)  
\( \text{nat2nats} b n \text{ } | \text{ } b>0 = \)  
\( \text{nAdicHead} b n : \text{nat2nats} b \text{ } \)  
\( \text{nat2nats} b (x:x's) = \text{nAdicCons} b (x,nat2nats b x's) \)  

The following example illustrate how they work:  

\*InfPair> nat2nats 3 2012 \n[0,2,2,0,0,0,0]  
\*InfPair> nats2nat 3 it \n2012  

Using the framework introduced in 16, 18 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:  

\( \text{nat} :: \text{Encoder } N \)  
\( \text{nat} = \text{nAdicNat } 2 \)  

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] \n873  
\*InfPair> as (nAdicNat 7) list [2,0,1,2] \n27146  
\*InfPair> as nat list [2,0,1,2] \n300  
\*InfPair> as list nat it \n[2,0,1,2]  

2) Deriving new families of Encoders and Permutations of \( N \): For each \( I, k \in N \) one can generate a family of permutations (bijections \( f : N \rightarrow N \)), parameterized by the pair \( (1,k) \), by composing nat2nats 1 and nats2nat k.
The following example illustrates their work on the initial segment \([0..31]\) of \(N\):

\[
\begin{align*}
\text{nat' list} & \equiv \text{nat' list} [2..] \\
\text{nat' list} & \equiv \text{nat' list} [2,0,1,2] \\
1644 & \equiv \text{nat' list} [2,0,1,2] \\
\text{nat' list} & \equiv \text{nat' list} [0,2,4,5,7,8,9] \\
1644 & \equiv \text{nat' list} [0,2,4,5,7,8,9]
\end{align*}
\]

The following examples illustrate the mechanism:

\[
\begin{align*}
\text{as nat' list} & \equiv \text{as nat' list} [2,0,1,2] \\
1644 & \equiv \text{as nat' list} [2,0,1,2] \\
\text{as nat' list} & \equiv \text{as nat' list} [0,2,4,5,7,8,9] \\
1644 & \equiv \text{as nat' list} [0,2,4,5,7,8,9]
\end{align*}
\]

Note that functions like as nat’ nat illustrate another general mechanism for defining permutations of \(N\).

III. PAIRING BIJECTIONS DERIVED FROM CHARACTERISTIC FUNCTIONS OF SUBSETS OF \(N\)

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

A. The bijection between lists and characteristic functions of sets

The function \(\text{list2bins}\) converts a sequence of natural numbers into a characteristic function of a subset of \(N\) 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.

The function \(\text{bin2list}\) converts a characteristic function represented as bitstrings back to a list of natural numbers.

Together they provide the Encoder \(\text{bins}\), that we will use to connect characteristic functions to various data types.

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

\[
\begin{align*}
\text{list2bins} & \equiv \text{list2bins} [2,0,1,2] \\
[0,0,1,0,1,0,1,0,0,1] & \equiv \text{list2bins} [2,0,1,2] \\
\text{as nat' list} & \equiv \text{as nat' list} [0,2,4,5,7,8,9] \\
[0,0,1,0,1,0,1,1,1] & \equiv \text{as nat' list} [0,2,4,5,7,8,9]
\end{align*}
\]

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.

B. Splitting and merging bitstrings with a characteristic function

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

The following example illustrates this correspondence:

\[
\begin{align*}
\text{bsplit} & \equiv \text{bsplit} [0,2,4,5,7,8,9] \\
[1,0,1,0,1,0,1,1,1] & \equiv \text{bsplit} [0,2,4,5,7,8,9]
\end{align*}
\]

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.
Guided by the characteristic function of a subset of \( \mathbb{N} \), represented as a bitstring, the function \( \text{bmerge} \) 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.

\[
\text{bmerge} :: [N] \to ([N], [N]) \to [N]
\]

\[
\text{bmerge} _{([],[])} = []
\]

\[
\text{bmerge} \text{bs} ([], [y]) = [y]
\]

\[
\text{bmerge} \text{bs} ([x], []) = [x]
\]

\[
\text{bmerge} \text{bs} ([y], [x]) = \text{bmerge} \text{bs} ([0], [y])
\]

\[
\text{bmerge} \text{bs} (xs, [y]) = \text{bmerge} \text{bs} (xs, [0])
\]

\[
\text{bmerge} (0::bs) (xs, ys) = y \cdot \text{bmerge} \text{bs} (xs, ys)
\]

\[
\text{bmerge} (1::bs) (xs, ys) = x \cdot \text{bmerge} \text{bs} (xs, ys)
\]

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

\[*\text{InfPair} > \text{bsplit} \ [0,1,0,1,0,1] \ [10,20,30,40,50,60]
\]
\(*\text{InfPair} > \text{bmerge} \ [0,1,0,1,0,1] \ [10,20,30,40,50,60]
\]

\[\text{bsplit} = \text{ns} \text{rs} \text{ls} = \text{bs} \text{as} \text{nats} \text{bins}
\]

\[\text{bs} = \text{as} \text{bins} \text{xEncoder x} = \text{as} \text{nats} \text{bins} \text{n}
\]

\[\text{ls,rs} = \text{bsplit} \text{bs} \text{as} \text{nats} \text{bins} \text{ns}
\]

\[\text{l} = \text{as} \text{nats} \text{bins} \text{ls}
\]

\[\text{r} = \text{as} \text{nats} \text{bins} \text{rs}
\]

\[\text{bpair k} = \text{genericPair set} \text{[0,k..]}
\]

\[\text{bunpair k} = \text{genericUnPair set} \text{[0,k..]}
\]

C. Defining pairing bijections, generically

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

\[
\text{genericUnpair} :: \text{Encoder t} \to \text{t} \to \text{N} \to ([N], [N]) \to [N]
\]

\[
\text{genericUnpair} \text{xs} = n \to \text{as} \text{nats} \text{bins} \text{xs}
\]

\[\text{as} = \text{as} \text{bins} \text{Encoder x}
\]

\[\text{as} = \text{as} \text{nats} \text{bins} \text{n}
\]

\[\text{bs} = \text{bsplit} \text{bs} \text{as} \text{nats} \text{bins} \text{ns}
\]

\[\text{ls,rs} = \text{bsplit} \text{bs} \text{as} \text{nats} \text{bins} \text{ns}
\]

\[\text{l} = \text{as} \text{nats} \text{bins} \text{ls}
\]

\[\text{r} = \text{as} \text{nats} \text{bins} \text{rs}
\]

Let us observe first that for termination of this functions depends on termination of the calls to \( \text{bsplit} \) and \( \text{bmerge} \), as illustrated by the following examples:

\[*\text{InfPair} > \text{genericPair bins (cycle [0])} \ (10,20)
\]
\(*\text{InfPair} > \text{genericUnPair bins (cycle [1])} \ 42
\]

\[\text{CInterrupted.}
\]

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

Clearly, this suggests restrictions on the acceptable characteristic functions.

We will now give sufficient conditions ensuring that the functions \( \text{genericUnpair} \) and \( \text{genericPair} \) 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 bloc of digits occurring in a characteristic function any (finite or infinite) contiguous sequence of digits. Note that an infinite bloc 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 bloc is larger than \( n \).

Proposition 6: If \( \{a_n\}_{n \in \mathbb{N}} \) is an infinite sequence of bits containing only finite blocks of 0 and 1 digits, \( \text{genericPair bins} \) and \( \text{genericUnPair bins} \) 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 \( \text{genericPair} \) and \( \text{genericUnPair} \), the functions \( \text{bmerge} \) and \( \text{bsplit} \) terminate.

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

\[
\text{bpair k} = \text{genericPair bins (cycle [1,0])}
\]

\[
\text{bunpair k} = \text{genericUnPair bins (cycle [1,0])}
\]

and working as follows:

\[*\text{InfPair} > \text{map} \text{bpair2} \ [0..10]
\]
\(*\text{InfPair} > \text{map} \text{bunpair2} \ [0..10]
\]

Proposition 7: If \( \{a_n\}_{n \in \mathbb{N}} \) is an infinite sequence of non-decreasing natural numbers, the functions \( \text{genericPair set} \) and \( \text{genericUnPair set} \) define a family of pairing bijections parameterized by \( \{a_n\}_{n \in \mathbb{N}} \).

Proof: Given that the sequence is non-decreasing, it represents canonically an infinite set such that its complement is also infinite, represented as a 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 \( \text{bpair k} \) and its inverse \( \text{bunpair k} \) are derived from a set representation (implicitly morphed into a characteristic function).

\[\text{bpair k} = \text{genericPair set} \text{[0,k..]}
\]

\[\text{bunpair k} = \text{genericUnPair set} \text{[0,k..]}
\]

Note that for \( k = 2 \) we obtain exactly the bijections \( \text{bpair2} \) and \( \text{bunpair2} \) derived previously, as illustrated by the following example:

\[*\text{InfPair} > \text{map} \text{bpair2} \ [0..10]
\]
\(*\text{InfPair} > \text{map} \text{bunpair2} \ [0..10]
\]

We conclude with a similar result for lists:

Proposition 8: If \( \{a_n\}_{n \in \mathbb{N}} \) is an infinite sequence of natural numbers only containing finite blocks of 0s, the functions \( \text{genericPair list} \) and \( \text{genericUnPair list} \) define a family of pairing bijections parameterized by \( \{a_n\}_{n \in \mathbb{N}} \).

Proof: It follows from Prop. 7 by observing that such sequences are transformed into infinite sets represented as non-decreasing sequences.

The Appendix discusses a few more examples of such pairing functions and visualizes a few space-filling curves associated to them.

Proposition 9: There are \( 2^n \) 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 bloc of 0 or 1 digits necessarily ends with the bloc. Therefore, by erasing the bloc 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 \( 2^n \).
IV. RELATED WORK

Pairing functions have been used in work on decision problems as early as [3], [8], [10]–[13]. There are about 19200 Google documents referring to the original “Cantor pairing function” among which we mention the surprising result that, together with the successor function it defines a decidable subset of arithmetic [2]. An extensive study of various pairing functions and their computational properties is presented in [1], [14]. They are also related to 2D-space filling curves (Z-order, Gray-code and Hilbert curves) [4]–[7]. 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 [4]–[7] and symbolic arbitrary length arithmetic computations [19]. Note also that $\text{bpair} 2$ and $\text{bunpair} 2$ are the same as the functions defined in [9] and also known as Morton-codes, with uses in indexing of spatial databases [6].

V. CONCLUSION

We have described 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 [16].

Given the space constraints, we have not explored the natural extensions to more general tupling / un tupling 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 / un tupling bijections and those defined between natural numbers and sequences, sets and multisets of natural numbers.

We hope that our adoption of the non-strict functional language Haskell (freely available from [haskell.org]), as a complement to conventional mathematical notation, enables the empirically curious reader to instantly validate our claims and encourage her/him to independently explore their premises and their consequences.

REFERENCES

[1] Patrick Cegielski and Denis Richard. On arithmetical first-order theories allowing encoding and decoding of lists. Theoretical Computer Science, 223(12):55 – 75, 1999.
[2] Patrick Cegielski and Denis Richard. Decidability of the Theory of the Natural Integers with the Cantor Pairing Function and the Successor. Theor. Comput. Sci., 257(1-2):51–77, 2001.
[3] Laszlo 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.
[4] J. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. In Brian Lings and Keith 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] Jonathan Lawder. The application of space-filling curves to the storage and retrieval of multi-dimensional data. 2000. PhD Thesis, University of London, UK.
[7] Bongki Moon, H.v. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. IEEE Transactions on Knowledge and Data Engineering, 13:124–141, 2001.
[8] Jozef Pepis. Ein verfahren der mathematischen logik. The Journal of Symbolic Logic, 3(2):61–76, jun 1938.
[9] Stephen Pigeon. Contributions à la compression de données. Ph.d. thesis, Université de Montréal, Montréal, 2001.
[10] Julia Robinson. General recursive functions. Proceedings of the American Mathematical Society, 1(6):703–718, dec 1950.
[11] Julia Robinson. A note on primitive recursive functions. Proceedings of the American Mathematical Society, 6(4):667–670, aug 1955.
[12] Julia Robinson. Finite generation of recursively enumerable sets. Proceedings of the American Mathematical Society, 19(6):1480–1486, dec 1968.
[13] Julia Robinson. Recursive functions of one variable. Proceedings of the American Mathematical Society, 19(4):815–820, aug 1968.
[14] Arnold L. Rosenberg. Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science, 14(1):3–17, 2003.
[15] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. 2010. published electronically at www.research.att.com/~njas/sequences.
[16] Paul 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.
[17] Paul Tarau. Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell, January 2009. Unpublished draft, http://arXiv.org/abs/0808.2953, updated version at http://logic.cse.unt.edu/tarau/research/2010/ISO.pdf, 150 pages.
[18] Paul Tarau. “Everything Is Everything” Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms. Complex Systems, (18):475–493, 2010.
[19] Paul Tarau and David 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, March 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 ( [16], [17]) 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 $g$.

$$X \xrightarrow{f = g^{-1}} Y$$

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')

id = id itself = Iso id id invert (Iso f g) = Iso g f

Assuming that for any pair of type $\text{Iso} a b$, $f \circ g = id_a$ 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.

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
```

The combinator “as” adds a convenient syntax such that converters between \( \lambda \) 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, list2mset :: [N] \rightarrow [N]
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
set2list, list2set :: [N] \rightarrow [N]
set2list = (map pred) . mset2list . (map succ)
list2list = (map pred) . list2mset . (map succ)
```

We will represent them as lists i.e. their Haskell type is `[N]`.

```
iso set2list list2set :: [N]
iso mset2list list2mset :: [N]
```

The combinator “as” provides an embedded transformation language for routing isomorphisms through two Encoders.

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

The function syracuse is used in an equivalent formulation of the Collatz conjecture. Interestingly, it can be computed using the nAdicTail which results after dividing a number \( n \) with \( \mu_2(n) \). Note that we derive our pairing function directly from the list representation of the range of this function as genericPair and genericUnpair implicitly construct the associated characteristic function.

```
syracuse :: N \rightarrow N
syracuse n = nAdicTail 2 (6*n+4)
syror 0 = [0]
syror n = n : syror (syracuse n)
syrnats = map syracuse [0..]
syrpair = genericPair list syrnats
syrunpair = genericUnpair list syrnats
```

Figures 1 and 2 show the “Z-order” (Morton code) path connecting successive values in the range of the function `bunpair 2` and `bunpair 3`. Figures 3 and 4 show the path connecting the values in the range of unpairing functions associated, respectively to the Syracuse function and the binary digits of \( \pi \). Interestingly, at a first glance, some regular patterns emerge even in the case of such notoriously irregular characteristic functions.

```
Fig. 1. Path connecting values of `bunpair 2`
```

Examples of pairing functions derived from characteristic functions

```
Syracuse function and the binary digits of \( \pi \). Interestingly, at a first glance, some regular patterns emerge even in the case of such notoriously irregular characteristic functions.
```
Fig. 2. Path connecting values of bunpair 3

Fig. 3. Path connecting values of an unpairing bijection based on the Syracuse function

Fig. 4. Path connecting values of an unpairing bijection based on binary digits of $\pi$