Two Mechanisms for Generating Infinite Families of Pairing Bijections

Paul Tarau[1]

[1]Department of Computer Science and Engineering
University of North Texas

RACS 2013

Pairing Bijections

Definition

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

- They 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.

Our families of pairing bijections

- like in the case of Cantor's $f(x,y) = \frac{1}{2}(x+y)(x+y+1)+y$, pairing bijections have been usually hand-crafted by putting to work geometric or arithmetic intuitions
- we introduce here two general mechanisms for building infinite families of pairing functions seen as instances of bijective data transformation mechanisms
  - deriving pairing bijections from n-adic valuations
  - deriving Pairing bijections from characteristic functions of subsets of $\mathbb{N}$
- why are pairing bijections useful? $\Rightarrow$ some applications:
  - Indexing multi-dimensional data
  - Distance search in GIS
  - Machine learning
  - Coding theory
  - Foundations of Computing Science
  - Recursion theory
  - Computability
  - Number Theory

Data Transformation Isomorphisms

- we implement an isomorphism between two objects X and Y as a Haskell data type encapsulating a bijection f and its inverse g
- for any pair of type `Iso a b`, $f \circ g = id_b$ and $g \circ f = id_a$

```
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
```

Data Transformations through a Hub

- 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
- we need to pick a representation that is relatively easily convertible to various others and scalable to accommodate large objects up to the runtime system's actual memory limits
- $\Rightarrow$ we 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]
```

An `Encoder` as an isomorphism connecting an object to the *Hub*

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

the combinator "`as`", provides 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:

```
from_A_to_B x = as B A x
from_B_to_A x = as A B x
```

Examples of `Encoders`

As `[N]` has been chosen as the root, the sequence data type *list* is simply the `Encoder` defined by the identity isomorphism on `[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)
```

A classic pairing function

A classic pairing function used in recursion theory around 1930 is:

$$f(x, y) = 2^x(2y + 1) - 1$$

- it is remarkable as it favors its first argument, which has an exponential impact on the result
- we will generalize this mechanism to obtain a family of bijections parameterized by an arbitrary base b instead of 2

n-adic valuations and a key arithmetic property

Definition

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

Proposition

$\forall b \in \mathbb{N}, b > 1, \forall y \in \mathbb{N}$ if $\exists q, m$ such that $b > m > 0, y = bq + m$, then there's exactly one pair (y', m'), $b - 1 > m' \geq 0$ such that $y' = (b-1)q + m'$ and the function associating (y', m') to (y, m) is a bijection.

A family of bijections between $\mathbb{N} \times \mathbb{N}$ and N^+

the functions `nAdicCons b` and `nAdicDeCons b`, form 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
```

Examples

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
```

The following examples illustrate their 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 $\nu_n(x)$.

The family of pairing functions parameterized by a base b

```
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]
```

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

For each base `b>1`, we 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
```

Defining the corresponding `Encoders`

- we can “reify” these bijections as `Encoders` between $\mathbb{N}$ and $]\mathbb{N}]$
- such Encoders can be “morphed” 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)
```

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

The following examples illustrate these operations,

```
*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]
```

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

For each $l, k \in \mathbb{N}$ we generate a family of permutations (bijections $f : \mathbb{N} \to \mathbb{N}$), parameterized by the pair `(l,k)`:

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

Examples:

```
*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]
```

Proposition

$$(nAdicBij\ k\ l) \circ (nAdicBij\ l\ k) \equiv id \qquad (1)$$

The bijection between lists and characteristic functions of sets of natural numbers

- 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
- we interpret each element of the list as the number of `0` digits before the next `1` digit
- The function `bin2list` converts a characteristic function represented as bitstrings back to a list of natural numbers
- infinite sequences are handled as well, resulting in infinite bitstrings

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

```
*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]
```

Characteristic functions and subsets

Proposition

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. $\square$

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]
```

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])
...
```

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]
...
```

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
```

An example: Morton codes

Morton codes 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]
```

A generalization of Morton codes

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]
```

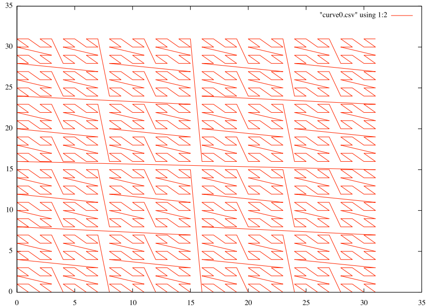

Figure : Path connecting values of `bunpair 2`

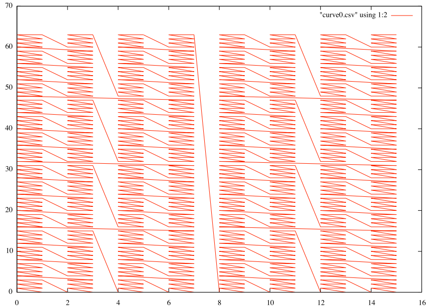

Figure : Path connecting values of bunpair 3

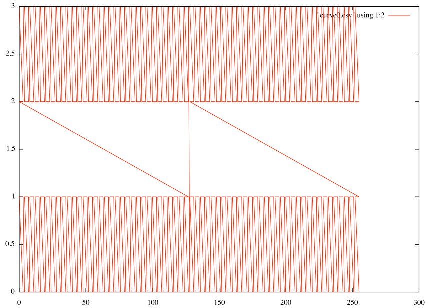

Figure : Path connecting values of an unpairing bijection based on binary digits of π

Conclusion

- we have shown two general mechanisms for generating infinite (countable and uncountable) families of pairing / unpairing bijections
- generalizations are possible to k-tupling / untupling bijections between $\mathbb{N}^k$ and $\mathbb{N}$
- we have made use of a simple but elegant data transformation framework to design our algorithms in a modular way
- $\Rightarrow$ convenient tools to “custom-tailor” such bijections for various applications