Ranking Catamorphisms and Unranking Anamorphisms on Hereditarily Finite Datatypes

– unpublished draft –

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cs.unt.edu

Abstract

Using specializations of *unfold* and *fold* on a generic tree data type we derive *unranking* and *ranking* functions providing natural number encodings for various Hereditarily Finite datatypes.

In this context, we interpret unranking operations as instances of a generic *anamorphism* and ranking operations as instances of the corresponding *catamorphism*.

Starting with Ackerman's Encoding from Hereditarily Finite Sets to Natural Numbers we define *pairings* and *finite tuple encodings* that provide building blocks for a theory of *Hereditarily Finite Functions*.

The more difficult problem of *ranking* and *unranking Hereditarily Finite Permutations* is then tackled using Lehmer codes and factoradics.

The self-contained source code of the paper, as generated from a literate Haskell program, is available at `http://logic.csci.unt.edu/tarau/research/2008/fFUN.zip`.

Keywords *ranking/unranking, pairing/tupling functions, Ackermann encoding, hereditarily finite sets, hereditarily finite functions, permutations and factoradics, computational mathematics, Haskell data representations*

1. Introduction

This paper is an exploration with functional programming tools of *ranking* and *unranking* problems on finite functions and bijections and their related hereditarily finite universes. The *ranking problem* for a family of combinatorial objects is finding a unique natural number associated to it, called its *rank*. The inverse *unranking problem* consists of generating a unique combinatorial object associated to each natural number.

The paper is organized as follows: section 2 introduces generic ranking/unranking framework parameterized by bijective transformers and terminating conditions based on *urelements*, section 3 introduces Ackermann's encoding and its inverse as instances of the framework. After discussing some classic pairing functions, section 4 introduces new pairing/unpairing and tuple operations on natural numbers and uses them for encodings of finite functions (section 5), resulting in encodings for "Hereditarily Finite Functions" (section 6). Ranking/unranking of permutations and Hereditarily Finite Permutations as well as Lehmer codes and factoradics are covered in section 7. Sections 8 and 9 discuss related work, future work and conclusions.

The paper is part of a larger effort to cover in a declarative programming paradigm, arguably more elegantly, some fundamental combinatorial generation algorithms along the lines of (Knuth 2006). The practical expressiveness of functional programming languages (in particular Haskell) are put at test in the process.

While the main focus of the paper was testdriving Haskell on the curvy tracks of non-trivial combinatorial generation problems, we have bumped, somewhat accidentally, into making a few new contributions to the field as such, that could be easily blamed on the quality of the vehicle we were testdriving:

1. the three ranking/unranking algorithms from finite functions to natural numbers are new
2. the universe of Hereditarily Finite Functions, as a functional analogue of the well known universe of Hereditarily Finite Sets is new
3. the universe of Hereditarily Finite Permutations, as an analogue of the well known universe of Hereditarily Finite Sets is new
4. the natural number tupling/untupling functions are new

5. the ranking/unranking algorithm for permutations of arbitrary sizes is new (although it is based on a known Lehmer code-based algorithm for permutations of fixed size)
6. the catamorphism/anamorphism view of ranking/unranking functions is new and it is likely to be reusable for various families of combinatorial generation problems

Through the paper, we will use the following set of primitive arithmetic functions:

```
double n = 2*n
half n = n `div` 2
exp2 n = 2^n
```

together with `succ`, `pred`, `even`, `odd` and `sum` Haskell functions, to emphasize that this subset is easily hardware implementable (by only using boolean operations, shifts and adders) and that these functions also have `O(log n)` or better software implementations for integers of (arbitrary) length `n`.

When possible, we will use point-free notations (unnecessary function arguments omitted) to emphasize the generic function composition dataflow.

As we have put significant effort to ensure that all our types can be inferred, we will omit type declarations, with apologies to the type-curious reader, who can have them displayed as needed, while interacting with the Haskell sources of the paper available online.

2. Generic Unranking and Ranking with Higher Order Functions

We will use, through the paper, a generic "rose tree" type `T` distinguishing between atoms tagged with `A`) and subforests (tagged with `F`).

```
data T a = A a | F [T a] deriving (Eq,Ord,Read,Show)
```

Atoms will be mapped to natural numbers in `[0..ulimit-1]`. When `ulimit` is fixed, we denote this set A. We denote Nat the set of natural numbers and T the set of trees of type T with atoms in A.

The unranking operation is seen here as an instance of a generic anamorphism mechanism *unfold*, while the ranking operation is seen as an instance of the corresponding catamorphism *fold* (Hutton 1999; Meijer and Hutton 1995).

Unranking As an adaptation of the *unfold* operation, natural numbers will be mapped to elements of T with a generic higher order function `unrank_ ulimit f`, defined from Nat to T, parameterized by the the natural number `ulimit` and the transformer function `f`:

```
unrank_ ulimit _ n | (n<ulimit) && (n≥0) = A n
unrank_ ulimit f n | n ≥ulimit =
  (F (map (unrank_ ulimit f) (f  (n-ulimit))))
```

A global constant `default_ulimit` will be used through the paper to fix the default range of atoms, allowing us to work with a default `unrank` function:

```
default_ulimit = 0

unrank = unrank_ default_ulimit
```

Ranking Similarly, as an adaptation of *fold*, generic inverse mappings `rank_ ulimit` and `rank`) from T to Nat are defined as:

```
rank_ ulimit _ (A n) | (n<ulimit) && (n≥0) = n
rank_ ulimit g (F ts) =
  ulimit+(g (map (rank_ ulimit g) ts))

rank = rank_ default_ulimit
```

Note that the guard in the second definition simply states correctness constraints ensuring that atoms belong to the same set A for `rank_` and `unrank_`. This ensures that the following holds:

PROPOSITION 1. *If the transformer function* $f : Nat \rightarrow [Nat]$ *is a bijection with inverse g, such that* $n \geq ulimit \wedge f(n) = [n_0, ...n_i, ...n_k] \Rightarrow n_i < n$, *then* `unrank` *is a bijection from* Nat *to* T, *with inverse* `rank` *and the recursive computations of both functions terminate in a finite number of steps.*

Proof: *by induction on the structure of* Nat *and* T, *using the fact that* `map` *preserves bijections.*

Ranking functions can be traced back to Gödel numberings (Gödel 1931; Hartmanis and Baker 1974) associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms (Martinez and Molinero 2003; Knuth 2006).

3. Hereditarily Finite Sets and Ackermann's Encoding

While the Universe of Hereditarily Finite Sets is best known as a model of the Zermelo-Fraenkel Set theory with the Axiom of Infinity replaced by its negation (Takahashi 1976; Meir et al. 1983), it has been the object of renewed practical interest in various fields, from representing structured data in databases (Leontjev and Sazonov 2000) to reasoning with sets and set constraints (Dovier et al. 2000; Piazza and Policriti 2004).

3.1 Ackermann's Encoding

The Universe of Hereditarily Finite Sets is built from the empty set (or a set of *Urelements*) by successively applying powerset and set union operations.

A surprising bijection, discovered by Wilhelm Ackermann in 1937 (Ackermann 1937; Abian and Lamacchia 1978; Kaye and Wong 2007) maps Hereditarily Finite Sets (HFS) to Natural Numbers (Nat):

$$f(x) = \texttt{if } x = \{\} \texttt{ then } 0 \texttt{ else } \sum_{a \in x} 2^{f(a)}$$

Assuming HFS extended with *Urelements* (objects not containing any elements) our generic "rose tree" representation can be used for Hereditarily Finite Sets, with *Urelements* seen as atoms, i.e. Natural Numbers in `[0..ulimit-1]`. The constructor `A a` marks *Urelements* of type `a` (usually the arbitrary length Integer type in Haskell) and the constructor `F` marks a list of recursively built HFS type elements. Note that if no elements are used with the `A` constructor, we obtain the "pure" HFS universe with everything built out from the empty set represented as `F []`.

Let us note that Ackermann's encoding can be seen as the recursive application of a bijection `set2nat` from finite subsets of Nat to Nat, that associates to a set of (distinct!) natural numbers a (unique!) natural number. With this representation, Ackermann's encoding from HFS to Nat `hfs2nat` can be expressed in terms of our generic `rank` function as:

```
hfs2nat = rank set2nat

set2nat ns = sum (map exp2 ns)
```

To obtain the inverse of the Ackerman encoding, let's first define the inverse `nat2set` of the bijection `set2nat`. It decomposes a natural number into a list of exponents of 2 (seen as bit positions equaling 1 in its bitstring representation, in increasing order).

```
nat2set n = nat2exps n 0 where
  nat2exps 0 _ = []
  nat2exps n x =
    if (even n) then xs else (x:xs) where
      xs=nat2exps (half n) (succ x)
```

The inverse of the Ackermann encoding, with urelements in `[0..default_ulimit-1]` and the empty set mapped to `F []` is defined as follows:

```
nat2hfs = unrank nat2set
```

This definition is motivated by the fact that `nat2hfs` and `hfs2nat` are obtained through recursive compositions of `nat2set` and `set2nat`, respectively. Generalizing the encoding mechanism to use other bijections with similar properties, naturally leads to the *anamorphism/catamorphism* view of `unrank/rank`.

The following proposition summarizes the results in this subsection:

PROPOSITION 2. *Given id = $\lambda x.x$, the following function equivalences hold:*

$$nat2set \circ set2nat \equiv id \quad (1)$$

$$set2nat \circ nat2set \equiv id \quad (2)$$

$$nat2hfs \circ hfs2nat \equiv id \quad (3)$$

$$hfs2nat \circ nat2hfs \equiv id \quad (4)$$

3.2 Combinatorial Generation as Iteration

Using the inverse of Ackermann's encoding, the infinite stream HFS can be generated simply by iterating over the infinite stream `[0..]`:

```
iterative_hfs_generator=map nat2hfs [0..]


take 5 iterative_hfs_generator
 [F [],F [F []],F [F [F []]],
    F [F [],F [F []]],F [F [F [F []]]]]
```

One can try out `nat2hfs` and its inverse `hfs2nat` and print out a canonical string representation of HFS with the `setShow` functions given in Appendix:

```
nat2hfs 42
  F [F [F []],F [F [],F [F []]],
     F [F [],F [F [F []]]]]
hfs2nat (nat2hfs 42)
  42
setShow 42
  "{{{}},{{},{{}}},{{},{{{}}}}}"
```

Note that `setShow n` will build a string representation of $n \in Nat$, implicitly "deforested" as a HFS with Urelements in `[0..default_ulimit-1]`. Figure 1 shows the directed acyclic graph obtained by merging shared nodes in the *rose tree* representation of the HFS associated to a natural number (with arrows pointing from sets to their elements).

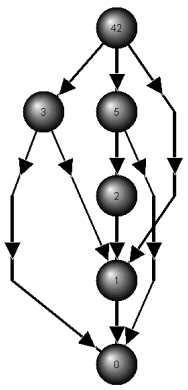

Figure 1: Hereditarily Finite Set associated to 42

4. Pairing Functions and Tuple Encodings

Pairings are bijective functions $Nat \times Nat \rightarrow Nat$. Following the classic notation for pairings of (Robinson 1950), given the pairing function J, its left and right inverses K and L are such that

$$J(K(z), L(z)) = z \tag{5}$$

$$K(J(x, y)) = x \tag{6}$$

$$L(J(x, y)) = y \tag{7}$$

We refer to (Cégielski and Richard 2001) for a typical use in the foundations of mathematics and to (Rosenberg 2002) for an extensive study of various pairing functions and their computational properties. We will start by overviewing two classic pairing functions.

4.1 Cantor's Pairing Function

Cantor's geometrically inspired pairing function (also present in earlier work by Cauchy) is defined as:

```
nat_cpair x y = (x+y)*(x+y+1) `div` 2+y
```

As the following example shows, it grows symmetrically in both arguments:

```
[nat_cpair i j|i<-[0..3],j<-[0..3]]
  [0,2,5,9,1,4,8,13,3,7,12,18,6,11,17,24]
```

4.2 The Pepis-Kalmar-Robinson Pairing Function

An interesting pairing function *asymmetrically growing, faster on the first argument*, is the function **pepis_J** and its left and right unpairing companions **pepis_K** and **pepis_L** that have been used, by Pepis, Kalmar and Robinson together with Cantor's functions, in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem in (Pepis 1938; Kalmar 1939; Kalmar, Laszlo and Suranyi, Janos 1947, 1950; Robinson 1950, 1955, 1968a,b, 1967). The function **pepis_J** combines two numbers reversibly by multiplying a power of 2 derived from the first and an odd number derived from the second:

$$f(x, y) = 2^x * (2 * y + 1) - 1 \tag{8}$$

Its Haskell implementation, together with its inverse is:

```
pepis_J x y  = pred ((exp2 x)*(succ (double y)))

pepis_K n = two_s (succ n)

pepis_L n = half (pred (no_two_s (succ n)))

two_s n | even n = succ (two_s (half n))
two_s _ = 0

no_two_s n = n `div` (exp2 (two_s n))
```

This pairing function (slower in the second argument) works as follows:

```
pepis_J 1 10
  41
pepis_J 10 1
  3071
[pepis_J i j|i<-[0..3],j<-[0..3]]
  [0,2,4,6,1,5,9,13,3,11,19,27,7,23,39,55]
```

As Haskell provides a built-in ordered pair, it is convenient to regroup `J, K, L` as mappings to/from built-in ordered pairs:

```
haskell2pepis (x,y) = pepis_J x y
pepis2haskell n = (pepis_K n,pepis_L n)
```

4.3 The BitMerge Pairing Function

We will introduce here an unusually simple pairing function (that we have found out recently as being the same as the one in defined in Steven Pigeon's PhD thesis on Data Compression (Pigeon 2001), page 114). It provides compact representations for various constructs involving ordered pairs.

The bijection `bitmerge_pair` from $Nat \times Nat$ to Nat and its inverse `bitmerge_unpair` are defined as follows:

```
bitmerge_pair (i,j) =
  set2nat ((evens i) ++ (odds j)) where
    evens x = map double (nat2set x)
    odds y = map succ (evens y)

bitmerge_unpair n = (f xs,f ys) where
  (xs,ys) = partition even (nat2set n)
  f = set2nat . (map half)
```

The function `bitmerge_pair` works by splitting a number's big endian bitstring representation into odd and even bits while its inverse `bitmerge_unpair` blends the odd and even bits back together. With help of the function `to_rbits` given in Appendix, that decomposes $n \in Nat$ into a list of bits (smaller units first) one can follow what happens, step by step:

```
to_rbits 2008
  [0,0,0,1, 1,0,1,1, 1,1,1]
bitmerge_unpair 2008
  (60,26)
to_rbits 60
  [0,0, 1,1, 1,1]
to_rbits 26
  [0,1, 0,1, 1]
bitmerge_pair (60,26)
  2008
```

PROPOSITION 3. *The following function equivalences hold:*

$$bitmerge\_pair \circ bitmerge\_unpair \equiv id \tag{9}$$

$$bitmerge\_unpair \circ bitmerge\_pair \equiv id \tag{10}$$

4.4 Tuple Encodings as Generalized BitMerge

We will now generalize this pairing function to k-tuples and then we will derive an encoding for finite functions.

The function `to_tuple`: $Nat \rightarrow Nat^k$ converts a natural number to a k-tuple by splitting its bit representation into k groups, from which the k members in the tuple are finally rebuilt. This operation can be seen as a transposition of a bit matrix obtained by expanding the number in base 2^k:

```
to_tuple k n = map from_rbits (
    transpose (
      map (to_maxbits k) (
        to_base (exp2 k) n
      )
    )
  )
```

To convert a k-tuple back to a natural number we will merge their bits, k at a time. This operation uses the transposition of a bit matrix obtained from the tuple, seen as a number in base 2^k, with help from bit crunching functions given in Appendix:

```
from_tuple ns = from_base (exp2 k) (
    map from_rbits (
      transpose (
        map (to_maxbits l) ns
      )
    )
  ) where
      k=genericLength ns
      l=max_bitcount ns
```

The following example shows the decoding of 42, its decomposition in bits (right to left), the formation of a 3-tuple and the encoding of the tuple back to 42.

```
to_rbits 42
  [0,1,0, 1,0,1]
to_tuple 3 42
  [2,1,2]
to_rbits 2
  [0,1]
to_rbits 1
  [1]
from_tuple [2,1,2]
  42
```

Fig. 2 shows multiple steps of the same decomposition, with shared nodes collected in a DAG. Note that cylinders represent markers on edges indicating argument positions, the cubes indicate leaf vertices (0,1) and the small pyramid indicates the root where the expansion has started.

The following proposition states that this tupling function is a generalization of `bitmerge_pair`

PROPOSITION 4. *The following function equivalences hold:*

$$bitmerge\_unpair\ n \equiv to\_tuple\ 2\ n \tag{11}$$

$$bitmerge\_pair\ (x,y) \equiv from\_tuple\ [x,y] \tag{12}$$

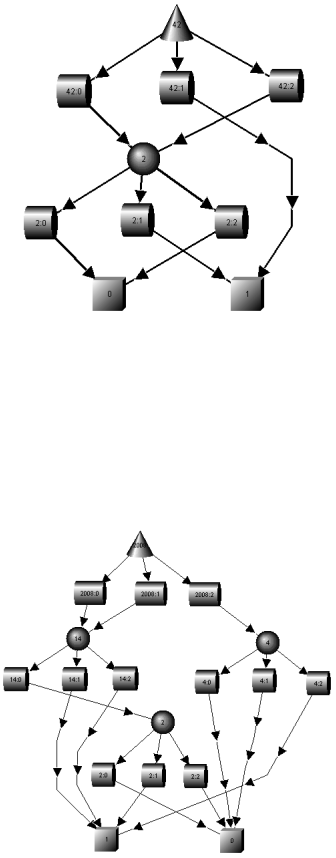

Figure 2: Repeated 3-tuple expansions: *42* and *2008*

5. Encoding Finite Functions

As finite sets can be put in a bijection with an initial segment of Nat we can narrow down the concept of finite function as follows:

DEFINITION 1. *A* `finite function` *is a function defined from an initial segment of* Nat *to* Nat*.*

This definition implies that a finite function can be seen as an array or a list of natural numbers except that we do not limit the size of the representation of its values.

5.1 Encoding Finite Functions as Tuples

We can now encode and decode a finite function from $[0..k-1]$ to Nat (seen as the list of its values), as a natural number:

```
ftuple2nat [] = 0
ftuple2nat ns = haskell2pepis (pred k,t) where
  k=genericLength ns
```

```
  t=from_tuple ns

nat2ftuple 0 = []
nat2ftuple kf = to_tuple (succ k) f where
  (k,f)=pepis2haskell kf
```

As the length of the tuple, k, is usually smaller than the number obtained by merging the bits of the k-tuple, we have picked the Pepis pairing function, exponential in its first argument and linear in its second, to embed the length of the tuple needed for the decoding. The encoding/decoding works as follows:

```
ftuple2nat [1,0,2,1,3]
  21295
nat2ftuple 21295
  [1,0,2,1,3]
map nat2ftuple [0..15]
  [[],[0,0],[1],[0,0,0],[2],[1,0],[3],
  [0,0,0,0],[4],[0,1],[5],[1,0,0],[6],
  [1,1],[7],[0,0,0,0,0]]
```

Note that

```
map nat2ftuple [0..]
```

provides an iterative generator for the stream of finite functions.

5.2 Deriving an Encoding of Finite Functions from Ackermann's Encoding

Given that a finite set with n elements can be put in a bijection with [0..n-1], a finite functions $f : [0..n-1] \to Nat$ can be represented as the list $[f(0)...f(n-1)]$. Such a list has however repeated elements. So how can we turn it into a set with distinct elements, bijectively?

The following two functions provide the answer.

First, we just sum up the list of the values of the function with `scanl`, resulting in a monotonically growing sequence (provided that we first increment every number by 1 to ensure that 0 values do not break monotonicity).

```
fun2set ns =
  map pred (tail (scanl (+) 0 (map succ ns)))
```

The inverse function reverting back from a set of distinct values collects the increments from a term to the next (and ignores the last one):

```
set2fun ns = map pred (genericTake l ys) where
  l=genericLength ns
  xs =(map succ ns)
  ys=(zipWith (-) (xs++[0]) (0:xs))
```

PROPOSITION 5. *The following function equivalences hold:*

$$nat2set \circ set2nat \equiv id \tag{13}$$

$$set2nat \circ nat2set \equiv id \tag{14}$$

The following example shows the conversion and its inverse.

```
fun2set [1,0,2,1,2]
  [1,2,5,7,10]
set2fun [1,2,5,7,10]
  [1,0,2,1,2]
```

By combining this with Ackermann encoding's basic step `set2nat` and its inverse `nat2set`, we obtain an encoding from finite functions to Nat follows:

```
nat2fun = set2fun . nat2set

fun2nat = set2nat . fun2set


nat2fun 2008
  [3,0,1,0,0,0,0]
fun2nat [3,0,1,0,0,0,0]
  2008
```

PROPOSITION 6. *The following function equivalences hold:*

$$nat2fun \circ fun2nat \equiv id \tag{15}$$

$$fun2nat \circ nat2fun \equiv id \tag{16}$$

One can see that this encoding ignores 0s in the binary representation of a number, while counting 1 sequences as increments. *Run Length Encoding* of binary sequences (Mkinen and Navarro 2005) encodes 0s and 1s symmetrically, by counting the numbers of 1s and 0s. This encoding is reversible, knowing that 1s and 0s alternate, and that the most significant digit is always 1:

```
bits2rle [] = []
bits2rle [_] = [0]
bits2rle (x:y:xs) | x==y = (c+1):cs where
  (c:cs)=bits2rle (y:xs)
bits2rle (_:xs) = 0:(bits2rle xs)

rle2bits [] = []
rle2bits (n:ns) =
  (genericReplicate (n+1) b) ++ xs where
    xs=rle2bits ns
    b=if []==xs then 1 else 1-(head xs)
```

By composing them with converters to/from bitlists, we obtain the bijection $nat2rle : Nat \to [Nat]$ and its inverse $rle2nat : [Nat] \to Nat$

```
nat2rle = bits2rle . to_rbits0

rle2nat = from_rbits . rle2bits

to_rbits0 0 = []
to_rbits0 n = to_rbits n
```

PROPOSITION 7. *The following function equivalences hold:*

$$nat2rle \circ rle2nat \equiv id \tag{17}$$

$$rle2nat \circ nat2rle \equiv id \tag{18}$$

6. Encodings for "Hereditarily Finite Functions"

One can now build a theory of "Hereditarily Finite Functions" (HFF) centered around using a transformer like `nat2ftuple`, `nat2fun`, `nat2rle` and its inverse `ftuple2nat`, `fun2nat`, `rle2nat` in way similar to the use of `nat2set` and `set2nat` for HFS, where the empty function (denoted `F []`) replaces the empty set as the quintessential *"urfunction"*. Similarly to Urelements in the HFS theory, "urfunctions" (considered here as atomic values) can be introduced as constant functions parameterized to belong to $[0..ulimit-1]$.

By using the generic `rank` function defined in section 2 we can extend the bijections defined in this section to encodings of Hereditarily Finite Functions. By instantiating the transformer function in `unrank` to `nat2ftuple`, `nat2fun` and `nat2rle` we obtain:

```
nat2hff = unrank nat2fun
nat2hff1 = unrank nat2ftuple
nat2hff2 =  unrank nat2rle
```

By instantiating the transformer function in `rank` we obtain:

```
hff2nat = rank fun2nat
hff2nat1 = rank ftuple2nat
hff2nat2 = rank rle2nat
```

The following examples show that `nat2hff`, `nat2hff1` and `nat2hff2` are indeed bijections, and that the resulting HFF-trees are typically more compact than the HFS-tree associated to the same natural number.

```
  F []
nat2hff 1
  F [F []]
nat2hff1 0
  F []
nat2hff1 1
  F [F [],F []]
nat2hff2 0
  F []
nat2hff2 1
  F [F []]

nat2hff 42
  F [F [F []],F [F []],F [F []]]
nat2hff1 42
  F [F [F [F [],F [],F []],F []]]
nat2hff2 42
  F [F [],F [],F [],F [],F [],F []]
nat2hfs 42
  F [F [F []],F [F [],F [F []]],
     F [F [],F [F [F []]]]]
  F [F [F []],F [F [],F [F []]],
     F [F [],F [F [F []]]]]

nat2hff 12345
  F [F [],F [F [F []]],F [],
     F [],F [F [F []],F []],F []]
nat2hff1 12345
  F [F [F [F [F [F [],F []]],
    F []]],F [F [],F [],F [F [],F []]]]
nat2hff2 12345
  F [F [],F [F []],F [F [],F []],
    F [F [],F [],F []],F [F []]]

hff2nat (nat2hff 12345)
  12345
hff2nat1 (nat2hff1 12345)
  12345
hff2nat2 (nat2hff2 12345)
  12345
```

Note that `map nat2hff [0..], nat2hff1 [0..], nat2hff1 [0..]` provide iterative generators for the (recursively enumerable!) stream of hereditarily finite functions.

The resulting HFF with urfunctions (seen as digits) can also be used as generalized *numeral systems* with applications to building arbitrary length integer implementations. Assuming `default_ulimit=10` we obtain:

```
nat2hff 1234567890
  F [A 3,A 2,A 0,A 1,A 7,
     A 0,A 1,A 2,A 0,A 2,A 2
  ]
nat2hff1 1234567890
  F [F [F [F [F [A 0,A 3]],
     F [F [F [A 2,A 0,A 1]]],A 1]]
  ]
nat2hff2 1234567890
  F [A 2,A 0,A 1,A 1,A 0,A 0,A 6,A 1,
     A 0,A 0,A 1,A 1,A 1,A 0,A 1,A 0
  ]
```

which display with the `funShow` functions given in Appendix as:

```
funShow 1234567890
  "(3 2 0 1 7 0 1 2 0 2 2)"
funShow1 1234567890
  "(((((0 3)) (((2 0 1))) 1)))"
funShow2 1234567890
  "(2 0 1 1 0 0 6 1 0 0 1 1 1 0 1 0)"
```

PROPOSITION 8. *The following function equivalences hold:*

$$nat2hff1 \circ hff2nat1 \equiv id \tag{19}$$

$$hff2nat1 \circ nat2hff1 \equiv id \tag{20}$$

$$nat2hff \circ hff2nat \equiv id \tag{21}$$

$$hff2nat \circ nat2hff \equiv id \tag{22}$$

7. Encoding Finite Bijections

To obtain an encoding for finite bijections (permutations) we will first review a ranking/unranking mechanism for permutations that involves an unconventional numeric representation, *factoradics*.

7.1 The Factoradic Numeral System

The factoradic numeral system (Knuth 1997) replaces digits multiplied by power of a base N with digits that multiply successive values of the factorial of N. In the increasing order variant `fr` the first digit d_0 is 0, the second is $d_1 \in \{0, 1\}$ and the N-th is $d_N \in [0..N-1]$. The left-to-right, decreasing order variant `fl` is obtained by reversing the digits of `fr`.

```
fr 42
  [0,0,0,3,1]
rf [0,0,0,3,1]
  42
fl 42
  [1,3,0,0,0]
lf [1,3,0,0,0]
  42
```

The function `fr` handles the special case for 0 and calls `fr1` which recurses and divides with increasing values of N while collecting digits with `mod`:

```
-- factoradics of n, right to left
fr 0 = [0]
fr n = f 1 n where
   f _ 0 = []
   f j k = (k `mod` j) :
           (f (j+1) (k `div` j))
```

The function `fl`, with digits left to right is obtained as follows:

```
fl = reverse . fr
```

The function `lf` (inverse of `fl`) converts back to decimals by summing up results while computing the factorial progressively:

```
rf ns = sum (zipWith (*) ns factorials) where
  factorials=scanl (*) 1 [1..]
```

Finally, `lf`, the inverse of `fl` is obtained as:

```
lf = rf . reverse
```

7.2 Ranking and unranking permutations of given size with Lehmer codes and factoradics

The Lehmer code of a permutation f is defined as the number of indices j such that $1 \leq j < i$ and $f(j) < f(i)$ (Mantaci and Rakotondrajao 2001).

PROPOSITION 9. *The Lehmer code of a permutation determines the permutation uniquely.*

The function `perm2nth` computes a `rank` for a permutation `ps` of `size>0`. It starts by first computing its Lehmer code `ls` with `perm2lehmer`. Then it associates a unique natural number `n` to `ls`, by converting it with the function `lf` from factoradics to decimals. Note that the Lehmer code `Ls` is used as the list of digits in the factoradic representation.

```
perm2nth ps = (l,lf ls) where
  ls=perm2lehmer ps
  l=genericLength ls

perm2lehmer [] = []
perm2lehmer (i:is) = l:(perm2lehmer is) where
  l=genericLength [j|j←is,j<i]
```

The function `nat2perm` provides the matching *unranking* operation associating a permutation `ps` to a given `size>0` and a natural number `n`.

```
-- generates n-th permutation of given size
nth2perm (size,n) =
  apply_lehmer2perm (zs++xs) [0..size-1] where
    xs=fl n
    l=genericLength xs
    k=size-l
    zs=genericReplicate k 0

-- converts Lehmer code to permutation
lehmer2perm xs = apply_lehmer2perm xs is where
  is=[0..(genericLength xs)-1]

-- extracts permutation from factoradic "digit" list
apply_lehmer2perm [] [] = []
apply_lehmer2perm (n:ns) ps@(x:xs) =
   y : (apply_lehmer2perm ns ys) where
   (y,ys) = pick n ps

pick i xs = (x,ys++zs) where
  (ys,(x:zs)) = genericSplitAt i xs
```

Note also that `lehmer2perm` is used this time to reconstruct the permutation `ps` from its Lehmer code, which in turn is computed from the permutation's factoradic representation.

One can try out this bijective mapping as follows:

```
nth2perm (5,42)
  [1,4,0,2,3]
perm2nth [1,4,0,2,3]
  (5,42)
nth2perm (8,2008)
  [0,3,6,5,4,7,1,2]
```

```
perm2nth [0,3,6,5,4,7,1,2]
  (8,2008)
```

7.3 A bijective mapping from permutations to Nat

One more step is needed to to extend the mapping between permutations of a given length to a bijective mapping from/to Nat: we will have to "shift towards infinity" the starting point of each new bloc of permutations in Nat as permutations of larger and larger sizes are enumerated.

First, we need to know by how much - so we compute the sum of all factorials up to $n!$.

```
-- fast computation of the sum of all factorials up to n!
sf n = rf (genericReplicate n 1)
```

This is done by noticing that the factoradic representation of [0,1,1,..] does just that. The stream of all such sums can now be generated as usual:

```
sfs = map sf [0..]
```

What we are really interested into, is decomposing `n` into the distance to the last sum of factorials smaller than `n`, `n_m` and the its index in the sum, `k`.

```
to_sf n = (k,n-m) where
  k=pred (head [x|x<-[0..],sf x>n])
  m=sf k
```

Unranking of an arbitrary permutation is now easy - the index `k` determines the size of the permutation and `n-m` determines the rank. Together they select the right permutation with `nth2perm`.

```
nat2perm 0 = []
nat2perm n = nth2perm (to_sf n)
```

Ranking of a permutation is even easier: we first compute its size and its rank, then we shift the rank by the sum of all factorials up to its size, enumerating the ranks previously assigned.

```
perm2nat ps = (sf l)+k where
  (l,k) = perm2nth ps


nat2perm 2008
  [1,4,3,2,0,5,6]
perm2nat [1,4,3,2,0,5,6]
  2008
```

As finite bijections are faithfully represented by permutations, this construction provides a bijection from Nat to the set of Finite Bijections.

PROPOSITION 10. *The following function equivalences hold:*

$$nat2perm \circ perm2nat \equiv id \equiv perm2nat \circ nat2perm \quad (23)$$

The stream of all finite permutations can now be generated as usual:

```
perms = map nat2perm [0..]
```

7.4 Hereditarily Finite Permutations

By using the generic `unrank` and `rank` functions defined in section 2 we can extend the `nat2perm` and `perm2nat` to encodings of Hereditarily Finite Permutations (HFP).

```
nat2hfp = unrank nat2perm
hfp2nat = rank perm2nat
```

The encoding works as follows:

```
nat2hfp 42
  F [F [],F [F [],F [F []]],
    F [F [F []],F [],F [F []],
    F [F [],F [F []],F [F [],F [F []]]]]
hfp2nat it
  42
```

Assuming `default_ulimit=10` and using the string representation provided by permShow (Appendix) we obtain:

```
nat2hfp 42
  F [F [],A 2,A 3,A 1,A 4]
permShow 42
  "(0 2 3 1 4)"
permShow 1234567890
  "(1 6 (0 1 3 2) 2 0 3 (0 1 2 3)
    7 8 5 9 4 (0 2 1 3))"
```

PROPOSITION 11. *The following function equivalences hold:*

$$nat2hfp \circ hfp2nat \equiv id \equiv hfp2nat \circ nat2hfp \quad (24)$$

8. Related work

Natural Number encodings of Hereditarily Finite Sets have triggered the interest of researchers in fields ranging from Axiomatic Set Theory and Foundations of Logic to Complexity Theory and Combinatorics (Takahashi 1976; Kaye and Wong 2007; Kirby 2007; Abian and Lamacchia 1978; Booth 1990; Meir et al. 1983; Leontjev and Sazonov 2000; Sazonov 1993; Avigad 1997). Computational and Data Representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in (Dovier et al. 2000; Piazza and Policriti 2004; Paulson 1994). Pairing functions have been used work on decision problems as early as (Pepis 1938; Kalmar 1939; Robinson 1950, 1968b). The tuple functions we have used to encode finite functions are new. While finite functions have been used extensively in various branches of mathematics and computer science, we have not seen any formalization of hereditarily Finite Functions or Hereditarily Finite Bijections as such in the literature.

9. Conclusion and Future Work

We have shown the expressiveness of Haskell as a metalanguage for executable mathematics, by describing natural number encodings, tupling/untupling and ranking/unranking functions for finite sets, functions and permutations and by

extending them in a generic way to Hereditarily Finite Sets, Hereditarily Finite Functions and Hereditarily Finite Permutations.

In a Genetic Programming context (Koza 1992; Poli et al.), the bijections between bitvectors/natural numbers on one side, and trees/graphs representing HFSs, HFFs, HPPs on the other side, suggest exploring the mapping and its action on various transformations as a phenotype-genotype connection.

We also foresee interesting applications in cryptography and steganography. For instance, in the case of the permutation related encodings - something as simple as the order of the cities visited or the order of names on a greetings card, seen as a permutation with respect to their alphabetic order, can provide a steganographic encoding/decoding of a secret message by using functions like `nat2perm` and `perm2nat`. It looks like an interesting topic to investigate if higher density and more random looking steganographic loads could be incorporated on top of Hereditarily Finite Permutations.

References

Alexander Abian and Samuel Lamacchia. On the consistency and independence of some set-theoretical constructs. *Notre Dame Journal of Formal Logic*, X1X(1):155–158, 1978.

Wilhelm Friedrich Ackermann. Die Widerspruchsfreiheit der allgemeinen Mengenlhere. *Mathematische Annalen*, (114):305–315, 1937.

Jeremy Avigad. The Combinatorics of Propositional Provability. In *ASL Winter Meeting*, San Diego, January 1997.

David Booth. Hereditarily Finite Finsler Sets. *J. Symb. Log.*, 55(2): 700–706, 1990.

Patrick Cégielski 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.

Agostino Dovier, Carla Piazza, and Alberto Policriti. Comparing Expressiveness of Set Constructor Symbols. In *Frontiers of Combining Systems*, pages 275–289, 2000.

K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

Juris Hartmanis and Theodore P. Baker. On simple goedel numberings and translations. In Jacques Loeckx, editor, *ICALP*, volume 14 of *Lecture Notes in Computer Science*, pages 301–316. Springer, 1974. ISBN 3-540-06841-4. URL `http://dblp.uni-trier.de/db/conf/icalp/icalp74.html#HartmanisB74`.

Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.*, 9(4):355–372, 1999.

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. ISSN 0022-4812.

Kalmar, Laszlo and Suranyi, Janos. On the reduction of the decision problem. *The Journal of Symbolic Logic*, 12(3):65–73, sep 1947. ISSN 0022-4812.

Kalmar, Laszlo and Suranyi, Janos. On the reduction of the decision problem: Third paper. pepis prefix, a single binary predicate. *The Journal of Symbolic Logic*, 15(3):161–173, sep 1950. ISSN 0022-4812.

Richard Kaye and Tin Lock Wong. On Interpretations of Arithmetic and Set Theory. *Notre Dame J. Formal Logic Volume*, 48 (4):497–510, 2007.

Laurence Kirby. Addition and multiplication of sets. *Math. Log. Q.*, 53(1):52–65, 2007.

Donald Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼/knuth/taocp.html.

Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0201896842. URL `http://portal.acm.org/citation.cfm?id=270146`.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

Alexander Leontjev and Vladimir Yu. Sazonov. Capturing LOGSPACE over Hereditarily-Finite Sets. In Klaus-Dieter Schewe and Bernhard Thalheim, editors, *FoIKS*, volume 1762 of *Lecture Notes in Computer Science*, pages 156–175. Springer, 2000. ISBN 3-540-67100-5.

Roberto Mantaci and Fanja Rakotondrajao. A permutations representation that knows what "eulerian" means. *Discrete Mathematics & Theoretical Computer Science*, 4(2):101–108, 2001.

Conrado Martinez and Xavier Molinero. Generic algorithms for the generation of combinatorial objects. In Branislav Rovan and Peter Vojtas, editors, *MFCS*, volume 2747 of *Lecture Notes in Computer Science*, pages 572–581. Springer, 2003. ISBN 3-540-40671-9. URL `http://dblp.uni-trier.de/db/conf/mfcs/mfcs2003.html#MartinezM03`.

Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *FPCA*, pages 324–333, 1995.

Amram Meir, John W. Moon, and Jan Mycielski. Hereditarily Finite Sets and Identity Trees. *J. Comb. Theory, Ser. B*, 35(2): 142–155, 1983.

Veli Mkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *CPM*, volume 3537 of *Lecture Notes in Computer Science*, pages 45–56. Springer, 2005. ISBN 3-540-26201-6. URL `http://dblp.uni-trier.de/db/conf/cpm/cpm2005.html#MakinenN05`.

Lawrence C. Paulson. A Concrete Final Coalgebra Theorem for ZF Set Theory. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 120–139. Springer, 1994. ISBN 3-540-60579-7.

Jozef Pepis. Ein verfahren der mathematischen logik. *The Journal of Symbolic Logic*, 3(2):61–76, jun 1938. ISSN 0022-4812.

Carla Piazza and Alberto Policriti. Ackermann Encoding, Bisimulations, and OBDDs. *TPLP*, 4(5-6):695–718, 2004.

Stephen Pigeon. *Contributions à la compression de données.* Ph.d. thesis, Université de Montréal, Montréal, 2001.

Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. *A Field Guide to Genetic Programming*. URL `http://www.gp-field-guide.org.uk`. e-book.

Julia Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950. ISSN 0002-9939.

Julia Robinson. A note on primitive recursive functions. *Proceedings of the American Mathematical Society*, 6(4):667–670, aug 1955. ISSN 0002-9939.

Julia Robinson. An introduction to hyperarithmetical functions. *The Journal of Symbolic Logic*, 32(3):325–342, sep 1967. ISSN 0022-4812.

Julia Robinson. Recursive functions of one variable. *Proceedings of the American Mathematical Society*, 19(4):815–820, aug 1968a. ISSN 0002-9939.

Julia Robinson. Finite generation of recursively enumerable sets. *Proceedings of the American Mathematical Society*, 19(6): 1480–1486, dec 1968b. ISSN 0002-9939.

Arnold L. Rosenberg. Efficient pairing functions - and why you should care. In *IPDPS*. IEEE Computer Society, 2002. ISBN 0-7695-1573-8.

Vladimir Yu. Sazonov. Hereditarily-Finite Sets, Data Bases and Polynomial-Time Computability. *Theor. Comput. Sci.*, 119(1): 187–214, 1993.

Moto-o Takahashi. A Foundation of Finite Mathematics. *Publ. Res. Inst. Math. Sci.*, 12(3):577–708, 1976.

A. Appendix

To make the code in the paper fully self contained, we list here some auxiliary functions.

String Representations The functions `setShow` and `funShow` provide a string representation of a natural number as a "pure" HFS or HFF. They are obtained as instances of `gshow` which provides a generic template parameterized with syntactic elements.

```
setShow = (gshow "{" "," "}") . nat2hfs
funShow = (gshow "(" " " ")") . nat2hff
funShow1 = (gshow "(" " " ")") . nat2hff1
funShow2 = (gshow "(" " " ")") . nat2hff2
permShow = (gshow "(" " " ")") . nat2hfp

gshow _ _ _ (A n) = show n
gshow l _ r (F []) =
  -- empty function shown as 0 rather than ()
  if default_ulimit > 1 then "0" else l++r
gshow l c r (F ns) = l++
  foldl (++) ""
    (intersperse c (map (gshow l c r) ns))
  ++r
```

Bit crunching functions The function bitcount computes the number of bits needed to represent an integer and max_bitcount computes the maximum bitcount for a list of integers.

```
bitcount n = head [x|x←[1..],(exp2 x)>n]
max_bitcount ns = foldl max 0 (map bitcount ns)
```

The following functions implement conversion operations between bitlists and numbers. Note that our bitlists represent binary numbers by selecting exponents of 2 in increasing order (i.e. "right to left").

```
-- from decimals to binary as list of bits
to_rbits n = to_base 2 n

-- from bits to decimals
from_rbits bs = from_base 2 bs

-- to binary, padded with 0s, up to maxbits
to_maxbits maxbits n =
  bs ++ (genericTake (maxbits-l)) (repeat 0) where
    bs=to_base 2 n
    l=genericLength bs

-- conversion to base n, as list of digits
to_base base n = d :
  (if q==0 then [] else (to_base base q)) where
    (q,d) = quotRem n base

-- conversion from any base to decimal
from_base base [] = 0
from_base base (x:xs) = x+base*(from_base base xs)
```