Ranking and Unranking of Hereditarily Finite Functions and Permutations

Paul Tarau

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

Abstract. Prolog’s ability to return multiple answers on backtracking provides an elegant mechanism to derive reversible encodings of combinatorial objects as Natural Numbers i.e. ranking and unranking functions. Starting from a generalization of Ackerman’s encoding of Hereditarily Finite Sets with Urelements and a novel tupling/untupling operation, we derive encodings for Finite Functions and use them as building blocks for an executable 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 paper is organized as a self-contained literate Prolog program available at http://logic.csci.unt.edu/tarau/research/2008/pHFF.zip.

Keywords: logic programming and computational mathematics, ranking/unranking, tupling/untupling functions, Ackermann encoding, hereditarily finite sets, hereditarily finite functions, hereditarily finite permutations, encodings of permutations, factoradics

1 Introduction

This paper is an exploration with logic programming tools of ranking and unranking problems on finite functions and bijections and their related hereditarily finite universes. The practical expressiveness of logic programming languages (in particular Prolog) are put at test in the process. 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 [13].

The paper is organized as follows: section 2 introduces generic ranking/unranking functions, section 3 introduces Ackermann’s encoding in the more general case when urelements are present. Section 4 introduces new tupling/untupling 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.

We will assume that the underlying Prolog system supports the usual higher order function-style predicates call/N, findall/3, maplist/N, sumlist/2 or their semantic equivalents and a few well known library predicates, used mostly
for list processing and arithmetics. Arbitrary length integers are needed for some
of the larger examples but their absence does not affect the correctness of the
code within the integer range provided by a given Prolog implementation. Other-
wise, the code in the paper, embedded in a literate programming LaTeX file,
is self contained and runs under SWI-Prolog. Note also that a few utility predi-
cates, not needed for following the main ideas of the paper, are left out from the
narrative and provided in the Appendix.

2 Generic Unranking and Ranking with Higher Order
Functions

We will use, through the paper, a generic multiway tree type distinguishing be-
tween atoms represented as (arbitrary length) integers and subforests represented
as Prolog lists. Atoms will be mapped to natural numbers in \([0..\text{ulimit}-1]\).
Assuming that \text{ulimit} is fixed, we denote \(A\) the set \([0..\text{ulimit}-1]\). We denote
\(\text{Nat}\) the set of natural numbers and \(T\) the set of trees of type \(T\) with atoms in
\(A\).

Definition 1 A ranking function on \(T\) is a bijection \(T \rightarrow \text{Nat}\). An unranking
function is a bijection \(\text{Nat} \rightarrow T\).

Ranking functions can be traced back to Gödel numberings \([7, 8]\) associated to
formulae. However, Gödel numberings are typically only injective functions, as
their use in the proofs of Gödel’s incompleteness theorems only requires injective
mappings from well-formed formulae to numbers. Together with their inverse
unranking functions they are also used in combinatorial and uniform random
instance generation \([18, 13]\) algorithms.

2.1 Unranking

As an adaptation of the unfold operation \([9, 19]\), elements of \(T\) will be mapped
to natural numbers with a generic higher order function \text{unrank}, parameterized
by the the natural number \text{ulimit} and the transformer function \(F\):

\[
\text{unrank}_{(\text{ulimit}, F, N, R)} :- \text{N} > 0, \text{N} < \text{ulimit}, !, \text{R} = \text{N}.
\]

\[
\text{unrank}_{(\text{ulimit}, F, N, R)} :- \text{N} = \text{ulimit},
\text{NO is N-ulimit},
\text{call}(F, \text{NO}, \text{Ns}),
\text{maplist(unrank}_{(\text{ulimit}, F), \text{Ns}, \text{R})}.
\]

A global constant provided by the predicate \text{default_ulimit}, will be used
through the paper to fix the default range of atoms as well as a default \text{unrank}
function: Note also that we will use a syntactically more convenient DCG nota-
tion, as \text{default_ulimit} will act as a modifier for functional style predicates,
composed by chaining their arguments automatically with Prolog’s DCG trans-
formation:
2.2 Ranking

Similarly, as an adaptation of fold, generic inverse mappings \( \text{rank}_(Ulimit,G) \) and \( \text{rank} \) from \( T \) to \( Nat \) are defined as:

\[
\text{rank}_(Ulimit,_,N,R):-\text{integer}(N),\text{N}>=0,\text{N}<Ulimit,!,\text{R}=N.
\]

\[
\text{rank}_(Ulimit,G,Ts,R):-\text{maplist}([\text{rank}_(Ulimit,G)],Ts,T),\text{call}(G,T,R0),R = R0+Ulimit.
\]

\[
\text{rank}(G)\rightarrow\text{default_ulimit}(Ulimit),\text{rank}_(Ulimit,G).
\]

Note that the guard in the second definition simply states correctness constraints ensuring that atoms belong to the same set \( A \) for \( \text{rank} \) and \( \text{unrank} \). This ensures that the following holds:

**Proposition 1** If the transformer function \( F : Nat \rightarrow [Nat] \) is a bijection with inverse \( G \), such that \( n \geq \text{ulimit} \land F(n) = [n_0,...,n_i,...,n_k] \Rightarrow n_i < n \), then \( \text{unrank} \) is a bijection from \( Nat \) to \( T \), with inverse \( \text{rank} \) 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 \( \text{maplist} \) preserves bijections.

3 Hereditarily Finite Sets and Ackermann’s Encoding

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 [32, 20]. In a Logic Programming framework, it has been used for reasoning with sets, set constraints, hypersets and bisimulations [6, 24].

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.

Ackermann’s encoding [2,1,11] is a bijection that maps Hereditarily Finite Sets (\( HFS \)) to Natural Numbers (\( Nat \)) as follows:

\[
f(x) = \text{if } x = \{\} \text{ then } 0 \text{ else } \sum_{a \in x} 2^{f(a)}
\]

Assuming \( HFS \) extended with *Urelements* (atomic objects not having any elements) our generic tree representation can be used for Hereditarily Finite Sets.

Ackermann’s encoding can be seen as the recursive application of a bijection \( \text{set2nat} \) from finite subsets of \( Nat \) to \( Nat \), that associates to a set of (distinct!) natural numbers a (unique!) natural number.
set2nat(Xs,N):-set2nat(Xs,0,N).

set2nat([],R,R).

set2nat([X|Xs],R1,Rn):-R2 is R1+(1<<X),set2nat(Xs,R2,Rn).

With this representation, Ackermann’s encoding from HFS to Nat hfs2nat can be expressed in terms of our generic rank function as:

\[ \text{hfs2nat} \rightarrow \text{default_ulimit(Ulimit)}, \text{hfs2nat_}(Ulimit). \]

\[ \text{hfs2nat_}(Ulimit) \rightarrow \text{rank_}(Ulimit, \text{set2nat}). \]

where the constant provided by default ulimit controls the segment \([0..Ulimit-1]\) of Nat to be mapped to urelements. The default value 0 defines “pure” sets, all built from the empty set only.

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

\[ \text{nat2set}(N,Xs):=\text{nat2elements}(N,Xs,0). \]

\[ \text{nat2elements}(0,\[],_K). \]

\[ \text{nat2elements}(N,\text{NewEs},K1):=-N>0, \]
\[ B \text{ is } \land (N,1), N1 \text{ is } N>>1, K2 \text{ is } K1+1, \text{add_el}(B,K1,\text{Es},\text{NewEs}), \]
\[ \text{nat2elements}(N1,\text{Es},K2). \]

\[ \text{add_el}(0,_,\text{Es}). \]
\[ \text{add_el}(1,K,\text{Es},[K|\text{Es}]). \]

The inverse of the Ackermann encoding, with urelements in \([0..Ulimit-1]\) and Ulimit mapped to \([\]) follows:

\[ \text{nat2hfs_}(Ulimit) \rightarrow \text{unrank_}(Ulimit, \text{nat2set}). \]

\[ \text{nat2hfs} \rightarrow \text{default_ulimit(Ulimit)}, \text{nat2hfs_}(Ulimit). \]

Using an equivalent functional notation, the following proposition summarizes the results in this subsection:

**Proposition 2** Given \(id = \lambda x.x\), the following function equivalences hold:

\[ \text{nat2set} \circ \text{set2nat} \equiv id \equiv \text{set2nat} \circ \text{nat2set} \quad (1) \]

\[ \text{nat2hfs} \circ \text{hfs2nat} \equiv id \equiv \text{hfs2nat} \circ \text{nat2hfs} \quad (2) \]

4 Pairing Functions and Tuple Encodings

Pairings are bijective functions \(Nat \times Nat \rightarrow Nat\). We refer to [5] for a typical use in the foundations of mathematics and to [30] for an extensive study of various pairing functions and their computational properties.
4.1 The Pepis-Kalmar-Robinson Pairing Function

The predicates `pepis_pair/3` and `pepis_unpair/3` are derived from 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 in some fundamental work on recursion theory, decidability and Hilbert’s Tenth Problem in [23, 10, 28]:

\[
\text{pepis_pair}(X,Y,Z) :- \text{pepis}_J(X,Y,Z).
\]

\[
\text{pepis_unpair}(Z,X,Y) :- \text{pepis}_K(Z,X), \text{pepis}_L(Z,Y).
\]

\[
\text{pepis}_J(X,Y,Z) :- Z \text{ is } ((1<<X)\times((Y<<1)+1))-1.
\]

\[
\text{pepis}_K(Z,X) :- Z1 \text{ is } Z+1, \text{two_s}(Z1,X).
\]

\[
\text{pepis}_L(Z,Y) :- Z1 \text{ is } Z+1, \text{no_two_s}(Z1,N), \text{Y is } (N-1)>>1.
\]

\[
\text{two_s}(N,R) :- \text{even}(N), !, \text{H is } N>>1, \text{two_s}(H,T), \text{R is } T+1.
\]

\[
\text{two_s}(_,0).
\]

\[
\text{no_two_s}(N,R) :- \text{two_s}(N,T), \text{R is } N // (1<<T).
\]

\[
\text{even}(X) :- 0 ::= \backslash(1,X).
\]

This pairing function given by the formula

\[
f(x,y) = 2^x \times (2 \times y + 1) - 1
\]

is asymmetrically growing, faster on the first argument. It works as follows:

?- `pepis_pair`(1,10,R).
R = 41.

?- `pepis_pair`(10,1,R).
R = 3071.

?- `findall`(R, (`between`(0,3,A), `between`(0,3,B), `pepis_pair`(A,B,R)), Rs).
Rs=[0, 2, 4, 6, 1, 5, 9, 13, 3, 11, 19, 27, 7, 23, 39, 55]

4.2 Tuple Encodings

We will now generalize pairing functions 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 \(2^k\):

\[
to\_tuple(K,N, Ns):-
\]

\[
\text{Base is } 1<<K, \text{to_base}(\text{Base},N,Ds), \text{maplist}(\text{to_maxbits}(K),Ds,Bss), \\
\text{mtranspose}(Bss,Xss),
\]

\[
\text{maplist}(\text{from_rbits},Xss,Ns).
\]
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:

\[
\text{from\_tuple}(Nss,R):=
\begin{align*}
&\max\_\text{bitcount}(Nss,L), \text{length}(Nss,K), \text{maplist}(\text{to\_maxbits}(L), Nss, Mss), \\
&\text{mtranspose}(Mss, Tss), \\
&\text{maplist}(\text{from\_rbits}, Tss, Ts), \text{Base is } 1\ll K, \text{from\_base}(\text{Base}, Ts, R).
\end{align*}
\]

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

\[
\begin{align*}
?-& \text{to\_tuple}(3,42,T), \text{to\_rbits}(2,Bs2), \text{to\_rbits}(1,Bs1), \text{from\_tuple}(T,N).
\end{align*}
\]

\[
\begin{align*}
T &= [2, 1, 2], \\
Bs2 &= [0, 1], \\
Bs1 &= [1], \\
N &= 42
\end{align*}
\]

Fig. 1 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.

![Fig. 1: 42 after repeated 3-tuple expansions](image)

Note that one can now define pairing functions as instances of the tupling functions:

\[
\begin{align*}
\text{to\_pair}(N,A,B):= \text{to\_tuple}(2,N,[A,B]). \\
\text{from\_pair}(X,Y,Z):= \text{from\_tuple}([X,Y],Z).
\end{align*}
\]
One can observe that `to_pair` and `from_pair` are the same as the functions defined in Steven Pigeon’s PhD thesis on Data Compression [25], page 114).

5 Encoding Finite Functions

As finite sets can be put in a bijection with an initial segment of \( \text{Nat} \), we can narrow down the concept of finite function as follows:

**Definition 2** A finite function is a function defined from an initial segment of \( \text{Nat} \) to \( \text{Nat} \).

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 \( \text{Nat} \) (seen as the list of its values), as a natural number:

\[
\text{ftuple2nat}([],0).
\]

\[
\text{ftuple2nat}(Ns, N) :- \text{Ns}=[_|_],
\]

\[
\text{length}(Ns, K), K1 \text{ is } K-1,
\]

\[
\text{from_tuple}(Ns, T), \text{pepis_pair}(K1, T, N).
\]

\[
\text{nat2ftuple}(0, []).
\]

\[
\text{nat2ftuple}(N, Ns) :- N>0,
\]

\[
\text{pepis_unpair}(N, K, F), K1 \text{ is } K+1,
\]

\[
\text{to_tuple}(K1, F, Ns).
\]

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:

?- \text{ftuple2nat}([1,0,2,1,3], N).
N = 21295
?- \text{nat2ftuple}(21295, T).
T=[1,0,2,1,3]
?- \text{ints_from}(0, 15, Is), \text{maplist(nat2ftuple, Is, Ts)}.
Ts=[[0], [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

\[
\text{nat}(0).
\]

\[
\text{nat}(N) :- \text{nat}(N1), N \text{ is } N1+1.
\]

\[
\text{iterative_fun_generator}(F) :- \text{nat}(N), \text{nat2ftuple}(N, F).
\]

provides an iterative generator for the stream of finite functions.
5.2 Deriving Encodings 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 \( f : [0..n-1] \rightarrow \text{Nat} \) can be represented as the list \([f(0)\ldots f(n-1)]\). Such a list has however repeated elements. So how can we turn it into a set with distinct elements, bijectively?

The following two predicates provide the answer.

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

\[
\text{fun2set([],[]).} \\
\text{fun2set([A|As],Xs):-findall(X,prefix\_sum(A,As,X),Xs).} \\
\text{prefix\_sum(A,As,R):-append(Ps,\_,As),length(Ps,L),sumlist(Ps,S),R is A+S+L.}
\]

The inverse of \text{fun2set} reverting back from a set of distinct values collects the increments from a term to the next (and ignores the last one):

\[
\text{set2fun([],[]).} \\
\text{set2fun([X|Xs], [X|Fs]):=set2fun(Xs,X,Fs).} \\
\text{set2fun([],\_,[]).} \\
\text{set2fun([X|Xs], Y, [A|As]):=A is (X-Y)-1, set2fun(Xs,X,As).}
\]

**Proposition 3** The following function equivalences hold:

\[
\text{fun2set} \circ \text{set2fun} \equiv \text{id} \equiv \text{set2fun} \circ \text{fun2set}
\]

The mapping and its inverse work as follows:

?- \text{fun2set([1,0,2,1,2],Set), set2fun(Set, Fun)}.  \\
Set = [1, 2, 5, 7, 10], \\
Fun = [1, 0, 2, 1, 2].

By combining this bijection with Ackermann’s encoding’s basic step \text{set2nat} and its inverse \text{nat2set}, we obtain an encoding from finite functions to \text{Nat} as follows (with DCG notation used to express function composition):

\[
\text{nat2fun} \rightarrow \text{nat2set, set2fun}. \\
\text{fun2nat} \rightarrow \text{fun2set, set2nat}.
\]

?- \text{nat2fun(2008, F), fun2nat(F, N)}.  \\
F = [3, 0, 1, 0, 0, 0, 0], N = 2008

**Proposition 4** The following function equivalences hold:

\[
\text{nat2fun} \circ \text{fun2nat} \equiv \text{id} \equiv \text{fun2nat} \circ \text{nat2fun}
\]
One can see that this encoding ignores 0s in the binary representation of a number, while counting 1 sequences as increments. Alternatively, Run Length Encoding of binary sequences [21] encodes 0s and 1s symmetrically, by counting the numbers of 1s and 0s. This encoding is reversible, given that 1s and 0s alternate, and that the most significant digit is always 1:

```
bits2rle([],[]):-!.
bites2rle([Y],[0]):-!.
bites2rle([X,Y|Xs],Rs):-X=Y,!,bits2rle([Y|Xs],[C|Cs]),C1 is C+1,Rs=[C1|Cs].
bites2rle([_|Xs],[0|Rs]):-bits2rle(Xs,Rs).
```

```
rle2bits([],[]).
rle2bits([N|Ns],NewBs):-rle2bits(Ns,Xs),
    ( []==Xs->B is 1
    ; Xs=[X1|_],B is 1-X1
    ),
    N1 is N+1,ndup(N1,B,Bs),append(Bs,Xs,NewBs).
```

By composing `bits2rle` and `rle2bits` with converters to/from bitlists, we obtain the bijection `nat2rle : Nat → [Nat]` and its inverse `rle2nat : [Nat] → Nat`

```
nat2rle --> to_rbits0,bits2rle.
rle2nat --> rle2bits,from_rbits .
```

```
to_rbits0(0,[]).
to_rbits0(N,R):-N>0,to_rbits(N,R).
```

**Proposition 5** The following function equivalences hold:

```
nat2rle ◦ rle2nat ≡ id ≡ rle2nat ◦ nat2rle
```

### 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 `ftuple2nat, fun2nat, rle2nat` in way similar to the use of `nat2set` and `set2nat` for HFS, where the empty function (denoted `[]`) 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 `unrank` and `rank` predicates 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 (with DCG notation expressing composition of functional predicates):

```
nat2hff --> default_ulimit(D),nat2hff_(D).
nat2hff1 --> default_ulimit(D),nat2hff1_(D).
```
nat2hff2  -->  default_ulimit(D),nat2hff2_(D).

nat2hff_(Ulimit)  -->  unrkn_(Ulimit,nat2fun).
nat2hff1_(Ulimit)  -->  unrkn_(Ulimit,nat2ftuple).
nat2hff2_(Ulimit)  -->  unrkn_(Ulimit,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.

?- nat2hff(42,H),hff2nat(H,N).
   H = [[[[]], [[]], [[]]]],
   N = 42

?- nat2hff1(42,H),hff2nat1(H,N).
   H = [[[[]], []], [[]], [[]]],
   N = 42

?- nat2hff2(42,H),hff2nat2(H,N).
   H = [[[]], [[]], [[]], []],
   N = 42

Note that

?-nat(N),nat2hff(N,HFF).
?-nat(N),nat2hff1(N,HFF).
?-nat(N),nat2hff2(N,HFF).

provide iterative generators for the (recursively enumerable!) stream of heredi-
tarily finite functions.

The resulting HFF with urfunctions (seen as digits) can also be used as gen-
eralized numeral systems with possible applications to building arbitrary length
integer implementations.

?- nat2hff_(10,1234567890,HFF).
   [3, 2, 0, 1, 7, 0, 1, 2, 0, 2, 2]

Proposition 6 The following function equivalences hold:

\[ nat2hff1 \circ hff2nat1 \equiv id \equiv hff2nat1 \circ nat2hff1 \]  \hspace{1cm} (7)

\[ nat2hff \circ hff2nat \equiv id \equiv hff2nat \circ nat2hff \]  \hspace{1cm} (8)
7 Encoding Finite Bijectons

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 [14] 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,R),rf(R,N).
R = [0, 0, 0, 3, 1],
N = 42

?- fl(42,R),lf(R,N).
R = [1, 3, 0, 0, 0],
N = 42

The Prolog predicate \( fr \) 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,[]).
fr(N,R):-N > 0,fr1(1,N,R).
fr1(_,0,[]).
fr1(J,K,\[KMJ | Rs\]):-K > 0,KMJ is K mod J,J1 is J+1,KDJ is K // J,
fr1(J1,KDJ,Rs).

The reverse \( fl \), is obtained as follows:
fl(N,Ds):-fr(N,Rs),reverse(Rs,Ds).

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

lf(Ls,S):-length(Ls,K),K1 is K-1,lf(K1,_,S,\[\]).

% from list of digits of factoradics, back to decimals
lf(0,1,0)-->[0].
lf(K,N,S)--->[D],{K>0,K1 is K-1},lf(K1,N1,S1),\[N is K+N1,S is S1+D+N\].

Finally, \( rf \), the inverse of \( fr \) is obtained by reversing \( fl \).
rf(Ls,S):-reverse(Ls,Rs),lf(Rs,S).
The Lehmer code of a permutation \( f \) is defined as the number of indices \( j \) such that \( 1 \leq j < i \) and \( f(j) < f(i) \) [17].

**Proposition 7** The Lehmer code of a permutation determines the permutation uniquely.

The predicate `perm2nth` computes a rank for a permutation \( Ps \) of \( Size>0 \). It starts by first computing its Lehmer code \( Ls \) with `perm_lehmer`. Then it associates a unique natural number \( N \) to \( Ls \), by converting it with the predicate `lf` from factoradics to decimals. Note that the Lehmer code \( Ls \) is used as the list of digits in the factoradic representation.

```prolog
perm2nth(Ps,Size,N):-
    length(Ps,Size), Last is Size-1,
    ints_from(0,Last,Is),
    perm_lehmer(Is,Ps,Ls),
    lf(Ls,N).
```

The generation of the Lehmer code is surprisingly simple and elegant in Prolog. We just instrument the usual backtracking predicate generating a permutation to remember the choices it makes, in the auxiliary predicate `select_and_remember`!

```prolog
% associates Lehmer code to a permutation
perm_lehmer([],[],[]).
perm_lehmer([X],[X|Zs],[K|Ks]):-
    select_and_remember(X,[X|Zs],Xs,K,K),
    perm_lehmer(Xs,Ps,Ls),
    lf(Ls,N).
```

The predicate `nat2perm` provides the matching unranking operation associating a permutation \( Ps \) to a given \( Size>0 \) and a natural number \( N \).

```prolog
nth2perm(Size,N,Ps):-
    fl(N,Ls),length(Ls,L),
    K is Size-L,Last is Size-1,ints_from(0,Last,Is),
    zeros(K,Zs),append(Zs,Ls,LehmerCode),
    perm_lehmer(Is,Ps,LehmerCode).
```

Note also that `perm_lehmer` is used (reversibly!) this time to reconstruct the permutation \( Ps \) from its Lehmer code. The Lehmer code is computed from the permutation’s factoradic representation obtained by converting \( N \) to \( Ls \) and then padding it with 0’s. One can try out this bijective mapping as follows:

```prolog
?- nth2perm(5,42,Ps),perm2nth(Ps,Length,Nth).
Ps = [1, 4, 0, 2, 3],
```
7.3 A bijective mapping from permutations to Nat

One more step is needed 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(0,0).
sf(N,R1):-N>0,N1 is N-1,ndup(N1,1,Ds),rf([0|Ds],R),R1 is R+1.

This is done by noticing that the factoradic representation of \([0,1,1,\ldots]\) does just that. The stream of all such sums can now be generated as usual:
sf(S):-nat(N),sf(N,S).

What we are really interested into, is decomposing \( N \) into the distance to the last sum of factorials smaller than \( N \), \( N_M \) and its index in the sum, \( K \).
to_sf(N, K, N_M):-nat(X),sf(X,S),S>N!,K is X-1,sf(K,M),N_M is N-M.

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 \( \text{nth2perm} \).
nat2perm(0,[]).
nat2perm(N,Ps):-to_sf(N, K, N_M),nth2perm(K,N_M,Ps).

Ranking of a permutation is even easier: we first compute its Size and its rank \( Nth \), then we shift the rank by the sum of all factorials up to \( \text{Size} \), enumerating the ranks previously assigned.
perm2nat([],0).
perm2nat(Ps,N):-perm2nth(Ps, Size,Nth),sf(Size,S),N is S+Nth.

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

**Proposition 8** The following function equivalences hold:

\[
nat2perm \circ \text{perm2nat} \equiv \text{id} \equiv \text{perm2nat} \circ \text{nat2perm}
\]  

(9)
7.4 Hereditarily Finite Permutations

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

\[
\text{nat2hfp} \rightarrow \text{default_ulimit}(D), \text{nat2hfp}_D.
\]
\[
\text{nat2hfp}_D(U\text{limit}) \rightarrow \text{unrank}_D(U\text{limit}, \text{nat2perm}).
\]
\[
\text{hfp2nat} \rightarrow \text{rank}(\text{perm2nat}).
\]

The encoding works as follows:

?- \text{nat2hfp}(42,H),\text{hfp2nat}(H,N),\text{write}(H),\text{nl}.
H = [[], [], [[]]], [[[]], []], []], [[]], [[[], [], []], [], [], []], N = 42

Proposition 9 The following function equivalences hold:

\[
\text{nat2hfp} \circ \text{hfp2nat} \equiv \text{id} \equiv \text{hfp2nat} \circ \text{nat2hfp}
\]

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 [32, 11, 12, 1, 4, 20, 16, 31, 3]. Computational and Data Representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in [6, 24, 22]. Pairing functions have been used work on decision problems as early as [23, 10, 27, 29]. 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 logic programming 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 [15, 26], 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 predicates like `nat2perm` and `perm2nat`.

Last but not least, the use of a logic programming language to express in a generic way some fairly intricate combinatorial algorithms predicts an interesting new application area.

References


A Appendix

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

**Integer list operations** These are some simple utility predicates:

\%
\% generates integers From..To
\%
\ints_from(From,To,Is):-\findall(I,between(From,To,I),Is).

\%
\% replicates X, N times
ndup(0, _, []).  
ndup(N, X, [X | Xs]) :- N > 0, N1 is N - 1, ndup(N1, X, Xs).

zeros(N, Zs) :- ndup(N, 0, Zs).

Matrix Transposition  This code transposes a matrix represented as list of lists.

mtranspose([], []).  
mtranspose([Xs], Css) :- !, to_columns(Xs, Css).

mtranspose([Xs | Xss], Css2) :- !,
    mtranspose(Xss, Css1),
    to_columns(Xs, Css1, Css2).

to_columns([], []).  
to_columns([X | Xs], [[X] | Zs]) :- to_columns(Xs, Zs).

to_columns([], Css, Css).  
to_columns([X | Xs], [Cs | Css1], [[X | Cs] | Css2]) :- to_columns(Xs, Css1, Css2).

Bit crunching functions  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”).

% conversion to list of digits in given base  
to_base(Base, N, Bs) :- to_base(N, Base, 0, Bs).

to_base(N, R, _K, Bs) :- N < R, Bs = [N].  
to_base(N, R, K, [B | Bs]) :- N >= R,
    B is N mod R, N1 is N // R, K1 is K + 1,
    to_base(N1, R, K1, Bs).

% conversion from list of digits in given base  
from_base(_, [], 0).  
from_base(Base, [X | Xs], N) :- from_base(Base, Xs, R), N is X + R * Base.

% conversion to list of bits, right to left  
to_rbits(N, Bs) :- to_base(2, N, Bs).

% conversion from list of bits, right to left  
from_rbits(Bs, N) :- from_base(2, Bs, N).

% counting how many bits a number needs  
bitcount(N, K) :- N < 1, K = 1.  
bitcount(N, K) :- N > 1, N1 is N // 1, bitcount(N1, K1), K is K1 + 1.

% finds the largest bitcount for a list  
max_bitcount(Nss, L) :- maplist(bitcount, Nss, Ls), max_list(Ls, L).

% pads up to maxbits, if needed
to_maxbits(Maxbits,N,Rs):-
    to_base(2,N,Bs),length(Bs,L),ML is Maxbits-L,
    ndup(ML,0,Zs),append(Bs,Zs,Rs).