Declarative Combinatorics: Ranking and Unranking of Hereditarily Finite Permutations

Paul Tarau

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

Abstract. Sets represent "content" in a pure way - order is immaterial. Permutations represent "order" in a pure way - what is actually ordered is immaterial. The paper will show that a similar "fractal" structure is shared when natural number encodings of both sets and permutations are expanded recursively.
Starting from encodings for finite permutations based on Lehmer codes and factoradics, we derive through a process similar to Ackermann's encoding of hereditarily finite sets, an encoding of *hereditarily finite permutations*.
The paper is organized as a self-contained literate Prolog program available at `http://logic.cse.unt.edu/tarau/research/2009/pHFP.zip`.
Keywords: logic programming and computational mathematics, hereditarily finite permutations, encodings of permutations, factoradics ranking/unranking bijections, Ackermann's encoding of hereditarily finite sets

1 Introduction

This paper is an exploration with logic programming tools of *ranking* and *unranking* problems on finite permutations and their related hereditarily finite universe. 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 [9].

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. Ranking/unranking of permutations and Hereditarily Finite Permutations as well as Lehmer codes and factoradics are covered in section 4. Sections 6 and 7 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. Otherwise, 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 predicates, not needed for following the main ideas of the paper, are left out from the narrative and provided in the Appendix.

The paper is organized as follows: section 2 introduces a general ranking/unranking framework for multiway tree data types with atoms/urelements and section 3 specializes it to hereditarily finite sets with urelements. An encoding of finite permutations is given in section 4 followed by the introduction of hereditarily finite permutations in section 5, the crux of the paper. Related work is discussed in section 6, followed by conclusions in section 7.

2 Generic unranking and ranking with higher order predicates

We will use, through the paper, a generic *multiway tree* type distinguishing between atoms represented as (arbitrary length) integers and sub-forests represented as Prolog lists. Atoms will be mapped to natural numbers in `[0..Ulimit-1]`. Assuming that `Ulimit` is fixed, we denote A the set `[0..Ulimit-1]`. We denote 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 $T \rightarrow Nat$. An unranking function is a bijection $Nat \rightarrow T$.*

Ranking functions can be traced back to Gödel numberings [5, 6] 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 [12, 9] algorithms.

2.1 Unranking

As an adaptation of the *unfold* operation [7, 13], elements of T will be mapped to natural numbers with a generic higher order function `unrank_` parameterized by the the natural number `Ulimit` and the transformer function `F`:

```
unrank_(Ulimit,_,N,R):-N>=0,N<Ulimit,!,R=N.
unrank_(Ulimit,F,N,R):-N>=Ulimit,
  N0 is N-Ulimit,
  call(F,N0,Ns),
  maplist(unrank_(Ulimit,F),Ns,R).
```

A global constant provided by the predicate `default_ulimit`, will be used through the paper to fix the default range of atoms as well as a default `unrank` function: Note also that we will use a syntactically more convenient `DCG` notation

for functional style predicates, composed by chaining their arguments automatically with Prolog's `DCG` transformation:

```
unrank(F)-->
  default_ulimit(Ulimit),
  unrank_(Ulimit,F).

default_ulimit(L)-->{clause(ulimit(L),_)},!.
default_ulimit(0)-->[].
```

Note also that `default_ulimit` provides a default global value for the number of atoms that can be customized with a dynamic clause `ulimit/1` if needed.

2.2 Ranking

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

```
rank_(Ulimit,_,N,R):-integer(N),N>=0,N<Ulimit,!,R=N.
rank_(Ulimit,G,Ts,R):-
  maplist(rank_(Ulimit,G),Ts,T),
  call(G,T,R0),
  R is R0+Ulimit.

rank(G)-->
  default_ulimit(Ulimit),
  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 `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* `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 [17, 14]. In a logic programming context, it has been used for reasoning with sets, set constraints, hypersets and bisimulations [4, 16].

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, 8, 18] is a bijection that maps hereditarily finite sets (HFS) to natural numbers (Nat) as follows:

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

Assuming HFS extended with *Urelements* (atomic objects not having any elements) a generic tree representation can be used for hereditarily finite sets with urelements.

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 [18].

```
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:

```
hfs2nat-->default_ulimit(Ulimit),hfs2nat_(Ulimit).

hfs2nat_(Ulimit)-->rank_(Ulimit,set2nat).
```

where the constant provided by `default_ulimit` controls the segment `[0..Ulimit-1]` of Nat to be mapped to urelements. For each natural number u this provides a generalization of Ackermann's mapping, to hereditarily finite sets with urelements in $[0..u-1]$ defined as:

$f_u(x) = \texttt{if } x < u \texttt{ then } x \texttt{ else } u + \sum_{a \in x} 2^{f_u(a)}$

For $u = 0$ this becomes Ackermann's original mapping from "pure" hereditarily finite sets, all built from the empty set only, to natural numbers.

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).

```
nat2set(N,Xs):-nat2elements(N,Xs,0).

nat2elements(0,[],_K).
nat2elements(N,NewEs,K1):-N>0,
  B is /\(N,1),
  N1 is N>>1,K2 is K1+1,
  add_el(B,K1,Es,NewEs),
  nat2elements(N1,Es,K2).

add_el(0,_,Es,Es).
add_el(1,K,Es,[K|Es]).
```

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

```
nat2hfs_(Ulimit)-->unrank_(Ulimit,nat2set).
```

```
nat2hfs-->default_ulimit(Ulimit),nat2hfs_(Ulimit).
```

We can represent the action of a hylomorphism unfolding a natural number into a hereditarily finite set as a directed graph with outgoing edges induced by by applying the inverse of the Ackermann encoding as shown in Fig. 1. Using an

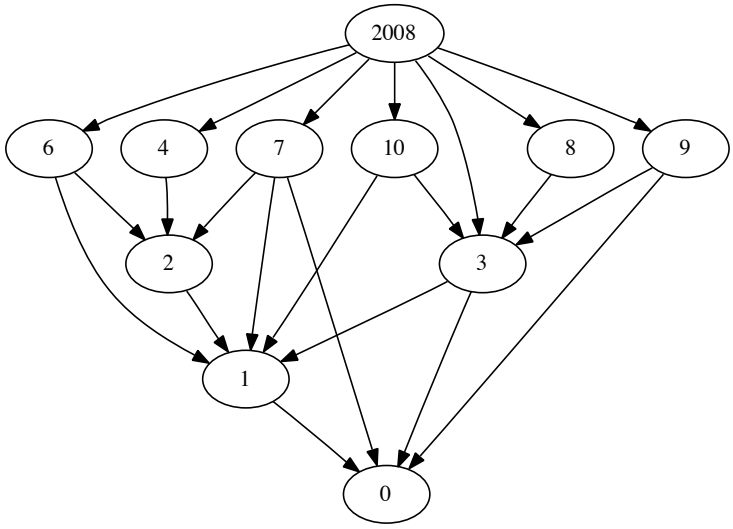

Fig. 1: 2008 as a HFS

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

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

$$nat2set \circ set2nat \equiv id \equiv set2nat \circ nat2set \tag{1}$$

$$nat2hfs \circ hfs2nat \equiv id \equiv hfs2nat \circ nat2hfs \tag{2}$$

4 Encoding finite permutations

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

4.1 The factoradic numeral system

The factoradic numeral system [10] 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,[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,Ls,[]).

% 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).
```

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

The Lehmer code of a permutation f of size n is defined as the sequence $l(f) = (l_1(f) \dots l_i(f) \dots l_n(f))$ where $l_i(f)$ is the number of elements of the set $\{j > i | f(j) < f(i)\}$ [11].

Proposition 3 *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.

```
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`!

```
% associates Lehmer code to a permutation
perm_lehmer([],[],[]).
perm_lehmer(Xs,[X|Zs],[K|Ks]):-
  select_and_remember(X,Xs,Ys,0,K),
  perm_lehmer(Ys,Zs,Ks).

% remembers selections - for Lehmer code
select_and_remember(X,[X|Xs],Xs,K,K).
select_and_remember(X,[Y|Xs],[Y|Ys],K1,K3):-K2 is K1+1,
  select_and_remember(X,Xs,Ys,K2,K3).
```

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

```
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:

```
?- nth2perm(5,42,Ps),perm2nth(Ps,Length,Nth).
Ps = [1, 4, 0, 2, 3],
Length = 5,
Nth = 42

?- nth2perm(8,2008,Ps),perm2nth(Ps,Length,Nth).
Ps = [0, 3, 6, 5, 4, 7, 1, 2],
```

```
Length = 8,
Nth = 2008
```

4.3 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(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,..] 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 `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 `Size`, enumerating the ranks previously assigned.

```
perm2nat([],0).
perm2nat(Ps,N) :-perm2nth(Ps, Size,Nth),sf(Size,S),N is S+Nth.


?- nat2perm(2008,Ps),perm2nat(Ps,N).
Ps = [1, 4, 3, 2, 0, 5, 6],
N = 2008
```

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

Proposition 4 *The following function equivalences hold:*

$$nat2perm \circ perm2nat \equiv id \equiv perm2nat \circ nat2perm \tag{3}$$

5 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).

```
nat2hfp --> default_ulimit(D),nat2hfp_(D).
nat2hfp_(Ulimit) --> unrank_(Ulimit,nat2perm).
hfp2nat --> rank(perm2nat).
```

The encoding works as follows:

```
?- nat2hfp(42,H),hfp2nat(H,N),write(H),nl.
H = [[], [[], [[]]], [[[]], []], [[]], [[], [[]], [[], [[]]]],
N = 42
?- nat2hfp(2008,S),write(S),nl,fail.
[[[]], [[], [[]], [[], [[]]]], [[[]], []], [[], [[]]], [],
       [[], [[], [[]]], [[]]], [[[]], [], [[], [[]]]].
```

Proposition 5 *The following function equivalences hold:*

$$nat2hfp \circ hfp2nat \equiv id \equiv hfp2nat \circ nat2hfp \tag{4}$$

Sets represent "content" in a pure way - order is immaterial. Permutations represent "order" in a pure way - what is actually ordered is immaterial. Let us note that a similar "fractal" structure is shared when natural number encodings of both sets and permutations are expanded recursively as HFSs and HFPs.

As shown in Fig 2 an ordered digraph (with labels starting from 0 representing the order of outgoing edges) can be used to represent the unfolding of a natural number to the associated hereditarily finite permutation. Note that as this mapping generates sequences where the order of the edges matters, therefore order is indicated by labeling the edges with integers starting from `0`. An interesting property of graphs associated to hereditarily finite permutations is that moving from a number n to its successor typically only induces a reordering of the labeled edges, as shown in Fig. 3.

It is interesting to see how "information density" of HFS and HFP compares. Intuitively that would answer the question: which is more efficient - codifying information as pure "content" or as pure "order"?

Figs. 4 and 5 compare sizes of HFS and HFP trees obtained from the same natural number up to 2^{10} and 2^{17} respectively.

We leave the study of the relative asymptotic behavior of the two curves as an example of interesting *open problem* derived from our data type hylomorphisms.

6 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 [17, 8, 1, 14, 3]. Computational and data

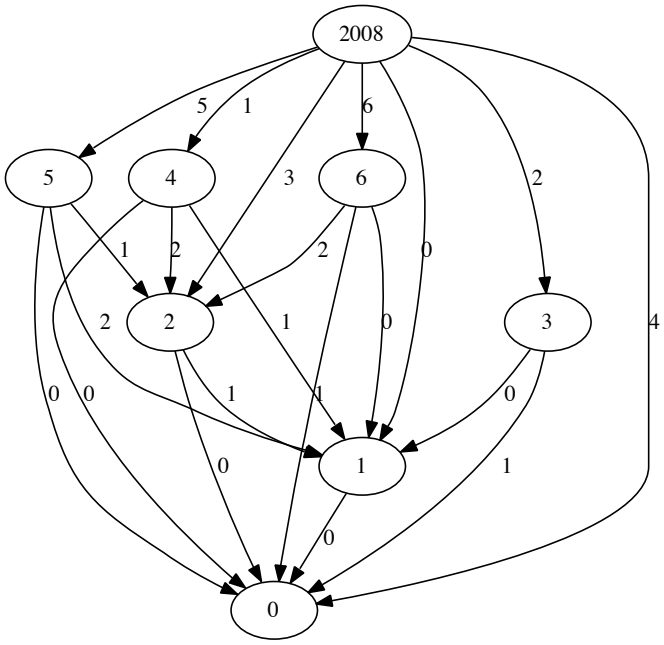

Fig. 2: 2008 as a HFP

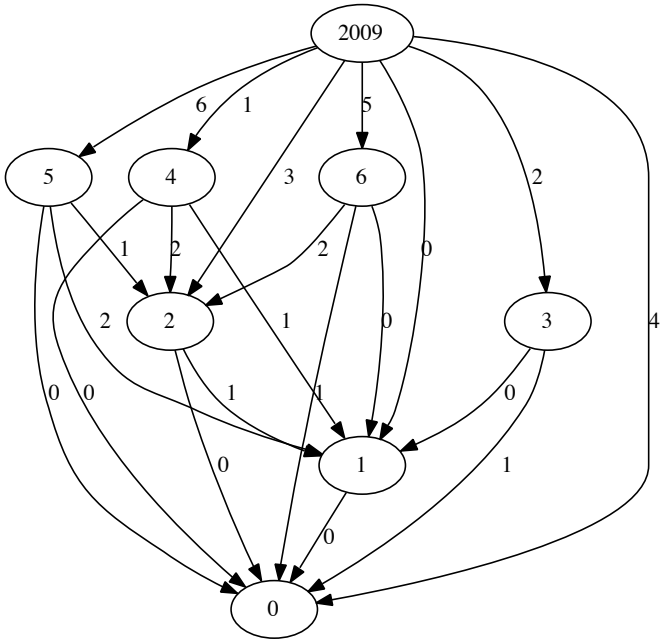

Fig. 3: 2009 as a HFP

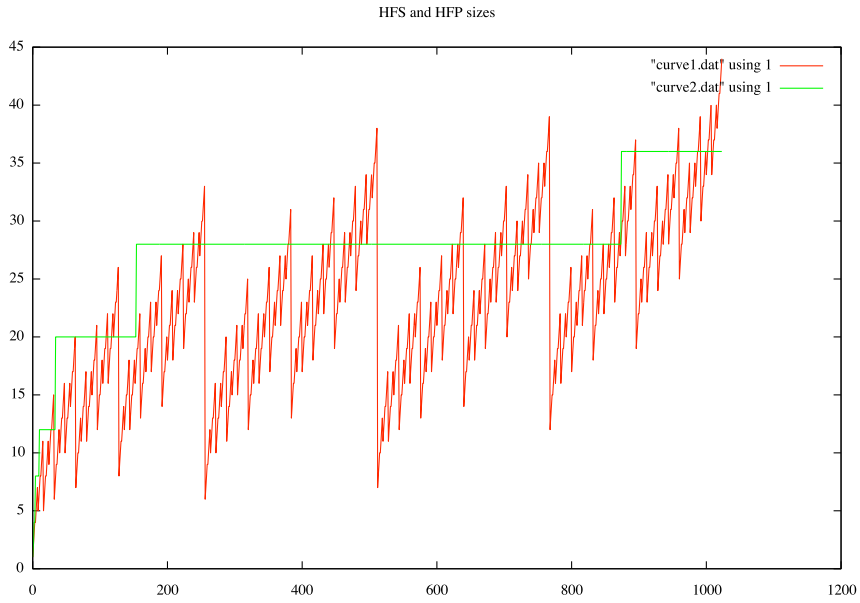

Fig. 4: Comparison of curve1=HFS and curve2=HFP sizes up to 2^{10}

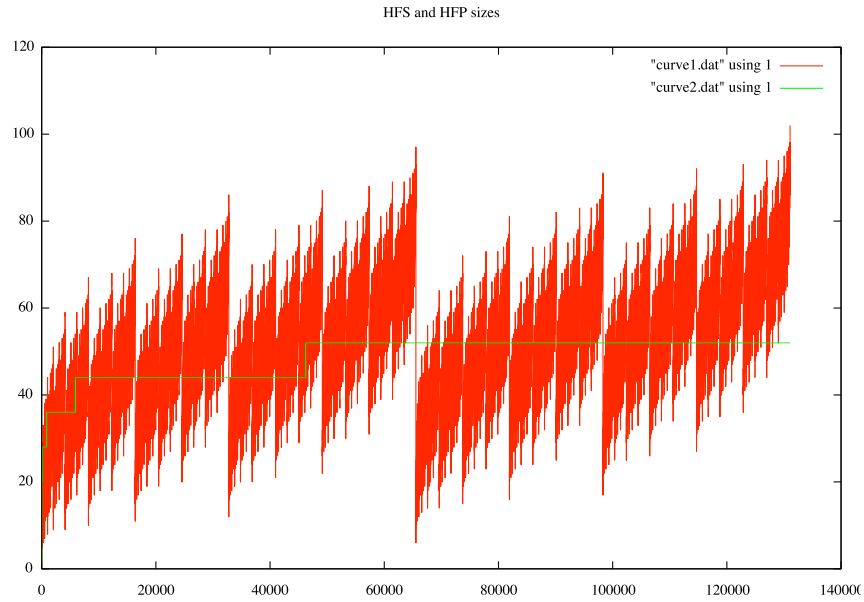

Fig. 5: Comparison of curve1=HFS and curve2=HFP sizes up to 2^{17}

representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in [4, 16, 15]. While finite permutations have been used extensively in various branches of mathematics and computer science, we have not seen any formalization of hereditarily finite permutations as such in the literature.

7 Conclusion and Future Work

We have shown the expressiveness of logic programming as a metalanguage for executable mathematics, by describing ranking/unranking functions for finite sets and permutations and by extending them in a generic way to hereditarily finite sets and hereditarily finite permutations.

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

1. 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.
2. Wilhelm Friedrich Ackermann. Die Widerspruchsfreiheit der allgemeinen Mengenlhere. *Mathematische Annalen*, (114):305–315, 1937.
3. Jeremy Avigad. The Combinatorics of Propositional Provability. In *ASL Winter Meeting*, San Diego, January 1997.
4. Agostino Dovier, Carla Piazza, and Alberto Policriti. Comparing Expressiveness of Set Constructor Symbols. In *Frontiers of Combining Systems*, pages 275–289, 2000.
5. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
6. 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.
7. Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.*, 9(4):355–372, 1999.
8. Richard Kaye and Tin Lock Wong. On Interpretations of Arithmetic and Set Theory. *Notre Dame J. Formal Logic Volume*, 48(4):497–510, 2007.
9. Donald Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.

10. Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
11. Roberto Mantaci and Fanja Rakotondrajao. A permutations representation that knows what "eulerian" means. *Discrete Mathematics & Theoretical Computer Science*, 4(2):101–108, 2001.
12. 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.
13. Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *FPCA*, pages 324–333, 1995.
14. Amram Meir, John W. Moon, and Jan Mycielski. Hereditarily Finite Sets and Identity Trees. *J. Comb. Theory, Ser. B*, 35(2):142–155, 1983.
15. 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.
16. Carla Piazza and Alberto Policriti. Ackermann Encoding, Bisimulations, and OBDDs. *TPLP*, 4(5-6):695–718, 2004.
17. Moto-o Takahashi. A Foundation of Finite Mathematics. *Publ. Res. Inst. Math. Sci.*, 12(3):577–708, 1976.
18. Paul Tarau. Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell, January 2009. http://arXiv.org/abs/0808.2953, unpublished draft, 104 pages.

A Appendix

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

% generator for the stream of natural numbers 0,1,2,...
nat(0).
nat(N):-nat(N1),N is N1+1.
```

The following predicates print out a HFS or HFP with *Urelements*

```
show_hfs(S):-gshow(S,"{,}"),nl.
show_hfp(S):-gshow(S,"( )"),nl.

gshow(0,[L,_C,R]):-put(L),put(R).
gshow(N,_):-integer(N),N>0,!,write(N).
gshow(Hs,[L,C,R]):-put(L),gshow_all(Hs,[L,C,R]),put(R).

gshow_all([],_).
gshow_all([H],LCR):-gshow(H,LCR).
gshow_all([H,G|Hs],[L,C,R]):-
  gshow(H,[L,C,R]),
  ([C]\=="~"->put(C);true),
  gshow_all([G|Hs],[L,C,R]).
```

as shown in the following examples:

```
?- nat2hfs(2009,H),show_hfs(H).
{{},{{},{{}}},{{{{}}}},{{{}},{{{}}}},{{},{{}},{{{}}}},
    {{{},{{}}}},{{},{{},{{}}}},{{{}},{{},{{}}}}

?- nat2hfp(2009,H),show_hfp(H).
((()) (() (()) (() (()))) ((()) ()) (() (())) ()
 ((()) () (() (()))) (() (() (())) (())))

?- assert(ulimit(4)).
true.

?- nat2hfs(2009,H),show_hfs(H).
{{},2,{},{1},{{},1},{2},{{},2},{1,2}}

?- nat2hfp(2009,H),show_hfp(H).
(1 () 3 () (()) (() 1) 2)
```