Executable Set Theory and Arithmetic Encodings in Prolog

Paul Tarau

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

Abstract. The paper is organized as a self-contained literate Prolog program that implements elements of an executable finite set theory with focus on combinatorial generation and arithmetic encodings. The complete SWI-Prolog tested code is available at `http://logic.csci.unt.edu/tarau/research/2008/pHFS.zip`.
First, ranking and unranking functions for some "mathematically elegant" data types in the universe of Hereditarily Finite Sets with Urelements are provided, resulting in arithmetic encodings for powersets, hypergraphs, ordinals and choice functions.
After implementing a digraph representation of Hereditarily Finite Sets we define *decoration functions* that can recover well-founded sets from encodings of their associated acyclic digraphs.
We conclude with an encoding of arbitrary digraphs and discuss a concept of duality induced by the set membership relation.
In the process, we uncover the surprising possibility of internally sharing isomorphic objects, independently of their language level types and meanings.
Keywords: logic programming and computational mathematics, hereditarily finite sets, ranking and unranking functions, executable set theory, arithmetic encodings, Prolog data representations

1 Introduction

This paper is an exploration with logic programming tools of interesting executable aspects of finite set theory, with focus on natural number encodings of various set-related data objects. Such encodings can be traced back to the Gödel numberings used to encode formulae as natural numbers in Gödel's famous incompleteness theorems. Their bijective variants are known as *ranking/unranking functions* in the field of combinatorial generation, where they are used to iterate over various combinatorial objects. Other typical uses are in the uniform generation of random objects of a given type and in designing succinct representations for data compression purposes.

The practical expressiveness of logic programming languages (in particular Prolog) are put at test in the process. This 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 [11].

The paper is organized as follows: section 2.1 introduces Ackermann's encoding in the more general case when *urelements* are present and shows an encoding for hypergraphs as a particular case. Section 3 gives examples of transporting common set and natural number operations from one side to the other. After discussing some classic pairing functions, section 4 introduces a new pairing/unpairing operation on natural numbers and applies them in section 5 to obtain encodings of powersets, ordinals and choice functions. Section 6 discusses graph representations and *decoration functions* on Hereditarily Finite Sets (6.1), and provides encodings for directed acyclic graphs (6.2). Sections 7 and 8 discuss related work, future work and conclusions.

2 Hereditarily Finite Sets and the Ackermann 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 [21], it has been the object of renewed practical interest in various fields, from representing structured data in databases [12] to reasoning with sets and set constraints in a Logic Programming framework [6, 15, 7].

2.1 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 [1, 9] 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* (i.e. objects not having any elements), we will use a recursively built *rose tree* for Hereditarily Finite Sets, where *Urelements* are represented as Natural Numbers in the interval `[0,Ulimit-1]` and sets are represented as Prolog lists without duplicated elements. The change to Ackermann's mapping, to accomodate a finite number of Urelements in $[0..ulimit-1]$ is a follows:

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

Proposition 1 *For $ulimit \in Nat$ the function f_{ulimit} is a bijection from Nat to HFS with Urelements in $[0..ulimit - 1]$.*

The proof follows from the fact that no sets map to values smaller than *ulimit* and that Urelements map into themselves. Note that if no Urelements are used, we obtain the "pure" HFS universe with the empty set represented as `[]` and mapped to `0`.

First, let's 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, defined as follows:

```
set2nat(Xs,N):-set2nat(Xs,0,N).

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

In fact, `set2nat` maps a set of exponents of 2 (implemented as bitshifts) to the associated sum of powers of 2. With this representation, Ackermann's encoding from HFS to Nat `hfs2nat` (parameterized by a fixed default_ulimit) value, becomes:

```
hfs2nat(N,R):-default_ulimit(D),hfs2nat_(D,N,R).

hfs2nat_(Ulimit,N,R):-integer(N),!,N>=0,N<Ulimit,!,R=N.
hfs2nat_(Ulimit,Ts,R):-
  maplist(hfs2nat_(Ulimit),Ts,T),
  set2nat(T,R0),
  R is R0+Ulimit.

default_ulimit(0).
```

Note that to ensure that `hfs2nat_` is a bijection we have shifted the result `R0` in the second clause by `Ulimit`, shuch that codes for sets will always be larger or equal to `Ulimit`.

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,Xs):-findall(X,nat2element(N,X),Xs).

nat2element(N,K):-nat2el(N,0,K).

nat2el(N,K1,Kn):-
  N>0, B is /\(N,1), N1 is N>>1,
  nat2more(B,N1,K1,Kn).

nat2more(1,_,K,K).
nat2more(_,N,K1,Kn):-K2 is K1+1,nat2el(N,K2,Kn).
```

The inverse of the (bijective) Ackermann encoding, with urelements in the interval `[0,Ulimit-1]` is defined as follows:

```
nat2hfs_(Ulimit,N,R):-N>=0,N<Ulimit,!,R=N.
nat2hfs_(Ulimit,N,R):-N>=Ulimit,
  N0 is N-Ulimit,
  nat2set(N0,Ns),
  maplist(nat2hfs_(Ulimit),Ns,R).
```

We can now define

```
nat2hfs(N,R):-default_ulimit(D),nat2hfs_(D,N,R).
```

where the constant given by `default_ulimit/1` controls the initial segment of *Nat* to be mapped to *Urelements*. We will assume in `default_ulimit(0)` unless specified otherwise. Note also that we shif back `N0` by `Ulimit` to ensure that `nat2hfs_` accurately reverses the action of `hfs2nat_`. One can try out `nat2hfs` and its inverse `hfs2nat` and their parametric variants for `Ulimit=3` as follows:

```
?- nat2hfs(42,S),hfs2nat(S,N).
S = [[[]], [[], [[]]], [[], [[[]]]]],
N = 42.

?-  nat2hfs_(3,42,S),hfs2nat_(3,S,N).
S = [0, 1, 2, [1]],
N = 42.
```

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).

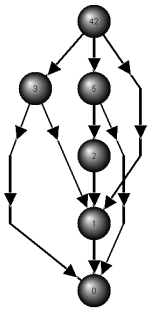


Fig. 1: Hereditarily Finite Set associated to 42

As both `nat2hfs` and `hfs2nat` are obtained through recursive compositions of `nat2set` and `set2nat`, respectively, one can generalize the encoding mechanism by replacing these building blocks with other bijections with similar properties.

2.2 Combinatorial Generation as Iteration

Using the inverse of Ackermann's encoding, the infinite stream HFS (with urelements in `[0,Ulimit-1]`) can be generated simply by iterating over all natural numbers:

```
nat(0).
nat(N):-nat(N1),N is N1+1.

iterative_hfs_generator(HFS):-default_ulimit(D),hfs_with_urelements(D,HFS).

hfs_with_urelements(Ulimit,HFS):-nat(N),nat2hfs_(Ulimit,N,HFS).


?- iterative_hfs_generator(HFS).
HFS = [] ;
HFS = [[]] ;
HFS = [[[]]] ;
HFS = [[], [[]]] ;
HFS = [[[[]]]] ;
HFS = [[], [[[]]]] ;
HFS = [[[]], [[[]]]] ;
...
```

2.3 Generating the Stream of Hereditarily Finite Sets Directly

To fully appreciate the elegance and simplicity of the combinatorial generation mechanism described previously, we will also provide a "hand-crafted" recursive generator for HFS. The reader will notice that it uses Prolog's "backtracking over infinite streams of answers" capability in an essential way. And arguably, that in a language without such features the algorithm is likely to get significantly more intricate.

If $P(x)$ denotes the powerset of x, the Universe of Hereditarily Finite Sets HFS is constructed inductively as follows:

1. the empty set $\{\}$ is in HFS
2. if x is in HFS then the union of its power sets $P^k(x)$ is in HFS

To implement in Prolog a simple HFS generator, conforming this definition, we start with a powerset predicate, working with sets represented as lists:

```
all_subsets([],[[]]).
all_subsets([X|Xs],Zss):-all_subsets(Xs,Yss),extend_subsets(Yss,X,Zss).

extend_subsets([],_,[]).
extend_subsets([Ys|Yss],X,[Ys,[X|Ys]|Zss]):-extend_subsets(Yss,X,Zss).
```

We can now backtrack over the infinite stream of "pure" hereditarily finite sets, one level at a time:

```
hfs_generator(NewSet):-nat(N),hfs_level(N,NewSet).

hfs_level(N,NewSet):-N1 is N+1,
  subsets_at_stage(N1,[],Hss1),subsets_at_stage(N,[],Hss),
  member(NewSet,Hss1),not(member(NewSet,Hss)).

subsets_at_stage(0,X,X).
subsets_at_stage(N,X,Xss):-N>0,N1 is N-1,
  all_subsets(X,Xs),
  subsets_at_stage(N1,Xs,Xss).
```

Note that redundant generation is avoided by keeping only new sets generated at each stage. Note also that `hfs_generator` produces the same stream of answers as `iterative_hfs_generator`.

```
?- hfs_generator(HFS),hfs2nat(HFS,N).
HFS = [], N = 0 ;
HFS = [[]], N = 1 ;
HFS = [[[]]], N = 2 ;
HFS = [[], [[]]],
...
```

2.4 Encoding Hypergraphs

By limiting recursion to one level in Ackermann's encoding, we can derive a bijective encoding of *hypergraphs* (also called *set systems*), represented as sets of sets of *Urelements*

```
nat2hypergraph(N,Nss):-nat2set(N,Ns),maplist(nat2set,Ns,Nss).

hypergraph2nat(Nss,N):-maplist(set2nat,Nss,Ns),set2nat(Ns,N).
```

as shown in the following example:

```
?- nat2hypergraph(2008,Nss),hypergraph2nat(Nss,N).
Nss = [[0, 1], [2], [1, 2], [0, 1, 2], [3], [0, 3], [1, 3]],
N = 2008.
```

Like in the case of combinatorial generation of HFS, the infinite stream of hypergraphs becomes simply

```
?-nat(N),nat2hypergraph(N,Nss).
```

Note also that a hypothetical application using integers, finite sets and hypergraphs can now *internally share the same data representation* - for instance arbitrary length integers - opening the doors for a form of generalized memoing mechanism.

In the following sections we will think about Ackermann's encoding and its inverse as *Functors* in Category Theory [16], transporting various operations from Natural Numbers to Hereditarily Finite Sets and back.

3 "Shapeshifting" between Nat and HFS with Fold operators and Functors

Given the *rose tree* structure of HFS, a natural `fold` operation [13] can be defined on them as a higher order predicate:

```
hfold(_,G,N,R):- integer(N),!,call(G,N,R).
hfold(F,G,Xs,R):-maplist(hfold(F,G),Xs,Rs),call(F,Rs,R).
```

For instance, it can count how many sets occur in a given HFS, as follows:

```
hsize(HFS,Size):-hfold(hsize_f,hsize_g,HFS,Size).

hsize_f(Xs,S):-sumlist(Xs,S1),S is S1+1.
hsize_g(_,1).
```

Note that recursing over `nat2set` has been used to build a member of HFS from a member of Nat. After lifting `nat2set` to a generic transformer predicate T, we can combine it with the action of a `fold` operator working directly on natural numbers (with urelements in `[0,Ulimit-1]`) as shown in the predicate `gfold/6`:

```
gfold(_,G,Ulimit,_,N,R):- integer(N),N⩽Ulimit,!,call(G,N,R).
gfold(F,G,Ulimit,T,N,R):-
  call(T,N,TransformedN),
  maplist(gfold(F,G,Ulimit,T),TransformedN,Rs),
  call(F,Rs,R).
```

We can now instantiate `gfold` to apply the transformer `nat2set`:

```
nfold(F,G,Ulimit,N,R):-gfold(F,G,Ulimit,nat2set,N,R).
nfold1(F,G,N,R):-default_ulimit(D),nfold(F,G,D,N,R).
```

Note that this can be seen as a form of implicit *deforestation by prevention*, equivalent to deforestation through program transformation in functional languages [22]. For instance, `nfold` allows counting the elements contained in the HFS representation of a number, without actually building the HFS tree as in:

```
nsize(N,R):-default_ulimit(Ulimit),nsize(Ulimit,N,R).
nsize(Ulimit,N,R):-nfold(hsize_f,hsize_g,Ulimit,N,R).
```

Note also that `nsize` can be seen as a *structural complexity* measure associated to a Natural Number or bitstring.

The action of the Ackermann encoding as a *Functor* from HFS to Nat on morphisms (seen as functions on a list of arguments) is defined as follows:

```
toNat(F,Hs,R):-maplist(hfs2nat,Hs,Ns),call(F,Ns,N),nat2hfs(N,R).
```

The same, acting on 1 and 2 argument operations is:

```
toNat1(F,X,R):-hfs2nat(X,N),call(F,N,NR),nat2hfs(NR,R).

toNat2(F,X,Y,R):-
  hfs2nat(X,NX),hfs2nat(Y,NY),
    call(F,NX,NY,NR),
  nat2hfs(NR,R).
```

The inverse Ackermann encoding from Nat to HFS seen as Functor is:

```
toHFS(F,Ns,N):-maplist(nat2hfs,Ns,Hs),call(F,Hs,H),hfs2nat(H,N).
```

with variants acting on a 1 and 2 argument functions:

```
toHFS1(F,X,R):-nat2hfs(X,N),call(F,N,NR),hfs2nat(NR,R).

toHFS2(F,X,Y,R):-
  nat2hfs(X,NX),nat2hfs(Y,NY),
  call(F,NX,NY,NR),hfs2nat(NR,R).
```

Using these functors we can define the equivalent of union, intersection, difference, ordered pair, cartesian product, powerset, adduction [9, 10], etc., on natural numbers seen as sets. We can also transport from Nat to HFS, operations like successor, sum, product, equality, with the practical idea in mind that one can pick the most efficient (or the simpler to implement) of the two representations.

4 Pairing Functions

Pairings are bijective functions $Nat \times Nat \to Nat$.

We refer to [5] for a typical use in the foundations of mathematics and to [20] for an extensive study of various pairing functions and their computational properties.

On top of the "set operations" defined in subsection 3 on Nat, the classic Kuratowski *ordered pair* $(a, b) = \{\{a\}, \{a, b\}\}$ can be easily implemented. However, the Kuratowski pair only provides an injective function $Nat \times Nat \to Nat$, resulting in fast growing integers very quickly, i.e. for $X, Y \in \{0, 1, 2, 3\}$ the generated sequence is:

```
2,10,34,514,12,4,68,1028,48,80,16,4112,768,1280,4352,256
```

4.1 Cantor's Pairing Function

We can do better by borrowing some interesting pairing functions defined on natural numbers. Starting from Cantor's pairing function

```
cantor_pair(K1,K2,P):-P is (((K1+K2)*(K1+K2+1))//2)+K2.
```

bijections from $Nat \times Nat$ to Nat have been used for various proofs and constructions of mathematical objects [18, 19, 5].

One can see that this time that the range is more compact, for $X, Y \in \{0, 1, 2, 3\}$ the sequence is:

```
0,2,5,9,1,4,8,13,3,7,12,18,6,11,17,24
```

4.2 An Efficient Pairing Function: BitMerge

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 [17], page 114).

The predicate `bitmerge_pair` implements a bijection from $Nat \times Nat$ to Nat that works by splitting a number's big endian bitstring representation into odd and even bits, while its inverse `to_pair` blends the odd and even bits back together. We will provide here an "elementary" implementation using exclusively bitshifting and addition operations, while abstracting away the set view of a natural number through the generator `nat2element` defined in subsection 2.1.

```
bitmerge_pair(A,B,P):-up0(A,X),up1(B,Y),P is X+Y.

bitmerge_unpair(P,A,B):-down0(P,A),down1(P,B).

even_up(A,R):-nat2element(A,X),E is X<<1,R is 1<<E.
odd_up(A,R):-nat2element(A,X),E is 1+(X<<1),R is 1<<E.
even_down(A,R):-nat2element(A,X),even(X),E is X>>1,R is 1<<E.
odd_down(A,R):-nat2element(A,X),odd(X),E is (X>>1), R is 1<<E.

even(X):- 0 =:= /\(1,X).
odd(X):- 1 =:= /\(1,X).

up0(A,P):-findall(R,even_up(A,R),Rs),sumlist(Rs,P).
up1(A,P):-findall(R,odd_up(A,R),Rs),sumlist(Rs,P).
down0(A,X):-findall(R,even_down(A,R),Rs),sumlist(Rs,X).
down1(A,X):-findall(R,odd_down(A,R),Rs),sumlist(Rs,X).
```

Note that `up` operations insert 0's in each even/odd position of the bitstrings while `down` operations remove even/odd bits while keeping odd/even bits, as shown in the following example with bitstrings aligned:

```
?- bitmerge_unpair(2008,X,Y),bitmerge_pair(X,Y,Z).
X = 60,
Y = 26,
Z = 2008.
% 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
%   60:[      0,    1,    1,    1,    1]
%   26:[   0,    1,    0,    1,    1   ]
```

Note also the significantly more compact packing, compared to Kuratowski pairs, and, like Cantor's pairing function, similar growth in both arguments.

It is convenient to see pairing/unpairing as one-to-one functions from/to the underlying language's ordered pairs:

```
bitmerge_pair(X-Y,Z):-bitmerge_pair(X,Y,Z).

bitmerge_unpair(Z,X-Y):-bitmerge_unpair(Z,X,Y).
```

This view as one-argument functions will allow using them in operations like `maplist`.

5 Powersets, Ordinals and Choice Functions

A concept of (finite) *powerset* can be associated to a number $n \in Nat$ by computing the powerset of the HFS associated to it, using the `toHFS1` functor:

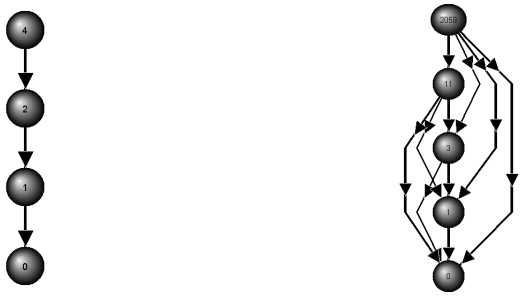
```
nat_powset(N,PN):-toHFS1(all_subsets,N,PN).
```

The von Neumann *ordinal* associated to a HFS, defined with interval notation as $\lambda = [0, \lambda)$, is implemented by the function `hfs_ordinal`, simply by transporting it from Nat:

```
hfs_ordinal(0,[]).
hfs_ordinal(N,Os):-N>0,N1 is N-1,findall(I,between(0,N1,I),Is),
  maplist(hfs_ordinal,Is,Os).

nat_ordinal(N,OrdN):-hfs_ordinal(N,H),hfs2nat(H,OrdN).
```

The following example shows the *transitive structure* of a von Neumann ordinal's set representation. It also shows its fast growing Nat encoding $(4 \rightarrow 2059)$ which can be seen as a somewhat unusual injective embedding of finite ordinals in Nat, seen as the set of finite cardinals.

```
?- hfs_ordinal(4,H),nat_ordinal(4,N),write(N:H),nl.
2059:[[], [[]], [[], [[]]], [[], [[]], [[], [[]]]]]
```



(a) as a pure HFS (b) its associated ordinal 2059

Fig. 2: 4 and its associated ordinal

Finally, a choice function, showing that Nat with the structure borrowed from HFS is actually a model for the Axiom of Choice, is implemented as an encoding of pairs of sets and their first elements with our compact $Nat \times Nat \rightarrow Nat$ pairing function `bitmerge_pair`:

```
nat_choice_fun(N,CFN):-nat2set(N,Es),
  maplist(nat2set,Es,Ess),maplist(choice_of_one,Ess,Hs),
  maplist(bitmerge_pair,Es,Hs,Ps),set2nat(Ps,CFN).

choice_of_one([X|_],X).
```

As *even* numbers represent sets that do not contain the empty set as an element, we compute Nat representations of the choice function as follows:

```
?- maplist(nat_choice_fun,[0,2,4,6,8,10,12,14,16],Funs).
Funs = [0, 2, 64, 66, 32, 34, 96, 98, 16777216].
```

6 Directed Graph Encodings

Directed Graphs are equivalent to binary relations seen as sets of ordered pairs. Equivalently, they can also be seen as vertices paired with lists of vertices of adjacent outgoing edges.

6.1 Directed Acyclic Graph representations for HFS

The tree representation of HFS can be seen as a *set* of edges, oriented to describe either set membership $\in$ or its transpose, set containment $\ni$:

```
nat2memb(N,XY):-default_ulimit(D),nat2memb(D,N,XY).
nat2memb(Ulimit,N,X-Y):-nat2contains(Ulimit,N,Y-X).

nat2contains(N,XY):-default_ulimit(D),nat2contains(D,N,XY).
nat2contains(Ulimit,N,E):-
  N⩾=Ulimit,
  N0 is N-Ulimit,
  nat2element(N0,X),
  ( E=N-X
  ; nat2contains(Ulimit,X,E)
  ).
```

The following examples show how the two predicates work.

```
?- findall(X,nat2memb(42,X),Xs),sort(Xs,S),write(S),nl.
[0-1, 0-3, 0-5, 1-2, 1-3, 1-42, 2-5, 3-42, 5-42]

?- findall(X,nat2contains(42,X),Xs),sort(Xs,S),write(S),nl.
[1-0, 2-1, 3-0, 3-1, 5-0, 5-2, 42-1, 42-3, 42-5]
```

The pair representation of $\in$ and its inverse $\ni$ can be turned directly into an actual graph data type (using SWI-Prolog's graph and associative list libraries). Given that our sets are well-founded, this graph is a directed acyclic graph.

```
nat2cdag(Ulimit,N,G):-
  findall(E,nat2contains(Ulimit,N,E),Es),
  vertices_edges_to_ugraph([],Es,G).
```

```
nat2mdag(Ulimit,N,G):-
  findall(E,nat2memb(Ulimit,N,E),Es),
  vertices_edges_to_ugraph([],Es,G).

?- nat2cdag(1,42,G).
G=[0-[], 1-[], 2-[0], 3-[1], 5-[2], 42-[0, 3, 5]].

?- nat2mdag(1,42,G).
[0-[2, 42], 1-[3], 2-[5], 3-[42], 5-[42], 42-[]].
```

However, as the associated HFS is usually sparse in case of large integers, we can think about compressing it to a canonical form. We can achieve this by replacing its n distinct vertex numbers with smaller integers in $[0, n-1]$, by progressively building a map describing this association. Given the fast growth of nodes from a level to another in a HFS tree, it makes sense to map the largest nodes to small integers first, as shown in the predicate `to_dag`:

```
to_dag(N,NewG):-
  findall(E,nat2contains(0,N,E),Es),
  vertices_edges_to_ugraph([],Es,G),
  vertices(G,Rs),reverse(Rs,Vs),
  empty_assoc(D),remap(Vs,0-D,_RVs,KD),remap(Es,KD,REs,_NewKD),
  vertices_edges_to_ugraph([],REs,NewG).

remap(Xs,Rs):-empty_assoc(D),remap(Xs,0-D,Rs,_KD).

remap([],KD,[],KD).
remap([X|Xs],KD1,[A|Rs],KD3):-integer(X),!,
  assoc(X,A,KD1,KD2),
  remap(Xs,KD2,Rs,KD3).
remap([X-Y|Xs],KD1,[A-B|Rs],KD4):-
  assoc(X,A,KD1,KD2),assoc(Y,B,KD2,KD3),
  remap(Xs,KD3,Rs,KD4).

assoc(X,R,K-D,KD):-get_assoc(X,D,A),!,R=A,KD=K-D.
assoc(X,K,K-D,NewK-NewD):-NewK is K+1,put_assoc(X,D,K,NewD).
```

Note that this construction assumes sets with no urelements, and the root of the graph (represented as 0) is the original natural number n from which the HFS has been built.

```
?- to_dag(42,G).
G = [0-[1, 2, 4], 1-[3, 5], 2-[4, 5], 3-[4], 4-[5], 5-[]]
```

An interesting question arises at this point. *Can we rebuild a natural number from its directed acyclic graph representation, assuming no labels are available, except 0?*

The answer is yes, and the key idea here is to apply `set2nat` recursively in a way similar to the implementation of `hfs2nat` and rebuild the values in

each vertex while progressing towards the root to recover the original value of $n \in Nat$, as shown in `from_dag`:

```
from_dag(G,N):-vertices(G,[Root|_]),compute_decoration(G,Root,N).

compute_decoration(G,V,Ds):-neighbors(V,G,Es),compute_decorations(G,Es,Ds).

compute_decorations(_,[],0).
compute_decorations(G,[E|Es],N):-
  maplist(compute_decoration(G),[E|Es],Ds),
  set2nat(Ds,N).

?- to_dag(42,G),from_dag(G,N).
G = [0-[1, 2, 4], 1-[3, 5], 2-[4, 5], 3-[4], 4-[5], 5-[]], N = 42
```

After implementing this predicate, we have found that it closely follows the *decoration* functions used in Aczel's book [2], and renamed it `compute_decoration`. In the simpler case of the HFS universe, with our well-founded sets represented as DAGs, the existence and unicity of the result computed by `from_dag` follows immediately from the Mostowski Collapsing Lemma [2].

6.2 Encodings of Directed Graphs as Natural Numbers

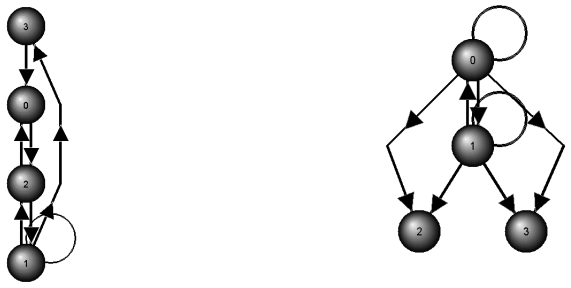
Hypersets [2] are defined by replacing the Foundation Axiom with the AntiFoundation axiom. Intuitively this means that the $\in$-graphs can be cyclical [3], provided that they are minimized through *bisimulation equivalence* [7]. We have not (yet) found an elegant encoding of hereditarily finite hypersets as natural numbers, similar to Ackerman's encoding. The main difficulty seems related to the fact that hypersets are modeled in HFS as equivalence classes with respect to bisimulation [2, 3, 15]. Toward this end, an easy first step seems to find a bijection from directed graphs (with no isolated vertices, corresponding to their view as binary relations), to Nat:

```
nat2digraph(N,G):-nat2set(N,Ns),
  maplist(bitmerge_unpair,Ns,Ps),
  vertices_edges_to_ugraph([],Ps,G).

digraph2nat(G,N):-edges(G,Ps),
  maplist(bitmerge_pair,Ps,Ns),
  set2nat(Ns,N).
```

With digraphs represented as lists of edges, this bijection works as follows:

```
2 ?- nat2digraph(255,G),digraph2nat(G,N).
G = [0-[0, 1], 1-[0, 1], 2-[0, 1], 3-[0, 1]]
N = 255.

3 ?- nat2digraph(2008,G),digraph2nat(G,N).
G = [0-[2, 3], 1-[1, 2], 2-[0, 1], 3-[1]],
N = 2008.
```

The resulting graphs are pictured in Figure 3. As usual



(a) 2008 as a digraph (b) 255 as a digraph

Fig. 3: Digraph Encodings

```
?-nat(N),nat2digraph(N,G).
```

provides a combinatorial generator for the infinite stream of directed acyclic graphs.

6.3 Duality

Idempotent operations like reversing the sense of the edges in a digraph (transpose/2 in SWI-Prolog) induce $Nat \rightarrow Nat$ bijections:

```
transpose_nat(N,TN):-nat2digraph(N,G),transpose(G,T),digraph2nat(T,TN).
```

```
1?- maplist(transpose_nat,[0,1,2,3,4,5,6,7],Ts),
    maplist(transpose_nat,Ts,Ns).
Ts = [0, 1, 4, 5, 2, 3, 6, 7],
Ns = [0, 1, 2, 3, 4, 5, 6, 7].
```

A more interesting form of duality arises when we interchange the $\in$ and $\ni$ relations themselves in HFS. Intuitively, it corresponds to the fact that intensions/concepts would become the building blocks of the theory, provided that something similar to the *axiom of extensionality* holds. In comments related to Russell's type theory [8] pp. 457-458 Gödel mentions an *axiom of intensionality* with the intuitive meaning that "different definitions belong to different notions". Gödel also notices the duality between "no two different properties

belong to exactly the same things" and "no two different things have exactly the same properties" but warns that contradictions in a simple type theory would result if such an axiom is used non-constructively. We will leave as a topic for future research to investigate various aspects of $\in$ / $\ni$ duality in HFS, in correlation with Natural Number their encodings.

7 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 [21, 9, 10, 4, 12]. Graph representations of sets and hypersets based on the variants of the Anti Foundation Axiom have been studied extensively in [2, 3]. Computational and Data Representation aspects of Finite Set Theory and hypersets have been described in a logic programming context in [6, 15, 7]. Pairing functions have been used work on decision problems as early as [14, 18, 19].

8 Conclusion and Future Work

Implementing with relative ease the encoding techniques typically used only in the foundations of mathematics recommends logic programming languages as effective tools for experimental mathematics.

While focusing on the ability to "shapeshift" between different data types, with the intent of internally sharing possibly heterogeneous data representations, we have described a variety of isomorphisms between mathematically interesting data structures, all centered around encodings as Natural Numbers. The possibility of sharing significant common parts of HFS-represented integers could be used in implementing shared stores for arbitrary length integers. Along the same lines, another application would be data compression using some "information theoretically minimal" variants of the graphs in subsection 6.1, from which larger, HFS and/or natural numbers can be rebuilt.

References

1. Wilhelm Friedrich Ackermann. Die Widerspruchsfreiheit der allgemeinen Mengenlhere. *Mathematische Annalen*, (114):305–315, 1937.
2. Peter Aczel. *Non-wellfounded sets.* Number 14 in CSLI Lecture Notes. Center for the Study of Language and Information (CSLI), 1988.
3. Jon Barwise and Lawrence Moss. *Vicious Circles.* Number 60 in CSLI Lecture Notes. Center for the Study of Language and Information (CSLI), 1996.
4. David Booth. Hereditarily Finite Finsler Sets. *J. Symb. Log.*, 55(2):700–706, 1990.
5. 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.

6. Agostino Dovier, Carla Piazza, and Alberto Policriti. Comparing Expressiveness of Set Constructor Symbols. In *Frontiers of Combining Systems*, pages 275–289, 2000.
7. Agostino Dovier, Carla Piazza, and Alberto Policriti. A Fast Bisimulation Algorithm. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 79–90. Springer, 2001.
8. Kurt Goedel. Russel's mathematical logic. In A.D. Irvine, editor, *Bertrand Russell: Critical Assessments*, London, 1999. Routledge.
9. Richard Kaye and Tin Lock Wong. On Interpretations of Arithmetic and Set Theory. *Notre Dame J. Formal Logic Volume*, 48(4):497–510, 2007.
10. Laurence Kirby. Addition and multiplication of sets. *Math. Log. Q.*, 53(1):52–65, 2007.
11. Donald Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼/knuth/taocp.html.
12. 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.
13. Tobias Nipkow and Lawrence C. Paulson. Proof Pearl: Defining Functions over Finite Sets. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2005.
14. Jozef Pepis. Ein verfahren der mathematischen logik. *The Journal of Symbolic Logic*, 3(2):61–76, jun 1938.
15. Carla Piazza and Alberto Policriti. Ackermann Encoding, Bisimulations, and OBDDs. *TPLP*, 4(5-6):695–718, 2004.
16. Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.
17. Stephen Pigeon. *Contributions à la compression de données.* Ph.d. thesis, Université de Montréal, Montréal, 2001.
18. Julia Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950.
19. Julia Robinson. Finite generation of recursively enumerable sets. *Proceedings of the American Mathematical Society*, 19(6):1480–1486, dec 1968.
20. Arnold L. Rosenberg. Efficient pairing functions - and why you should care. In *IPDPS*. IEEE Computer Society, 2002.
21. Moto-o Takahashi. A Foundation of Finite Mathematics. *Publ. Res. Inst. Math. Sci.*, 12(3):577–708, 1976.
22. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.