A Functional Hitchhiker's Guide to Hereditarily Finite Sets, Ackermann Encodings and Pairing Functions

– unpublished draft –

Paul Tarau

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

Abstract

The paper is organized as a self-contained literate Haskell program that implements elements of an executable finite set theory with focus on combinatorial generation and arithmetic encodings. The code, tested under GHC 6.6.1, is available at `http://logic.csci.unt.edu/tarau/research/2008/fSET.zip`.

We introduce ranking and unranking functions generalizing Ackermann's encoding to the universe of Hereditarily Finite Sets with Urelements. Then we build a lazy enumerator for Hereditarily Finite Sets with Urelements that matches the unranking function provided by the inverse of Ackermann's encoding and we describe functors between them 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.

Keywords *hereditarily finite sets, ranking and unranking functions, executable set theory, arithmetic encodings, Haskell data representations, functional programming and computational mathematics*

1. Introduction

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 in a Logic Programming framework (Dovier et al. 2000; Piazza and Policriti 2004; Dovier et al. 2001).

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) from Hereditarily Finite Sets to Natural Numbers, was the original trigger for our work on building in a mathematically elegant programming language, a concise and *executable* hereditarily finite set theory. The arbitrary size of the data objects brought in the need for arbitrary length integers. The focus on potentially infinite enumerations brought in the need for lazy evaluation. These have made *Haskell* a natural choice.

We will describe our constructs in a subset of *Haskell* (Peyton Jones 2002, 2003a,b) seen as a concrete syntax for a generic lambda calculus based functional language[1].

We will only make the assumptions that non-strict functions (higher order included), with call-by-need evaluation and arbitrary length integers are available in the language. While our code will conform Haskell's type system, we will do that without any type declarations, by ensuring that the types of our functions are all inferred. This increases chances that the code can be ported, through simple syntax transformations, to any programming language that implements our basic assumptions.

The paper is organized as follows: section 2 introduces the reader to combinatorial generation with help of a bit-string example, section 3 introduces Ackermann's encod-

WXYZ '08 date, City.

[1] As a courtesy to the reader wondering about the title, the author confesses being a hitchhiker in the world of functional programming, coming from the not so distant galaxy of logic programming but still confused by recent hitchhiking trips in the exotic worlds of logic synthesis, foundations of mathematics, natural language processing, conversational agents and virtual reality. And not being afraid to go boldly where . . . a few others have already been before.

ing in the more general case when *urelements* are present and shows an encoding for hypergraphs as a particular case. Section 4 gives examples of transporting common set and natural number operations from one side to the other. After discussing some classic pairing functions, section 5 introduces new pairing/unpairing on natural numbers. Section 6 discusses graph representations and *decoration functions* on Hereditarily Finite Sets (6.1), and provides encodings for directed acyclic graphs (6.3). Sections 7 and 8 discuss related work, future work and conclusions.

2. What's in a Bit?

Let us observe first that the well known bitstring representation of Natural Numbers (see `to_rbits` and `from_rbit` in Appendix and notice the reversed bit order) is a first hint at their genuinely polymorphic, "shapeshifting" nature:

```
to_rbits 2008
  [0,0,0,1,1,0,1,1,1,1,1]
from_rbits [0,0,0,1,1,0,1,1,1,1,1]
  2008
```

The effect is trivial here - these transformers turn a number into a list of bits and back. One step further, we will now define two one argument *functions*, that implement the "bits" `o` and `i`:

```
o x = 2*x+0
i x = 2*x+1
```

One can recognize now that 2008 is just the result of composing "bits", with a result similar to the result of `from_rbits`:

```
(o.o.o.i.i.o.i.i.i.i.i) 0
  2008
```

The reader will notice that we have just "shapeshifted" to yet another view: a number is now a composition of *bits, seen as transformers*, where each bit does its share by leftshifting the string one position and then adding its contribution to it. Note the analogy with Church numerals, which represent numbers as iterations of function application, except that here n will only need O($log_2(n)$) of space.

Like with the usual bitstring representation, the dominant digit is always `1`, zeros after that have no effect, from where we can infer that the mapping between such bitstrings and numbers is not one-to-one. A variant of the *2-adic bijective* numeral representation fixes this, and shows one of the simplest bijective mappings from natural numbers to bitstrings (i.e. the regular language $\{0,1\}^*$):

```
nat2bits = drop_last . to_rbits . succ where
  drop_last bs=genericTake (l-1) bs where
    l=genericLength bs

bits2nat bs = (from_rbits (bs ++ [1]))-1

nat2bits 42
  [1,1,0,1,0]
bits2nat it
  42
map nat2bits [0..15]
  [[],[0],[1],[0,0],[1,0],[0,1],[1,1],
   [0,0,0],[1,0,0],[0,1,0],[1,1,0],
   [0,0,1],[1,0,1],[0,1,1],[1,1,1],
   [0,0,0,0]]
```

The last example suggests that we are now able to generate the *infinite stream of all possible bitstrings* simply as as:

```
all_bitstrings = map nat2bits [0..]
```

We will now hitchhike with this *design pattern* in our toolbox to a more interesting universe.

3. Hereditarily Finite Sets and the Ackermann Encoding

The Universe of Hereditarily Finite Sets (HFS) is built from the empty set (or a set of *Urelements*) by successively applying powerset and set union operations. Assuming HFS extended with *Urelements* (i.e. objects not having any elements), the following data type defines a recursive "rose tree" for Hereditarily Finite Sets:

```
data HFS t = U t | S [HFS t] deriving (Show, Eq)
```

We will assume that *Urelements* are represented as Natural Numbers in `[0..ulimit-1]`. The constructor `U t` marks *Urelements* of type `t` (usually the arbitrary length Integer type in Haskell) and the constructor `S` marks a list of recursively built HFS type elements. Note that if no elements are used with the `U` constructor, we obtain the "pure" HFS universe by representing the empty set as `S []`.

3.1 Ackermann's Encoding

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):

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

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.

A simple change to Ackermann's mapping, will accomodate a finite number of Urelements in $[0..u-1]$, as follows:

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

PROPOSITION 1. *For $u \in Nat$ the function f_u is a bijection from Nat to HFS with Urelements in $[0..u-1]$.*

The proof follows from the fact that no sets map to values smaller than $ulimit$ and that Urelements map into themselves.

With this representation, Ackermann's encoding from HFS with Urelements in `[0..ulimit-1]` to Nat `hfs2nat_` becomes:

```
hfs2nat_ _ (U n) = n
hfs2nat_ ulimit (S es) =
  ulimit + set2nat (map (hfs2nat_ ulimit) es)

set2nat ns = sum (map (2^) ns)
```

where `set2nat` maps a set of exponents of 2 to the associated sum of powers of 2.

We can now define

```
hfs2nat = hfs2nat_ urelement_limit

urelement_limit=0
```

where the constant `urelement_limit` controls the initial segment of Nat to be mapped to *Urelements*. Note that to keep our Haskell code as simple as possible we assume that `urelement_limit` is a global parameter that implicitly fixes the set of Urelements.

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 = nat2right_exps n 0 where
  nat2right_exps 0 _ = []
  nat2right_exps n e = add_rexp (n `mod` 2) e
    (nat2right_exps (n `div` 2) (e+1)) where
      add_rexp 0 _ es = es
      add_rexp 1 e es = (e:es)

nat2set 42
  [1,3,5]
set2nat [1,3,5]
  42
nat2set 2008
  [3,4,6,7,8,9,10]
set2nat [3,4,6,7,8,9,10]
  2008
```

The inverse of the (bijective) Ackermann encoding (generalized to work with urelements in `[0..ulimit-1]`) is defined as follows:

```
nat2hfs_ ulimit n | n<ulimit = U n
nat2hfs_ ulimit n =
  S (map (nat2hfs_ ulimit) (nat2set (n-ulimit)))
```

We can now define

```
nat2hfs = nat2hfs_ urelement_limit
```

where the constant `urelement_limit` controls the initial segment of Nat to be mapped to *Urelements*.

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.

One can try out `nat2hfs` and its inverse `hfs2nat` and print out a HFS with the `setShow` function (given in Appendix):

```
nat2hfs 42
  S [S [U 0],S [U 0,S [U 0]],S [U 0,S [S [U 0]]]]
hfs2nat (nat2hfs 42)
  42
setShow 42
  "{{{}},{{},{{}}},{{},{{{}}}}}"
```

Assuming `urelement_limit=3` the HFS representation becomes:

```
nat2hfs 42
  S [U 0,U 1,U 2,S [U 1]]
setShow 42
  "{0,1,2,{1}}"
```

Note that `setShow n` will build a string representation of $n \in Nat$, "shapeshifted" as a HFS with Urelements. Figure 1 shows directed acyclic graphs 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).

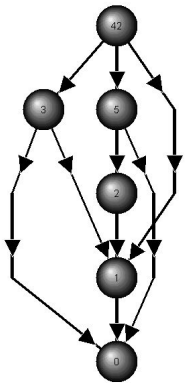

Figure 1: Hereditarily Finite Set associated to 42

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
  [U 0,S [U 0],S [S [U 0]],
   S [U 0,S [U 0]],S [S [S [U 0]]]]
```

3.3 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 this uses some fairly high level Haskell constructs like list comprehensions and lazy evaluation, and that in a language without such features the algorithm might 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 Haskell a simple HFS generator, conforming this definition, we start with a powerset function, working with sets represented as lists:

```
list_subsets [] = [[]]
list_subsets (x:xs) =
  [zs|ys←list_subsets xs,zs←[ys,(x:ys)]]
```

We can generate the infinite stream of "pure" hereditarily finite sets using Haskell's lazy evaluation mechanism, as follows:

```
hfs_generator = uhfs_from 0 where
  uhfs_from k = union (old_hfs k) (uhfs_from (k+1))

  old_hfs k = elements_of (hpow k (U 0))
  elements_of (U _) = []
  elements_of (S hs) = hs

  hpow 0 h = h
  hpow k h = hpow (k-1) (S (hsubsets h))

  hsubsets (U n) =  []
  hsubsets (S hs) = (map S (list_subsets hs))
```

One can now extract a finite number of HFS from the stream

```
take 5 hfs_generator
 [S [],S [S []],S [S [S []]],
  S [S [],S [S []]],S [S [S [S []]]]]
```

and notice the identical behavior of `hfs_generator` and `iterative_hfs_generator`.

3.4 Encoding Hypergraphs

DEFINITION 1. *A hypergraph (also called* set system*) is a pair $H = (X, E)$ where X is a set and E is a set of non-empty subsets of X.*

By limiting recursion to one level in Ackermann's encoding, we can derive a bijective encoding of *hypergraphs*, represented as sets of sets:

```
nat2hypergraph = (map nat2set) . nat2set
hypergraph2nat = set2nat . (map set2nat)
```

as shown in the following example:

```
nat2hypergraph 2008
  [[0,1],[2],[1,2],[0,1,2],[3],[0,3],[1,3]]
hypergraph2nat (nat2hypergraph 2008)
  2008
```

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

```
map nat2hypergraph [0..]
```

Note also that a hypothetical application using integers, finite sets and hypergraphs can use internally the same immutable data type, with opportunities to *share* common structures.

In the following sections we will think about Ackermann's encoding and its inverse as *Functors* in Category Theory (Pierce 1991), transporting various operations from Natural Numbers to Hereditarily Finite Sets and back.

4. Shapeshifting Operations between Nat and HFS

4.1 Fold operators and functors

Given the *rose tree* structure of HFS, a natural `fold` operation (Nipkow and Paulson 2005) can be defined on them as a higher order Haskell function:

```
hfold f g (U n) = g n
hfold f g (S xs) = f (map (hfold f g) xs)
```

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

```
hsize = hfold f g where
  f xs = 1+(sum xs)
  g _ =1
```

Note that recursing over `nat2set` has been used to build a member of HFS from a member of Nat. Thus, we can combine it with the action of a `fold` operator working directly on natural numbers as follows:

```
nfold f g n = nfold_ f g urelement_limit n

nfold_ f g ulimit n | n<ulimit = g n
nfold_ f g ulimit n =
  f (map (nfold_ f g ulimit) (nat2set n))
```

For instance, `nfold` allows counting the elements contained in the HFS representation of a number:

```
nsize = nfold  f g where
  f xs = 1+(sum xs)
  g _ =1
```

as if defined by

```
nsize_alt n = hsize (nat2hfs n)
```

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 = nat2hfs . f . (map  hfs2nat)
```

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

```
toNat1 f i = nat2hfs (f (hfs2nat i))
toNat2 f i j = nat2hfs (f (hfs2nat i) (hfs2nat j))
```

The inverse Ackermann encoding acts as a Functor from Nat to HFS:

```
toHFS f = hfs2nat . f . (map nat2hfs)
```

with variants acting on a 1 and 2 argument functions:

```
toHFS1 f x = hfs2nat (f (nat2hfs x))
toHFS2 f x y = hfs2nat (f (nat2hfs x) (nat2hfs x))
```

Note that the `nat2set` and `set2nat` functions used in the Ackerman encoding and its inverse can also be seen as providing Functors connecting Nat and $[Nat]$ (seen as a representation of finite subsets of Nat):

```
toExps f = set2nat . f . (map nat2set)
fromExps f = nat2set . f . (map set2nat)
```

4.2 Mappings between Arithmetic and Set Operations

After extending 2 argument set operations to lists, using `foldl`

```
setOp f []=[]
setOp f (x:xs) = foldl f x xs
```

we can define the equivalent of adduction (i.e. $\{i\} \cup s$ - see (Kaye and Wong 2007; Kirby 2007)), union, intersection etc., on natural numbers seen as (lists of) sets:

```
nat_adduction i is =
  set2nat (union [i] (nat2set is))

nat_singleton i = 2^i

nat_intersect = nats_intersect . nat2set
nats_intersect = toExps (setOp intersect)

nat_union = nats_union . nat2set
nats_union = toExps (setOp union)

nat_equal i j = if i==j then 1 else 0
```

Similarly, we can transport from Nat to HFS, operations like successor, addition, product, equality as follows:

```
hsucc = toNat1 succ
hsum = toNat sum
hproduct = toNat product
hequal = toNat2 nat_equal
hexp2  = toNat1 (2^)
```

with the practical idea in mind that one can pick the most efficient (or the simpler to implement) of the two representations at will.

As current computer architectures tend to support Natural Numbers and underlying arbitrary integer representations quite well, we can pick them as the hub that mediates the "shapeshiftings" between various data types. However, in an application where lazy structure building would be instrumental for performance, something like HFS (or one of the encodings described in the next sections) could be the most appropriate internal representation.

5. Pairing Functions

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

$$J(K(z), L(z)) = z \tag{1}$$

$$K(J(x, y)) = x \tag{2}$$

$$L(J(x, y)) = y \tag{3}$$

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.

On top of the "set operations" defined in subsection 4.2 on Nat, the classic Kuratowski ordered pair $(a, b) = \{\{a\}, \{a, b\}\}$ can be implemented with adductions and singletons as follows:

```
nat_kpair x y = nat_adduction sx ssxy where
  sx = nat_singleton x
  sy = nat_singleton y
  sxy = nat_adduction x sy
  ssxy = nat_singleton sxy
```

However, the Kuratowski pair only provides an injective function $Nat \times Nat \rightarrow Nat$, resulting in fast growing integers very quickly:

```
[nat_kpair x y|x<-[0..3],y<-[0..3]]
   [2,10,34,514,12,4,68,1028,48,
   80,16,4112,768,1280,4352,256]
```

5.1 Cantor's Pairing Function

We can do better by borrowing some interesting pairing functions defined on natural numbers. Starting from Cantor's pairing function bijections from $Nat \times Nat$ to Nat have been used for various proofs and constructions of mathematical objects (Robinson 1950, 1955, 1968a,b; Cégielski and Richard 2001).

Cantor's pairing function is defined as:

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

Note that its range is more compact

```
[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]
```

Unfortunately, its inverse involves floating point operations that do not combine well with arbitrary length integers.

5.2 A new Pairing Function

We will introduce here a new pairing function, that provides compact representations for various set theoretic constructs involving ordered pairs while only using elementary integer arithmetic operations.

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

```
bitmerge_pair (i,j) =
  set2nat ((evens i) ++ (odds j)) where
    evens x = map (*2) (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 (`div` 2))
```

The function `bitmerge_pair` 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. With help of the function `to_rbits` given in Appendix, that decomposes $n \in Nat$ into a list of bits (smaller units first) on 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
```

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

```
map bitmerge_unpair [0..15]
  [(0,0),(1,0),(0,1),(1,1),(2,0),(3,0),
  (2,1),(3,1),(0,2),(1,2),(0,3),(1,3),
  (2,2),(3,2),(2,3),(3,3)]
[bitmerge_pair (i,j)|i<-[0..3],j<-[0..3]]
  [0,2,8,10,1,3,9,11,4,6,12,14,5,7,13,15]
```

5.3 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:

```
nat_powset i = set2nat
  (map set2nat (list_subsets (nat2set i)))
```

or, directly, as in (Abian and Lamacchia 1978):

```
nat_powset_alt i = product
  (map (λk→1+2^(2^k)) (nat2set i))
```

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:

```
nat_ordinal 0 = 0
nat_ordinal n =
  set2nat (map nat_ordinal [0..(n-1)])

hfs_ordinal = nat2hfs . nat_ordinal
```

The following example shows the *transitive structure* of a von Neumann ordinal's set representation (see Fig. 2). 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
  S [S [],S [S []],S [S [],
     S [S []]],S [S [],S [S []],
     S [S [],S [S []]]]]
nat_ordinal 4
  2059
```

Finally, a choice function is implemented as an encoding of pairs of sets and their first elements with our compact $Nat \times Nat \rightarrow Nat$ pairing function:

```
nat_choice_fun i =  set2nat xs where
  es = nat2set i
  hs = map (head . nat2set) es
  xs = zipWith (curry bitmerge_pair) es hs
```

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:

```
map nat_choice_fun [0,2..16]
  [0,2,64,66,32,34,96,98,16777216]
```

Note that `nat_choice_function` computes a natural number representation i.e. Göedel number for a function that picks an element of each set of any family of sets not containing the empty set. Constructing such a natural number proves that Nat, with the structure borrowed from HFS is actually *a model* for the *Axiom of Choice*. Such models are important in the foundations of mathematics as they show that interpretations of sets and functions other the usual ones are compatible with various axiomatizations of set theory (Kaye and Wong 2007; Kirby 2007).

6. Directed Graph Encodings

Directed Graphs are equivalent to binary relations seen as sets of ordered pairs. Equivalently, (as implemented in the Haskell `Data.Graph` package), they can also be seen as

Figure 2: 4 and its associated ordinal: *as a pure* HFS and *its associated ordinal 2059*

arrays of vertices in [0..n] paired with lists of vertices of adjacent outgoing edges. We will freely alternate between these two representations in this section.

6.1 Directed Acyclic Graph representations for HFS

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

```
nat2memb = nat2pairs with_memb
nat2contains = nat2pairs with_contains

with_memb a x = (x,a)
with_contains a x = (a,x)
```

Note that this uses the function `nat2pairs` (see Appendix) that provides the actual decomposition of a number into Haskell ordered pairs. The following examples show how this works:

```
nat2memb 42
  [(0,1),(0,3),(0,5),(1,2),(1,3),
   (1,42),(2,5),(3,42),(5,42)]
nat2contains 42
  [(1,0),(2,1),(3,0),(3,1),(5,0),
   (5,2),(42,1),(42,3),(42,5)]
```

These list of pair representations can be easily converted to Haskell's graph data type (imported from Data.Graph) as follows:

```
nat2member_dag = nat2dag_ nat2memb

nat2contains_dag = nat2dag_ nat2contains

nat2dag_ f n = buildG (0,l) es where
  es=reverse (f n)
  l=foldl max 0 (nat2parts n)
```

where `nat2parts`, given in the Appendix, converts n to the set of Natural Numbers occurring in its HFS representation.

Moreover, the pair representation of $\in$ and its inverse can be turned into a more compact graph by replacing its n distinct vertex numbers with smaller integers from $[0..n-1]$, by progressively building a map describing this association, as shown in the function `to_dag`

```
to_dag n = (buildG (0,l)
  (map (remap m) (nat2contains n))) where
    is = [0..l]
    ns =  reverse (nat2parts n)
    l = (genericLength ns)-1
    m= (zip ns is)
    remap m (f,t) =  (lf,lt) where
      (Just lf)=(lookup f m)
      (Just lt)=(lookup t m)
```

Dually, one can convert $n \in Nat$ to the containment graph of its HFS as follows

```
to_ddag = transposeG . to_dag
```

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?* Surprisingly, the answer is yes, and the function `from_dag` provides the conversion:

```
from_dag g =
  compute_decoration g (fst (bounds g))

compute_decoration g v =
  compute_decorations g (g!v) where
    compute_decorations _ [] = 0
    compute_decorations g es =
      sum (map ((2^) . (compute_decoration g)) es)

to_dag 42
  array (0,5) [(0,[1,2,4]),(1,[3,5]),(2,[4,5]),
```

```
               (3,[4]),(4,[5]),(5,[])]
from_dag (to_dag 42)
  42
to_ddag 42
  array (0,5) [(0,[]),(1,[0]),(2,[0]),(3,[1]),
               (4,[3,2,0]),(5,[4,2,1])]
```

After implementing this function, we have found that it closely follows the *decoration* functions used in Aczel's book (Aczel 1988), 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 ((Aczel 1988)).

6.2 Extensional/Intensional Duality

What can be said about the graphs obtained by reversing the direction of the arrows representing the $\in$ relation? 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 (Goedel 1999) 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 can now look for the presence of intensional/extensional symmetry in HFS by trying to rebuild a HFS representation from the transpose of $\in$, $\ni$:

```
from_ddag g =
  compute_decoration g (snd (bounds g))

intensional_dual_of = from_ddag . to_ddag
```

Are such representations self-dual? Let's define as *self-dual* a number $n \in Nat$ that equals its intensional dual and then filter self-dual numbers in an interval:

```
self_idual n = n==intensional_dual_of n

self_iduals from to =
  filter self_idual [from..to]
```

Unfortunately, as the following example shows, relatively few numbers are self-duals:

```
self_iduals 0 1000
  [0,1,2,3,4,5,10,11,16,17,34,35,
   64,65,130,131,264,265,522,523]
```

Figures 3 and 4 show some HFS graphs of natural numbers equal to their intensional duals.

We will leave it as a topic for future research to investigate more in depth, various aspects of $\in$ / $\ni$ duality in HFS, in correlation with Natural Numbers and their encodings.

Figure 3: self-dual and its intensional dual as HFS graphs: *131* and *the dual of 131*

6.3 Encodings of Directed Graphs as Natural Numbers

Hypersets (Aczel 1988) are defined by replacing the Foundation Axiom with the AntiFoundation axiom. Intuitively this means that the $\in$-graphs can be cyclical (Barwise and Moss 1996), provided that they are minimized through *bisimulation equivalence* (Dovier et al. 2001). 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 (Aczel 1988; Barwise and Moss 1996; Piazza and Policriti 2004). 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 = map bitmerge_unpair (nat2set n)
```

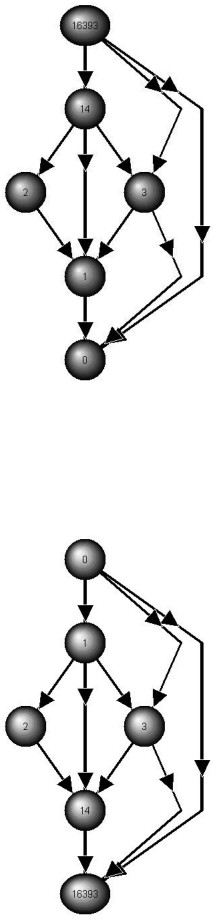

Figure 4: self-dual and its intensional dual as HFS graphs: *16393* and *the dual of 16393*

```
digraph2nat ps = set2nat (map bitmerge_pair ps)
```

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

```
nat2digraph 2008
  [(1,1),(2,0),(2,1),(3,1),
   (0,2),(1,2),(0,3)]
digraph2nat (nat2digraph 2008)
  2008
nat2digraph (255)
  [(0,0),(1,0),(0,1),(1,1),
  (2,0),(3,0),(2,1),(3,1)]
digraph2nat (nat2digraph 255)
  255
```

As usual

```
map nat2digraph [0..]
```

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

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 (Takahashi 1976; Kaye and Wong 2007; Kirby 2007; Abian and Lamacchia 1978; Kirby 2007; Meir et al. 1983; Leontjev and Sazonov 2000; Sazonov 1993; Avigad 1997). Graph representations of sets and hypersets based on the variants of the Anti Foundation Axiom have been studied extensively in (Aczel 1988; Barwise and Moss 1996). Computational and Data Representation aspects of Finite Set Theory and hypersets have been described in logic programming and theorem proving contexts in (Dovier et al. 2000; Piazza and Policriti 2004; Dovier et al. 2001; Paulson 1994). Pairing functions have been used work on decision problems as early as (Pepis 1938; Kalmar 1939; Robinson 1950, 1955, 1968a,b). Various mappings from natural number encodings to Rational Numbers are described in (Gibbons et al. 2006), also in a functional programming framework.

8. Conclusion and Future Work

Implementing with relative ease the encoding techniques typically used only in the foundations of mathematics recommends Haskell as a surprisingly effective tool for experimental mathematics.

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.

Last but not least, making more accessible to computer science students some of the encoding techniques typically used only in the foundations of mathematics (and related reasoning techniques), suggests applications to teaching discrete mathematics and/or functional languages in the tradition of (Hall and O'Donnell 2000).

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.

Peter Aczel. *Non-wellfounded sets*. Number 14 in CSLI Lecture Notes. Center for the Study of Language and Information (CSLI), 1988.

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

Jon Barwise and Lawrence Moss. *Vicious Circles*. Number 60 in CSLI Lecture Notes. Center for the Study of Language and Information (CSLI), 1996.

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.

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. ISBN 3-540-42345-1.

Jeremy Gibbons, David Lester, and Richard Bird. Enumerating the rationals. *Journal of Functional Programming*, 16(4), 2006. URL `http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/rationals.pdf`.

Kurt Goedel. Russel's mathematical logic. In A.D. Irvine, editor, *Bertrand Russell: Critical Assessments*, London, 1999. Routledge.

C. Hall and J. O'Donnell. *Discrete Mathematics Using a Computer*. Springer, 2000.

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.

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.

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.

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

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. ISBN 3-540-28372-2.

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.

Simon L. Peyton Jones. Haskell 98: Introduction. *J. Funct. Program.*, 13(1):0–6, 2003a.

Simon L. Peyton Jones. Haskell 98: Standard prelude. *J. Funct. Program.*, 13(1):103–124, 2003b.

Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. September 2002. http://haskell.org/definition/haskell98-report.pdf.

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

Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

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. 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.

Appendix

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

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").

```
-- 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

-- 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)
```

String Representations The function `setShow` provides a string representation of a natural number as a "pure" HFS.

```
setShow n = sShow urelement_limit n
```

The function `sShow` provides a string representation of a natural number as a HFS with *Urelements*.

```
sShow 1 0 = "{}"
sShow ulimit n | n<ulimit = show n
sShow ulimit n = "{"++
  foldl (++) ""
    (intersperse "," (map (sShow ulimit) (nat2set (n-ulimit))))
  ++"}"
```

Conversion to Ordered Pairs The function `nat2pairs` converts a natural number to a set of Haskell ordered pairs expressing the $\in$ relation on its associated HFS or its dual $\ni$.

```
nat2pairs withF n  = (sort . nub)  (nat2ps withF n)

nat2ps withF 0 = []
nat2ps withF from =
  ((n2rel ns) ++ (ns2rel ns)) where
    f = withF from
    n2rel = map f
    ns2rel = concatMap (nat2ps withF)
    ns=nat2set from
```

The function `nat2parts` converts n to the set of Natural Numbers occurring in the HFS representation of n.

```
nat2parts = sort . nub . nat2repeated where
  nat2repeated 0 = [0]
  nat2repeated from = from : (nat2more ns) where
    nat2more = concatMap nat2repeated
    ns=nat2set from
```