An Embedded Declarative Data Transformation Language

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

Abstract

We introduce a logic programming framework for data type transformations based on isomorphisms between elementary data types (natural numbers, finite functions, sets and permutations, digraphs, hypergraphs, etc.) and automatically derived extensions to hereditarily finite universes through *ranking/unranking* operations.

An embedded higher order combinator language provides any-to-any encodings automatically.

Applications range from stream iterators on combinatorial objects and uniform generation of random instances to succinct data representations and serialization of Prolog terms.

The self-contained source code of the paper, as generated from a literate Prolog program, is available at `http://logic.cse.unt.edu/tarau/research/2009/pISO.zip`

Keywords Prolog data representations, computational mathematics, ranking/unranking bijections, hereditarily finite sets and functions, digraph and hypergraph encodings

1. Introduction

Data structures in imperative languages have traditionally been designed with *mutability* in mind and therefore with space saving strategies based on in-place updates. On the contrary, the dominance of *immutable* data structures in declarative languages suggests sharing equivalent immutable components as an effective space saving alternative. Moreover, in the presence of higher order constructs, function sharing among heterogeneous data objects, is also appealing, as a way to borrow or lend "free algorithms". A relatively small number of universal data types are used as basic building blocks in programming languages and their runtime interpreters, corresponding to a few well tested mathematical abstractions like sets, functions, graphs, groups, categories etc.

However, this rises the question: what guaranties do we have that doing this between data types is useful and safe?

Also sharing heterogeneous data objects faces two problems:

- some form of equivalence needs to be proven between two objects `A` and `B` before `A` can replace `B` in a data structure, a possibly tedious and error prone task
- the fast growing diversity of data types makes harder and harder to recognize sharing opportunities.

The techniques introduced in this paper provide a generic solution to these problems, through isomorphic mappings between heterogeneous data types, such that unified internal representations make equivalence checking and sharing possible. The added benefit of these "shapeshifting" data types is that the functors transporting their data content will also transport their operations, resulting in shortcuts that provide, for free, implementations of interesting algorithms.

By ensuring that all data transformations are isomorphisms, i.e. reversible mappings that also transport operations, one obtains *encodings* from arbitrary data types to simpler and easier to manipulate representations – for instance natural numbers. Such encodings can be traced back to Gödel numberings [Gödel 1931, Hartmanis and Baker 1974] associated to formulae, but a wide diversity of common computer operations, ranging from wireless data transmissions to cryptographic codes qualify.

Encodings between data types provide a variety of services ranging from free iterators and random objects to data compression and succinct representations. Tasks like serialization and persistence are facilitated by simplification of reading or writing operations without the need of special purpose parsers. Sensitivity to internal data representation format or size limitations can be circumvented without extra programming effort.

The paper is organized as follows. Section 2 introduces the general framework for the paper, in the form of an embedded data transformation language, that is applied in section 3 to obtain encodings of some basic data types. Section 4 discusses a mechanism for lifting our transformations to hereditarily finite functions and sets. Sections 5 and 6 describe encodings of permutations and hereditarily finite permutations. After discussing some classic pairing functions, section 7 introduces pairing/unpairing operations that are used in section 8 to provide bijective encodings of graphs and hypergraphs as natural numbers. Section 9 describes natural number encodings of list operations in a simple LIPS-like programming language. Section 10 describes applications with focus on combinatorial generation, random instances, succinct representations of various data types and an encoding of Prolog terms. Sections 11 and 12 discuss related work, future work and conclusions.

The main contributions of the paper can be summarized as follows (with section/subsection numbers in parenthesis):

- a general framework for bijective encodings between heterogeneous datatypes in Prolog and an embedded combinator language providing automatic any-to-any encoding by routing through a common representation (a "hub") (2)
- a novel use of Prolog as an executable specification language for computational mathematics, involving emulation of lazy application and composition of higher order functions (closures) encapsulated as Prolog data objects (2.1)

- a mechanism for lifting encodings to hereditarily finite data types (4.1) and its application to derive Ackermann's encoding for hereditarily finite sets (4.1.1)
- two new instances of hereditarily finite representations derived from finite function and permutation encodings (4.1.2, 6)
- a number of applications, including a novel Prolog term encoder (10.4)
- a significant number of one-to-one encoders, claimed to be new, unless specified otherwise (through various sections the paper)
- a bijective natural number encoding of list processing code (3.1,9)
- a presentation of our results as a literate Prolog program directly testable for technical correctness and reusable as a public domain Prolog library

2. An Embedded Data Transformation Language

It is important to organize such encodings as a flexible embedded language to accommodate any-to-any conversions without the need to write one-to-one converters. We organize our encodings in terms of basic category theory [Mac Lane 1998] constructs as a *groupoid of isomorphisms* connecting various data types.

DEFINITION 1. *A groupoid is a category where every morphism is an isomorphism.*

We start by designing an embedded transformation language as a set of operations on this groupoid of isomorphisms. We then extend it with a set of higher order combinators mediating the composition of encodings and the transfer of operations between data types.

2.1 The Groupoid of Isomorphisms

We implement an isomorphism between two objects X and Y as a Prolog data type (a term with functor `iso/2`) `iso(F,G)`, encapsulating a bijection F and its inverse G.

$$\begin{tikzcd} X \arrow[r, shift left, "f = g^{-1}"] & Y \arrow[l, shift left, "g = f^{-1}"] \end{tikzcd}$$

As a well-known mechanism to embed higher order functions in Prolog [Warren 1981], we will use `iso/2` as a *closure* (higher order predicate) to be applied to an input argument and an output argument. We assume the presence of Prolog's `call/N` predicate that applies a closure to `N-1` extra arguments and `maplist/N` that applies a closure to `N-1` extra list arguments.

We organize our *groupoid* of isomorphisms as follows.

First we define the groupoid structure as a set of isomorphism transformers, designed to be encapsulated as Prolog terms, ready for future (lazy) evaluation:

```
compose(iso(F,G),iso(F1,G1),
  iso(fcompose(F1,F),fcompose(G,G1))).
itself(iso(id,id)).
invert(iso(F,G),iso(G,F)).
```

Then, we provide evaluators for isomorphisms, that apply their left or right functions to actual arguments. Note that like `iso/2`, compose/3 is a closure to be applied to 2 extra arguments with `call/2` or `maplist/2`.

```
fcompose(G,F,X,Y):-call(F,X,Z),call(G,Z,Y).
id(X,X).
from(iso(F,_),X,Y):-call(F,X,Y).
to(iso(_,G),X,Y):-call(G,X,Y).
```

The *from* function extracts the first component (a *section* in category theory parlance) and the *to* function extracts the second component (a *retraction*) defining the isomorphism. We can now formulate *laws* about isomorphisms that can be used to test correctness of implementations.

PROPOSITION 1. *The data type* `iso/2` *specifies a groupoid structure, i.e. the* compose *operation (when it can be applied) is associative,* itself *acts as an identity element and* invert *computes the inverse of an isomorphism.*

It is convenient to give a name to each isomorphism as a unary predicate

```
<name>(iso(From,To)).
```

We can transport operations from an object to another with *borrow* and *lend* combinators defined as follows:

```
borrow(IsoName,H,X,Y):-call(IsoName,iso(F,G)),
  fcompose(F,fcompose(H,G),X,Y).

lend(IsoName,H,X,Y):-call(IsoName,Iso),
  invert(Iso,iso(F,G)),
  fcompose(F,fcompose(H,G),X,Y).
```

We can see the combinators `from, to, compose, itself, invert, borrow, lend` as part of an *embedded data transformation language*. Various examples for their use will be given as soon as we populate our universe with interesting isomorphisms.

2.2 Routing isomorphisms through a *Hub*

To avoid defining $n(n-1)/2$ isomorphisms between n objects, we choose a *Hub* object to/from which we will actually implement isomorphisms. We will extend our embedded combinator language using the groupoid structure of the isomorphisms to connect any two objects through isomorphisms to/from the *Hub*.

Choosing a *hub* object is somewhat arbitrary, but it makes sense to pick a representation that is relatively easy convertible to various others, efficiently implementable and, last but not least, scalable to accommodate large objects up to the runtime system's actual memory limits.

We denote Nat the set of natural numbers. We will choose as our *hub* object *finite sequences of natural numbers*. They can be seen as *finite functions* from an initial segment of Nat, say $[0..n]$, to Nat. We will represent them as lists of (*arbitrary*) natural numbers denoted $[Nat]$. Note that in the case of a Prolog not supporting arbitrary precision integers or rationals, such lists could be used, in principle, to emulate them at source level, through the use of isomorphisms mapping them to natural numbers, signed integers and then dyadic rational numbers, following the techniques described in [Tarau 2009] in a functional programming context.

We can now define an *encoder* as an isomorphism connecting an object to our *hub* together with the combinators *with* and *as* providing an *embedded transformation language* for routing isomorphisms through two *encoders*.

```
with(Iso1,Iso2,Iso):-
  invert(Iso2,Inv2),compose(Iso1,Inv2,Iso).

as(That,This,X,Y):-
  % gets the actual encoders by calling <Name>(Iso)
  call(That,ThatIso), call(This,ThisIso),
  % routes through our hub
  with(ThatIso,ThisIso,Iso),
  % activates the isomorphism
  to(Iso,X,Y).
```

The combinator `with` turns two encoders into an arbitrary isomorphism, i.e. acts as a connection hub between their domains. The combinator `as` adds a more convenient syntax such that converters between "a" and "b" can be designed as:

```
'a2b'(X,Y) :- as('b','a',X,Y).
'b2a'(X,Y) :- as('a','b',X,Y).
```

$$\begin{tikzcd}
A \arrow[rr, shift left, "a2b"] \arrow[dr, shift left, "a^{-1}"'] & & B \arrow[ll, shift left, "b2a"] \arrow[dl, shift left, "b^{-1}"] \\
& Hub \arrow[ul, shift left, "a"'] \arrow[ur, shift left, "b"] &
\end{tikzcd}$$

We will provide extensive use cases for these combinators as we populate our groupoid of isomorphisms. For now, given that $[Nat]$ has been chosen as the *hub*, we will define our finite function data type *fun* simply as the identity isomorphism on sequences in $[Nat]$.

```
fun(Iso) :-itself(Iso).
```

3. Extending the groupoid of isomorphisms

We will now populate our groupoid of isomorphisms with combinators based on a few primitive converters.

3.1 An isomorphism between finite functions and natural numbers

We define the following predicates working on natural numbers:

```
cons(X,Y,XY):-X>=0,Y>=0,XY is (2**X)*(2*Y+1).
hd(XY,X):-XY>0,P is XY mod 2,hd1(P,XY,X).

hd1(1,_,0).
hd1(0,XY,X):-Z is XY // 2,hd(Z,H),X is H+1.

tl(XY,Y):-hd(XY,X),Y is XY // (2**(X+1)).

null(0).
```

Note that these operations are defined exclusively in terms of elementary arithmetic operations. Surprisingly, the following holds:

PROPOSITION 2. *The predicates* `cons/3,hd/2,tl/2,null/1` *emulate faithfully the list functions CONS,CAR,CDR,NIL as defined in [McCarthy 1960]*

Indeed, let's observe that this is a consequence of the fact that $\forall z \in \mathbb{N} - \{0\}$ the diophantic equation

$$2^x(2y+1) = z \qquad (1)$$

has exactly one solution $x, y \in \mathbb{N}$, which follows immediately from the unicity of the decomposition of a natural number as a multiset of prime factors.

Note also that following John McCarthy's *EVAL* construct one can now build relatively easily a programming language interpreter working directly and exclusively through simple arithmetic operations on natural numbers[1].

Using these predicates we define a bijection between finite functions and natural numbers:

```
fun2nat([],0).
fun2nat([X|Xs],N):-fun2nat(Xs,N1),cons(X,N1,N).

nat2fun(0,[]).
nat2fun(N,[X|Xs]):-N>0,hd(N,X),tl(N,T),nat2fun(T,Xs).
```

working as follows:

```
?- nat2fun(2009,Xs).
Xs = [0, 2, 0, 1, 0, 0, 0, 0] .
```

[1] We will describe this in more detail in section 9.

```
?- fun2nat([0, 2, 0, 1, 0, 0, 0, 0],N).
N = 2009.

nat(iso(nat2fun,fun2nat)).
```

The resulting encoder (`nat/1`) can now interoperate with the encoder `fun`:

```
?- as(fun,nat,42,F).
F = [1, 1, 1]

?- as(fun,nat,2008,F).
F = [3, 0, 1, 0, 0, 0, 0]

?- lend(nat,reverse,2008,R).
R = 1135 % different, sequence depends on order
```

3.2 An isomorphism to finite sets of natural numbers

The isomorphism is specified with two bijections `set2fun` and `fun2set`.

```
set(iso(set2fun,fun2set)).
```

It maps arbitrary lists of natural numbers representing finite functions to strictly increasing sequences of natural numbers representing sets. While finite sets and sequences share a common representation $[Nat]$, sets are subject to the implicit constraint that all their elements are distinct[2]. This suggest that a set like $\{7, 1, 4, 3\}$ could be represented by first ordering it as $\{1, 3, 4, 7\}$ and then compute the differences between consecutive elements. This gives $[1, 2, 1, 3]$, with the first element 1 followed by the increments $[2, 1, 3]$. To turn it into a bijection, including 0 as a possible member of a sequence, another adjustment is needed: elements in the sequence of increments should be replaced by their predecessors. This gives $[1, 1, 0, 2]$ as implemented by `set2fun/2`:

```
set2fun(Xs,Ns):-sort(Xs,Is),set2fun(Is,-1,Ns).

set2fun([],_,[]).
set2fun([X|Xs],Y,[A|As]):-A is (X-Y)-1,set2fun(Xs,X,As).
```

Incremental sums of the successors of numbers in such a sequence, return the original set in sorted form, as implemented by `fun2set/2`:

```
fun2set(Ns,Xs):-fun2set(Ns,-1,Xs).

fun2set([],_,[]).
fun2set([X|Xs],Y,[A|As]):-A is (X+Y)+1,fun2set(Xs,A,As).
```

The encoder (`set`) is now ready to interoperate with any another encoder:

```
?- as(set,fun,[0, 1, 0, 0, 4],S).
S = [0, 2, 3, 4, 9].

?- as(fun,set,[0, 2, 3, 4, 9],F).
F = [0, 1, 0, 0, 4].

?- as(set,nat,2009,Set).
Set = [0, 3, 4, 6, 7, 8, 9, 10].

?- as(nat,set,[0, 3, 4, 6, 7, 8, 9, 10],N).
N = 2009.
```

[2] Such constraints can be regarded as *laws/assertions* that we assume holding for a given data type, when needed, restricting it to the appropriate domain of the underlying mathematical concept. They assume the existence of an injective embedding from a *client* representation (sets of natural numbers in this case) to a *host* representation (finite sequences of natural numbers in this case). This assumption has been in use implicitly in various data representations and algorithms, but it is important to keep in mind that it is always based on the existence of such an injective embedding.

Note that as/4 works exactly as if isomorphisms were defined directly:

```
nat_set(iso(nat2set,set2nat)).

nat2set(N,Set):-nat2fun(N,F),fun2set(F,Set).
set2nat(Set,N):-set2fun(Set,F),fun2nat(F,N).


?- nat2set(2009,S).
S = [0, 3, 4, 6, 7, 8, 9, 10] .

?- set2nat([0, 3, 4, 6, 7, 8, 9, 10],N).
N = 2009.
```

4. Generic unranking and ranking hylomorphisms

The *ranking problem* for a family of combinatorial objects is finding a unique natural number associated to it, called its *rank*. The inverse *unranking problem* consists of generating a unique combinatorial object associated to each natural number.

4.1 Hereditarily finite data types

The unranking operation is seen here as an instance of a generic *anamorphism* mechanism (an *unfold* operation), while the ranking operation is seen as an instance of the corresponding catamorphism (a *fold* operation) [Meijer and Hutton 1995]. Together they form a mixed transformation called *hylomorphism*.

We will use such hylomorphisms to lift isomorphisms between lists and natural numbers to isomorphisms between a derived "self-similar" tree data type and natural numbers. In particular we will derive Ackermann's encoding from hereditarily finite sets to natural numbers.

The data type T representing hereditarily finite structures will be a generic multiway tree with a single leaf type [].

The two sides of our hylomorphism are parameterized by two transformations F and G forming an isomorphism iso(F,G):

```
unrank(F,N,R):-call(F,N,Y),unranks(F,Y,R).
unranks(F,Ns,Rs):-maplist(unrank(F),Ns,Rs).

rank(G,Ts,Rs):-ranks(G,Ts,Xs),call(G,Xs,Rs).
ranks(G,Ts,Rs):-maplist(rank(G),Ts,Rs).
```

Both combinators can be seen as a form of "structured recursion" that propagate a simpler operation guided by the structure of the data type. For instance, the size of a tree of type T is obtained as:

```
tsize1(Xs,N):-sumlist(Xs,S),N is S+1.

tsize(T,N) :- rank(tsize1,T,N).
```

Note also that unrank and rank work on trees in cooperation with unranks and ranks working on lists of trees.

We can now combine an anamorphism+catamorphism pair into an isomorphism hylo defined with rank and unrank on the corresponding hereditarily finite data types:

```
hylo(IsoName,iso(rank(G),unrank(F))):-
  call(IsoName,iso(F,G)).

hylos(IsoName,iso(ranks(G),unranks(F))):-
  call(IsoName,iso(F,G)).
```

4.1.1 A hylomorphism encoding hereditarily finite sets

Hereditarily finite sets will be represented as an encoder from multiway trees:

```
hfs(Iso):-
  hylo(nat_set,Hylo),
  nat(Nat),
  compose(Hylo,Nat,Iso).
```

The hfs encoder can now borrow operations from sets or natural numbers as follows:

```
hfs_succ(H,R):-borrow(nat_hfs,succ,H,R).
nat_hfs(Iso):-nat(Nat),hfs(HFS),with(Nat,HFS,Iso).


?- hfs_succ([],R).
R = [[]] ;
```

Otherwise, hylomorphism induced isomorphisms work as usual with our embedded transformation language:

```
?- as(hfs,nat,42,H).
H = [[[]], [[], [[]]], [[], [[[]]]]]
```

One can notice that we have just derived as a "free algorithm" Ackermann's encoding [Ackermann 1937, Piazza and Policriti 2004], from Hereditarily Finite Sets to natural numbers:

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

together with its inverse:

```
ackermann(H,N):-as(nat,hfs,H,N).
inverse_ackermann(N,H):-as(hfs,nat,N,H).
```

One 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_ackermann function as shown in Fig. 1.

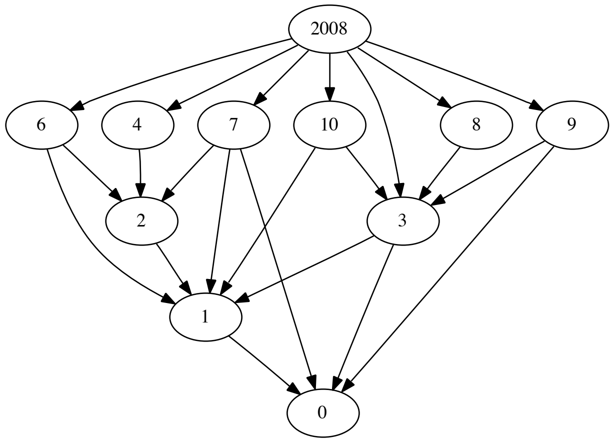

Figure 1. 2008 as a HFS

4.1.2 A hylomorphism encoding Hereditarily Finite Functions

The same tree data type can host a hylomorphism derived from finite functions instead of finite sets:

```
hff(Iso) :-
  hylo(nat,Hylo),nat(Nat),
  compose(Hylo,Nat,Iso).
```

The hff encoder can be seen as a "free algorithm", providing data compression/succinct representation for Hereditarily Finite Sets. Note, for instance, the significantly smaller tree size in:

```
?- as(hff,nat,42,H).
H = [[[]], [[]], [[]]]
```

As the cognoscenti might observe this is explained by the fact that hff provides higher information density than hfs, by incorporating order information that matters in the case of sequence and is ignored in the case of a set.

One can represent the action of a hylomorphism unfolding a natural number into a hereditarily finite function as a directed ordered multi-graph as shown in Fig. 2. Note that as the mapping as fun n generates a sequence where the order of the edges matters, this order is indicated with integers starting from 0 labeling the edges.

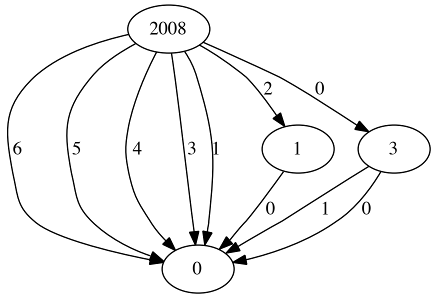

Figure 2. 2008 as a HFF

4.2 Mapping hereditarily finite representations to a parenthesis languages

An encoder for a parenthesis language is obtained by combining a parser and writer. As hereditarily finite functions naturally map one-to-one to parenthesis expressions expressed as bitstrings, we will choose them as target of the transformers.

The parser recurses over a bitstring encoding balanced parentheses [= 0,] = 1 and builds a HFF as follows:

```
pars2hff(Xs,T):-pars2term(0,1,T,Xs,[]).

pars2term(L,R,Xs) --> [L],pars2args(L,R,Xs).

pars2args(_,R,[]) --> [R].
pars2args(L,R,[X|Xs])-->
   pars2term(L,R,X),
   pars2args(L,R,Xs).
```

Note also that pars2hff is bidirectional i.e. it works both as an encoder and decoder.

```
?- pars_hff([0,0,1,0,1,1],T),pars_hff(Ps,T).
T = [[], []],
Ps = [0, 0, 1, 0, 1, 1]
```

We obtain the encoder:

```
pars(Iso):-hff(HFF),compose(iso(pars2hff,hff2pars),HFF,Iso).

hff2pars(HFF,Ps):-pars2hff(Ps,HFF).
```

working as follows:

```
?- as(pars,nat,42,Ps),as(nat,pars,Ps,N).
Ps = [0, 0, 0, 1, 1, 0, 0, 1, 1, 1],
N = 42
```

5. Encoding finite permutations

Sets represent "content" in a pure way - order is immaterial. Permutations represent "order" in a pure way - what is actually ordered is immaterial. We will show that a similar hereditarily finite 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*.

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

5.1 The factoradic numeral system

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

```
?- fr(42,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).
```

5.2 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 $\{j > i | f(j) < f(i)\}$ [Mantaci and Rakotondrajao 2001].

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),ndup(K,0,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
```

5.3 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_stream(N),sf(N,S).

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

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_stream(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:*

$$nat2perm \circ perm2nat \equiv id \equiv perm2nat \circ nat2perm \quad (2)$$

We obtain the encoder:

```
perm(Iso):-nat(Nat),compose(iso(perm2nat,nat2perm),Nat,Iso).
```

6. Hereditarily finite permutations

By using the generic ranking and unranking hylomorphism mechanism described in section 4 we can extend the nat2perm and perm2nat to encodings of hereditarily finite permutations (HFP).

```
nat_perm(iso(nat2perm,perm2nat)).

hfp(Iso):-
  hylo(nat_perm,Hylo),
  nat(Nat),
  compose(Hylo,Nat,Iso).
```

The encoding works as follows:

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

As shown in Fig 3, 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

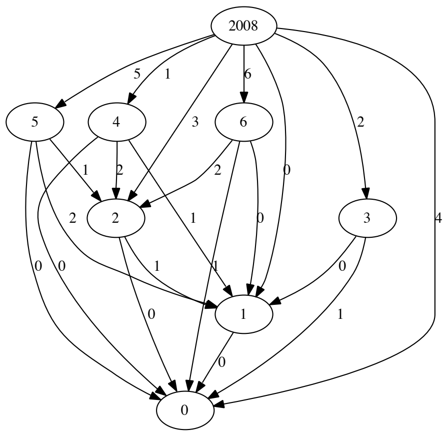

Figure 3. 2008 as a HFP

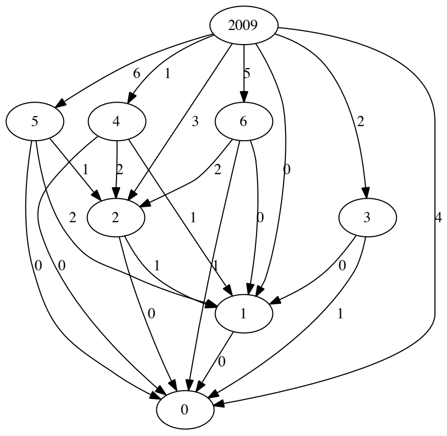

Figure 4. 2009 as a HFP

that moving from a number n to its successor typically only induces a reordering of the labeled edges, as shown in Fig. 4.

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. 5 compares sizes of HFS and HFP trees obtained from the same natural number up to 2^{10}.

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.

7. Pairing and Unpairing Functions

DEFINITION 2. *A* pairing function *is a bijection* $f : Nat \times Nat \rightarrow Nat$. *An* unpairing function *is a bijection* $g : Nat \rightarrow Nat \times Nat$.

Following Julia Robinson's notation [Robinson 1950], given a pairing function J, its left and right inverses K and L are such that

$$J(K(z), L(z)) = z \tag{3}$$

$$K(J(x, y)) = x \tag{4}$$

$$L(J(x, y)) = y \tag{5}$$

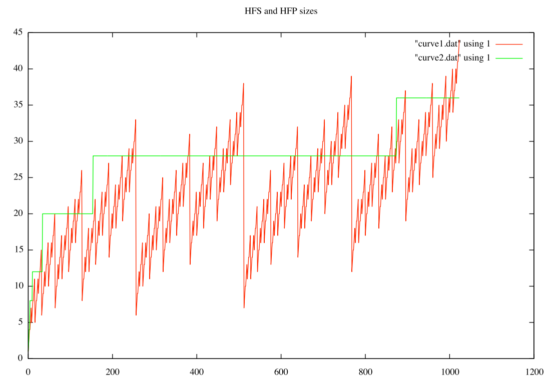

Figure 5. Comparison of curve1=HFS and curve2=HFP sizes up to 2^{10}

We refer to [Cégielski and Richard 2001] for a typical use in the foundations of mathematics and to [Rosenberg 2003] for an extensive study of various pairing functions and their computational properties.

7.1 Cantor's Pairing Function

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 [Robinson 1950, Cégielski and Richard 2001].

For $X, Y \in \{0, 1, 2, 3\}$ the sequence of values of this pairing function is:

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

Note however, that the inverse of Cantor's pairing function involves floating point operations that require emulation in terms of arbitrary length integers to avoid loosing precision.

```
cantor_unpair(Z,K1,K2):-
  I is floor((sqrt(8*Z+1)-1)/2),
  K1 is ((I*(3+I))//2)-Z,
  K2 is Z-((I*(I+1))//2).
```

7.2 Pairing/Unpairing operations acting directly on bitlists

We will describe here pairing operations, that are expressed exclusively as bitlist transformations of `bitunpair` and its inverse `bitpair`, and are therefore likely to be easily hardware implementable. As we have found out recently, they turn out to be the same as the functions defined in Steven Pigeon's PhD thesis on Data Compression [Pigeon 2001], page 114).

The predicate `bitpair` 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. The helper predicates `to_rbits` and `from_rbits`, given in the Appendix, convert to/from integers to bitlists.

```
bitpair(X,Y,P):-
  to_rbits(X,Xs),
  to_rbits(Y,Ys),
  bitmix(Xs,Ys,Ps),!,
  from_rbits(Ps,P).
```

```
bitunpair(P,X,Y):-
  to_rbits(P,Ps),
  bitmix(Xs,Ys,Ps),!,
  from_rbits(Xs,X),
  from_rbits(Ys,Y).

bitmix([X|Xs],Ys,[X|Ms]):-!,bitmix(Ys,Xs,Ms).
bitmix([],[X|Xs],[0|Ms]):-!,bitmix([X|Xs],[],Ms).
bitmix([],[],[]).
```

The transformation of the bitlists, done by the bidirectional predicate `bitmix/2` is shown in the following example with bitstrings aligned:

```
?- bitunpair(2008,X,Y),bitpair(X,Y,Z).
X = 60,
Y = 26,
Z = 2008

% 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
%   60:[0,    0,    1,    1,    1,    1]
%   26:[   0,    1,    0,    1,    1   ]
```

Note that we represent numbers with bits in reverse order (least significant first). Like in the case of Cantor's pairing function, we can see similar growth in both arguments:

```
?- between(0,15,N),bitunpair(N,A,B),
   write(N:(A,B)),write(' '),fail;nl.
0: (0, 0) 1: (1, 0) 2: (0, 1) 3: (1, 1)
4: (2, 0) 5: (3, 0) 6: (2, 1) 7: (3, 1)
8: (0, 2) 9: (1, 2) 10: (0, 3) 11: (1, 3)
12: (2, 2) 13: (3, 2) 14: (2, 3) 15: (3, 3)

?- between(0,3,A),between(0,3,B),bitpair(A,B,N),
   write(N:(A,B)),write(' '),fail;nl.
0: (0, 0) 2: (0, 1) 8: (0, 2) 10: (0, 3)
1: (1, 0) 3: (1, 1) 9: (1, 2) 11: (1, 3)
4: (2, 0) 6: (2, 1) 12: (2, 2) 14: (2, 3)
5: (3, 0) 7: (3, 1) 13: (3, 2) 15: (3, 3)
```

It is also convenient sometimes to see pairing/unpairing as one-to-one functions from/to the underlying language's ordered pairs, i.e. `X-Y` in Prolog :

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

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

We can derive the following encoder:

```
bnat2(Iso):-nat(Nat),
  compose(iso(bitpair,bitunpair),Nat,Iso).
```

working as follows:

```
?- as(bnat2,nat,0,Pair).
Pair = 0-0 .
?- as(bnat2,nat,2008,Pair).
Pair = 60-26
?- as(nat,bnat2,60-26,N).
N = 2008.
```

The following figures visualize properties of our pairing/unpairing functions. Given that unpairing functions are bijections from Nat to $Nat \times Nat$ they will progressively cover all points having natural number coordinates in their range in the plane. Figure 7 show the curve generated by `bitunpair`.

7.3 Deriving pairing/unpairing operations from hd,tl,cons

We will introduce here an unusually simple pairing/unpairing operation based on `cons, hd, tl` defined in subsection 3.1.

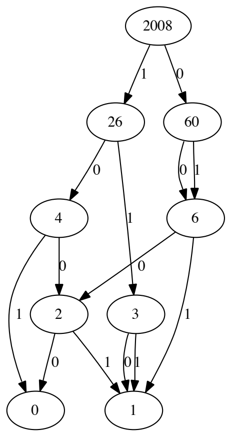

Figure 6. Graph obtained by recursive application of `bitunpair` for 2008

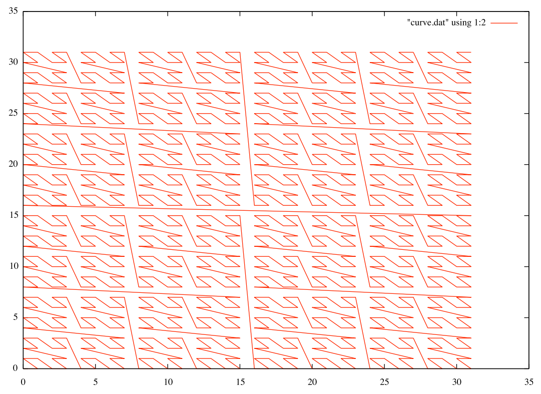

Figure 7. 2D curve connecting values of `bitunpair n` for $n \in [0..2^{10} - 1]$

By representing objects in $Nat \times Nat$ as terms of the form `p(_,_)`, we can now extend `cons/hd/tl` to a pairing/unpairing operation such that `(0,0)` corresponds to `0` as follows:

```
consUnPair(XY,p(X,Y)):-Z is XY+1,hd(Z,X),tl(Z,Y).
consPair(p(X,Y),XY):-cons(X,Y,Z),XY is Z-1.
```

We can derive the following encoder:

```
nat2(Iso):-nat(Nat),
  compose(iso(consPair,consUnPair),Nat,Iso).
```

working as follows:

```
?- as(nat2,nat,0,Pair).
Pair = p(0, 0) .
?- as(nat2,nat,2008,Pair).
Pair = p(0, 1004)
?- as(nat,nat2,p(0,1004),N).
N = 2008.
```

Figure 8 shows the directed graphs describing recursive application of `consUnPair`. Labels `0,1` on edges indicate position in the ordered pairs.

As the cognoscenti might notice, this is in fact a classic *pairing/unpairing function* that has been used, by Pepis, Kalmar and Robinson in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem in [Pepis 1938, Kalmar 1939, Robinson 1950].

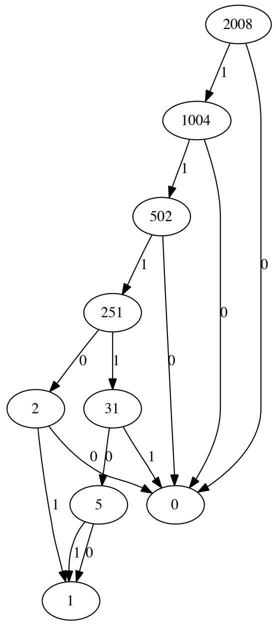

Figure 8. Graph obtained by recursive application of `consUnPair` for 2008

8. Encoding Directed Graphs and Hypergraphs

We will now show that more complex data types like digraphs and hypergraphs have extremely simple encoders. This shows once more the importance of compositionality in the design of our embedded transformation language.

8.1 Encoding Directed Graphs

First we will define an encoding for edges seen as pairs of vertices. We can find a bijection from directed graphs to finite sets by fusing their list of ordered pair representation into finite sets with a pairing function:

```
digraph2set(Ps,Ns) :- maplist(consPair,Ps,Ns).
set2digraph(Ns,Ps) :- maplist(consUnPair,Ns,Ps).
```

The resulting encoder is:

```
digraph(Iso):-set(Set),
   compose(iso(digraph2set,set2digraph),Set,Iso).
```

working as follows:

```
?- as(digraph,nat,2008,D),as(nat,digraph,D,N).
D = [p(2, 0), p(0, 2), p(0, 3), p(3, 0),
     p(0, 4), p(1, 2), p(0, 5)],
N = 2008.
```

Note that these encodings are generic in the sense that by changing the pairing/unpairing functions, a different encoder is obtained.

```
bdigraph2set(Ps,Ns) :- maplist(bitpair,Ps,Ns).
bset2digraph(Ns,Ps) :- maplist(bitunpair,Ns,Ps).
```

The resulting encoder

```
bdigraph(Iso):-set(Set),
   compose(iso(bdigraph2set,bset2digraph),Set,Iso).
```

works as follows:

```
?- as(bdigraph,nat,2008,D),as(nat,bdigraph,D,N).
D = [1-1, 2-0, 2-1, 3-1, 0-2, 1-2, 0-3],
N = 2008
```

Fig. 9 shows the digraph associated to 2008.

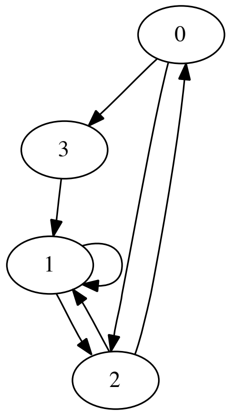

Figure 9. 2008 as a digraph

8.2 Encoding Hypergraphs

DEFINITION 3. *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.*

We can easily derive a bijective encoding of *hypergraphs*, represented as sets of sets:

```
set2hypergraph(S,G) :- maplist(nat2nonempty,S,G).
hypergraph2set(G,S) :- maplist(nonempty2nat,G,S).

nat2nonempty(N,S):-N1 is N+1,nat2set(N1,S).
nonempty2nat(S,N):-set2nat(S,N1),N is N1-1.
```

The resulting encoder is:

```
hypergraph(Iso):-set(Set),
  compose(iso(hypergraph2set,set2hypergraph),Set,Iso).
```

working as follows

```
?- as(hypergraph,nat,2009,G),as(nat,hypergraph,G,N).
G = [[0],[2],[0,2],[0,1,2],[3],[0,3],[1,3],[0,1,3]],
N = 2009 .
```

9. Encoding Programming Language Constructs

We have seen that `hd, tl, cons` have provided simple and elegant pairing/unpairing functions useful for encoding digraphs. We will now show that simple programming language constructs, higher order functions included, can be encoded on top of the purely arithmetic operations `hd,tl,cons`. A "list" concatenation operation `app` is defined as follows:

```
app(0,Ys,Ys).
app(XXs,Ys,XZs):-XXs>0,
  hd(XXs,X),tl(XXs,Xs),
  app(Xs,Ys,Zs),
  cons(X,Zs,XZs).
```

Note that `app/3` works on natural numbers "seen" as lists, for instance:

```
?- app(2008,2009,R).
R = 4116440.
```

One can observe that this emulates what happens when operands are first turned into conventional lists and the result is converted back after `append/3` is called:

```
?- as(fun,nat,2008,Ns).
Ns = [3, 0, 1, 0, 0, 0, 0].
```

```
?- as(fun,nat,2009,Ns).
Ns = [0, 2, 0, 1, 0, 0, 0, 0] .
?- append([3, 0, 1, 0, 0, 0, 0],
          [0, 2, 0, 1, 0, 0, 0, 0], R).
R = [3, 0, 1, 0, 0, 0, 0, 0, 2, 0, 1, 0, 0, 0, 0]
?- as(nat,fun,
      [3, 0, 1, 0, 0, 0, 0, 0, 2, 0, 1, 0, 0, 0, 0],N).
N = 4116440.
```

One can continue this process by defining the dictionary predicates `getAssoc/3, addAssoc/4` capable to host function definitions and binding environments.

```
getAssoc(_,0,0).
getAssoc(K,Ps,V):-K>0,hd(Ps,XY),
  getAssoc1(XY,K,Ps,V).

getAssoc1(0,_,0).
getAssoc1(XY,K,Ps,V):-
  tl(Ps,Qs),hd(XY,X),tl(XY,Y),
  getAssoc2(K,X,Y,Qs,V).

getAssoc2(K,K,Y,_,Y).
getAssoc2(K,X,_,Qs,V):-K=\=X,getAssoc(K,Qs,V).

addAssoc(K,V,Ps,Qs):-cons(K,V,KV),cons(KV,Ps,Qs).
```

working as follows:

```
?- addAssoc(1,2,0,Ps),addAssoc(2,1,Ps,Qs),
   getAssoc(2,Qs,V1),getAssoc(1,Qs,V2),
   getAssoc(3,Qs,V3).
Ps = 1024, Qs = 8392704,
V1 = 1,V2 = 2,V3 = 0.
```

Following [McCarthy 1960] one can build a fully arithmetized theory of recursive functions together with a Turing-equivalent `eval` predicate along the lines of [Chaitin 1975]. Our encodings are likely to be practical enough for experimenting with various concepts of algorithmic complexity and randomness [Li and Vitányi 1993].

10. Applications

Besides their utility as a uniform basis for a general purpose data conversion/serialization library, let us point out some specific applications of our isomorphisms.

10.1 Combinatorial Generation

A free combinatorial generation algorithm (providing a constructive proof of recursive enumerability) for a given structure is obtained simply through an isomorphism from *nat*:

```
nth(Thing,N,X) :- as(Thing,nat,N,X).

stream_of(Thing,X) :- nat_stream(N),nth(Thing,N,X).

?- nth(set,42,S).
S = [1, 3, 5]

?- stream_of(hfs,H).
H = [] ;
H = [[]] ;
H = [[[]]] ;
H = [[], [[]]] ;
H = [[[[]]]] ;
...
```

10.2 Random Generation

Combining `nth` with a random generator for *nat* provides free algorithms for random generation of complex objects of customizable size:

```
random_gen(Thing,Max,Len,X):-
  random_fun(Max,Len,Ns),
  as(Thing,fun,Ns,X).

random_fun(Max,Len,Ns):-length(Ns,Len),
  maplist(random_nat(Max),Ns).

random_nat(Max,N):-random(X),N is integer(Max*X).

?- random_gen(nat,100,4,R).
R = 26959946667150641291244691713864218\
      914210413126375567920582101041152.

?- random_gen(set,100,4,R).
R = [16, 39, 118, 168].

?- random_gen(fun,100,4,R).
R = [92, 60, 47, 76].

?- random_gen(hff,100,4,R).
R = [[[[],[]],[[]]],[[],[[]],[],[],
    [[]]],[[[]],[],[],[]],[[],[],[[]]]].

?- random_gen(hypergraph,100,4,R).
R = [[0, 1, 3, 6], [3, 5, 7],
     [1, 2, 6, 7], [0, 4, 5, 6, 7]].
```

Besides providing with `random_gen(nat,..)` arbitrary precision random numbers on top of the built-in limited precision floating point generator `random/1`, one can see that this technique can be used to implement elegantly random test generators in tools like QuickCheck [Claessen and Hughes 2002] without having to write data structure specific scripts.

10.3 Succinct Representations

Depending on the information theoretical density of various data representations as well as on the constant factors involved in various data structures, significant data compression can be achieved by choosing an alternate isomorphic representation, as shown in the following examples:

```
?- as(hff,hfs,[[[]], [[], [[]]], [[], [[[]]]]],HFF).
HFF = [[[]], [[]], [[]]]

?- as(nat,hff,[[[]], [[]], [[]]],N).
N = 42
```

In particular, mapping to efficient arbitrary length integer implementations (usually C-based libraries), can provide more compact representations or improved performance for isomorphic higher level data representations.

10.4 Encoding Prolog terms

An encoding of Prolog terms code has applications in succinct representation and serialization of terms - usable to send terms over a network connection, for instance.

We will sketch here an encoding mechanism that might also be useful to Prolog implementors interested in designing alternative heap representations for new Prolog runtime systems / abstract machine architectures.

First we provide an encoding that separates the “structure” of a term `T`, encoded as a parenthesis language representation of a hereditarily finite function `Ps` and a list of atomic terms and Prolog variables `As`, seen as a symbol table that stores the “content” of the terms:

```
term2bitpars(T,Ps,As):-term2bitpars(T,Ps,[],As,[]).

term2bitpars(T,Ps,Ps)-->{var(T)},[T].
term2bitpars(T,Ps,Ps)-->{atomic(T)},[T].
```

```
term2bitpars(T,[0|Ps],NewPs)-->{compound(T),T=..Xs},
  args2bitpars(Xs,Ps,NewPs).

args2bitpars([],[1|Ps],Ps)-->[].
args2bitpars([X|Xs],[0|Ps],NewPs)-->
  term2bitpars(X,Ps,[1|XPs]),
  args2bitpars(Xs,XPs,NewPs).
```

The encoding is reversible, i.e. the term T can be recovered:

```
bitpars2term(Ps,As,T):-bitpars2term(T,Ps,[],As,[]).

bitpars2term(T,Ps,Ps)-->[T].
bitpars2term(T,[0|Ps],NewPs)-->
  bitpars2args(Xs,Ps,NewPs),{T=..Xs}.

bitpars2args([],[1|Ps],Ps)-->[].
bitpars2args([X|Xs],[0|Ps],NewPs)-->
  bitpars2term(X,Ps,[1|XPs]),
  bitpars2args(Xs,XPs,NewPs).
```

The two transformations work as follows:

```
?- term2bitpars(f(g(a,X),X,42),Ps,As),bitpars2term(Ps,As,T).
Ps = [0,0,1,0,0,0,1,0,1,0,1,1,1,0,1,0,1,1],
As = [f,g,a,X,X,42],
T=f(g(a,X),X,42) .
```

After aggregating bitlists into natural numbers we obtain:

```
term2code(T,N,As):-term2bitpars(T,Ps,As),
  from_base(2,Ps,N).

code2term(N,As,T):-to_base(2,N,Ps),
  bitpars2term(Ps,As,T).
```

working as follows:

```
?- term2code(f(g(a,X),X,42),N,As),code2term(N,As,T).
N = 220484,
As = [f, g, a, X, X, 42],
T = f(g(a, X), X, 42) .
```

One can complete the encoding by hashing the symbol table into a list of small integers that can be encoded as a natural number using `nat2fun` and then aggregated with the result of `term2code` using a pairing function.

10.5 Other Applications

A fairly large number of useful algorithms in fields ranging from data compression, coding theory and cryptography to compilers, circuit design and computational complexity involve bijective functions between heterogeneous data types. Their systematic encapsulation in a generic API that coexists well with strong typing can bring significant simplifications to various software modules with the added benefits of reliability and easier maintenance.

In a Genetic Programming context [Koza 1992] the use of isomorphisms between bitvectors/natural numbers on one side, and trees/graphs representing HFSs, HFFs on the other side, looks like a promising phenotype-genotype connection. Mutations and crossovers in a data type close to the problem domain are transparently mapped to numerical domains where evaluation functions can be computed easily.

11. Related work

A preliminary draft of this paper is part of the CICLOPS'08 workshop's informal (online only) proceedings [Tarau 2008] and a large (104 pages) unpublished draft [Tarau 2009] discusses the same data type encoding methodology in the form of a literate Haskell program.

Ranking functions can be traced back to Gödel numberings [Gödel 1931, Hartmanis and Baker 1974] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [Martinez and Molinero 2003, Knuth 2006]. However the generic view, given in this paper, of such transformations as hylomorphisms obtained compositionally from simpler isomorphisms as well as the techniques used to encode them in Prolog are new.

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, Abian and Lamacchia 1978, Kirby 2007].

Computational and data representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in [Dovier et al. 2000, Piazza and Policriti 2004, Paulson 1994].

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.

An extensive study of various pairing functions and their computational properties is presented in [Rosenberg 2003].

A number of papers of J. Vuillemin develop similar techniques aiming to unify various data types, with focus on theories of boolean functions and arithmetics [Vuillemin 1994, 2003] and the use of Lehmer codes and permutation encodings [Vuillemin 1980].

12. Conclusion

We have shown the expressiveness of Prolog as a metalanguage for executable mathematics, by describing encodings for finite functions, sets and permutations in a uniform framework as data type isomorphisms with a groupoid structure. Prolog's higher order predicates and recursion patterns have helped the design of an embedded data transformation language.

The framework has been extended with hylomorphisms providing generic mechanisms for encoding hereditarily finite sets and Hereditarily finite functions. In the process, surprising "free algorithms" have emerged like Ackermann's encoding from hereditarily finite sets to natural numbers. We plan to explore in depth in the near future, some of the results that are likely to be of interest in fields ranging from combinatorics to data compression and arbitrary precision numerical computations.

While we have not explicitly provided a complexity analysis for various isomorphisms, it is clear from the actual code that our transformations typically work in time and space proportional to the overall size of the representation. In particular, when natural numbers are the source or the target, complexity is $O(log(N))$, given that $log(N)$ is the bitsize of the representation of N.

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.

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.

Gregory J. Chaitin. A theory of program size formally identical to information theory. *J. Assoc. Comput. Mach*, 22:329–340, 1975.

Koen Claessen and John Hughes. Testing monadic code with quickcheck. *SIGPLAN Notices*, 37(12):47–59, 2002.

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

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

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

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.

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

Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0201896842.

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

Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1993. ISBN 0-387-94053-7.

Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, New York, NY, USA, 1998.

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

Conrado Martinez and Xavier Molinero. Generic algorithms for the generation of combinatorial objects. In Branislav Rovan and Peter Vojtas, editors, *MFCS*, volume 2747 of *Lecture Notes in Computer Science*, pages 572–581. Springer, 2003.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367177.367199.

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

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

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

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

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

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

Arnold L. Rosenberg. Efficient pairing functions - and why you should care. *International Journal of Foundations of Computer Science*, 14(1):3–17, 2003.

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

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.

Paul Tarau. Declarative Combinatorics in Prolog: ShapeShifting Data Objects with Isomorphisms and Hylomorphisms. In Manuel Carro and Bart Demoen, editors, *Proceedings of CICLOPS 2008, 8th International Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 107–123, December 2008. URL `http://clip.dia.fi.upm.es/Conferences/CICLOPS-2008/CICLOPS-2008-proceedings.pdf`.

Jean Vuillemin. Digital algebra and circuits. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 733–746. Springer, 2003. ISBN 3-540-21002-4.

Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4): 229–239, 1980.

Jean Vuillemin. On circuits and numbers. *IEEE Trans. Computers*, 43(8): 868–879, 1994.

D. H. D. Warren. Higher-order extensions to Prolog – are they needed? In D. Michie, J. Hayes, and Y. H. Pao, editors, *Machine Intelligence 10*. Ellis Horwood, 1981.

Appendix

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

```
% converts an integer to a list of bits
% least significant first
to_rbits(0,[]).
to_rbits(N,[B|Bs]):-N>0,B is N mod 2, N1 is N//2,
  to_rbits(N1,Bs).

% converts a list of bits (least significant first)
% into an integer
from_rbits(Rs,N):-nonvar(Rs),from_rbits(Rs,0,0,N).

from_rbits([],_,N,N).
from_rbits([X|Xs],E,N1,N3):-NewE is E+1,N2 is X<<E+N1,
  from_rbits(Xs,NewE,N2,N3).

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

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

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

% 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).
```