Declarative Combinatorics in Prolog: Shapeshifting Data Objects with Isomorphisms and Hylomorphisms

Paul Tarau

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

Abstract. This paper is an exploration in a logic programming framework of isomorphisms between elementary data types (natural numbers, sets, finite functions, graphs, hypergraphs) and their extension to hereditarily finite universes through *hylomorphisms* derived from *ranking/unranking* and *pairing/unpairing* operations.
An embedded higher order combinator language provides any-to-any encodings automatically.
A few examples of "free algorithms" obtained by transferring operations between data types are shown. Other applications range from stream iterators on combinatorial objects to succinct data representations and generation of random instances.
The self-contained source code of the paper, as generated from a literate Prolog program, is available at `http://logic.csci.unt.edu/tarau/research/2008/pISO.zip`
Keywords: Prolog data representations, computational mathematics, ranking/unranking, Ackermann encoding, hereditarily finite sets and functions, pairing/unpairing

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".

The closest analogy to this, drawn from everyday thinking, is ... *analogy*. Analogical/metaphoric thinking routinely shifts entities and operations from a field to another hoping to uncover similarities in representation or use.

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. The simplest instance is the case of isomorphisms – reversible mappings that also transport operations. In their simplest form such isomorphisms show up as *encodings* – to some simpler and easier to manipulate representation – for instance natural numbers.

Such encodings can be traced back to Gödel numberings [8, 10] 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.

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. Toward this end we will organize our encodings as a group of isomorphisms within a (mildly) category theory-inspired design.

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

2.1 The Group 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.

$$X \underset{g = f^{-1}}{\overset{f = g^{-1}}{\rightleftarrows}} Y$$

As a well-known mechanism to embed higher order functions in Prolog [30], 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` extra arguments and `maplist/N` that applies a closure to `N` extra list arguments. We can organize the *group* of isomorphisms as follows.

First we define the group structure as a set of isomorphism transformers:

```
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 group structure, i.e. the* compose *operation is associative,* itself *acts as an identity element and* invert *computes the inverse of an isomorphism.*

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).
```

The combinators `fit` and `retrofit` just transport an object `x` through an isomorphism and and apply to it an operation `op` available on the other side:

```
fit(Op,IsoN,X,Y):-
  call(IsoN,Iso),
  fit_iso(Op,Iso,X,Y).

fit_iso(Op,Iso,X,Y):-
  from(Iso,X,Z),
  call(Op,Z,Y).
```

```
retrofit(Op,IsoN,X,Y):-
  call(IsoN,Iso),
  retrofit_iso(Op,Iso,X,Y).

retrofit_iso(Op,Iso,X,Y):-
  to(Iso,X,Z),
  call(Op,Z,Y).
```

We can see the combinators `from, to, compose, itself, invert, borrow, lend, fit etc.` as part of an *embedded data transformation language*.

2.2 Choosing a Root

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

Choosing a *Root* 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 will choose as our *Root* object *Finite Sequences of Natural Numbers*. They can be seen as as finite functions from an initial segment of Nat, say $[0..n]$, to Nat. We will represent them as lists i.e. their Prolog type is $[Nat]$. Alternatively, an array representation can be chosen.

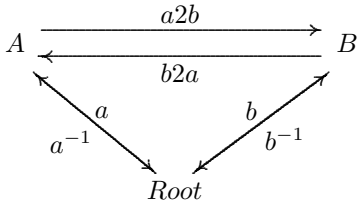
We can now define an *Encoder* as an isomorphism connecting an object to *Root* together with the combinators *with* and *as* providing an *embedded transformation language* for routing isomorphisms through two *Encoders*.

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

as(That,This,X,Y):-
  call(That,ThatF),
  call(This,ThisF),
  with(ThatF,ThisF,Iso),
  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('a','b',X,Y).
'b2a'(X,Y) :- as('b','a',X,Y).
```



We will provide extensive use cases for these combinators as we populate our group of isomorphisms. Given that $[Nat]$ has been chosen as the root, 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 Group of Isomorphisms

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

3.1 An Isomorphism to Finite Sets of Natural Numbers

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

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

While finite sets and sequences share a common representation $[Nat]$, sets are subject to the implicit constraint that all their elements are distinct[1]. 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`:

```
set2fun([],[]).
set2fun([X|Xs],[X|Fs]):-
  sort([X|Xs],[_|Ys]),
  set2fun(Ys,X,Fs).

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

It can now be verified easily that incremental sums of the successors of numbers in such a sequence, return the original set in sorted form, as implemented by `fun2set`:

[1] Such constraints can be regarded as *laws* that we assume about a given data type, when needed, restricting it to the appropriate domain of the underlying mathematical concept.

```
fun2set([],[]).
fun2set([A|As],Xs):-findall(X,prefix_sum(A,As,X),Xs).

prefix_sum(A,As,R):-append(Ps,_,As),length(Ps,L),
  sumlist(Ps,S),R is A+S+L.
```

The resulting Encoder (`set`) is now ready to interoperate with 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 the example shows, this encoding maps arbitrary lists of natural numbers representing finite functions to strictly increasing sequences of natural numbers representing sets.

3.2 Folding Sets into Natural Numbers

We can fold a set, represented as a list of distinct natural numbers into a single natural number, reversibly, by observing that it can be seen as the list of exponents of 2 in the number's base 2 representation.

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

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

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

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

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

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

We will standardize this pair of operations as an *Encoder* for a natural number using our Root as a mediator:

```
nat(Iso):-
  nat_set(NatSet),
  set(Set),
  compose(NatSet,Set,Iso).
```

The resulting Encoder (`nat`) is now ready to interoperate with any other Encoder:

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

?- as(set,nat,42,F).
F = [1, 3, 5]

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

?- as(set,nat,2008,S).
S = [3, 4, 6, 7, 8, 9, 10]

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

?- lend(nat_set,reverse,2008,R).
R = 2008 % same, set is order independent

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

?- fit(length,nat,42,L).
L = 3

?- retrofit(succ,nat_set,[1,3,5],N).
N = 43
```

The reader might notice at this point that we have already made full circle - as finite sets can be seen as instances of finite sequences. Injective functions that are not surjections with wider and wider gaps can be generated using the fact that one of the representations is information theoretically "denser" than the other, for a given range:

```
?- as(set,fun,[0,1,2,3],S1).
S1 = [0, 2, 5, 9].

?- as(set,fun,[0,2,5,9],S2).
S2 = [0, 3, 9, 19].

?- as(set,fun,[0,3,9,19],S3).
S3 = [0, 4, 14, 34].
```

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 Pure 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) [12, 19]. 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(IsoN,iso(rank(G),unrank(F))):-call(IsoN,iso(F,G)).

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

Hereditarily Finite Sets Hereditarily Finite Sets will be represented as an Encoder for the tree type `T`:

```
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 [2, 22], 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(N,H):-as(nat,hfs,N,H).
inverse_ackermann(H,N):-as(hfs,nat,H,N).
```

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 another "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.

5 Pairing/Unpairing

A *pairing* function is an isomorphism $f : Nat \times Nat \to Nat$. Its inverse is called *unpairing*.

We will introduce here an unusually simple pairing function (also mentioned in [23], p.142).

The function `bitpair` works by splitting a number's big endian bitstring representation into odd and even bits.

```
bitpair(p(I,J),P):-
  evens(I,Es),
  odds(J,Os),
  append(Es,Os,Ps),
  set2nat(Ps,P).

evens(X,Es):-nat2set(X,Ns),maplist(double,Ns,Es).
odds(X,Os):-evens(X,Es),maplist(succ,Es,Os).
double(N,D):-D is 2*N.
```

The inverse function `bitunpair` blends the odd and even bits back together.

```
bitunpair(N,p(E,O)):-
  nat2set(N,Ns),
  split_evens_odds(Ns,Es,Os),
  set2nat(Es,E),
  set2nat(Os,O).

split_evens_odds([],[],[]).
split_evens_odds([X|Xs],[E|Es],Os):-
  X mod 2 =:= 0,
  E is X // 2,
  split_evens_odds(Xs,Es,Os).
split_evens_odds([X|Xs],Es,[O|Os]):-
  X mod 2 =:= 1,
  O is X // 2,
  split_evens_odds(Xs,Es,Os).
```

The transformation of the bitlists is shown in the following example with bitstrings aligned:

```
?-bitunpair(2008,R)
R = p(60,26)

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

We can derive the following Encoder:

```
nat2(Iso):-
  nat(Nat),
  compose(iso(bitpair,bitunpair),Nat,Iso).
```

working as follows:

```
?- as(nat2,nat,2008,Pair).
Pair = p(60, 26)

?- as(nat,nat2,p(60,26),N).
N = 2008
```

6 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.

6.1 Encoding Directed Graphs

We can find a bijection from directed graphs (with no isolated vertices, corresponding to their view as binary relations), to finite sets by fusing their list of ordered pair representation into finite sets with a pairing function:

```
digraph2set(Ps,Ns) :- maplist(bitpair,Ps,Ns).
set2digraph(Ns,Ps) :- maplist(bitunpair,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(1, 1), p(2, 0), p(2, 1), p(3, 1), p(0, 2), p(1, 2), p(0, 3)],
N = 2008
```

6.2 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.*

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

```
set2hypergraph(S,G) :- maplist(nat2set,S,G).
hypergraph2set(G,S) :- maplist(set2nat,G,S).
```

The resulting Encoder is:

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

working as follows

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

7 Applications

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

7.1 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).

nat_stream(0).
nat_stream(N):-nat_stream(N1),succ(N1,N).


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

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

7.2 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(set,100,4,R).
R = [16, 39, 118, 168].

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

?- random_gen(nat,100,4,R).
R = 269599466671506412912446917138642189142104131263755679205821010411 52.

?- random_gen(hfs,4,3,R).
R = [[[]], [[], [[[]]]], [[[]], [[], [[]]]]

?- random_gen(hff,4,3,R).
R = [[], [], [[]]]
```

Besides providing arbitrary precision random numbers as a "free algorithm" on top of a builtin limited precision floating point generator, one can see that this technique can be used to implement elegantly random test generators in tools like QuickCheck [6] without having to write data structure specific scripts.

7.3 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. We can compare representations sharing a common datatype to conjecture about their asymptotic information density.

7.4 Experimental Mathematics

For instance, after defining:

```
length_as(Thing,X,Len) :-
  nat(Nat),
  call(Thing,T),
  with(Nat,T,Iso),
  fit_iso(length,Iso,X,Len).

sum_as(Thing,X,Len) :-
```

```
  nat(Nat),
  call(Thing,T),
  with(Nat,T,Iso),
  fit_iso(sumlist,Iso,X,Len).

size_as(Thing,X,Len) :-
  nat(Nat),
  call(Thing,T),
  with(Nat,T,Iso),
  fit_iso(tsize,Iso,X,Len).
```

one can conjecture that finite functions are more compact than sets asymptotically

```
1 ?- length_as(fun,123456789012345678901234567890,L).
L = 54

2 ?- length_as(set,123456789012345678901234567890,L).
L = 54

3 ?- length_as(fun,123456789012345678901234567890,L).
L = 54

4 ?- sum_as(set,123456789012345678901234567890,L).
L = 2690

5 ?- sum_as(fun,123456789012345678901234567890,L).
L = 43
```

and then observe that the same trend applies also to their hereditarily finite derivatives:

```
?- size_as(hfs,123456789012345678901234567890,L).
L = 627

?- size_as(hff,123456789012345678901234567890,L).
L = 91
```

7.5 A surprising "free algorithm": strange_sort

A simple isomorphism like `nat_set` can exhibit interesting properties as a building block of more intricate mappings like Ackermann's encoding, but let's also note a (surprising to us) "free algorithm" – sorting a list of distinct elements without explicit use of comparison operations:

```
strange_sort(Unsorted,Sorted):-
  nat_set(Iso),
  to(Iso,Unsorted,Ns),
  from(Iso,Ns,Sorted).
```

```
?- strange_sort([2,9,3,1,5,0,7,4,8,6],Sorted).
Sorted = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This algorithm emerges as a consequence of the commutativity of addition and the unicity of the decomposition of a natural number as a sum of powers of 2. The cognoscenti might notice that such surprises are not totally unexpected. In a functional programming context, they go back as early as Wadler's Free Theorems [29].

7.6 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 [16] 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. In the context of Software Transaction Memory implementations (like Haskell's STM [9]), encodings through isomorphisms are subject to efficient shortcuts, as undo operations in case of transaction failure can be performed by applying inverse transformations without the need to save the intermediate chain of data structures involved.

8 Related work

This work can be seen as part of a larger effort to cover in a declarative programming paradigm some fundamental combinatorial generation algorithms along the lines of Donald Knuth's recent work [15].

The closest reference on encapsulating bijections as a data type is [3] and Connan Eliot's composable bijections Haskell module [7], where, in a more complex setting, Arrows [11] are used as the underlying abstractions. While our `Iso` data type is similar to the *Bij* data type in [7] and BiArrow concept of [3], the techniques for using such isomorphisms as building blocks of an embedded composition language centered around encodings as Natural Numbers are new.

Ranking functions can be traced back to Gödel numberings [8, 10] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [18, 15, 27, 20]. However the generic view of such transformations as hylomorphisms obtained compositionally from simpler isomorphisms, as described in this paper, is 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 [28, 13, 1, 4, 14, 17].

Computational and Data Representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in [22, 21].

Pairing functions have been used in work on decision problems as early as [24, 25]. A typical use in the foundations of mathematics is [5]. An extensive study of various pairing functions and their computational properties is presented in [26].

9 Conclusion

We have shown the expressiveness of Prolog as a metalanguage for executable mathematics, by describing encodings for functions and finite sets in a uniform framework as data type isomorphisms with a group structure. Prolog's higher order predicates and recursion patterns have helped the design of an embedded data transformation language. Using higher order combinators a simplified random testing mechanism has been implemented as an empirical correctness test. The framework has been extended with hylomorphisms providing generic mechanisms for encoding Hereditarily Finite Sets and Hereditarily Finite Functions. In the process, a few surprising "free algorithms" have emerged, including 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.

References

1. Alexander Abian and Samuel Lamacchia. On the consistency and independence of some set-theoretical constructs. *Notre Dame Journal of Formal Logic*, X1X(1):155–158, 1978.
2. Wilhelm Friedrich Ackermann. Die Widerspruchsfreiheit der allgemeinen Mengenlhere. *Mathematische Annalen*, (114):305–315, 1937.
3. Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 86–97, New York, NY, USA, 2005. ACM Press.
4. Jeremy Avigad. The Combinatorics of Propositional Provability. In *ASL Winter Meeting*, San Diego, January 1997.
5. Patrick Cégielski and Denis Richard. Decidability of the theory of the natural integers with the cantor pairing function and the successor. *Theor. Comput. Sci.*, 257(1-2):51–77, 2001.
6. Koen Claessen and John Hughes. Testing monadic code with quickcheck. *SIGPLAN Notices*, 37(12):47–59, 2002.
7. Connan Eliot. Data.Bijections Haskell Module. http://haskell.org/haskellwiki/TypeCompose.
8. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
9. Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

10. 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.
11. John Hughes. Generalizing Monads to Arrows. Science of Computer Programming 37, pp. 67-111, May 2000.
12. Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.*, 9(4):355–372, 1999.
13. Richard Kaye and Tin Lock Wong. On Interpretations of Arithmetic and Set Theory. *Notre Dame J. Formal Logic Volume*, 48(4):497–510, 2007.
14. Laurence Kirby. Addition and multiplication of sets. *Math. Log. Q.*, 53(1):52–65, 2007.
15. Donald Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
16. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
17. 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.
18. 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.
19. Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *FPCA*, pages 324–333, 1995.
20. Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79:281–284, 2001.
21. 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.
22. Carla Piazza and Alberto Policriti. Ackermann Encoding, Bisimulations, and OBDDs. *TPLP*, 4(5-6):695–718, 2004.
23. Stephen Pigeon. *Contributions à la compression de données.* Ph.d. thesis, Université de Montréal, Montréal, 2001.
24. Julia Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950.
25. Julia Robinson. Finite generation of recursively enumerable sets. *Proceedings of the American Mathematical Society*, 19(6):1480–1486, dec 1968.
26. Arnold L. Rosenberg. Efficient pairing functions - and why you should care. In *IPDPS*. IEEE Computer Society, 2002.
27. Frank Ruskey and Andrzej Proskurowski. Generating binary trees by transpositions. *J. Algorithms*, 11:68–84, 1990.
28. Moto-o Takahashi. A Foundation of Finite Mathematics. *Publ. Res. Inst. Math. Sci.*, 12(3):577–708, 1976.
29. Philip Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, New York, NY, USA, 1989. ACM.
30. 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.