An Embedded Declarative Data Transformation Language

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

PPDP'09, Sept. 2009

Motivation

- analogies (and analogies between analogies) emerge when we transport objects and operations on them
- this is a creative process - it has been one of the most rewarding ones in terms of interesting outcomes (geometry and coordinates, primes and complex functions, cryptography and number theory, Turing machines and combinators, types and proofs etc.)
- $\longrightarrow$ let's provide a computational catalyst!

- to be able to encode something as something else we need isomorphisms $\longrightarrow$ bijections that transport structures
- $\longrightarrow$ the paper is about how we can (somewhat) automate this process by organizing such encodings . . . nicely. . .

Practical uses of datatype isomorphisms?

- the tip of the iceberg:
  - we can transfer operations between datatypes
  - "free algorithms" can emerge
  - sharing opportunities across heterogeneous data types
  - free iterators and random instance generators
  - data compression and succinct representations
  - a general mechanism for serialization and persistence
- more generally:
  $\longrightarrow$ automate the process of finding "computational analogies"
  $\longrightarrow$ discover new things about of a given datatype by "teleporting" operations from other datatypes

Outline

- a logic programming framework to encode and propagate isomorphisms between elementary data types that borrows some *very basic* concepts and results from a few different fields: combinatorics, recursion theory, foundations of set theory, categories, number theory, graph theory ...
- ranking/unranking operations (bijective Gödel numberings)
- isomorphisms using pairing/unpairing functions
- generating new isomorphisms through hylomorphisms (folding/unfolding into hereditarily finite universes)
- applications

the paper is also a *literate Prolog program* – see more about the same research thread in a functional programming context in a 150++ pages Haskell paper at `http://logic.csci.unt.edu/tarau/research/2009/fISO.pdf`

The Groupoid of Isomorphisms

```
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)).
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).
```

Proposition

We have a groupoid of isomorphisms: when defined, `compose` *is associative,* `itself` *is an identity element,* `invert` *computes the inverse of an isomorphism.*

Transporting operations

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

```
borrow(iso(F,G),Op,X,Y):-
   fcompose(F,fcompose(Op,G),X,Y).

lend(iso(G,F),Op,X,Y):-
   fcompose(F,fcompose(Op,G),X,Y).
```

To simplify operations on the groupoid, it is convenient to give a name to each isomorphism as a unary predicate of the form

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

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 can now define an *Encoder* as an isomorphism connecting an object to our *hub*
- the combinators *with* and *as* provide 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).
```

Finite Functions to/from Sets

We will choose as our *hub* object *finite sequences of natural numbers*. Connecting finite sets to our *hub*:

```
?- 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 (distinct) natural numbers representing sets.

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.

```
?- 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]
```

Ranking/unranking hereditarily finite datatypes

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).
```

"structured recursion": propagate a simpler operation guided by the structure of the data type obtained as:

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

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

Extending isomorphisms with hylomorphisms

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)).
```

Hereditarily finite sets

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

?- as(hfs,nat,2008,H),as(nat,hfs,H,N).
H=[[[], []]], [[[[]]]], [[[]]], [[[]]]], [[], []],
   [[[]]]], [[[], []]]], [[], []]]], [[[]],
   [[], []]]]]
N = 2008 .
```

we derive as a “free algorithm” *Ackermann’s encoding*

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

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

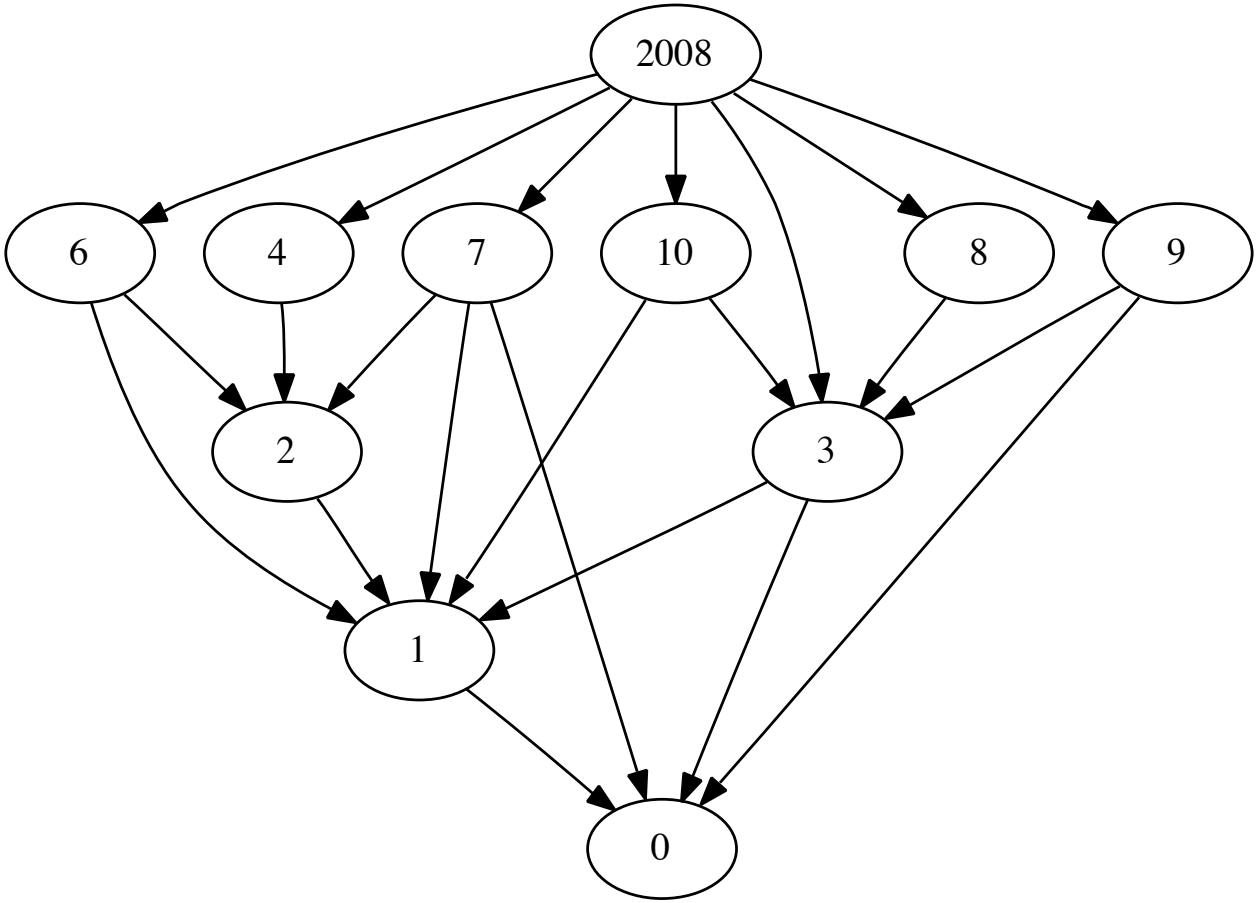


Figure: 2008 as a HFS

Hereditarily finite functions

this `hff` Encoder can be seen as another (new this time!) “free algorithm”, providing data compression/succinct representation for hereditarily finite sets (note the significantly smaller tree size):

```
?- as(hff,nat,2008,H),as(nat,hff,H,N).
H = [[[], []], [], [[]], [], [], [], []],
N = 2008 .
```

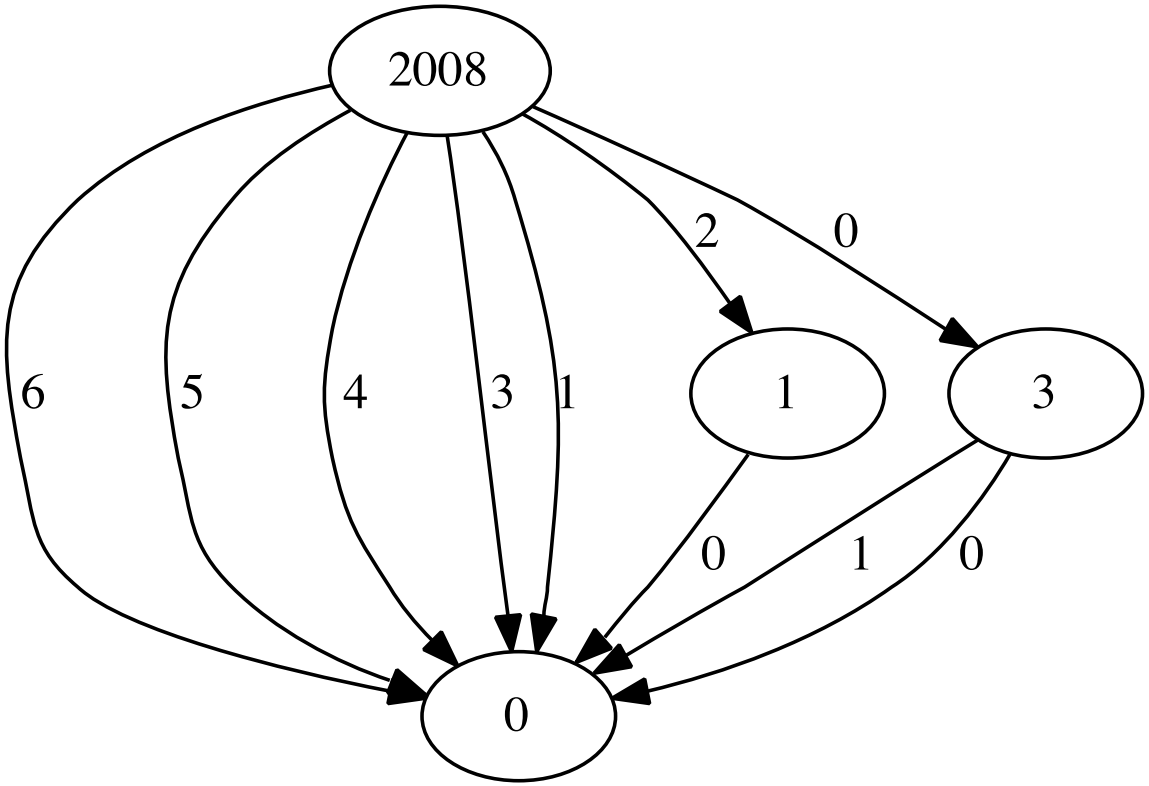


Figure: 2008 as a HFF

Ranking/unranking of permutations of given length

- The factoradic numeral system replaces digits multiplied by a 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 $d_n \in [0..n]$.
- $42 = 0 * 0! + 0 * 1! + 0 * 2! + 3 * 3! + 1 * 4!$.

```
42 : [0,0,0,3,1]
```

ranking/unranking combines factoradics and the Lehmer code of a permutation f of size n i.e. 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 $\{j > i | f(j) < f(i)\}$

Ranking/unranking of arbitrary finite permutations

To extend the mapping from permutations of a given length to arbitrary permutations we "shift towards infinity" the starting point of each new block of permutations as permutations of larger and larger sizes are enumerated. The Encoder is `perm/1`:

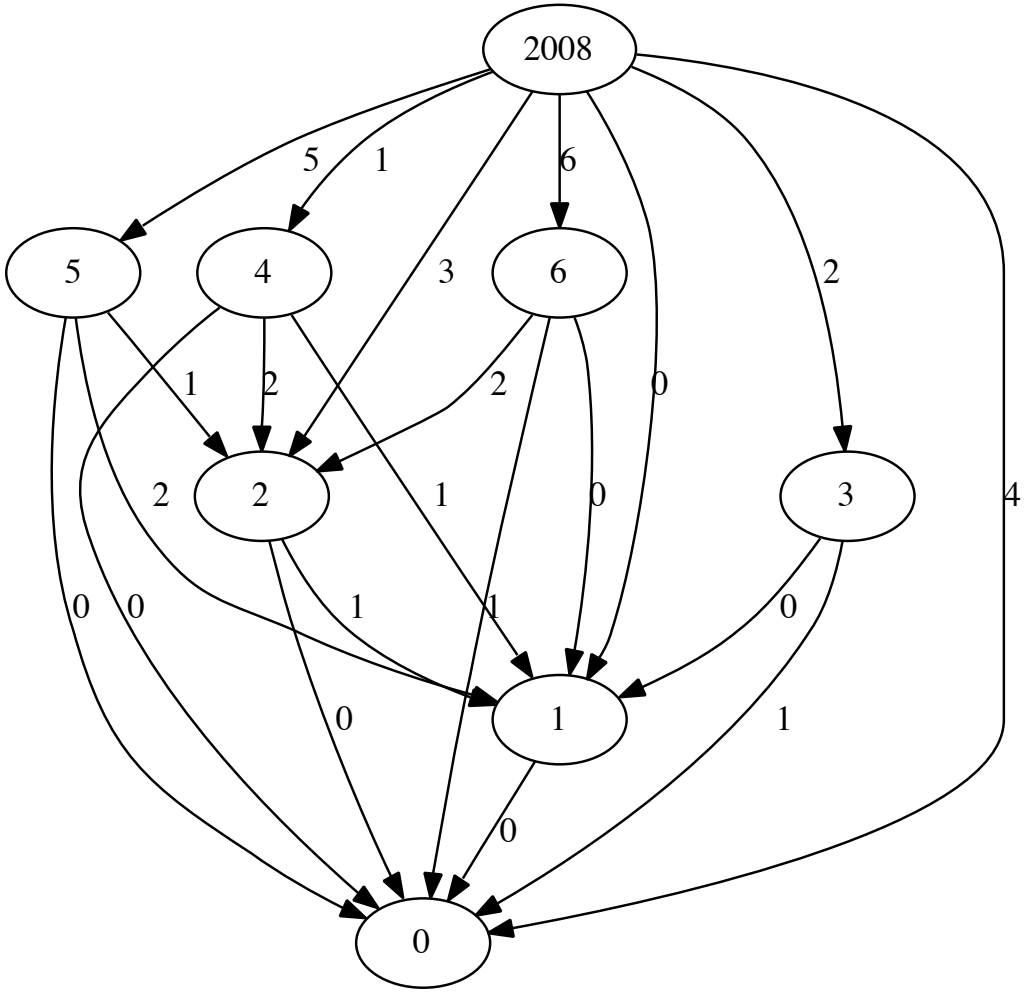
```
perm(Iso):-
  nat(Nat),
  compose(iso(perm2nat,nat2perm),Nat,Iso).

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

Hereditarily finite permutations

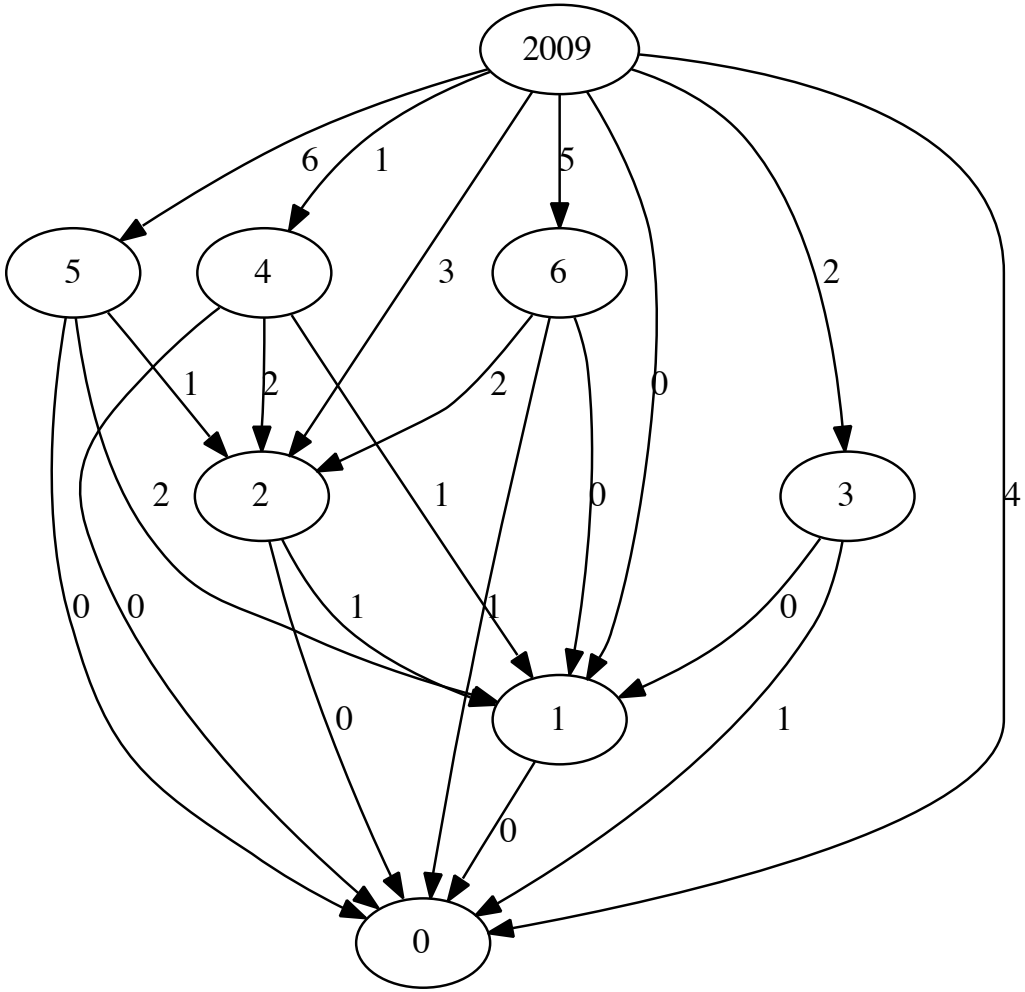
Interesting to note:

- sets: order unimportant
- permutations: content unimportant
- sequences - encodings use both order and content

Hereditarily finite permutation associated to **2008**



Hereditarily finite permutation associated to **2009**



Pairing/Unpairing

- *pairing* function: isomorphism $f : Nat \times Nat \longrightarrow Nat$; inverse: *unpairing*
- Cantor's encoding: requires integer sqrt, Newton's method.
- we need more efficient ones i.e. by using just simple bitstring operations.

```
?- 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   ]
```

Recursive unpairing graph for a *well-founded* pairing/unpairing bijection

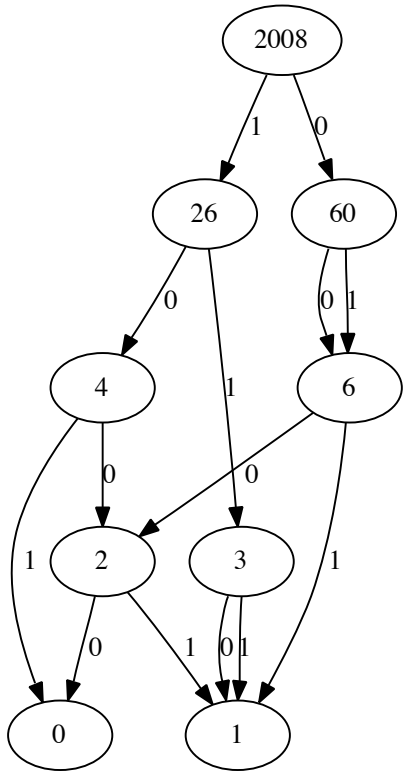


Figure: Graph obtained by recursive application of `bitunpair` for 2008

Encoding digraphs, DAGs, hypergraphs by combining set-encodings and pairing/unpairing functions

```
?- as(digraph,nat,2009,D),as(nat,digraph,D,N).
D = [0-0, 2-0, 0-2, 0-3, 3-0, 0-4, 1-2, 0-5],
N = 2009

?- as(dag,nat,2009,Dag),as(nat,dag,Dag,N).
Dag = [0-1, 2-3, 0-3, 0-4, 3-4, 0-5, 1-4, 0-6],
N = 2009

?- 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 .
```

Digraph encoding

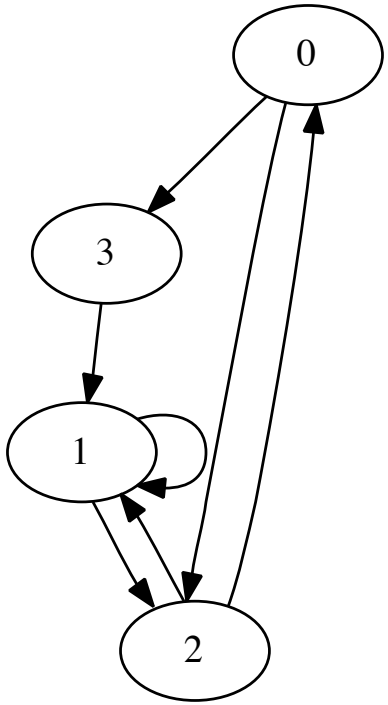


Figure: 2008 as a digraph

Beyond data structures: a simple Turing equivalent functional language

Proposition

$\forall z \in \mathbb{N} - \{0\}$ *the diophantic equation*

$$2^x(2y+1) = z \tag{1}$$

has exactly one solution $x, y \in \mathbb{N}$.

hd/2, tl/2, cons/3, null/1

```
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).
```

The predicates `cons/3, hd/2, tl/2, null/1` emulate CONS, CAR, CDR, NIL as defined in McCarthy's 1960 paper!

Applications: succinct representations

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

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

as an alternative to *algorithmic complexity* - are such succinct representations a measure of “structural” complexity?

Applications: encoding Prolog terms

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

```
?- 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) .
```

Applications: Random Generation

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

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

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

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

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

Conclusion

- an embedded combinator language that shapeshifts datatypes at will using a small groupoid of isomorphisms
- lifting isomorphisms to hereditarily finite datatypes
- a practical tool to experiment with various universal encoding mechanisms - but what are the *real* applications?
- $\longrightarrow$ *"You can observe a lot just by watching."* - Yogi Berra :-)

Literate Prolog program: `http://logic.cse.unt.edu/tarau/research/2009/pISO.zip`

Meta-conclusion: the MAGIC that *anywhere is possible*

Figure: “anywhere is possible” (PG13)

Our framework ensures that jumping from a “pyramid” to another is safe i.e. rated PG13 :-)

Open question: can we learn from the *magic*?

```
the magic: a bijective mapping  f:S→T, g:T→S
we can borrow an operation in T, ``!'' as ``?'' in S:
x `?` y = g  ((f x) `!` (f y))
```

- "the magic" allows us to teleport content and structure, but we need to be able to learn from it, i.e. *reverse engineer* it to the point where we can do it without using "the magic"
- example: assuming "!" has an *inductive definition* in T can we automatically derive a natural inductive definition for "?" in S that does not refer to `f` and `g` anymore?
- we can do it in some cases manually, and we learn very nice things in the process, but can we automate that?
- ongoing work - some very interesting results on the way...