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 \textit{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 \url{http://logic.cse.unt.edu/tarau/research/2009/pISO.zip}

\textbf{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 \textit{mutability} in mind and therefore with space saving strategies based on in-place updates. On the contrary, the dominance of \textit{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 guarantees 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 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.

\[
\begin{align*}
  f &= g^{-1} \\
  g &= f^{-1}
\end{align*}
\]

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:

\[
\begin{align*}
  \text{compose}(\text{iso}(F,G),\text{iso}(F_1,G_1)) &= \text{iso}(\text{fcompose}(F,F_1),\text{fcompose}(G,G_1)) \\
  \text{invert}(\text{iso}(F,G)) &= \text{iso}(\text{invert}(F),\text{invert}(G)) \\
  \text{from}(\text{iso}(F,G),X,Y) &= \text{call}(F,X,Y) \\
  \text{to}(\text{iso}(F,G),X,Y) &= \text{call}(G,X,Y)
\end{align*}
\]

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.

\[
\begin{align*}
  \text{fcompose}(G,F,X,Y) &= \text{call}(F,X,Z) \cdot \text{call}(G,Z,Y) \\
  \text{id}(X) &= X \\
  \text{from}(\text{iso}(F,\_),X,Y) &= \text{call}(F,X,Y) \\
  \text{to}(\text{iso}(\_\_),G,X,Y) &= \text{call}(G,X,Y)
\end{align*}
\]

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

\[
\text{<name>(iso(From,To)).}
\]

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

\[
\begin{align*}
  \text{borrow}(\text{IsoName},H,X,Y) &= \text{call}(\text{IsoName},\text{iso}(F,G)), \\
  \text{fcompose}(F,F\text{compose}(H,G)),X,Y).
\end{align*}
\]

\[
\begin{align*}
  \text{lend}(\text{IsoName},H,X,Y) &= \text{call}(\text{IsoName},\text{iso}(F,G)), \\
  \text{invert}(\text{Iso},\text{iso}(F,G)), \\
  \text{fcompose}(F,F\text{compose}(H,G)),X,Y).
\end{align*}
\]

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.

\[
\begin{align*}
  \text{with}(\text{Isol},\text{Isol2},\text{Isol2}) &= \text{invert}(\text{Isol2},\text{Inv2}),\text{compose}(\text{Isol1},\text{Inv2}2,\text{Isol}).
\end{align*}
\]

\[
\begin{align*}
  \text{as}(\text{That},\text{This},X,Y) &= \% \text{ gets the actual encoders by calling <Name>(Iso) \\
  \text{call}(\text{That},\text{ThatIso}), \text{call}(\text{This},\text{ThisIso}), \% \text{ routes through our hub \\
  \text{with}(\text{ThatIso},\text{ThisIso},\text{iso}), \% \text{ activates the isomorphism \\
  \text{to}(\text{iso},X,Y)).
\end{align*}
\]

The combinator as adds 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:
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:

\[
\begin{align*}
\text{cons}(X, Y, XY) & : \text{X} = 0, Y = 0, XY \text{ is } (2 \times X) + (2 \times Y + 1). \\
\text{hd}(XY, X) & : \text{XY} = 0, P \text{ is } XY \mod 2, \text{hd}(P, XY, X). \\
\text{hd1}(1, \ldots, 0). \\
\text{tl}(XY, X) & : Z = XY / 2, \text{hd}(Z, H), X = H + 1. \\
\text{null}(0).
\end{align*}
\]

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 diophantine equation

\[
2^z(2y + 1) = z
\]

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.

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

\[
\begin{align*}
\text{fun2nat}([X], N) & : \text{fun2nat}(X, N1), \text{cons}(X, N1, N). \\
\text{nat2fun}(0, [\ldots, 0]. \\
\text{nat2fun}(N, [X | Xs]) & : N = 0, \text{hd}(N, X), \text{tl}(N, T), \text{nat2fun}(T, Xs).
\end{align*}
\]

working as follows:

\[
\begin{align*}
? \leftarrow \text{fun2nat}([0, 2, 0, 1, 0, 0, 0]). \\
N & = 2009.
\end{align*}
\]

\[
\begin{align*}
\text{nat} & (\text{iso}(\text{nat2fun}, \text{fun2nat})). \\
\text{The resulting encoder } & \text{(nat/1) can now interoperate with the encoder fun:} \\
? \leftarrow \text{as(fun, nat, 42, F).} \\
F & = [1, 1, 1] \\
? \leftarrow \text{as(fun, nat, 2008, F).} \\
F & = [3, 0, 1, 0, 0, 0, 0] \\
? \leftarrow \text{lend(nat, reverse, 2008, R).} \\
R & = 1135 \% \text{ different, sequence depends on order}
\end{align*}
\]

3.2 An isomorphism to finite sets of natural numbers

The isomorphism is specified with two bijections set2fun and fun2set.

\[
\begin{align*}
\text{set} & (\text{iso}(\text{set2fun}, \text{fun2set})). \\
\text{It maps arbitrary lists of natural numbers representing finite functions to strictly increasing sequences of natural numbers representing sets. While finite sets and sequences share a common representation } [\text{Nat}], \text{ subjects are set to the implicit constraint that all their elements are distinct. This suggest that a set like } \{7, 1, 4, 3\} \text{ could be represented by first ordering it as } \{1, 3, 4, 7\} \text{ and then compute the differences between consecutive elements. This gives } [1, 2, 1, 3] \text{ with the first element followed by the increments } [2, 1, 3]. \text{ To turn it into a bijection, including } 0 \text{ as a possible member of a sequence, another adjustment is needed: elements in the sequence of increments should be replaced by their predecessors. This gives } [1, 1, 0, 2] \text{ as implemented by set2fun/2:} \\
\text{set2fun}(Xs, Ns) & : \text{sort}(Xs, Is), \text{set2fun}(Is, -1, Ns).
\end{align*}
\]

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

\[
\begin{align*}
? \leftarrow \text{as(set, fun, } [0, 1, 0, 0, 4], S). \\
S & = [0, 2, 3, 4, 9] \\
? \leftarrow \text{as(set, fun, } [0, 2, 3, 4, 9], P). \\
F & = [0, 1, 0, 0, 4] \\
? \leftarrow \text{as(set, nat, 2009, Set).} \\
\text{Set} & = [0, 3, 4, 6, 7, 8, 9, 10] \\
? \leftarrow \text{as(nat, set, } [0, 3, 4, 6, 7, 8, 9, 10], N). \\
N & = 2009.
\end{align*}
\]

\[\text{2 Such constraints can be regarded as laws/assertions that we assume holding for a given data type, when needed, restricting it to the appropriate domain of the underlying mathematical concept. They assume the existence of an injective embedding from a client representation (sets of natural numbers in this case) to a host representation (finite sequences of natural numbers in this case). This assumption has been in use implicitly in various data representations and algorithms, but it is important to keep in mind that it is always based on the existence of such an injective embedding.}\]
Note that as/4 works exactly as if isomorphisms were defined directly:

\[
\text{nat_set}(\text{iso(nat2set, set2nat)}).
\]
\[
\text{nat2set}(N, Set) :\text{-} \text{nat2fun}(N, F), \text{fun2set}(F, Set).
\]
\[
\text{set2nat}(Set, N) :\text{-} \text{set2fun}(Set, F), \text{fun2nat}(F, N).
\]

\[
S = [0, 3, 4, 6, 7, 8, 9, 10].
\]

?- 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 hylomorphisms are parameterized by two transformations \( F \) and \( G \) forming an isomorphism \( \text{iso}(F, G) \):

\[
\text{unrank}(F, N, R) :\text{-} \text{call}(F, N, Y), \text{unranks}(F, Y, R).
\]

\[
\text{unranks}(F, Ns, Rs) :\text{-} \text{maplist}(\text{unrank}(F), Ns, Rs).
\]

\[
\text{rank}(G, Ts, Rs) :\text{-} \text{ranks}(G, Ts, Xs), \text{call}(G, Xs, Rs).
\]

\[
\text{ranks}(G, Ts, Rs) :\text{-} \text{maplist}(\text{rank}(G), Ts, Rs).
\]

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:

\[
\text{tsize1}(Xs, N) :\text{-} \text{sumlist}(Xs, S), N \text{ is } S+1.
\]

\[
\text{tsize}(T, N) :\text{-} \text{rank}(\text{tsize1}, T, N).
\]

Note also that \( \text{unrank} \) and \( \text{rank} \) work on trees in cooperation with \( \text{unranks} \) and \( \text{ranks} \) working on lists of trees.

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

\[
\text{hylo}(\text{isolate}, \text{iso}((\text{rank}(G), \text{unrank}(F)))) :\text{-} \text{call}(\text{isolate}, \text{iso}(F, G)).
\]

\[
\text{hylos}(\text{isolate}, \text{iso}((\text{ranks}(G), \text{unranks}(F)))) :\text{-} \text{call}(\text{isolate}, \text{iso}(F, G)).
\]

4.1.1 A hylomorphism encoding hereditarily finite sets

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

\[
\text{hfs}(\text{Iso}) :\text{-} \text{hylo}((\text{nat_set}, \text{Hylo}), \text{nat}(\text{Nat})), \text{compose}(\text{Hylo}, \text{Nat}, \text{Iso}).
\]

The \( \text{hfs} \) encoder can now borrow operations from sets or natural numbers as follows:

\[
\text{hfs}\_\text{succ}(H, R) :\text{-} \text{borrow}((\text{nat}\_\text{hfs}, \text{succ}, H, R)).
\]

\[
\text{nat}\_\text{hfs}(\text{Iso}) :\text{-} \text{nat}(\text{Nat}), \text{hfs}(\text{HFS}), \text{with}(\text{Nat}, \text{HFS}, \text{Iso}).
\]

?- \text{hfs}\_\text{succ}([], R).
\[
R = [[]];
\]

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

?- \text{as}(\text{hfs}, \text{nat}, 42, 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:

\[
f(x) = \text{if } x = \{\} \text{ then } 0 \text{ else } \sum_{a \in x} 2^{f(a)}
\]

together with its inverse:

\[
\text{ackermann}(H, N) :\text{-} \text{as}(\text{nat}, \text{hfs}, H, N).
\]

\[
\text{inverse}\_\text{ackermann}(N, H) :\text{-} \text{as}(\text{hfs}, \text{nat}, N, H).
\]

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 \text{inverse}\_\text{ackermann} function as shown in Fig. 1.

\[
\begin{align*}
\text{Figure 1.} & \quad 2008 \text{ as a HFS}
\end{align*}
\]

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:

\[
\text{hff}(\text{Iso}) :\text{-} \text{hylo}((\text{nat}, \text{Hylo}), \text{nat}(\text{Nat})), \text{compose}(\text{Hylo}, \text{Nat}, \text{Iso}).
\]

The \( \text{hff} \) 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:

?- \text{as}(\text{hff}, \text{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](image)

### 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 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:

\[
\begin{align*}
\text{pars2hff}(Xs,T) & \rightarrow \text{pars2term}(0,1,T,Xs,[]) \\
\text{pars2term}(L,R,Xs) & \rightarrow [L],\text{pars2args}(L,R,Xs) \\
\text{pars2args}(\_\_,R,[]) & \rightarrow [R] \\
\text{pars2args}(L,R,[X,Xs]) & \rightarrow \text{pars2term}(L,R,X), \\
& \quad \text{pars2args}(L,R,Xs)
\end{align*}
\]

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

\[
\begin{align*}
?\text{- pars.hff}([0,0,1,0,1,1],T),\text{pars.hff}(Ps,T). \\
T & = [[],[]], \\
Ps & = [0,0,1,0,1,1]
\end{align*}
\]

We obtain the encoder:

\[
\begin{align*}
\text{pars}(\text{Isos}) & \rightarrow \text{hff}(\text{HFF}),\text{compose}(\text{iso}(\text{pars2hff},\text{hff2pars}),\text{HFF},\text{Iso}). \\
\text{hff2pars}(\text{HFF},Ps) & \rightarrow \text{pars2hff}(Ps,\text{HFF}).
\end{align*}
\]

working as follows:

\[
\begin{align*}
?\text{- as}(\text{pars},\text{nat},42,Ps),\text{as}(\text{nat},\text{pars},Ps,N). \\
Ps & = [0,0,0,1,0,0,1,1,1] \\
N & = 42
\end{align*}
\]

### 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 \(fr\).

\[
\begin{align*}
?\text{- fr}(42,R),\text{rf}(R,N). \\
R & = [0,0,0,3,1] \\
N & = 42
\end{align*}
\]

The Prolog predicate \(fr\) provides higher information density than \(hff\).

The predicate \(fl\) which recurses and divides with increasing values of \(N\) while collecting digits with \(mod\) values of the factorial of \(N\) while \(lfr\) is the left-to-right variant.

\[
\begin{align*}
?\text{fr}(42,R),\text{lf}(R,N). \\
R & = [0,0,1,0,1,1] \\
N & = 42
\end{align*}
\]

Finally, \(rf\), the inverse of \(fr\) is obtained by reversing \(fl\).

\[
\begin{align*}
?\text{rf}(N,Ds)\rightarrow\text{lf}(R,Ds).
\end{align*}
\]

The predicate \(lfr\) which divides with increasing values of \(N\) while \(rfr\) is the right-to-left variant.

#### 5.2 Ranking and unranking permutations of given size with Lehmer codes and factoradics

The Lehmer code of a permutation determines the permutation uniquely.

The Lehmer code of a permutation \(P\) of size \(n\) is defined as the sequence \(l(f) = (l_1(f), \ldots, l_n(f))\) 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 \(n\). 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 \(lfr\) from factoradics to decimals.

Note that the Lehmer code \(Ls\) is used as the list of digits in the factoradic representation.
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 \texttt{select\_and\_remember}!

\% associates Lehmer code to a permutation
\% \texttt{perm\_lehmer([L1],[L2],[L3]):=}
\% \texttt{select\_and\_remember(L1,[L1],[L2],[L3]),}
\% \texttt{perm\_lehmer([L1],[L2],[L3]),}
\% \texttt{perm\_lehmer([L1],[L2],[L3])}.

\% remembers selections - for Lehmer code
\% \texttt{select\_and\_remember(L1,L2,L3,K):=
% \texttt{K \text{ is } Size-L,Last is Size-1,
% \texttt{ints\_from(0,Last,Is),ndup(K,0,Zs),
% \texttt{append(Zs,Ls,LehmerCode),
% \texttt{perm\_lehmer(Is,Ps,Ls),}}}
\% \texttt{Xs}}\%Xs,Ps,Ls,Ks}\%

The predicate \texttt{nat\_2perm} provides the matching \textit{unranking} operation associating a permutation \texttt{Ps} to a given \texttt{Size}>0 and a natural number \texttt{N}.

\texttt{nth2perm(Size,N,Ps):=}
\% \texttt{sf(S):-nat\_stream(N),sf(N,S).}
\% \texttt{sf(0,0).}
\% \texttt{sf(N,R1):-N>0,N1 is N-1,
% \texttt{ndup(N1,1,De),
% \texttt{rf([0|D|],R),
% \texttt{R1 is R+1}.}
\texttt{sf(S):-nat\_stream(N),sf(N,S).}
\texttt{nat\_stream(0).}
\texttt{nat\_stream(N):=nat\_stream(N1),N is Ni+1.}
\texttt{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.}
\texttt{to\_sf(N, K,N\_M):-}
\texttt{nat\_stream(X),sf(X,S),S>N,}
\texttt{sf(S):-}
\texttt{N is S-N\_M.}
\% \texttt{Unranking of an arbitrary permutation is now easy - the index K}
\% \texttt{determines the size of the permutation and N_M determines the rank.}
\% \texttt{Together they select the right permutation with nth2perm.}
\texttt{nat2perm(0,[]).}
\texttt{nat2perm(N,Ps):-to\_sf(N, K,N\_M),nth2perm(K,N\_M,Ps).}

\texttt{Ranking of a permutation is even easier; we first compute its Size}
\texttt{and its rank \texttt{Nth}, then we shift the rank by the sum of all factorials}
\texttt{up to Size, enumerating the ranks previously assigned.}
\texttt{perm2nat([].0).}
\texttt{perm2nat(Ps,N) :- perm2nth(Ps,Size,Nth),
% \texttt{sf(Size,S),
% \texttt{N is S-Nth.}
\texttt{7- nth2perm(5,42,Ps),perm2nth(Ps,Length,Nth).
% Ps = [1, 4, 2, 0, 3],
% Length = 5,
% Nth = 42}
\texttt{7- 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 \textit{Nat}

One more step is needed to extend the mapping between permutations of a given length to a bijective mapping from/to \textit{Nat}: we will have to “shift towards infinity” the starting point of each new block of permutations in \textit{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 \texttt{N}!
\% \texttt{fast\_computation\_of\_the\_sum\_of\_all\_factorials\_up\_to\_N!}
\texttt{sf(0,0).}
\texttt{sf(N,R1):-N>0,N1 is N-1,
% \texttt{ndup(N1,1,De),
% \texttt{rf([0|D|],R),
% \texttt{R1 is R+1.}
\texttt{This is done by noticing that the factoradic representation of}
\texttt{[0,1,1,...] does just that. The stream of all such sums can now be}
\texttt{generated as usual:}
\texttt{sf(S):-nat\_stream(N),sf(N,S).}
\texttt{nat\_stream(0).}
\texttt{nat\_stream(N):=nat\_stream(N1),N is Ni+1.}

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
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 \( f : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \). An unpairing function is a bijection \( g : \text{Nat} \rightarrow \text{Nat} \times \text{Nat} \).

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

\[
J(K(z), L(z)) = z
\]

\[
K(J(x, y)) = x
\]

\[
L(J(x, y)) = y
\]

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

\[
cantor_pair(K1, K2, P) \leftarrow P \text{ is } (((K1+K2) \times (K1+K2+1))/2)+K2.
\]

bijections from \( \text{Nat} \times \text{Nat} \) to \( \text{Nat} \) 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:

\[
?- \text{findall}(R, (\text{between}(0,3,A), \text{between}(0,3,B), \text{cantor_pair}(A,B,R)), Rs).
\]

\[
Rs = [0, 2, 4, 6, 1, 5, 9, 13, 3, 11, 19, 27, 7, 23, 39, 55]
\]

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.

\[
cantor_unpair(Z, K1, K2) \leftarrow
1 \text{ is floor}((\sqrt{8 \times Z + 1})-1)/2),
K1 \text{ is } ((1+3-1)/2)-Z,
K2 \text{ is } Z-(((I+1))/2).
\]

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 \( \text{Nat} \times \text{Nat} \) to \( \text{Nat} \) 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.

\[
\text{bitpair}(X,Y,P) \leftarrow
to_rbits(X,Xs),
to_rbits(Y,ys),
\text{bitmix}(Xs,ys,Ps),!,
from_rbits(Ps,P).
\]
bitunpair(P,X,Y):-
    to_rbits(P,Ps),
    bitmix(Xs,Ys,Ps),!,
    from_rbits(Xs,X),
    from_rbits(Ys,Y).

bitmix([],[],[]).

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

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

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

X = 60,
Y = 26,
Z = 2008

? 2008:[0, 0, 1, 0, 1, 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)

? 0: (0, 0) 2: (0, 1) 3: (1, 1) 4: (2, 1) 5: (3, 1)
6: (0, 2) 7: (1, 2) 8: (0, 3) 9: (1, 3)
10: (2, 2) 11: (3, 2) 12: (0, 4) 13: (1, 4)
14: (2, 4) 15: (3, 4)

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).

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).

We can derive the following encoder:

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

working as follows:

?- as(nat2,nat,0,Pair).
Pair = 0-0 .
?- as(nat2,nat,2008,Pair).
Pair = 60-26
?- as(nat,nat2,60-26,N).
N = 2008.

The following figures visualize properties of our pairing/unpairing functions. Given that unpairing functions are bijections from Nat to Nat × 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 7. 2D curve connecting values of bitunpair n for n ∈ [0.2^{10} – 1]

By representing objects in Nat × 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(N,X,Y)):-Z is XY+1,hd(Z,X),tl(Z,Y).

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].
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:

\[
\text{digraph2set}(P,N) \Leftarrow \text{maplist}(\text{consPair}, P, N).
\]

\[
\text{set2digraph}(N,P) \Leftarrow \text{maplist}(\text{consUnPair}, N, P).
\]

The resulting encoder is:

\[
\text{digraph}(\text{Iso}) \Leftarrow \text{set}(S),
\text{compose}(\text{iso}(\text{digraph2set}, \text{set2digraph}), S, \text{Iso}).
\]

working as follows:

\[
?\text{- as(digraph,nat,2008,D),as(nat,digraph,D,N)}.
\]

\[
D = [p(2, 0), p(0, 2), p(0, 3), p(3, 0), p(0, 4), p(1, 2), p(0, 5)],
N = 2008.
\]

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

\[
\text{bdigraph2set}(P,N) \Leftarrow \text{maplist}(\text{bitpair}, P, N).
\]

\[
\text{bset2digraph}(N,P) \Leftarrow \text{maplist}(\text{bitunpair}, N, P).
\]

The resulting encoder

\[
\text{bdigraph}(\text{Iso}) \Leftarrow \text{set}(S),
\text{compose}(\text{iso}(\text{bdigraph2set}, \text{bset2digraph}), S, \text{Iso}).
\]

works as follows:

\[
?\text{- as(bdigraph,nat,2008,D),as(nat,bdigraph,D,N)}.
\]

\[
D = [1-1, 2-0, 2-1, 3-1, 0-2, 1-2, 0-3],
N = 2008.
\]

Fig. 9 shows the digraph associated to 2008.

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:

\[
\text{set2hypergraph}(S,G) \Leftarrow \text{maplist}(\text{nat2nonempty}, S, G).
\]

\[
\text{hypergraph2set}(G,S) \Leftarrow \text{maplist}(\text{nonempty2nat}, G, S).
\]

\[
\text{nat2nonempty}(N,S) \Leftarrow N_1 + 1, \text{nat2set}(N_1, S).
\]

\[
\text{nonempty2nat}(S,N) \Leftarrow \text{set2nat}(S), N\text{ is } N_1 - 1.
\]

The resulting encoder is:

\[
\text{hypergraph}(\text{Iso}) \Leftarrow \text{set}(S),
\text{compose}(\text{iso}(\text{hypergraph2set}, \text{set2hypergraph}), S, \text{Iso}).
\]

working as follows:

\[
?\text{- as(hypergraph,nat,2009,G),as(nat,hypergraph,G,N)}.
\]

\[
G = [[0], [0, 2], [0, 1, 2], [3], [0, 3], [1, 3], [0, 1, 3]],
N = 2009.
\]

9. Encoding Programming Language Constructs

We have seen that \( \text{hd}, \text{tl}, \text{cons} \) 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 \( \text{hd}, \text{tl}, \text{cons} \). A “list” concatenation operation \( \text{app} \) is defined as follows:

\[
\text{app}(0, Ys, Ys).
\]

\[
\text{app}(\text{XXs}, Ys, Zs) \Leftarrow \text{XXs} > 0,
\text{hd}(\text{XXs}, X), \text{tl}(\text{XXs}, Xs),
\text{app}(Xs, Ys, Zs),
\text{cons}(X, Xs, Zs).
\]

Note that \( \text{app}/3 \) works on natural numbers “seen” as lists, for instance:

\[
?\text{- app(2008,2009,R)}.
\]

\[
R = 4116440.
\]

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

\[
?\text{- as(fun,nat,2008,Ns)}.
\]

\[
Ns = [3, 0, 1, 0, 0, 0, 0].
\]
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 $nat$:

$$nth(Thing, N, X) := as(Thing, nat, N, X).$$

$$stream_of(Thing, X) := nat_stream(N), nth(Thing, N, X).$$

10.2 Random Generation

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

$$random_gen(Thing, Max, Len, X) :=$$

$$random_fun(Max, Len, Ns), as(Thing, fun, Ns, X).$$

$$random_gen(Thing, Max, Len, X) :=$$

$$random_fun(Max, Len, Ns), maplist(random_gen(Thing, Max), Ns).$$

$$random_gen(Max, N) := - random(X), N is integer(Max+1).$$

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:

$$as(hff, hfs, [], [], [[], []], [[], []]), hff).$$

$$N = 42$$

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:

$$term2bitpars(T, Ps, As) := term2bitpars(T, Ps, [], As, []).$$

$$term2bitpars(T, Ps, Ps) ::= \{\text{var}(T), [T].$$

$$term2bitpars(T, Ps, Ps) ::= \{\text{atomic}(T), [T].$$
The encoding is reversible, i.e. the term \( T \) can be recovered:

\[
\text{bitpars2term}(X, Ps, Ps) \rightarrow \text{bitpars2term}(Ps, As, T)
\]

The two transformations work as follows:

?\(- term2bitpars(f(g(a,X),X,42),Ps,As),\text{bitpars2term}(Ps,As,T).
\]

\[\text{Ps} = [0,0,1,0,0,1,0,1,0,1,1,1,0,1,0,1,1], \text{As} = [f,g,a,x,x,42], T = f(g(a,x),x,42).\]

After aggregating bitlists into natural numbers we obtain:

\[
\text{term2code}(T,N,As) : = \text{term2bitpars}(T,Ps,As), \text{from}_\text{base}(2,N).
\]

\[
\text{code2term}(N,As,T) : = \text{to}_\text{base}(2,N), \text{bitpars2term}(Ps,As,T).
\]

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 \text{nat2fun} and then aggregated with the result of \text{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 HERs, HFIs 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 [Goedel 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 arithmetic [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


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
\( \text{to_rbits}(0, []). \)

\( \text{to_rbits}(N, [B | Rs]) \rightarrow B \equiv N \mod 2, N_1 \equiv N/2, \)  
\( \text{to_rbits}(N_1, Bs). \)

% converts a list of bits (least significant first)
% into an integer
\( \text{from_rbits}([], N). \)

\( \text{from_rbits}([X | Xs], E, N1, N3) \rightarrow E \equiv X \equiv E + N1, \)  
\( \text{from_rbits}(Xs, N3). \)

% conversion to list of digits in given base
\( \text{to_base}(N, Base, 0, Bs). \)

\( \text{to_base}(N, R, Bs) \rightarrow N \equiv R, Base = [N]. \)

\( \text{to_base}(N, R, [B | Bs]) \rightarrow \)  
\( B \equiv N \mod R, N_1 \equiv N/R, K_1 \equiv K_1 + 1, \)  
\( \text{to_base}(N_1, K_1, Bs). \)

% conversion from list of digits in given base
\( \text{from_base}([_ | 0], 0). \)

\( \text{from_base}([X | Xs], N) \rightarrow \text{from_base}(Base, Xs, R), N \equiv X + Base. \)

% generates integers From..To
\( \text{ints_from}([From, To], Is). \)

\( \text{findall}(I, \text{between}(From, To, I), Is). \)

% replicates X, N times
\( \text{ndup}(0, [_]). \)

\( \text{ndup}(N, [X | Xs]) \rightarrow N \equiv 0, N_1 \equiv N-1, \text{ndup}(N_1, X, Xs). \)