Deriving a Fast Inverse of the Generalized Cantor N-tupling Bijection

Paul Tarau

Dept. of Computer Science and Engineering

University of North Texas, Denton, Texas, USA

E-mail: tarau@cs.unt.edu

submitted TBD; revised TBD; accepted TBD

Abstract

We attack an interesting open problem (an efficient algorithm to invert the generalized Cantor N-tupling bijection) and solve it through a sequence of equivalence preserving transformations of logic programs, that take advantage of unique strengths of this programming paradigm. An extension to set and multiset tuple encodings, as well as a simple application to a "fair-search" mechanism illustrate practical uses of our algorithms.

The code in the paper (a literate Prolog program, tested with SWI-Prolog and Lean Prolog) is available at `http://logic.cse.unt.edu/tarau/research/2012/pcantor.pl` .

KEYWORDS: generalized Cantor n-tupling bijection, bijective data type transformations, combinatorial number system, solving combinatorial problems in Prolog, optimization through program transformation, logic programming and software engineering

1 Introduction

It is by no means a secret that logic programming is an ideal paradigm for solving combinatorial problems. Built-in backtracking, unification and availability of constraint solvers facilitates quick prototyping for problems involving search or generation of combinatorial objects. It also provides an easy path from executable specification to optimal implementation through a well-understood set of program transformations. From a software engineering perspective, problem solving with help of logic programming tools is a natural fit to *agile development* practices as it encourages a fast moving iterative process consisting of incremental refinements.

This paper reports on tackling a somewhat atypical problem solving instance: finding a fast inverse of a generalization of Cantor's pairing bijection to n-tuples. This generalization is mentioned in two relatively recent papers (Cegielski and Richard 1999; Lisi 2007) with a possible attribution in (Cegielski and Richard 1999) to Skolem as a first reference.

The formula, given in (Cegielski and Richard 1999) p.4, looks as follows:

$$K_n(x_1,\ldots,x_n) = \binom{n-1+x_1+\ldots+x_n}{n} + \cdots + \binom{k-1+x_1+\ldots+x_k}{k} + \ldots + \binom{1+x_1+x_2}{2} + \binom{x_1}{1}$$

where $\binom{n}{k}$ represents the number of subsets of k elements of a set of n elements and $K_n(x_1,\ldots,x_n)$ denotes the natural number associated to the tuple $x_1,\ldots,x_n$. So the problem of inverting it means finding a solution of the *Diophantine equation*

$$\binom{x_1}{1} + \binom{1+x_1+x_2}{2} + \ldots + \binom{n-1+x_1+\ldots+x_n}{n} = z \tag{1}$$

and proving that it is unique. Unfortunately, despite extensive literature search, we have not found any attempt to devise an algorithm that computes the inverse of the function K_n, so we had to accept the fact that we were looking at an *open problem* with possibly interesting implications, given that for $n = 2$ it reduces to Cantor's pairing function that has been used in hundreds of papers on foundations of mathematics, logic, recursion theory as well as in some practical applications (dynamic n-dimensional arrays) like (Rosenberg 2003).

As an inductive proof that K_n is a bijection is given in (Lisi 2007) (Theorem 2.1), we know that a solution exists and is unique, so the problem reduces to computing the first solution of the Diophantine equation (1).

Unfortunately, solving an arbitrary Diophantine equation is Turing-equivalent. This is a consequence of the negative answer to Hilbert's 10-th problem, proven by Matiyasevich (Matiyasevich 1993), based on earlier work by Robinson, Davis and Putnam (Robinson 1969; Davis et al. 1961). And some fairly simple instances of it, like Fermat's $\exists x,y,z > 0, \exists n \geq 3, x^n + y^n = z^n$ have waited for centuries before being solved.

On the other hand, things do not look that bad in this case, as it is easy to show that in the equation (1), $\forall i, x_i \leq z$ holds. Therefore, an enumeration of all tuples $x_1,\ldots,x_n$ for $0 \leq x_i \leq z$ provides an obvious but dramatically inefficient solution.

So the our open problem reduces to *finding an efficient, linear or low polynomial algorithm for computing the inverse.*

And this paper provides a surprisingly simple solution to it in section 7, after telling the story of our incremental refinements (as well as backtracking steps) leading to it. Section 2 overviews the well-known solution for $n = 2$. Section 3 provides the Prolog implementation of the mapping from n-tuples to natural numbers. Section 4 describes the successive refinements of the inverse function, from its specification to a moderately useful implementation. Section 5 introduces a *list-to-set bijection* that will turn out to be helpful in "connecting the dots" to a well-known combinatorial problem that leads to our solution described in section 7 (after a small "backtracking step" shown in section 6). Section 8 compares the performance our intermediate refinements and our final result. Section 9 extends the bijection to sets and multisets. Section 10 shows a simple application implementing a "fair search" mechanism, section 11 discusses related work and section 12 concludes the paper.

2 The Classic Result: Cantor's Pairing Function and its Inverse

Cantor's pairing function is a polynomial of degree 2, obtained from the generalized one for $n = 2$, given by the formula $f(x_1, x_2) = x_1 + \frac{(x_1+x_2+1)(x_1+x_2)}{2}$.
The following Prolog code implements it:

```
cantor_pair(X1,X2,P) :- P is X1 + (((X1+X2+1) * (X1+X2)) // 2).
```

Note that by composing it n times, one can obtain an n-tupling function, but unfortunately the resulting polynomial is of degree 2^n, in contrast to the generalized n-tupling bijection which is a polynomial of degree n. On the other, hand, as the following Prolog code shows, the problem of finding its inverse efficiently is relatively easy. Basically, the inverse of Cantor's pairing function is obtained by solving a second degree equation while keeping in mind that solutions should be natural numbers (Wikipedia 2011b).

```
cantor_unpair(P,K1,K2) :-
  E is 8*P+1,
  intSqrt(E,R),
  I is (R-1)//2,
  K1 is P-((I*(I+1))//2),
  K2 is ((I*(3+I))//2)-P.
```

We face a small bump here – Prolog's ordinary square root returning a fixed size float or double does not make sense when working with arbitrary size integers, so we need to implement an "integer square root" returning the natural number that provides the largest perfect square $\leq N$. Fortunately, we can ensure fast convergence using Newton's method:

```
intSqrt(0,0).
intSqrt(N,R) :- N>0,
  iterate(N,N,K),
  K2 is K*K,
  (K2>N → R is K-1 ; R=K).

iterate(N,X,NewR) :-
  R is (X+(N//X))//2,
  A is abs(R-X),
  (A<2 → NewR=R ; iterate(N,R,NewR)).
```

As the following example shows, computations with larger than 64-bit operands are handled, provided that the underlying Prolog system supports arbitrary length integers.

```
?- cantor_pair(1234567890,9876543210,P),cantor_unpair(P,A,B).
P = 61728394953703760,
A = 1234567890,
B = 9876543210.
```

3 Implementing the Generalized Cantor n-tupling Bijection

Tupling/untupling functions are a natural generalization of pairing/unpairing operations. They are called *ranking/unranking* functions by combinatorialists as they map bijectively various combinatorial objects to $\mathbb{N}$ (ranking) and back (unranking).

The natural generalization of Cantor's pairing bijection described in (Cegielski and Richard 1999) is introduced using geometric considerations that make it obvious that it defines a bijection $K_n : \mathbb{N}^n \to \mathbb{N}$. More precisely, they observe that the enumeration in $\mathbb{N}^2$ of integer coordinate pairs laying on the anti-diagonals $x_1+x_2 = c$ can be lifted to points with integer coordinates laying on hyperplanes of the form $x_1 + x_2 + \ldots + x_k = c$. The same result, using a slightly different formula is proven algebraically, by induction in (Lisi 2007). We remind that the bijection K_n is defined by the formula

$$K_n(x_1, \ldots, x_n) = \sum_{k=1}^{n} \binom{k-1+x_1+\ldots+x_k}{k} \tag{2}$$

where $\binom{n}{k}$, also called "binomial coefficient" denotes the number of subsets of n with k elements as well as the coefficient of x^k in the expansion of the binomial $(x+y)^n$.

It is easy to see that the generalized Cantor n-tupling function defined by equation (2) is a polynomial of degree n in its arguments, and a conjecture, attributed in (Cegielski and Richard 1999) to Rudolf Fueter (1923), states that it is the only one, up to a permutation of the arguments. And, as mentioned in section 1, as we have found out through extensive literature search, while hoping for the contrary, it was also and *open problem* to find an efficient inverse for it.

Our first step is an efficient implementation of the function $K_n : \mathbb{N}^k \to \mathbb{N}$. By all means, this is the easy part, just summing up a set of binomial coefficients.

3.1 Binomial Coefficients, efficiently

Computing binomial coefficients efficiently is well-known (see (Wikipedia 2012a)).

$$\binom{k}{n} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\ldots(n-(k-1))}{k!} \tag{3}$$

However, we will need to make sure that we avoid unnecessary computations and reduce memory requirements by using a tail-recursive loop. After simplifying the slow formula in the first part of the equation (3) with the faster one based on falling factorial $n(n-1)\ldots(n-(k-1))$, and performing divisions as early as possible to avoid generating excessively large intermediate results, one can derive the `binomial_loop` tail-recursive predicate

```
binomial_loop(_,K,I,P,R) :- I>=K, !, R=P.
binomial_loop(N,K,I,P,R) :-
   I1 is I+1,
   P1 is ((N-I)*P) // I1,
   binomial_loop(N,K,I1,P1,R).
```

And, as a simple optimization, when $N-K \leq K$, the shorter computation of $\binom{N}{N-K}$ is used to reduce the number of steps in `binomial_loop`. The resulting predicate `binomial(N,K,R)` computes $\binom{N}{K}$ and unifies the result with `R`.

```
binomial(N,K,R) :- N<K, !, R=0.
binomial(N,K,R) :- K1 is N-K, K>K1, !,
  binomial_loop(N,K1,0,1,R).
binomial(N,K,R) :- binomial_loop(N,K,0,1,R).
```

3.2 The $\mathbb{N}^k \to \mathbb{N}$ bijection

We are ready to implement a first version of the $\mathbb{N}^k \to \mathbb{N}$ ranking function as a tail-recursive computation using the accumulator pairs `L1` $\to$ `L2`, that hold the states of the length of the list processed so far, and `S1` $\to$ `S2`, that hold the state of the prefix sum of $X_1, X_2, \ldots X_k$ computed so far.

```
from_cantor_tuple1(Xs,R) :- from_cantor_tuple1(Xs,0,0,0,R).

from_cantor_tuple1([],_L,_S,B,B).
from_cantor_tuple1([X|Xs],L1,S1,B1,Bn) :-
  L2 is L1+1,
  S2 is S1+X,
  N is S2+L1,
  binomial(N,L2,B),
  B2 is B1+B,
  from_cantor_tuple1(Xs,L2,S2,B2,Bn).
```

The following examples illustrate the fact that the values of the result are relatively small, independently of the length or the size of the values on the input list.

```
?- from_cantor_tuple([],N).
N = 0.
?- from_cantor_tuple1([2,0,1,2],N).
N = 85.
?- from_cantor_tuple([0,2012,999,0,10],N).
N = 2107259417045595.
?- from_cantor_tuple([9,8,7,6,5,4,3,2,1,0,0,1,2,3,4,5,6,7,8,9],N).
N = 3706225144988231392404.
```

4 Refining the Specification of the Inverse

We start with an executable specification of the inverse, seen as defining, for a given K, a bijection $g_K : \mathbb{N} \to \mathbb{N}^K$.

4.1 Enumerating, naively

The predicate `to_cantor_tuple1(K,N,Ns)` computes, for each `K` the function g_K associating to the natural number `N` a tuple represented as a list `Ns` of length `K`.

```
to_cantor_tuple1(K,N,Ns) :-
  numlist(0,N,Is),
  cartesian_power(K,Is,Ns),
  from_cantor_tuple1(Ns,N),
  !. % just an optimization - no other solutions exists
```

Note that the built-in `numlist(From, To, Is)` generates a list of integers in the interval `[From..To]`).

The predicate `to_cantor_tuple1` uses `cartesian_power(K,Is,Ns)` to enumerate candidates of length `K`, drawn from the initial segment `0..N` of $\mathbb{N}$.

```
cartesian_power(0,_,[]).
cartesian_power(K,Is,[X|Xs]) :- K>0,
  K1 is K-1,
  member(X,Is),
  cartesian_power(K1,Is,Xs).
```

As `cartesian_power` backtracks over this finite set of potential solutions, the predicate `from_cantor_tuple1(Ns,N)` is called until the first (and known to be unique) solution is found. Given the unicity of the solution, the CUT in the predicate `to_cantor_tuple1` is simply an optimization without an effect on the meaning of the program.

The following example illustrates the correctness of this executable specification.

```
?- to_cantor_tuple1(3,42,R), from_cantor_tuple(R,S).
R = [1, 2, 2],
S = 42.
```

Unfortunately, performance deteriorates quickly around `K` larger than `5` and `N` larger than `100` as the time complexity of this program is at least $O(N^K)$. However, given our reliance on Prolog's backtracking, the search uses at most $O(Klog(N))$ space when filtering through lists of length `K` containing numbers of at most the bitsize of `N`.

4.2 A better algorithm, using a tighter upper limit

The next step in deriving an efficient untupling function is a bit trickier. First we observe that, as `from_cantor_tuple(K,Ns,N)` runs through successive hyperplanes $X_1 + \ldots + X_k = M$, for each of them the sum maxes out when $X_1 = M$ and $X_k = 0$ for $1 \leq K \leq N$. We can compute directly this maximum value with the predicate `largest_binomial_sum` as follows:

```
largest_binomial_sum(K,M,R) :- largest_binomial_sum(K,M,0,R).

largest_binomial_sum(0,_,R,R).
largest_binomial_sum(K,M,R1,Rn) :- K>0, K1 is K-1,
  M1 is M+K1,
  binomial(M1,K,B),
  R2 is R1+B,
  largest_binomial_sum(K1,M,R2,Rn).
```

Note that the predicate `largest_binomial_sum(K,M,R)` computes the same `R` as `cantor_tuple([M,0,...,0],R)`, with `K-1` `0`'s on the list.

Next we compute the upper limit for possible values of the sum `M` of $[X_1, ..., X_k]$ such that the relation `to_cantor_tuple([X1,...,Xk],N)` holds, i.e. we find the hyperplane $X_1 + \ldots + X_k = M$ defining the Cantor `K`-tuple. This computation, is implemented by the predicate `find_hyper_plane(K,N,M)` which, when given the inputs `K` and `M`, finds the value of the sum `M` that defines the hyperplane containing our tuple.

```
find_hyper_plane(0,_,0).
find_hyper_plane(K,N,M) :- K>0,
  between(0,N,M),
  largest_binomial_sum(K,M,R),
  R>=N,
  !.
```

Note the use of the built-in `between(From,To,I)` that backtracks over integers in the interval `[From..To]`.

We are now ready to define a more efficient inverse of the `from_cantor_tuple1` bijection, called `to_cantor_tuple2`, as a search through the set of lists such that the relation `from_cantor_tuple1(Xs,N)` holds.

```
to_cantor_tuple2(K,N,Ns) :-
 find_hyper_plane(K,N,M),
 sum_bounded_cartesian_power(K,M,Xs),
 from_cantor_tuple1(Xs,N),
 !,
 Ns=Xs.
```

The search, restricted this time to integers in the interval `[0..M]` is implemented by the predicate `sum_bounded_cartesian_power`.

```
sum_bounded_cartesian_power(0,0,[]).
sum_bounded_cartesian_power(K,M,[X|Xs]) :-  K>0, M>=0,
  K1 is K-1,
  between(0,M,X),
  M1 is M-X,
  sum_bounded_cartesian_power(K1,M1,Xs).
```

Note that, after applying the upper limit `M` computed by `find_hyper_plane`, to ensure that only tuples summing up to `M` are explored, we are using a customized cartesian product computation, in the predicate `sum_bounded_cartesian_power` backtracking over lists `[X1...Xk]` that sum-up to `M`.

The following examples illustrate that we obtain a correct implementation of our specification:

```
?- to_cantor_tuple2(10,23456,R), from_cantor_tuple1(R,S).
R = [2, 1, 0, 1, 1, 0, 0, 1, 0, 2],
S = 23456.
```

The code so far works well for small values of `K` up to `10-15` and `N` up to `20000-30000`. For larger values of `K`, e.g. `K=10` this upper limit grows very slowly and it helps reducing the search space significantly:

```
?- findall(M,(between(0,31,N),P is 2^N,find_hyper_plane(10,P,M)),Ms).
Ms = [1,1,1,1,2,2,2,3,3,4,5,5,6,7,7,8,...,15,16,17,19,21,23,25,27,29,31,34].
```

However, as the query

```
?- findall(M,(between(0,31,N),P is 2^N,find_hyper_plane(2,P,M)),Ms).
Ms = [1,1,2,3,5,7,10,15,22,31,44,63,90,127,180,255,361,511,723,1023,
     1447,2047,2895,4095,5792,8191,11584,16383,23169,32767,46340,65535]
```

indicates, while `M` grows significantly slower than `P`, it still grows linearly with `N`.

The predicate `to_cantor_tuple2` is a good improvement over `to_cantor_tuple1`, but it is by no means the efficient algorithm we are seeking.

Clearly, a "paradigm shift" is needed at this point, as obvious optimizations only promise diminishing returns. The highest hope would be to find a deterministic predicate similar to the integer square root based inverse for the case $N = 2$, but this time the arbitrary degree N of our polynomial looks like an insurmountable obstacle.

5 The Missing Link: from Lists to Sets and Back

After rewriting the formula for the $\mathbb{N}^k \to \mathbb{N}$ bijection as:

$$K_n(x_1, \ldots, x_n) = \sum_{k=1}^{n} \binom{k-1+s_k}{k} \tag{4}$$

where $s_k = \sum_{i=1}^{k} x_i$, we recognize the *prefix sums* s_k incremented with values of k starting at 0.

And, as our key "Eureka step", we instantly recognize here the "set side" of the bijection between sequences of n natural numbers and sets of n natural numbers described in (Tarau 2009a)[1].

We can compute the bijection `list2set` together with its inverse `set2list` as

```
list2set(Ns,Xs) :- list2set(Ns,-1,Xs).

list2set([],_,[]).
list2set([N|Ns],Y,[X|Xs]) :- X is (N+Y)+1, list2set(Ns,X,Xs).

set2list(Xs,Ns) :- set2list(Xs,-1,Ns).

set2list([],_,[]).
set2list([X|Xs],Y,[N|Ns]) :- N is (X-Y)-1, set2list(Xs,X,Ns).
```

The following examples illustrate how it works:

```
?- list2set([2,0,1,2],Set).
Set = [2, 3, 5, 8].
```

[1] In (Tarau 2009a) a general framework for bijective data transformations provides such conversion algorithms between a large number of fundamental data types.

```
?- set2list([2, 3, 5, 8],List).
List = [2, 0, 1, 2].
```

As a side note, we have also found this bijection it in (Knuth 2005) and implicitly in (Cegielski and Richard 1999), with indications that it might even go back to the early days of the theory of recursive functions.

6 Backtracking one step: revisiting the $\mathbb{N}^k \to \mathbb{N}$ bijection

It is time to step back at this point, and factor out `list2set` from our tail-recursive "untupling" loop `from_cantor_tuple1`.

The predicate `from_cantor_tuple` implements the the $\mathbb{N}^k \to \mathbb{N}$ bijection in Prolog, using the iterative computation of the binomial $\binom{n}{k}$ as well as the sequence to set transformer `list2set`. In contrast to `from_cantor_tuple1`, `untupling_loop` does not need to add the increments $1, 2, ..L-1$ as this task has been factored out and processed by `list2set`.

```
from_cantor_tuple(Ns,N) :-
  list2set(Ns,Xs),
  untupling_loop(Xs,0,0,N).

untupling_loop([],_L,B,B).
untupling_loop([X|Xs],L1,B1,Bn) :-
  L2 is L1+1,
  binomial(X,L2,B),
  B2 is B1+B,
  untupling_loop(Xs,L2,B2,Bn).
```

This shifts the problem of computing its inverse from lists to sets, an apparently minor use of a *bijective data type transformation*, that will turn out to be the single most critical step toward our solution.

7 The Efficient Inverse

We have now split our problem in two simpler ones: inverting `untupling_loop` and then applying `set2list` to get back from sets to lists.

Our first attempt was to try out constraint solving as it can sometime reverse arithmetic operations. Moreover, global constraints like `all_different` can take advantage of the fact that we are dealing with sets. However, the code, despite of the fact that we have tried also to take advantage of the optimizations implemented by the predicate `to_cantor_tuple2` turned out to be orders of magnitude slower than `to_cantor_tuple2`, mostly because delaying computations brought unnecessary overhead without essentially changing the nondeterministic nature of the search.

The key "Eureka step" at this point is to observe that `untupling_loop` implements the sum of the combinations $\binom{X_1}{1} + \binom{X_2}{2} + \ldots + \binom{X_K}{K} = N$, which is nothing but the representation of N in the *combinatorial number system of degree* K, (Wikipedia 2011a), due to (Lehmer 1964). And, fortunately, efficient conversion

algorithms between the conventional and the combinatorial number system are well known, (Buckles and Lybanon 1977; Knuth 2005).

For instance, theorem **L** in (Knuth 2005) describes the precise position of a given sequence in the lexicographic order enumeration of all sequences of length k.

Theorem 1 (*Knuth*)
The combination $[c_k, \ldots c_2, c_1]$ is visited after exactly $\binom{c_k}{k} + \ldots + \binom{c_2}{2} + \binom{c_1}{1}$ other combinations have been visited.

We are ready to implement the Prolog predicate `tupling_loop(K,N,Ds)`, which, given the degree `K` indicating the number of "combination digits", finds and repeatedly subtracts the greatest binomial smaller than `N`.

```
tupling_loop(0,_,[]).
tupling_loop(K,N,[D|Ns]) :- K>0,
  NewK is K-1,
  I is K+N,
  between(NewK,I,M),
  binomial(M,K,B),
  B>N,
  !, % no more search is needed
  D is M-1, % the previous binomial gives the "digit" D
  binomial(D,K,BM),
  NewN is N-BM,
  tupling_loop(NewK,NewN,Ns).
```

The predicate `tupling_loop` implements a deterministic greedy search algorithm, by subtracting the combination containing the most significant "digit" `D` at each step from the variable `N`. At a given step, this results in the variable `NewN` that carries on the result in the tail-recursive loop. At the same time, the decreased value of `K`, used in the binomial is carried on as the variable `NewK`.

And the efficient inverse of Cantor's N-tupling is now simply:

```
to_cantor_tuple(K,N,Ns) :-
  tupling_loop(K,N,Xs),
  reverse(Xs,Rs),
  set2list(Rs,Ns).
```

Note that we reverse the intermediate result `Xs` to ensure that `set2list` receives it in increasing order - our canonical representation for sets. The following example illustrates that it works as expected, including on very large numbers:

```
?- to_cantor_tuple(12,34567890,Ns), from_cantor_tuple(Ns,N).
Ns = [1,0,0,2,2,0,2,1,6,0,0,3],
N = 34567890 .
?- to_cantor_tuple(1234,666677778888999900000031415,Ns),
   from_cantor_tuple(Ns,N).
Ns = [0, 0, 0, 0, 0, 0, 0, 0, 0...,0, 0, 1, 0],
N = 666677778888999900000031415 .
```

K	N	Computation	Inferences	CPU Time (in secs)
5	10	to_cantor_tuple1	461,549	0.175
5	10	to_cantor_tuple2	310	< 0.001
5	10	to_cantor_tuple	71	< 0.001
5	10	*from_cantor_tuple*	25	< 0.001
5	20	to_cantor_tuple1	12,600,515	4.821
5	20	to_cantor_tuple2	416	< 0.001
5	20	to_cantor_tuple	90	< 0.001
5	20	*from_cantor_tuple*	29	< 0.001
5	100000	to_cantor_tuple2	369,797	0.146
5	100000	to_cantor_tuple	394	< 0.001
5	100000	*from_cantor_tuple*	39	< 0.001
50	100000	to_cantor_tuple2	14,688,168	5.442
50	100000	to_cantor_tuple	840	< 0.001
50	100000	*from_cantor_tuple*	242	< 0.001
100	100000	to_cantor_tuple2	17,659,214	6.302
100	100000	to_cantor_tuple	1,528	0.001
100	100000	*from_cantor_tuple*	457	< 0.001
1000	10000000	to_cantor_tuple	10,568	0.004
1000	10000000	*from_cantor_tuple*	3,585	0.001
2000	10000000	to_cantor_tuple	20,115	0.008
2000	10000000	*from_cantor_tuple*	7,027	0.002

Fig. 1: Comparison of the computational effort

8 Evaluating Performance

We have compared in Fig. 1 our 3 refinements of the predicate `to_cantor_tuple`, all computing the inverse of the Cantor n-tupling bijection. For reference, we have added also the direct computation `from_cantor_tuple`, applied on the results of the inverse computation. Fig. 1 shows the results, obtained with SWI-Prolog 6.0.0 running on Mac Pro with two QuadCore 2.26GHz Intel Xeon CPUs.

We have used SWI-Prolog's `time/1` built-in, which also provides an estimate of the number of inferences used in a computation. This machine independent parameter is likely to ignore the extra effort hidden under the GMP layer when performing arithmetic operations with large integers that require multiple machine words. This parameter gets reflected as part of the total CPU Time, although

given the exponential differences between the 3 implementations, this value is less meaningful as it often falls bellow `0.001 sec`.

For large values of `K` and `N` we have dropped the slower predicates. As a result, the last rows focus exclusively on the fast inverse `to_cantor_tuple`. The table shows that `to_cantor_tuple` scales up within a constant factor of the direct function, but suggest some room for further speed-up through memoing of already computed binomials and binary search for finding the first "combination digit". It also shows that "naturally derived" `to_cantor_tuple2` benefits significantly from the customized cartesian product, that drastically limits its search space, and keeps up with the "perfect solution" `to_cantor_tuple` up to fairly large values of `K` and `N`.

9 Extending the Bijection to Sets and Multisets of `K` Natural Numbers

We obtain a bijection from natural numbers to *sets of* `K` *natural numbers*, canonically represented as lists of strictly increasing elements, by simply dropping the `set2list` and `list2set` operations.

```
from_cantor_set_tuple(Xs,N) :- untupling_loop(Xs,0,0,N).

to_cantor_set_tuple(K,N,Xs) :-
  tupling_loop(K,N,Ts),
  reverse(Ts,Xs).
```

Multisets of `K` *natural numbers* are represented canonically as sequences of nondecreasing but possibly duplicated elements.

Following (Tarau 2009a), a transformation, similar to `list2set/set2list` can be derived for multisets. After a few unfoldings, the resulting code, using tail recursive helper predicates, becomes:

```
mset2set(Ns,Xs) :- mset2set(Ns,0,Xs).

mset2set([],_,[]).
mset2set([X|Xs],I,[M|Ms]) :- I1 is I+1, M is X+I, mset2set(Xs,I1,Ms).

set2mset(Xs,Ns) :- set2mset(Xs,0,Ns).

set2mset([],_,[]).
set2mset([X|Xs],I,[M|Ms]) :- I1 is I+1, M is X-I, set2mset(Xs,I1,Ms).
```

The two transformations can be seen as defining a bijection between strictly increasing and nondecreasing sequences of natural numbers:

```
?- set2mset([2,5,6,8,9],Mset), mset2set(Mset,Set).
Mset = [2, 4, 4, 5, 5], Set = [2, 5, 6, 8, 9].
```

We can combine this bijection with the Cantor n-tupling bijection and obtain

```
from_cantor_multiset_tuple(Ms,N) :-
  mset2set(Ms,Xs),
  from_cantor_set_tuple(Xs,N).
```

```
to_cantor_multiset_tuple(K,N,Ms) :-
  to_cantor_set_tuple(K,N,Xs),
  set2mset(Xs,Ms).
```

As the following application shows, when dealing with *commutative and associative operations*, such multiset encodings turn out to be a natural match.

10 A Simple Application: Fair Search

One might ask, legitimately, why would one bother with pairing and n-tupling bijections. While the case has been made (see for instance (Rosenberg 2003)) for various applications besides theoretical computer science, that range from indexing multi-dimensional data and geographic information systems, to cryptography and coding theory, we will focus here on a simple application with immediate relevance to logic programming: fair search through a multi-parameter search space.

A theorem conjectured by Bachet and proven by Lagrange, states that "*every natural number is the sum of at most four squares*". Let's assume that one wants to find, a "simple" solution to the equation (5), knowing that, as a consequence of this theorem, a solution always exists.

$$N = X^2 + Y^2 + Z^2 + U^2 \tag{5}$$

Let us define "simple solution" as a solution bounded by $O(X + Y + Z + U)$. And we want to enumerate "simpler" candidates first, efficiently. To this end, we can use the fast inverse of the Cantor n-tupling function (specialized to multisets, given that both the "`*`" and "`+`" operations, involved in the equation 5, are associative and commutative). And we can write a generic `fair_multiset_tuple_generator` as:

```
fair_multiset_tuple_generator(From,To,Length, Tuple) :-
  between(From,To,N),
  to_cantor_multiset_tuple(Length,N,Tuple).
```

We can specialize `fair_multiset_tuple_generator` for our specific problem as:

```
to_lagrange_squares(N,Ms) :-
  M is N^2, % conservative upper limit
  fair_multiset_tuple_generator(0,M,4,Ms),
  maplist(square,Ms,MMs),
  sumlist(MMs,N),
  !. % keep the first solution only

square(X,S) :- S is X*X.
```

The algorithm is quite efficient, for instance, it takes only a few seconds to find a decomposition for `2012`:

```
?- time(to_lagrange_squares(2012,Xs)), maplist(square,Xs,Ns), sumlist(Ns,N).
% 9,685,955 inferences, 4.085 CPU in 4.085 seconds (100% CPU, 2371347 Lips)
Xs = [15, 23, 23, 27],
Ns = [225, 529, 529, 729], N = 2012.
```

The algorithm is simple enough to be used as an executable specification and it ensures optimality of the solution in the sense that our search scans hyperplanes of the form $X_1 + X_2 + X_3 + X_4 = K$ for progressively larger and larger values of K. Also, given the multiset representation, the associativity and commutativity of "`*`" and "`+`" are factored in, reducing the search space significantly.

However, our simple algorithm is no match to the $O(log^2(N))$ randomized algorithm of (Rabin and Shallit 1986). As a side note, deriving a faster algorithm for this decomposition is a fascinating task on its own, starting with the observation that it needs only to be computed for the prime factors of a number and involving some elegant identities holding for Hurwitz quaternions (Wikipedia 2012b).

More importantly, the mechanism sketched here can also be used in iterative deepening algorithm as a fair a goal selector, (for both conjunctions and disjunctions). This can be done initially in a meta-interpreter and possibly partially evaluated or moved to the underlying Prolog abstract machine.

Note also that, depending on the natural representation of the candidate data tuple (i.e. set, multiset or sequence), one can customize the fair tuple generator accordingly.

11 Related Work

We have found the first reference to the generalization of Cantor's pairing function to n-tuples in (Cegielski and Richard 1999), and benefited from the extensive study of its properties in (Lisi 2007). There are about 19200 Google documents referring to the original "Cantor pairing function" among which we mention the surprising result that, together with the successor function it defines a decidable subset of arithmetic (Cégielski and Richard 2001). Combinatorial number systems can be traced back to (Lehmer 1964) and one can find efficient conversion algorithms to conventional number systems in (Knuth 2005) and (Buckles and Lybanon 1977). Finally, the "once you have seen it, obvious" `list2set / set2list` bijection is borrowed from (Tarau 2009a) but not unlikely to be common knowledge for people working in combinatorics or recursion theory. This simple bijection between lists and sets of natural numbers shows the unexpected usefulness of the framework supporting bijective data type transformations (Tarau 2009a), of which, a large Haskell-based[2] instance is described in (Tarau 2009b).

12 Conclusion

We have derived through iterative refinements a fairly surprising solution to an open problem for which we had no a priori idea if it is solvable, or within which complexity bounds could be solved. The key "Eureka step" was to recognize a bijective data type transformation that suddenly brought us to a relatively well known equivalent problem for which efficient algorithms were available. Through the process, the ability to automate search algorithms relying directly on an executable

[2] but designed in a guarded Horn-clause style, for virtually automatic transliteration to Prolog

declarative specification has been a major catalyst. The ability to derive equivalent logic programs using simple transformations has been also unusually helpful. From a software engineering perspective, this recommends logic programming as an ideal problem solving tool. Last but not least, proven sources of fundamental algorithms like (Knuth 2005) and the unusually high quality of Wikipedia articles on related topics have helped "connecting the dots" quickly and effectively.

Acknowledgement

This research has been supported by NSF research grant 1018172.

References

Buckles, B. P. and Lybanon, M. 1977. Generation of a Vector form the Lexicagraphical Index [G6]. *ACM Transactions on Mathematical Software 5,* 2 (June), 180–182.

Cegielski, P. and Richard, D. 1999. On arithmetical first-order theories allowing encoding and decoding of lists. *Theoretical Computer Science 222,* 12, 55 – 75.

Cégielski, P. and Richard, D. 2001. Decidability of the Theory of the Natural Integers with the Cantor Pairing Function and the Successor. *Theor. Comput. Sci. 257,* 1-2, 51–77.

Davis, M., Putnam, H., and Robinson, J. 1961. The decision problem for exponential diophantine equations. *The Annals of Mathematics 74,* 4 (nov), 425–436.

Knuth, D. E. 2005. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions.* Addison-Wesley Professional.

Lehmer, D. H. 1964. The machine tools of combinatorics. In *Applied combinatorial mathematics.* Wiley, New York, 5–30.

Lisi, M. 2007. Some remarks on the cantor pairing function. *Le Matematiche 62,* 1.

Matiyasevich, Y. 1993. *Hilbert's Tenth Problem.* MIT Press, Cambridge, London.

Rabin, M. O. and Shallit, J. O. 1986. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics 39,* S1, S239–S256.

Robinson, J. 1969. Unsolvable diophantine problems. *Proceedings of the American Mathematical Society 22,* 2 (aug), 534–538.

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

Tarau, P. 2009a. An Embedded Declarative Data Transformation Language. In *Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009.* ACM, Coimbra, Portugal, 171–182.

Tarau, P. 2009b. Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. Unpublished draft, http://arXiv.org/abs/0808.2953, updated version at http://logic.cse.unt.edu/tarau/research/2010/ISO.pdf, 150 pages.

Wikipedia. 2011a. Combinatorial number system — wikipedia, the free encyclopedia. [Online; accessed 21-March-2012].

Wikipedia. 2011b. Pairing function — wikipedia, the free encyclopedia. [Online; accessed 23-March-2012].

Wikipedia. 2012a. Binomial coefficient — wikipedia, the free encyclopedia. [Online; accessed 21-March-2012].

Wikipedia. 2012b. Lagrange's four-square theorem — wikipedia, the free encyclopedia. [Online; accessed 22-March-2012].