Pairing Functions, Boolean Evaluation and Binary Decision Diagrams

Paul Tarau

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

Abstract. A "pairing function" J associates a unique natural number z to any two natural numbers x,y such that for two "unpairing functions" K and L, the equalities K(J(x,y))=x, L(J(x,y))=y and J(K(z),L(z))=z hold. Using pairing functions on natural number representations of truth tables, we derive an encoding for Binary Decision Diagrams with the unique property that its boolean evaluation faithfully mimics its structural conversion to a a natural number through recursive application of a matching pairing function. We then use this result to derive *ranking* and *unranking* functions for BDDs and reduced BDDs. The paper is organized as a self-contained literate Prolog program, available at `http://logic.csci.unt.edu/tarau/research/2008/pBDD.zip`.
Keywords: logic programming and computational mathematics, pairing/unpairing functions, encodings of boolean functions, binary decision diagrams, natural number representations of truth tables

1 Introduction

This paper is an exploration with logic programming tools of *ranking* and *unranking* problems on Binary Decision Diagrams. The practical expressiveness of logic programming languages (in particular Prolog) are put at test in the process. The paper is part of a larger effort to cover in a declarative programming paradigm, arguably more elegantly, some fundamental combinatorial generation algorithms along the lines of [9].

The paper is organized as follows: Sections 2 and 3 overview efficient evaluation of boolean formulae in Prolog using bitvectors represented as arbitrary length integers and Binary Decision Diagrams (BDDs).

Section 4 discusses classic pairing and unpairing operations and introduces pairing/unpairing predicates acting directly on bitlists.

Section 5 introduces a novel BDD encoding (based on our unpairing functions) and discusses the surprising equivalence between boolean evaluation of BDDs and the inverse of our encoding, the main result of the paper.

Section 6 describes *ranking* and *unranking* functions for BDDs and reduced BDDs.

Sections 7 and 8 discuss related work, future work and conclusions.

The code in the paper, embedded in a literate programming LaTeX file, is entirely self contained and has been tested under *SWI-Prolog*.

2 Parallel Evaluation of Boolean Functions with Bitvector Operations

Evaluation of a boolean function can be performed one value at a time as in the predicate `if_then_else/4`

```
if_then_else(X,Y,Z,R):-
  bit(X),bit(Y),bit(Z),
  ( X==1->R=Y
  ; R=Z
  ).

bit(0).
bit(1).
```

resulting in

```
?- if_then_else(X,Y,Z,R),write([X,Y,Z]:R),nl,fail;nl.
[0, 0, 0]:0
[0, 0, 1]:1
[0, 1, 0]:0
[0, 1, 1]:1
[1, 0, 0]:0
[1, 0, 1]:0
[1, 1, 0]:1
[1, 1, 1]:1
```

Clearly, this does not take advantage of the ability of modern hardware to perform such operations one word a time - with the instant benefit of a speed-up proportional to the word size. An alternate representation, adapted from [9] uses integer encodings of 2^n bits for each boolean variable $X_0, \ldots, X_{n-1}$. Bitvector operations evaluate all value combinations at once.

Proposition 1 *Let x_k be a variable for $0 \leq k < n$ where n is the number of distinct variables in a boolean expression. Then column k in the matrix representation of the inputs in the the truth table represents, as a bitstring, the natural number:*

$$x_k = (2^{2^n} - 1)/(2^{2^{n-k-1}} + 1) \tag{1}$$

For instance, if $n = 2$, the formula computes $x_0 = 3 = [0, 0, 1, 1]$ and $x_1 = 5 = [0, 1, 0, 1]$.

The following predicates, working with arbitrary length bitstrings are used to evaluate variables x_k with $k \in [0..n-1]$ with formula 1 and map the constant boolean function 1 to the bitstring of length 2^n, `111..1`, representing $2^{2^n} - 1$

```
% maps variable K in [0..NbOfBits-1] to Xk
var_to_bitstring_int(NbOfBits,K,Xk):-
  all_ones_mask(NbOfBits,Mask),
```

```
    var_to_bitstring_int(NbOfBits,Mask,K,Xk).

var_to_bitstring_int(NbOfBits,Mask,K,Xk):-
  NK is NbOfBits-(K+1),
  D is (1≪(1≪NK))+1,
  Xk is Mask//D.

% represents constant 1 as 11...1 build with NbOfBits bits
all_ones_mask(NbOfBits,Mask):-Mask is (1≪(1≪NbOfBits))-1.
```

We have used in `var_to_bitstring_int` an adaptation of the efficient bitstring-integer encoding described in the Boolean Evaluation section of [9]. Intuitively, it is based on the idea that one can look at n variables as bitstring representations of the n columns of the truth table.

Variables representing such bitstring-truth tables (seen as *projection functions*) can be combined with the usual bitwise integer operators, to obtain new bitstring truth tables, encoding all possible value combinations of their arguments. Note that the constant 0 is represented as 0 while the constant 1 is represented as $2^{2^n} - 1$, corresponding to a column in the truth table containing ones exclusively.

3 Binary Decision Diagrams

We have seen that Natural Numbers in $[0..2^{2^n} - 1]$ can be used as representations of truth tables defining n-variable boolean functions. A binary decision diagram (BDD) [1] is an ordered binary tree obtained from a boolean function, by assigning its variables, one at a time, to 0 (left branch) and 1 (right branch).

The construction is known as Shannon expansion [24], and is expressed as

$$f(x) = (\bar{x} \wedge f[x \leftarrow 0]) \vee (x \wedge f[x \leftarrow 1]) \tag{2}$$

where $f[x \leftarrow a]$ is computed by uniformly substituting a for x in f. Note that by using the more familiar boolean if-the-else function Shannon expansion can also be expressed as:

$$f(x) = if \ x \ then \ f[x \leftarrow 0] \ else \ f[x \leftarrow 1] \tag{3}$$

We represent a *BDD* in Prolog as a binary tree with constants `0` and `1` as leaves, marked with the function symbol `c/1`. Internal *if-then-else* nodes marked with `ite/3` are controlled by variables, ordered identically in each branch, as first arguments of `ite/1`. The two other arguments are subtrees representing the `Then` and `Else` branches. Note that, in practice, reduced, canonical DAG representations are used instead of binary tree representations.

Alternatively, we observe that the Shannon expansion can be directly derived from a 2^n size truth table, using bitstring operations on encodings of its n variables. Assuming that the first column of a truth table corresponds to variable x, $x = 0$ and $x = 1$ mask out, respectively, the upper and lower half of the truth table.

```
% splits a truth table of NV variables in 2 tables of NV-1 variables
shannon_split(NV,X, Hi,Lo):-
  all_ones_mask(NV,M),
  NV1 is NV-1,
  all_ones_mask(NV1,LM),
  HM is xor(M,LM),
  Lo is /\(LM,X),
  H is /\(HM,X),
  Hi is H>>(1<<NV1).
```

Note that the operation `shannon_split` can be reversed as follows:

```
% fuses 2 truth tables of NV-1 variables into one of NV variables
shannon_fuse(NV,Hi,Lo, X):-
  NV1 is NV-1,
  H is Hi<<(1<<NV1),
  X is \/(H,Lo).
```

```
?- shannon_split(2, 7, X,Y),shannon_fuse(2, X,Y, Z).
X = 1,
Y = 3,
Z = 7.
```

```
?- shannon_split(3, 42, X,Y),shannon_fuse(3, X,Y, Z).
X = 2,
Y = 10,
Z = 42.
```

Another way to look at these two operations (for a fixed value of NV), is as bijections associating a pair of natural numbers to a natural number, i.e. as *pairing* functions.

4 Pairing Functions and Tuple Encodings

Definition 1 *A* pairing function *is a bijection* $f : Nat \times Nat \rightarrow Nat$. *An* unpairing function *is a bijection* $g : Nat \rightarrow Nat \times Nat$.

Following Julia Robinson's notation [16], given a pairing function J, its left and right inverses K and L are such that

$$J(K(z), L(z)) = z \tag{4}$$

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

$$L(J(x, y)) = y \tag{6}$$

We refer to [2] for a typical use in the foundations of mathematics and to [21] for an extensive study of various pairing functions and their computational properties.

4.1   Cantor's Pairing Function

Starting from Cantor's pairing function

```
cantor_pair(K1,K2,P):-P is (((K1+K2)*(K1+K2+1))//2)+K2.
```

bijections from $Nat \times Nat$ to Nat have been used for various proofs and constructions of mathematical objects [16, 17, 20, 19, 2].

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

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

Note however, that the inverse of Cantor's pairing function involves potentially expensive floating point operations that are also likely to loose precision for arbitrary length integers:

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

4.2   The Pepis-Kalmar Pairing Function

Another pairing function that can be implemented using only elementary integer operations is the following:

$$f(x, y) = 2^x * (2 * y + 1) - 1 \tag{7}$$

The predicates `pepis_pair/3` and `pepis_unpair/3` are derived from the function **pepis_J** and its left and right unpairing companions **pepis_K** and **pepis_L** that have been used, by Pepis, Kalmar and Robinson in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem in [12, 6, 18]:

```
pepis_pair(X,Y,Z):-pepis_J(X,Y,Z).

pepis_unpair(Z,X,Y):-pepis_K(Z,X),pepis_L(Z,Y).

pepis_J(X,Y, Z):-Z is ((1<<X)*((Y<<1)+1))-1.
pepis_K(Z, X):-Z1 is Z+1,two_s(Z1,X).
pepis_L(Z, Y):-Z1 is Z+1,no_two_s(Z1,N),Y is (N-1)>>1.

two_s(N,R):-even(N),!,H is N>>1,two_s(H,T),R is T+1.
two_s(_,0).

no_two_s(N,R):-two_s(N,T),R is N // (1<<T).

even(X):- 0 =:= /\(1,X).
odd(X):- 1 =:= /\(1,X).
```

This pairing function is asymmetrically growing (faster growth on the first argument). It works as follows:

```
?- pepis_pair(1,10,R).
R = 41.

?- pepis_unpair(10,1,R).
R = 3071.

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

4.3 Pairing/Unpairing operations acting directly on bitlists

We will describe here pairing operations, that are expressed exclusively as bitlist transformations of `bitmerge_unpair` and its inverse `bitmerge_pair`, 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 [13], page 114).

The predicate `bitmerge_pair` implements a bijection from $Nat \times Nat$ to Nat that works by splitting a number's big endian bitstring representation into odd and even bits, while its inverse `to_pair` blends the odd and even bits back together. The helper predicates `to_rbits` and `from_rbits`, given in the Appendix, convert to/from integers to bitlists.

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

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

bitmix([],[],[]).
bitmix([A],[],[A]).
bitmix([],[Y|Ys],[A,B|Ps]):-bitmix2(0,Y,A,B),bitmix([],Ys,Ps).
bitmix([X|Xs],[],[A,B|Ps]):-bitmix2(X,0,A,B),bitmix(Xs,[],Ps).
bitmix([X|Xs],[Y|Ys],[A,B|Ps]):-bitmix2(X,Y,A,B),bitmix(Xs,Ys,Ps).

bitmix2(X,Y,X,Y).
```

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

```
?- bitmerge_unpair(2008,X,Y),bitmerge_pair(X,Y,Z).
X = 60,
Y = 26,
Z = 2008
```

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

Note also, like in the case of Cantor's pairing function, similar growth in both arguments:

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

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

It is also convenient sometimes to see pairing/unpairing as one-to-one functions from/to the underlying language's ordered pairs:

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

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

5 Encodings of Binary Decision Diagrams

We will build a BDD by applying bitmerge_unpair recursively to a Natural Number TT, seen as an N-variable 2^N bit truth table. This results in a complete binary tree of depth N. As we will show later, this binary tree represents a BDD that returns TT when evaluated applying its boolean operations.

```
% NV=number of varibles, TT=a truth table, BDD the result
plain_bdd(NV,TT, bdd(NV,BDD)):-
  Max is (1<<(1<<NV)),
  TT<Max,
  isplit(NV,TT, BDD).

% recurses to depth NV, splitting TT into pairs
isplit(0,TT,c(TT)).
isplit(NV,TT,R):-NV>0,
  NV1 is NV-1,
  bitmerge_unpair(TT,Hi,Lo),
  isplit(NV1,Hi,H),
```

```
    isplit(NV1,Lo,L),
    ite(NV1,H,L)=R.
```

The following examples show the results returned by `plain_bdd` for all 2^{2^k} truth tables associated to k variables, with $k = 2$.

```
?- between(0,15,TT),plain_bdd(2,TT,BDD),write(TT:BDD),nl,fail;nl
0:bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(0), c(0))))
1:bdd(2, ite(1, ite(0, c(1), c(0)), ite(0, c(0), c(0))))
2:bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(1), c(0))))
...
13:bdd(2, ite(1, ite(0, c(1), c(1)), ite(0, c(0), c(1))))
14:bdd(2, ite(1, ite(0, c(0), c(1)), ite(0, c(1), c(1))))
15:bdd(2, ite(1, ite(0, c(1), c(1)), ite(0, c(1), c(1))))
```

5.1 Reducing the *BDDs*

The predicate `bdd_reduce` reduces a *BDD* by trimming identical left and right subtrees, and the predicate `bdd` associates this reduced form to $N \in Nat$.

```
bdd_reduce(BDD,bdd(NV,R)):-nonvar(BDD),BDD=bdd(NV,X),bdd_reduce1(X,R).

bdd_reduce1(c(TT),c(TT)).
bdd_reduce1(ite(_,A,B),R):-A==B,bdd_reduce1(A,R).
bdd_reduce1(ite(X,A,B),ite(X,RA,RB)):-A\==B,
  bdd_reduce1(A,RA),bdd_reduce1(B,RB).

bdd(NV,TT, ReducedBDD):-
  plain_bdd(NV,TT, BDD),
  bdd_reduce(BDD,ReducedBDD).
```

Note that we omit here the reduction step consisting in sharing common subtrees, as it is obtained easily by replacing trees with DAGs. The process is facilitated by the fact that our unique encoding provides a perfect hashing key for each subtree. The following examples show the results returned by `bdd` for NV=2.

```
?- between(0,15,TT),bdd(2,TT,BDD),write(TT:BDD),nl,fail;nl
0:bdd(2, c(0))
1:bdd(2, ite(1, ite(0, c(1), c(0)), c(0)))
2:bdd(2, ite(1, c(0), ite(0, c(1), c(0))))
3:bdd(2, ite(0, c(1), c(0)))
...
13:bdd(2, ite(1, c(1), ite(0, c(0), c(1))))
14:bdd(2, ite(1, ite(0, c(0), c(1)), c(1)))
15:bdd(2, c(1))
```

5.2   From BDDs to Natural Numbers

One can "evaluate back" the binary tree representing the BDD, by using the
pairing function `bitmerge_pair`. The inverse of `plain_bdd` is implemented as
follows:

```
plain_inverse_bdd(bdd(_,X),TT):-plain_inverse_bdd1(X,TT).

plain_inverse_bdd1(c(TT),TT).
plain_inverse_bdd1(ite(_,L,R),TT):-
  plain_inverse_bdd1(L,X),
  plain_inverse_bdd1(R,Y),
  bitmerge_pair(X,Y,TT).


?- plain_bdd(3,42, BDD),plain_inverse_bdd(BDD,N).
BDD = bdd(3,
         ite(2,
             ite(1,
                 ite(0, c(0), c(0)),
                 ite(0, c(0), c(0))),
             ite(1,
                 ite(0, c(1), c(1)),
                 ite(0, c(1), c(0))))),
N = 42
```

Note however that `plain_inverse_bdd/2` does not act as an inverse of `bdd/3`,
given that the *structure* of the *BDD* tree is changed by reduction.

5.3   Boolean Evaluation of BDDs

This rises the obvious question: how can we recover the original truth table from
a reduced BDD? The obvious answer is: by evaluating it as a boolean function!
The predicate `ev/2` describes the *BDD* evaluator:

```
ev(bdd(NV,B),TT):-
  all_ones_mask(NV,M),
  eval_with_mask(NV,M,B,TT).

evc(0,_,0).
evc(1,M,M).

eval_with_mask(_,M,c(X),R):-evc(X,M,R).
eval_with_mask(NV,M,ite(X,T,E),R):-
  eval_with_mask(NV,M,T,A),
  eval_with_mask(NV,M,E,B),
  var_to_bitstring_int(NV,M,X,V),
  ite(V,A,B,R).
```

The predicate `ite/4` used in `eval_with_mask` implements the boolean function
`if X then T else E` using arbitrary length bitvector operations:

```
ite(X,T,E, R):-R is xor(/\(X,xor(T,E)),E).
```

Note that this equivalent formula for `ite` is slightly more efficient than the obvious one with $\wedge$ and $\vee$ as it requires only 3 boolean operations. We will use `ite/4` as the basic building block for implementing a boolean evaluator for BDDs.

5.4 The Equivalence

A surprising result is that boolean evaluation and structural transformation with repeated application of *pairing* produce the same result, i.e. the predicate `ev/2` also acts as an inverse of `bdd/2` and `plain_bdd/2`.
As the following example shows, boolean evaluation `ev/2` *faithfully emulates* `plain_inverse_bdd/2`, *on both plain and reduced BDDs.*

```
?- plain_bdd(3,42,BDD),ev(BDD,N).
BDD = bdd(3,
        ite(2,
            ite(1,
                ite(0, c(0), c(0)),
                ite(0, c(0), c(0))),
            ite(1,
                ite(0, c(1), c(1)),
                ite(0, c(1), c(0))))),
N = 42

?- bdd(3,42,BDD),ev(BDD,N).
BDD = bdd(3,
        ite(2,
            c(0),
            ite(1,
                c(1),
                ite(0, c(1), c(0))))),
N = 42
```

The main result of this subsection can now be summarized as follows:

Proposition 2 *Let B be the complete binary tree of depth N, obtained by recursive applications of* `bitmerge_unpair` *on a truth table T, as described by the predicate* `plain_bdd(N,T,B)`.

Then for any NV and any T, when B is interpreted as an (unreduced) BDD, the result V of its boolean evaluation using the predicate ev(N, B, V) and the result R obtained by applying plain_inverse_bdd(N, B, R) are both identical to T. Moreover, the operation ev(N, B, V) reverses the effects of both `plain_bdd` *and* `bdd` *with an identical result.*

Proof: The predicate `plain_bdd` builds a binary tree by splitting the bitstring $tt \in [0..2^N - 1]$ up to depth N. Observe that this corresponds to the Shannon

expansion [24] of the formula associated to the truth table, using variable order $[n-1, ..., 0]$. Observe that the effect of `bitstring_unpair` is the same as

- the effect of `var_to_bitstring_int(N,M,(N-1),R)` acting as a mask selecting the left branch
- and the effect of its complement, acting as a mask selecting the right branch.

Given that 2^N is the double of 2^{N-1}, the same invariant holds at each step, as the bitstring length of the truth table reduces to half. On the other hand, it is clear that *ev* reverses the action of both `plain_bdd` and `bdd` as BDDs and reduced BDDs represent the same boolean function [1].

This result can be seen as a yet another intriguing isomorphism between boolean, arithmetic and symbolic computations.

6 Ranking and Unranking of BDDs

One more step is needed to extend the mapping between *BDDs* with N variables to a bijective mapping from/to *Nat*: we will have to "shift towards infinity" the starting point of each new block of BDDs in *Nat* as BDDs of larger and larger sizes are enumerated.

First, we need to know by how much - so we compute the sum of the counts of boolean functions with up to N variables.

```
bsum(0,0).
bsum(N,S):-N>0,N1 is N-1,bsum1(N1,S).

bsum1(0,2).
bsum1(N,S):-N>0,N1 is N-1,bsum1(N1,S1),S is S1+(1<<(1<<N)).
```

The stream of all such sums can now be generated as usual:

```
bsum(S):-nat(N),bsum(N,S).

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

What we are really interested into, is decomposing `N` into the distance to the last `bsum` smaller than N, `N_M` and the index of that generates the sum, `K`.

```
to_bsum(N, X,N_M):-
  nat(X),bsum(X,S),S>N,!,
  K is X-1,
  bsum(K,M),
  N_M is N-M.
```

Unranking of an arbitrary BDD is now easy - the index `K` determines the number of variables and `N_M` determines the rank. Together they select the right BDD with `plain_bdd` and `bdd/3`.

```
nat2plain_bdd(N,BDD):-to_bsum(N, K,N_M),plain_bdd(K,N_M,BDD).

nat2bdd(N,BDD):-to_bsum(N, K,N_M),bdd(K,N_M,BDD).
```

Ranking of a BDD is even easier: we first compute its `NumberOfVars` and its rank `Nth`, then we shift the rank by the `bsums` up to `NumberOfVars`, enumerating the ranks previously assigned.

```
plain_bdd2nat(bdd(NumberOfVars,BDD),N) :-
  B=bdd(NumberOfVars,BDD),
  plain_inverse_bdd(B,Nth),
  K is NumberOfVars-1,
  bsum(K,S),N is S+Nth.

bdd2nat(bdd(NumberOfVars,BDD),N) :-
  B=bdd(NumberOfVars,BDD),
  ev(B,Nth),
  K is NumberOfVars-1,
  bsum(K,S),N is S+Nth.
```

As the following example shows, `nat2plain_bdd/2` and `plain_bdd2nat/2` implement inverse functions.

```
?- nat2plain_bdd(42,BDD),plain_bdd2nat(BDD,N).
BDD = bdd(4,
         ite(3,
             ite(2,
                 ite(1,
                     ite(0, c(0), c(0)),
                     ite(0, c(1), c(0))),
                 ite(1,
                     ite(0, c(1), c(0)),
                     ite(0, c(0), c(0)))),
             ite(2,
                 ite(1,
                     ite(0, c(0), c(0)),
                     ite(0, c(0), c(0))),
                 ite(1, ite(0, c(0), c(0)),
                        ite(0, c(0), c(0)))))),
N = 42
```

The same applies to `nat2bdd/2` and its inverse `bdd2nat/2`.

```
?- nat2bdd(42,BDD),bdd2nat(BDD,N).
BDD = bdd(4,
         ite(3,
             ite(2,
             ite(1, c(0),
                    ite(0, c(1), c(0))),
                 ite(1,
                     ite(0, c(1),c(0)), c(0))),
                 c(0))),
N = 42
```

We can now generate infinite streams of BDDs as follows:

```
plain_bdd(BDD):-nat(N),nat2plain_bdd(N,BDD).

bdd(BDD):-nat(N),nat2bdd(N,BDD).


?- plain_bdd(BDD).
11 ?- plain_bdd(BDD).
BDD = bdd(1, ite(0, c(0), c(0))) ;
BDD = bdd(1, ite(0, c(1), c(0))) ;
BDD = bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(0), c(0)))) ;
BDD = bdd(2, ite(1, ite(0, c(1), c(0)), ite(0, c(0), c(0)))) ;
BDD = bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(1), c(0)))) ;
BDD = bdd(2, ite(1, ite(0, c(1), c(0)), ite(0, c(1), c(0))))
...
?- bdd(BDD).
BDD = bdd(1, c(0)) ;
BDD = bdd(1, ite(0, c(1), c(0))) ;
BDD = bdd(2, c(0)) ;
BDD = bdd(2, ite(1, ite(0, c(1), c(0)), c(0))) ;
BDD = bdd(2, ite(1, c(0), ite(0, c(1), c(0)))) ;
BDD = bdd(2, ite(0, c(1), c(0))) ;
BDD = bdd(3, c(0)) ;
BDD = bdd(3, ite(2, ite(1, ite(0, c(1), c(0)), c(0)), c(0))) ;
BDD = bdd(3, ite(2, c(0), ite(1, ite(0, c(1), c(0)), c(0)))) ;
BDD = bdd(3, ite(1, ite(0, c(1), c(0)), c(0)))
...
```

7   Related work

Pairing functions have been used work on decision problems as early as [12, 6–8, 16, 17, 20, 19]. Binary Decision Diagrams are the dominant boolean function representation in the field of circuit design automation [11, 4]. BDDs have been used in a Genetic Programming context [23, 22, 3] as a representation of evolving individuals subject to crossovers and mutations expressed as structural transformations and recently in a machine learning context for compressing probabilistic Prolog programs [15] representing candidate theories. Other interesting use of BDDs in a logic and constraint programming context are related to representations of finite domains. In [5] an algorithm for finding minimal reasons for inferences is given.

8   Conclusion and Future Work

The surprising connection of pairing/unpairing functions and BDDs, is the indirect result of implementation work on a number of practical applications. Our

initial interest has been triggered by applications of the encodings to combinational circuit synthesis in a logic programming framework [26, 25]. We have found them also interesting as uniform blocks for Genetic Programming applications of Logic Programming. In a Genetic Programming context [10, 14], the bijections between bitvectors/natural numbers on one side, and trees/graphs representing BDDs on the other side, suggest exploring the mapping and its action on various transformations as a phenotype-genotype connection. Given the connection between BDDs to boolean and finite domain constraint solvers it would be interesting to explore in that context, efficient succinct data representations derived from our BDD encodings.

References

1. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
2. Patrick Cégielski and Denis Richard. Decidability of the theory of the natural integers with the cantor pairing function and the successor. *Theor. Comput. Sci.*, 257(1-2):51–77, 2001.
3. Shan-Tai Chen, Shun-Shii Lin, Li-Te Huang, and Chun-Jen Wei. Towards the exact minimization of bdds-an elitism-based distributed evolutionary algorithm. *J. Heuristics*, 10(3):337–355, 2004.
4. Rolf Drechsler, Junhao Shi, and Görschwin Fey. Synthesis of fully testable circuits from bdds. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(3):440–443, 2004.
5. Peter Hawkins and Peter J. Stuckey. A hybrid bdd and sat finite domain constraint solver. In Pascal Van Hentenryck, editor, *PADL*, volume 3819 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2006.
6. Laszlo Kalmar. On the reduction of the decision problem. first paper. ackermann prefix, a single binary predicate. *The Journal of Symbolic Logic*, 4(1):1–9, mar 1939.
7. Kalmar, Laszlo and Suranyi, Janos. On the reduction of the decision problem. *The Journal of Symbolic Logic*, 12(3):65–73, sep 1947.
8. Kalmar, Laszlo and Suranyi, Janos. On the reduction of the decision problem: Third paper. pepis prefix, a single binary predicate. *The Journal of Symbolic Logic*, 15(3):161–173, sep 1950.
9. Donald Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼/knuth/taocp.html.
10. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.
11. Christoph Meinel and Thorsten Theobald. Ordered binary decision diagrams and their significance in computer-aided design of vlsi circuits. *Journal of Circuits, Systems, and Computers*, 9(3-4):181–198, 1999.
12. Jozef Pepis. Ein verfahren der mathematischen logik. *The Journal of Symbolic Logic*, 3(2):61–76, jun 1938.
13. Stephen Pigeon. *Contributions à la compression de données.* Ph.d. thesis, Université de Montréal, Montréal, 2001.
14. Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. *A Field Guide to Genetic Programming.* e-book.

15. Luc De Raedt, Kristian Kersting, Angelika Kimmig, Kate Revoredo, and Hannu Toivonen. Compressing probabilistic prolog programs. *Machine Learning*, 70(2-3):151–168, 2008.

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

17. Julia Robinson. A note on primitive recursive functions. *Proceedings of the American Mathematical Society*, 6(4):667–670, aug 1955.

18. Julia Robinson. An introduction to hyperarithmetical functions. *The Journal of Symbolic Logic*, 32(3):325–342, sep 1967.

19. Julia Robinson. Finite generation of recursively enumerable sets. *Proceedings of the American Mathematical Society*, 19(6):1480–1486, dec 1968.

20. Julia Robinson. Recursive functions of one variable. *Proceedings of the American Mathematical Society*, 19(4):815–820, aug 1968.

21. Arnold L. Rosenberg. Efficient pairing functions - and why you should care. In *IPDPS*. IEEE Computer Society, 2002.

22. Franz Rothlauf, Jürgen Branke, Stefano Cagnoni, Ernesto Costa, Carlos Cotta, Rolf Drechsler, Evelyne Lutton, Penousal Machado, Jason H. Moore, Juan Romero, George D. Smith, Giovanni Squillero, and Hideyuki Takagi, editors. *Applications of Evolutionary Computing, EvoWorkshops 2006: EvoBIO, EvoCOMNET, Evo-HOT, EvoIASP, EvoINTERACTION, EvoMUSART, and EvoSTOC, Budapest, Hungary, April 10-12, 2006, Proceedings*, volume 3907 of *Lecture Notes in Computer Science*. Springer, 2006.

23. Hidenori Sakanashi, Tetsuya Higuchi, Hitoshi Iba, and Yukinori Kakazu. Evolution of binary decision diagrams for digital circuit design using genetic programming. In Tetsuya Higuchi, Masaya Iwata, and Weixin Liu, editors, *ICES*, volume 1259 of *Lecture Notes in Computer Science*, pages 470–481. Springer, 1996.

24. Claude E. Shannon. *Claude Elwood Shannon: collected papers*. IEEE Press, Piscataway, NJ, USA, 1993.

25. Paul Tarau and Brenda Luderman. A Logic Programming Framework for Combinational Circuit Synthesis. In *23rd International Conference on Logic Programming (ICLP), LNCS 4670*, pages 180–194, Porto, Portugal, September 2007. Springer.

26. Paul Tarau and Brenda Luderman. Exact Combinational Logic Synthesis and Non-Standard Circuit Design. In *Proceedings of ACM Computing Frontiers'08*, Ischia, Italy, May 2008.

9 Appendix

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

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

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

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