Ranking and Unranking Functions for BDDs and MTBDDs with Applications to Circuit Minimization

Paul Tarau[1] and Brenda Luderman[2]

[1] Department of Computer Science and Engineering
University of North Texas
Denton, Texas
tarau@cs.unt.edu
[2] ACES CAD
Lewisville, Texas, USA
brenda.luderman@gmail.com

Abstract. We describe Haskell implementations of combinatorial generation algorithms with focus on boolean functions and logic circuit representations. Using *pairing* and *unpairing* functions on natural number representations of truth tables, we derive an encoding for Binary Decision Diagrams (BDDs) with the unique property that its boolean evaluation faithfully mimics its structural conversion to 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. Finally, a generalization of the encoding techniques to Multi-Terminal BDDs and an application to BDD minimization are described. The paper is organized as a self-contained literate Haskell program, available at `http://logic.csci.unt.edu/tarau/research/2008/fBDD.zip`.
Keywords: *binary decision diagrams, encodings of boolean functions, pairing/unpairing functions, ranking/unranking functions for BDDs and MTBDDs, computational mathematics in Haskell*

1 Introduction

This paper is an exploration with functional programming tools of *ranking* and *unranking* problems on Binary Decision Diagrams. The paper is part of a larger effort to cover in a declarative programming paradigm some fundamental combinatorial generation and boolean function manipulation algorithms along the lines of Knuth's recent work [4].

The paper is organized as follows:

Sections 2 and 3 overview efficient evaluation of boolean formulae in Haskell using bitvectors represented as arbitrary length integers and Binary Decision Diagrams (BDDs). Section 4 introduces pairing/unpairing functions 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. Section 6 describes *ranking*

and *unranking* functions for BDDs and reduced BDDs. Section 7 extends our techniques to BDDs with arbitrary variable order and Multi-Terminal BDDs and applications to BDD-based circuit minimization and generation of random test data. Sections 8 and 9 discuss related work, future work and conclusions.

2 Evaluation of Boolean Functions with Bitvector Operations

Evaluation of a boolean function can be performed one bit at a time as in the function `if_then_else`

```
if_then_else 0 _ z = z
if_then_else 1 y _ = y
```

Unfortunately 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 [4] uses integer encodings of 2^n bits for each boolean variable $x_0, \ldots, x_{n-1}$. Bitvector operations are used to 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 of 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 functions, working with arbitrary length bitstrings are used to evaluate the [0..n-1] variables x_k with formula 1 and map the constant `1` to the bitstring of length 2^n, `111..1`:

```
var_n n k = var_mn (bigone n) n k
var_mn mask n k = mask `div` (2^(2^(n-k-1))+1)
bigone nvars = 2^2^nvars - 1
```

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` (right branch) and `1` (left branch).

The construction is known as Shannon expansion [10], and is expressed as a decomposition of a function in two *cofactors*, $f[x \leftarrow 0]$ and $f[x \leftarrow 1]$

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

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, the Shannon expansion can also be expressed as:

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

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.

Seen as an operation on bitvectors, the Shannon expansion (for a fixed number of variables) defines a bijection associating a pair of natural numbers (the cofactor's truth tables) to a natural number (the function's truth table), i.e. it works as a unpairing function*.*

4 Pairing/Unpairing

Let Nat denote the set of natural numbers (0 included). A *pairing function* is an isomorphism $f : Nat \times Nat \rightarrow Nat$. Its inverse is called an *unpairing function.*

We introduce here an unusually simple pairing function (also mentioned in [7], p.142). The function `bitpair` works by splitting a number's big endian bitstring representation into odd and even bits, while its inverse `bitunpair` blends the odd and even bits back together.

```
type Nat = Integer
data Nat2 = P Nat Nat deriving (Eq,Ord,Read,Show)

bitpair ::  Nat2 → Nat
bitpair (P i j) =
  set2nat ((evens i) ++ (odds j)) where
    evens x = map (2*) (nat2set x)
    odds y = map succ (evens y)

bitunpair :: Nat→Nat2
bitunpair n = P (f xs) (f ys) where
  (xs,ys) = partition even (nat2set n)
  f = set2nat . (map (`div` 2))
```

The functions `set2nat` and `nat2set` convert to/from natural numbers to lists of exponents of 2 representing positions of bits=1.

```
nat2set n | n≥0 = nat2exps n 0 where
  nat2exps 0 _ = []
```

```
  nat2exps n x =
    if (even n) then xs else (x:xs) where
      xs=nat2exps (n `div` 2) (succ x)

set2nat ns = sum (map (2^) ns)
```

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

```
*BDD> bitunpair 2008
  (60,26)

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

5 Pairing Functions and Encodings of Binary Decision Diagrams

We show in this section that a Binary Decision Diagram (BDD) representing the same logic function as an n-variable 2^n bit truth table can be obtained by applying `bitunpair` recursively to `tt`. More precisely, we show that applying this *unfolding* operation results in a complete binary tree of depth n representing a BDD that returns `tt` when evaluated applying its boolean operations.

The binary tree type `BT` has the constants `B0` and `B1` as leaves representing the boolean values 0 and 1. Internal nodes (that represent `if-then-else` decision points), are marked with the constructor `D`. We also add integers, representing logic variables, ordered identically in each branch, as first arguments of `D`. The two other arguments are subtrees that represent `THEN` and `ELSE` branches:

```
data BT a = B0 | B1 | D a (BT a) (BT a) deriving (Eq,Ord,Read,Show)
```

The constructor `BDD` wraps together a tree of type `BT` and the number of logic variables occurring in it.

```
data BDD a = BDD a (BT a) deriving (Eq,Ord,Read,Show)
```

5.1 Unfolding natural numbers to binary trees with `bitunpair`

The following functions apply `bitunpair` recursively, on a Natural Number `tt`, seen as an n-variable 2^n bit truth table, to build a complete binary tree of depth n, that we represent using the `BDD` data type.

```
unfold_bdd :: Nat2 → BDD Nat
unfold_bdd (P n tt) = BDD n bt where
  bt=if tt<max then shf bitunpair n tt
     else error
       ("unfold_bdd: last arg "++ (show tt)++
```

```
      " should be < " ++ (show max))
    where max = 2^2^n

  shf _ n 0 | n<1 =  B0
  shf _ n 1 | n<1 =  B1
  shf f n tt = D k (shf f k tt1) (shf f k tt2) where
    k=pred n
    P tt1 tt2=f tt
```

The examples below show results returned by `unfold_bdd` for the 2^{2^n} truth tables associated to n variables, for $n = 2$:

```
BDD 2 (D 1 (D 0 B0 B0) (D 0 B0 B0))
BDD 2 (D 1 (D 0 B1 B0) (D 0 B0 B0))
BDD 2 (D 1 (D 0 B0 B0) (D 0 B1 B0))
...
BDD 2 (D 1 (D 0 B1 B1) (D 0 B1 B1))
```

Note that no boolean operations have been performed so far and that we still have to prove that such trees actually represent BDDs associated to truth tables.

5.2 Folding binary trees to natural numbers with `bitpair`

One can "evaluate back" the binary tree of data type BDD, by using the pairing function `bitpair`. The inverse of `unfold_bdd` is implemented as follows:

```
fold_bdd :: BDD Nat → Nat2
fold_bdd (BDD n bt) =
  P n (rshf bitpair bt) where
    rshf rf B0 = 0
    rshf rf B1 = 1
    rshf rf (D _ l r) =
      rf (P (rshf rf l) (rshf rf r))
```

Note that this is a purely structural operation and that integers in first argument position of the constructor `D` are actually ignored.

The two bijections, inverses of each other, work as follows:

```
*BDD>unfold_bdd (P 3 42)
  BDD 3
    (D 2
      (D 1 (D 0 B0 B0)
           (D 0 B0 B0))
      (D 1 (D 0 B1 B1)
           (D 0 B1 B0)))

*BDD>fold_bdd it
  42
```

5.3 Boolean Evaluation of BDDs

Practical uses of BDDs involve reducing them by sharing nodes and eliminating identical branches [1]. Note that in this case `bdd2nat` might give a different result as it computes different pairing operations. Fortunately, we can try to fold the binary tree back to a natural number by evaluating it as a boolean function.

The function `eval_bdd` describes the *BDD* evaluator:

```
eval_bdd (BDD n bt) = eval_with_mask (bigone n) n bt

eval_with_mask m _ B0 = 0
eval_with_mask m _ B1 = m
eval_with_mask m n (D x l r) =
  ite_ (var_mn m n x)
       (eval_with_mask m n l)
       (eval_with_mask m n r)
```

The *projection functions* `var_mn` defined in section 2 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 evaluates to 0 while the constant 1 is evaluated as $2^{2^n} - 1$ by the function `bigone`.

The function `ite_` used in `eval_with_mask` implements the boolean function `if x then t else e` using arbitrary length bitvector operations:

```
ite_ x t e = ((t `xor` e).&.x) `xor` e
```

As the following example shows, it turns out that boolean evaluation `eval_bdd` *faithfully emulates* `fold_bdd`*!*

```
*BDD> unfold_bdd (P 3 42)
BDD 3 (D 2 (D 1 (D 0 B0 B0) (D 0 B0 B0))
           (D 1 (D 0 B1 B1) (D 0 B1 B0)))
*BDD> eval_bdd it
42
```

5.4 The Equivalence

We now state the surprising (and new!) result that boolean evaluation and structural transformation with repeated application of *pairing* produce the same result:

Proposition 2 *The complete binary tree of depth n, obtained by recursive applications of* `bitunpair` *on a truth table tt computes an (unreduced) BDD, that, when evaluated, returns the truth table, i.e.*

$$fold\_bdd \circ unfold\_bdd \equiv id \tag{4}$$

$$eval\_bdd \circ unfold\_bdd \equiv id \tag{5}$$

Proof. The function `unfold_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 [10] of the formula associated to the truth table, using variable order $[n-1, ..., 0]$. Observe that the effect of `bitunpair` is the same as

- the effect of `var_mn m n (n-1)` 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.

We can thus assume from now on, that the BDD data type defined in section 5 actually represents BDDs mapped one-to-one to truth tables given as natural numbers.

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[3] of BDDs in Nat as BDDs of larger and larger sizes are enumerated.

First, we need to know by how much - so we count the number of boolean functions with up to n variables.

```
bsum 0 = 0
bsum n | n>0 = bsum1 (n-1)

bsum1 0 = 2
bsum1 n | n>0 = bsum1 (n-1)+ 2^2^n
```

The stream of all such sums can now be generated as usual[4]:

```
bsums = map bsum [0..]

*BDD> genericTake 7 bsums
  [0,2,6,22,278,65814,4295033110]
```

What we are really interested in, is decomposing `n` into the distance `n-m` to the last `bsum m` smaller than `n`, and the index that generates the sum, `k`.

```
to_bsum n = (k,n-m) where
  k=pred (head [x|x←[0..],bsum x>n])
  m=bsum k
```

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 `unfold_bdd` and `bdd`.

[3] defined by the same number of variables

[4] `bsums` is sequence A060803 in The On-Line Encyclopedia of Integer Sequences, `http://www.research.att.com/~njas/sequences`

```
nat2bdd n = unfold_bdd (P k n_m)
  where (k,n_m)=to_bsum n
```

Ranking of a BDD is even easier: we shift its rank within the set of BDDs with `nv` variables, by the value `(bsum nv)` that counts the ranks previously assigned.

```
bdd2nat bdd@(BDD nv _) =  ((bsum nv)+tt) where
  P _ tt =fold_bdd bdd
```

As the following example shows, `bdd2nat` implements the inverse of `nat2bdd`.

```
*BDD> nat2bdd 42
BDD 3 (D 2 (D 1 (D 0 B0 B1) (D 0 B1 B0))
           (D 1 (D 0 B0 B0) (D 0 B0 B0)))
*BDD> bdd2nat it
42
```

We can now repeat the *ranking* function construction for `eval_bdd`:

```
ev_bdd2nat bdd@(BDD nv _) = (bsum nv)+(eval_bdd bdd)
```

We can confirm that `ev_bdd2nat` also acts as an inverse to `nat2bdd`:

```
*BDD> ev_bdd2nat (nat2bdd 2008)
2008
```

6.1 Reducing the *BDDs*

We sketch here a simplified reduction mechanism for BDDs eliminating identical branches. Note that the general mechanism involves DAGs and provides also node sharing [1].

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

```
bdd_reduce (BDD n bt) = BDD n (reduce bt) where
  reduce B0 = B0
  reduce B1 = B1
  reduce (D _ l r) | l == r = reduce l
  reduce (D v l r) = D v (reduce l) (reduce r)

unfold_rbdd = bdd_reduce . unfold_bdd
```

The results returned by `unfold_rbdd` for `n=2` are:

```
BDD 2 (C 0)
BDD 2 (D 1 (D 0 (C 1) (C 0)) (C 0))
BDD 2 (D 1 (C 0) (D 0 (C 1) (C 0)))
BDD 2 (D 0 (C 1) (C 0))
...
BDD 2 (D 1 (D 0 (C 0) (C 1)) (C 1))
BDD 2 (C 1)
```

We can now define the *unranking* operation on reduced BDDs

```
nat2rbdd = bdd_reduce . nat2bdd
```

To be able to compare its space complexity with other representations we define a size operation on a BDD as follows:

```
bdd_size (BDD _ t) = 1+(size t) where
  size B0 = 1
  size B1 = 1
  size (D _ l r) = 1+(size l)+(size r)
```

7 Generalizing BDD ranking/unrankig functions

7.1 Encoding BDDs with Arbitrary Variable Order

While the encoding built around the equivalence described in Prop. 2 between bitwise pairing/unpairing operations and boolean decomposition is arguably as simple and elegant as possible, it is useful to parametrize BDD generation with respect to an arbitrary variable order. This is of particular importance when using BDDs for circuit minimization, as different variable orders can make circuit sizes flip from linear to exponential in the number of variables [1].

Given a permutation of n variables represented as natural numbers in $[0..n-1]$ and a truth table $tt \in [0..2^{2^n} - 1]$ we can define:

```
to_bdd vs tt | 0≤tt && tt ≤ m =
  BDD n (to_bdd_mn vs tt m n) where
    n=genericLength vs
    m=bigone n
to_bdd _ tt = error
   ("bad arg in to_bdd⇒" ++ (show tt))
```

where the function `to_bdd_mn` recurses over the list of variables `vs` and applies Shannon expansion [10], expressed as bitvector operations. This computes the cofactor functions $f1$ and $f0$, to be used as `then` and `else` branches, when evaluating back the BDD to a truth table with if-the-else functions.

```
to_bdd_mn []      0 _ _ = B0
to_bdd_mn []      _ _ _ = B1
to_bdd_mn (v:vs) tt m n = D v l r where
  cond=var_mn m n v
  f0= (m `xor` cond) .&. tt
  f1= cond .&. tt
  l=to_bdd_mn vs f1 m n
  r=to_bdd_mn vs f0 m n
```

Proposition 3 *The function* `to_bdd` *builds an (unreduced) BDD corresponding to a truth table* `tt` *for variable order* `vs` *that returns* `tt`*, when evaluated as a boolean function.*

We can reduce the resulting BDDs, and convert back from BDDs and reduced BDDs to truth tables with boolean evaluation:

```
to_rbdd vs tt = bdd_reduce (to_bdd vs tt)
from_bdd bdd = eval_bdd bdd
```

Finally, we can, obtain an optimal BDD expressing a logic function of n variables given as a truth table as follows:

```
search_bdd minORmax n tt = snd $ foldl1 minORmax
 (map (sized_rbdd tt) (all_permutations n)) where
    sized_rbdd tt vs = (bdd_size b,b) where
      b=to_rbdd vs tt

all_permutations n = permute [0..n-1] where

  permute [] = [[]]
  permute (x:xs) = [zs| ys←permute xs, zs←insert x ys]

  insert a [] = [[a]]
  insert a (x:xs) = (a:x:xs):[(x:ys) | ys←(insert a xs)]
```

The function `search_bdd min` can be used for multilevel boolean formula minimization on functions with up to 6-7 arguments.

```
*BDD> search_bdd min 3 42
BDD 3 (D 2 B0 (D 0 B1 (D 1 B1 B0)))
*BDD> search_bdd min 4 2008
BDD 4 (D 0 (D 3 (D 1 B0 B1) (D 2 B0 B1))
           (D 3 (D 1 B1 B0) (D 1 (D 2 B1 B0) B0)))
*BDD> search_bdd min 3 2008
BDD 7 (D 1 (D 2 (D 4 (D 3 (D 0 (D 5 ...
      ... (D 0 (D 5 B0 B1) B0)))))
*BDD> bdd_size it
110
```

Let us consider the classic problem of synthesizing a half adder, composed of an XOR (`^`) and an AND (`*`) function.

It is interesting to see how the BDD minimization algorithm compares with "perfect circuits" provided by an exact synthesizer as the one described in [11, 12]. The output of the exact synthesizer uses a graph representation where the 4-th argument names the output connection:

```
?- syn([ite],[0,1],[ite(A,B^C,B*C)]).
[TTs = [22],MG = 24]
syn(3,24,[ite],[0,1],[22]).
[0,0,0]:0
[0,0,1]:0
[0,1,0]:0
[0,1,1]:1
[1,0,0]:0
[1,0,1]:1
[1,1,0]:1
[1,1,1]:0
```

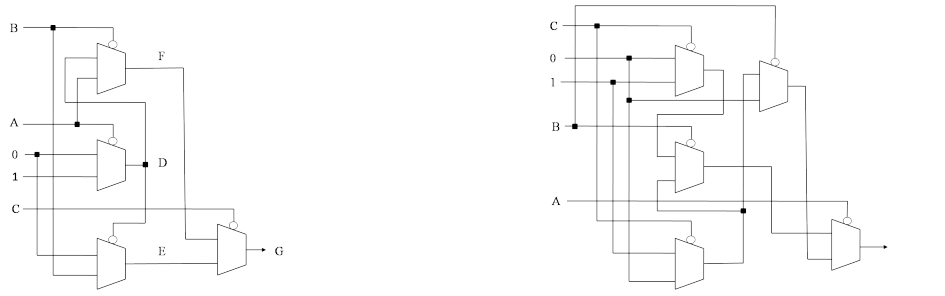
```
[A,B,C]:
   [ite(A,0,1,D),
    ite(D,0,B,E),
    ite(B,D,A,F),
    ite(C,F,E,G)] = [G]:[22].
```

Note that 4 `ITE` gates (2-1 mux with 1-0 selection lines) are used to combine the `XOR` and `AND` functions into a single output function, with the upper half of the truth table representing the `AND` and the lower half representing the `XOR`.

When running `to_min_bdd` on the 3-variable function 22 representing as a natural number the truth table of our our half adder, we obtain:

```
*BDD> search_bdd min 3 22
BDD 3 (D 0 (D 1 (D 2 B0 B1)
                (D 2 B1 B0))
           (D 1 (D 2 B1 B0)
                B0))
```

Assuming the sharing of the two `(D 2 B1 B0)` nodes we can see that while we do not obtain the exact minimum of 4 in this case, we get close enough (5 gates). The diagrams in Fig. 1 show the actual circuits, with variables `0,1,2` renamed as `A,B,C` for easier comparison.



(a) As minimal BDDs (exact synthesis) (b) As minimal ROBDDs

Fig. 1: Binary decision diagrams associated to half adder $ITE(A \oplus B, A \wedge B)$

7.2 Multi-Terminal Binary Decision Diagrams (MTBDD)

MTBDDs [3, 2] are a natural generalization of BDDs allowing non-binary values as leaves. Such values are typically bitstrings representing the outputs of a multi-terminal boolean function, encoded as unsigned integers.

We shall now describe an encoding of $MTBDDs$ that can be extended to ranking/unranking functions, in a way similar to $BDDs$ as shown in section 6.

Our `MTBDD` data type is a binary tree like the one used for $BDDs$, parameterized by two integers `m` and `n`, indicating that an MTBDD represents a function from $[0..n-1]$ to $[0..m-1]$, or equivalently, an n-input/m-output boolean function.

```
data MT a = L a | M a (MT a) (MT a)
            deriving (Eq,Ord,Read,Show)
data MTBDD a = MTBDD a a (MT a) deriving (Show,Eq)
```

The function `to_mtbdd` creates, from a natural number tt representing a truth table, an MTBDD representing functions of type $N \to M$ with $M = [0..2^m - 1], N = [0..2^n - 1]$. Similarly to a BDD, it is represented as binary tree of n levels, except that its leaves are in $[0..2^m - 1]$.

```
to_mtbdd m n tt = MTBDD m n r where
  mlimit=2^m
  nlimit=2^n
  ttlimit=mlimit^nlimit
  r=if tt<ttlimit
    then (to_mtbdd_ mlimit n tt)
    else error
      ("bt: last arg "++ (show tt)++
      " should be < " ++ (show ttlimit))
```

Given that correctness of the range of `tt` has been checked, the function `to_mtbdd_` applies `bitunpair` recursively up to depth n, where leaves in range $[0..mlimit-1]$ are created.

```
to_mtbdd_ mlimit n tt|(n<1)&&(tt<mlimit) = L tt
to_mtbdd_ mlimit n tt = (M k l r) where
   P x y=bitunpair tt
   k=pred n
   l=to_mtbdd_ mlimit k x
   r=to_mtbdd_ mlimit k y
```

Converting back from $MTBDDs$ to natural numbers is basically the same thing as for $BDDs$, except that assertions about the range of leaf data are enforced.

```
from_mtbdd (MTBDD m n b) = from_mtbdd_ (2^m) n b

from_mtbdd_ mlimit n (L tt)|(n<1)&&(tt<mlimit)=tt
from_mtbdd_ mlimit n (M _ l r) = tt where
   k=pred n
   x=from_mtbdd_ mlimit k l
   y=from_mtbdd_ mlimit k r
   tt=bitpair (P x y)
```

The following examples show that `to_mtbdd` and `from_mtbdd` are indeed inverses values in $[0..2^n - 1] \times [0..2^m - 1]$.

```
>to_mtbdd 3 3 2008
  MTBDD 3 3
    (M 2
      (M 1
         (M 0 (L 2) (L 1))
         (M 0 (L 2) (L 1)))
      (M 1
         (M 0 (L 2) (L 0))
         (M 0 (L 1) (L 1))))

>from_mtbdd it
2008

>mprint (to_mtbdd 2 2) [0..3]
  MTBDD 2 2
    (M 1 (M 0 (L 0) (L 0)) (M 0 (L 0) (L 0)))
  MTBDD 2 2
    (M 1 (M 0 (L 1) (L 0)) (M 0 (L 0) (L 0)))
  MTBDD 2 2
    (M 1 (M 0 (L 0) (L 0)) (M 0 (L 1) (L 0)))
  MTBDD 2 2
    (M 1 (M 0 (L 1) (L 0)) (M 0 (L 1) (L 0)))
```

7.3 Generating Random BDDs and MTBDDs

Random generation of BDDs and MTBDDs have practical uses in testing and benchmarking of various electronic design automation tools and methodologies.

Deriving mechanisms for uniform generation of random instances is a classic application of ranking/unranking functions. Given a one-to-one mapping to Nat it reduces to the simpler problem of uniform generation of natural numbers.

After customizing Haskell's library random generator

```
nrandom_nats smallest largest n seed=
  genericTake n
    (randomRs (smallest,largest) (mkStdGen seed))
```

one can define:

```
nrandom converter smallest largest n seed =
  map converter (nrandom_nats smallest largest n seed)
```

To generate 3 small instances of reduced BDD mapped to natural numbers from 10 to 20 one can write:

```
*BDD> nrandom nat2rbdd 10 20 3 77
[ BDD 2 (D 1 (D 0 B1 B0) B1),
  BDD 2 (D 1 (D 0 B0 B1) B1),
  BDD 2 (D 0 B0 B1)]
```

To generate an instance of a random 3-in/3-out MTBDD mapped to natural numbers from 1000 to 2000 one can write:

```
*BDD> head $ nrandom (to_mtbdd 3 3) 1000 2000 1 1
MTBDD 3 3 (M 2 (M 1 (M 0 (L 2) (L 1)) (M 0 (L 2) (L 1)))
          (M 1 (M 0 (L 0) (L 1)) (M 0 (L 0) (L 1))))
```

One can see the average size reduction from BDDs to reduced BDDs with something like:

```
*BDD> sum $ map bdd_size $ nrandom nat2bdd 1000 2000 10 7
320
*BDD> sum $ map bdd_size $ nrandom nat2rbdd 1000 2000 10 7
194
```

Or, one can see the size reductions due to trying all possible variable orders on random BDDs (in the case of 100 random 4 and 5 variable functions):

```
*BDD> sum $ map bdd_size (nrandom (search_bdd max 4) 0 (2^2^4-1) 100 77)
2384
*BDD> sum $ map bdd_size (nrandom (search_bdd min 4) 0 (2^2^4-1) 100 77)
1744
*BDD> sum $ map bdd_size (nrandom (search_bdd max 5) 0 (2^2^5-1) 100 77)
4812
*BDD> sum $ map bdd_size (nrandom (search_bdd min 5) 0 (2^2^5-1) 100 77)
3432
```

Such results are useful in evaluating the pros/cons of various circuit minimization strategies. Needless to say, one might notice the compactness and elegance of a declarative language like Haskell for such ad-hoc tasks, recommending it as a powerful scripting language for electronic design automation tools.

8 Related work

Pairing functions have been used for work on decision problems as early as [8].

BDDs are the dominant boolean function representation in the field of circuit design automation [6].

Besides their uses in circuit design automation, MTBDDs have been used in model-checking and verification of arithmetic circuits [3, 2].

BDDs have also been used in a Genetic Programming context [9] as a representation of evolving individuals subject to crossovers and mutations expressed as structural transformations.

9 Conclusion and Future Work

The surprising connection of bitstring based pairing/unpairing functions and to BDDs came out as 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 [11, 12] and ongoing work on genetic programming algorithms.

We have found such encodings interesting as uniform building blocks for Genetic Programming applications. In a Genetic Programming context [5], 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. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
2. F. Ciesinski, C. Baier, M. Groesser, and D. Parker. Generating compact MTBDD-representations from Probmela specifications. In *Proc. 15th International SPIN Workshop on Model Checking of Software (SPIN'08)*, 2008.
3. M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.
4. D. Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
5. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
6. C. Meinel and T. 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.
7. S. Pigeon. *Contributions à la compression de données.* Ph.d. thesis, Université de Montréal, Montréal, 2001.
8. J. Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950.
9. H. Sakanashi, T. Higuchi, H. Iba, and Y. Kakazu. Evolution of binary decision diagrams for digital circuit design using genetic programming. In T. Higuchi, M. Iwata, and W. Liu, editors, *ICES*, volume 1259 of *Lecture Notes in Computer Science*, pages 470–481. Springer, 1996.
10. C. E. Shannon. *Claude Elwood Shannon: collected papers.* IEEE Press, Piscataway, NJ, USA, 1993.
11. P. Tarau and B. Luderman. A Logic Programming Framework for Combinational Circuit Synthesis. In *23rd International Conference on Logic Programming (ICLP), LNCS 4670*, pages 180–194, Porto, Portugal, Sept. 2007. Springer.
12. P. Tarau and B. Luderman. Exact combinational logic synthesis and non-standard circuit design. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 179–188, New York, NY, USA, 2008. ACM.