Abstract
We describe Haskell implementations of interesting combinatorial generation algorithms with focus on boolean functions and logic circuit representations.

First, a complete exact combinational logic circuit synthesizer is described as a combination of catamorphisms and anamorphisms.

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 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 is provided.


Keywords exact combinational logic synthesis, binary decision diagrams, encodings of boolean functions, pairing/unpairing functions, ranking/unranking functions for BDDs and MTBDDs, declarative combinatorics in Haskell

1. Introduction
This paper is an exploration with functional programming tools of ranking and unranking problems on Binary Decision Diagrams. The practical expressiveness of functional programming languages (in particular Haskell) 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 (Knuth 2006).

The paper is organized as follows:

Sections 2 and 4 overview efficient evaluation of boolean formulae in Haskell using bitvectors represented as arbitrary length integers and Binary Decision Diagrams (BDDs).

Section 3 describes an exact combinational circuit synthesizer.

Section 5 discusses classic pairing and unpairing operations and introduces new pairing/unpairing functions acting directly on bitlists.

Section 6 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 7 describes ranking and unranking functions for BDDs and reduced BDDs.

Section 8 extends our results to Multi-Terminal BDDs.

Sections 9 and 10 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 GHC 6.4.3.

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

\[
\text{if}\_\text{then}\_\text{else}\ 0\ z = z
\]

\[
\text{if}\_\text{then}\_\text{else}\ 1\ y = y
\]

resulting in

\[
\{(x, y, z), \text{if}\_\text{then}\_\text{else}\ x\ y\ z | x \leftarrow [0, 1], y \leftarrow [0, 1], z \leftarrow [0, 1]\}
\]

\[
\{(0, 0, 0), 0\},
\]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]…$5.00
3.1 Encoding the Primary Inputs

First, let us extend the encoding to cover constants 1 and 0, that we will represent as “variables” n and n+1 and encode as vectors of n zeros or n ones (i.e. \(2^n - 1\), passed as the precomputed parameter n to avoid costly recomputation).

\[
\begin{align*}
\text{encode_var} \ m \ n \ k &| k=0 = 0 \\
\text{encode_var} \ m \ n \ k &| k=n+1 = 0 \\
\text{encode_var} \ m \ n \ k & = \text{var}_m \ m \ n \ k
\end{align*}
\]

Next we can precompute all the inputs knowing the number n of primary inputs for the circuit we want to synthesize:

\[
\begin{align*}
\text{init_inputs} \ n & = \\
& 0:m:(\text{map} (\text{encode_var} m n) [0..n-1]) \text{ where } m=\text{bigone} \ n
\end{align*}
\]

Given that inputs have all distinct encodings, we can decode them back - this function will be needed after the circuit is found.

\[
\begin{align*}
\text{decode_var} \ nvars \ v &| v== (\text{bigone} \ nvars) = nvars \\
\text{decode_var} \ nvars \ 0 & = nvars+1 \\
\text{decode_var} \ nvars \ v & = \text{head} \ [k|k->[0..nvars-1],(\text{encode_var} m nvars k)==v] \text{ where } m=\text{bigone} \ nvars
\end{align*}
\]

3. Exact Combinational Circuit Synthesis

A first application of these variable encodings is combinational circuit synthesis, known to be intractable for anything beyond a few input variables. Clearly, a speed-up by a factor proportional to the machine’s wordsize matters in this case.

\[
\begin{align*}
(0,0,1,1), \\
(0,1,0,0), \\
(0,1,1,1), \\
(1,0,0,0), \\
(1,0,1,0), \\
(1,1,0,1), \\
(1,1,1,1)
\end{align*}
\]

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 (Knuth 2006) 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:

\[
x_k = (2^{n} - 1)/(2^{n-k-1} + 1)
\]

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 \(2^n\), \(11\ldots1\):

\[
\begin{align*}
\text{var}_n \ n \ k & = \text{var}_m (\text{bigone} \ n) \ n \ k \\
\text{var}_m \ n \ k & = \text{mask} \ ('\text{div}' (2^n(2^n(n-k-1)))+1) \\
\text{bigone} \ nvars & = 2^n2^n-1
\end{align*}
\]

We have used in \(\text{var}_n \ n\) an adaptation of the efficient bitstring-integer encoding described in the Boolean Evaluation section of (Knuth 2006). 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 \(2^n-1\), corresponding to a column in the truth table containing ones exclusively.
3.2 The Folds and the Unfolds

We are ready now to generate trees with library operations marking internal nodes of type \(F\) and primary inputs marking the leaves of type \(V\).

```haskell
data T a = V a | F a (T a) (T a) deriving (Show, Eq)
generateT lib n = unfoldT lib n 0

unfoldT _ 1 k = [V k]
unfoldT lib n k = [F op l r | i ← [1..n-1],
                           l ← unfoldT lib i k,
                           r ← unfoldT lib (n-i) (k+i),
                           op ← lib]
```

For later use, we will also define the dual fold operation (catamorphism) parameterized by a function \(f\) describing action on the leaves and a function \(g\) describing action on the internal nodes.

```haskell
foldT _ g (V i) = g i
foldT f g (F i l r) = f i (foldT f g l) (foldT f g r)
```

This catamorphism will be used later in the synthesis process for things like boolean evaluation. A simpler use would be to compute the size of a formula as follows:

```haskell
fsize t = foldT f g t where
  g _ = 0
  f _ l r = 1+l+r
```

A first use of foldT will be to decode the constants and variables occurring in the result:

```haskell
decodeV nvars is i = V (decode_var nvars (is!i))
decodeF i x y = F i x y
decodeResult nvars (leafDAG, varMap,_) = foldT decodeF (decodeV nvars varMap) leafDAG
```

The following example shows the action of the decoder:

```haskell
> decodeV 2 (array (0,1) [(0,5),(1,3)]) 0
V 1
> decodeV 2 (array (0,1) [(0,5),(1,3)]) 1
V 0
> decodeResult 2 ((F 1 (V 0) (V 1)),
  (array (0,1) [(0,5),(1,3)]), 4)
  (F 1 (V 1) (V 0))
The following function uses foldT to generate a human readable string representation of the result (using the opname function given in Appendix):

```haskell
showT nvars t = foldT f g t where
  g i =
    if i<nvars
      then "x"++(show i)
    else show (nvars+1-i)
  f i l r = (opname i)++"("++l++","++r++")"
```

3.3 Assembling the Circuit Synthesizer

A Leaf-DAG generalizes an ordered tree by fusing together equal leaves. Leaf equality in our case means sharing a primary input variable or a constant.

In the next function we build candidate Leaf-DAGs by combining two generators: the inputs-to-occurrences generator `generateVarMap` and the expression tree generator `generateT`. Then we compute their bitstring value with a foldT based boolean formula evaluator. The function is parameterized by a library of logic gates `lib`, the number of primary inputs `nvars` and the maximum number of leaves it can use `maxleaves`:

```haskell
buildAndEvalLeafDAG lib nvars maxleaves = [
  (leafDAG, varMap, foldT (opcode mask) (varMap!) leafDAG | k ← [1..maxleaves],
  varMap ← generateVarMap k vs,
  leafDAG ← generateT lib k]
] where
  mask = bigone nvars
  vs = init_inputs nvars
```

We are now ready to test if the candidate matches the specification given by the truth table of \(n\) variables \(ttn\).

```haskell
findFirstGood lib nvars maxleaves ttn =
  head [r | r ← buildAndEvalLeafDAG lib nvars maxleaves,
            testspec ttn r]
  where
    testspec spec (_,_,v) = spec == v
```

The final steps of the circuit synthesizer consist in converting to a human readable form the successful first candidate (guaranteed to be minimal as they have been generated by increasing order of nodes).

```haskell
synthesize_from lib nvars maxleaves ttn =
  decodeResult nvars candidate where
  candidate = firstFindGood lib nvars maxleaves ttn
synthesize_with lib nvars ttn =
  synthesize_from lib nvars (bigone nvars) ttn
```

The following function uses `foldT` to generate a human readable string representation of the result (using the `showT` function given in Appendix):

```haskell
show syn lib nvars ttn =
  (show ttn)++":"++
  (showT nvars (synthesize_with lib nvars ttn))
The following example shows a minimal circuit for the 2 variable boolean function with truth table.

```
> syn [0] 2 6
"6:nand(nand(x0,nand(x1,1)),nand(x1,nand(x0,1)))"
```

The following examples show circuits synthesized for 3 argument function if-the-else in terms of a few different libraries. As this function is the building block of boolean circuit representations like Binary Decision Diagrams, having perfect minimal circuits for it in terms of a given library has clearly practical value. The reader might notice that it is quite unlikely to come up intuitively with some of these synthesized circuits.

```
> syn symops 3 83
"83:nor(nor(x2,x0),nor(x1,nor(x0,0)))"
>syn asymops 3 83
"83:impl(impl(x2,x0),less(x1,impl(x0,0)))"
>syn mixops 3 83
"83:nand(impl(x2,x0),nand(x1,x0))"
> syn [3,4] 3 83
"83:xor(x1,less(xor(x2,x1),x0))"
```

We refer to the Appendix for a few details, related to the bitvector operations on various boolean functions used in the libraries, as well as a few tests.

4. Binary Decision Diagrams

We have seen that Natural Numbers in [0..2^n-1] can be used as representations of truth tables defining n-variable boolean functions. A binary decision diagram (BDD) (Bryant 1986) 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 (Shannon 1993), and is expressed as a decomposition of a function in two cofactors, \( f[x ← 0] \) and \( f[x ← 1] \)

\[
f(x) = (\bar{x} ∧ f[x ← 0]) ∨ (x ∧ f[x ← 1])
\]  

where \( f[x ← a] \) 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:

\[
f(x) = if \ x \ then \ f[x ← 0] \ else \ f[x ← 1]
\]

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.

```
map (syn lib nvars) [0..(bigone nvars)]
```

5. Pairing Functions

**Definition 1.** A pairing function is a bijection \( f : \text{Nat} × \text{Nat} → \text{Nat} \). An unpairing function is a bijection \( g : \text{Nat} → \text{Nat} × \text{Nat} \).

5.1 Classic Pairing Functions

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

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

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

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

We refer to (Cegielski and Richard 2001) for a typical use in the foundations of mathematics and to (Rosenberg 2002) for an extensive study of various pairing functions and their computational properties.

Starting from Cantor’s pairing function

\[
f(x, y) = (x + y) * (x + y + 1)/2 + y
\]

and the Pepis-Kalmar-Robinson function

\[
f(x, y) = 2^x * (2 * y + 1) - 1
\]

bijections from \( \text{Nat} × \text{Nat} \) to \( \text{Nat} \) have been used for various proofs and constructions of mathematical objects (Pepis 1938; Kalmar 1939; Robinson 1950, 1955, 1968; Cegielski and Richard 2001).

5.2 Pairing/Unpairing operations acting directly on bitlists

We will introduce here a pairing function, expressed as simple bitlist transformations. This unusually simple pairing function (that we have found out recently as being the same as the one defined in Steven Pigeon’s PhD thesis on Data Compression (Pigeon 2001), page 114), provides compact representations for various constructs involving ordered pairs.

The function `bitmerge_pair` implements a bijection from \( \text{Nat} × \text{Nat} \) to \( \text{Nat} \) that works by splitting a number’s big endian bitstring representation into odd and even bits, while its inverse `bitmerge_unpair` blends the odd and
even bits back together. The helper functions nat2set and set2nat, given in the Appendix, convert from/to natural numbers to sets of nonzero bit positions.

```
bitmerge_pair (i,j) =
  set2nat ((evens i) ++ (odds j)) where
  evens x = map (2+) (nat2set x)
  odds y = map succ (evens y)
```

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

```
> bitmerge_unpair 2008
(60,26)
-- 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1]
-- 60:[ 0, 1, 1, 1, 1]
-- 26:[ 0, 1, 0, 1, 1 ]
```

**PROPOSITION 2.** The following function equivalences hold:

\[
\text{bitmerge\_pair} \circ \text{bitmerge\_unpair} \equiv \text{id} \quad (9)
\]

\[
\text{bitmerge\_unpair} \circ \text{bitmerge\_pair} \equiv \text{id} \quad (10)
\]

### 6. Pairing Functions and 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.

We represent a BDD in Haskell as a binary tree BT with constants 0 and 1 as leaves, marked with the function symbol C. Internal nodes representing if-then-else decisions, marked with D, are controlled by variables, ordered identically in each branch, as first arguments of D. 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.

```
data BT a = C a | D a (BT a) (BT a)
  deriving (Eq, Show)
```

The constructor BDD wraps together the number of variables of a binary decision diagram and the binary tree representation it.

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

The following functions apply bitmerge_unpair 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 will represent using the BDD data type.

```
-- n-number of variables, tt-a truth table
plain_bdd n t = BDD n tt where
  bt = if tt<max then shf bitmerge_unpair n tt
  else error
  (*"plain_bdd: last arg "++ (show tt)++
   " should be < " ++ (show max))
  where max = 2^2^n

-- recurses to depth n, splitting tt into pairs
shf f n tt | n<1 = C tt
shf f n tt = D k (ash f k tt1) (ash f k tt2) where
  k = pred n
  (tt1,tt2)=f tt
```

The following examples show the results returned by plain_bdd for all \(2^n\) truth tables associated to \(n\) variables for \(n = 2\), with help from printing function print_plain given in Appendix.

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

### 6.1 Reducing the BDDs

The function bdd_reduce reduces a BDD by collapsing identical left and right subtrees, and the function bdd associates this reduced form to \(n \in \mathbb{N}\).

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

```
bdd n = bdd_reduce . plain_bdd n
```

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 \(n=2\), with help from printing function print_reduced given in Appendix.

```
> print_reduced 2
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 (D 1 (C 0) (C 1) (C 1))
BDD 2 (C 1)
```
6.2 From BDDs to Natural Numbers

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

```
plain_inverse_bdd (BDD _ bt) =
  rshf bitmerge_pair bt
rshf rf (C tt) = tt
rshf rf (D _ l r) = rf ((rshf rf l),(rshf rf r))
```

```
plain_bdd 3 42
BDD 3
(D 2
  (D 1 (D 0 (C 0) (C 0))
   (D 0 (C 0) (C 0)))
  (D 0 (C 1) (C 1)))
) ev it
42
bddd 3 42
BDD 3
(D 2
  (C 0)
  (D 1
    (C 1)
    (D 0 (C 1) (C 0))))
) ev it
42
```

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

6.3 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 function `ev` describes the BDD evaluator:

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

```
eval_with_mask m _ (C c) = eval_constant m c
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)
eval_constant _ 0 = 0
eval_constant m 1 = m
```

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
```

We will use `ite_` as the basic building block for implementing a boolean evaluator for BDDs.

6.4 The Equivalence

A surprising result is that boolean evaluation and structural transformation with repeated application of pairing produce the same result, i.e. the function `ev` also acts as an inverse of `bdd` and `plain_bdd`.

As the following example shows, boolean evaluation `ev` faithfully emulates `plain_inverse_bdd`, on both plain and reduced BDDs.

7. 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 \( \text{Nat} \) as BDDs of larger and larger sizes are enumerated.

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

\[
\text{bsum 0 = 0} \\
\text{bsum n | n>0 = bsum1 (n-1)} \\
\text{bsum1 0 = 2} \\
\text{bsum1 n | n>0 = bsum1 (n-1)+ 2^{2^n}}
\]

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

\[
\text{bsums = map bsum [0..]} \\
\text{genericTake 7 bsums} \\
\text{[0,2,6,22,278,65814,4295033110]}
\]

What we are really interested into, is decomposing \( n \) into the distance \( n-m \) to the last \( \text{bsum m} \) smaller than \( n \), and the index that generates the sum, \( k \).

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

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 plain_bdd and bdd.

\[
\text{nat2plain_bdd n = plain_bdd k n_m} \\
\text{where (k,n_m)=to_bsum n} \\
\text{nat2bdd n = bdd k n_m} \\
\text{where (k,n_m)=to_bsum n}
\]

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

\[
\text{plain_bdd2nat bdd@(# (bsum \ nv)+(# (plain_inverse_bdd bdd)) bdd)} \\
\text{bdd2nat bdd@(# (bsum \ nv)+(ev bdd))}
\]

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

\[
\text{nat2plain_bdd 42} \\
\text{BDD 3} \\
\text{(D 2} \\
\text{(D 1} \\
\text{(D 0 (C 0) (C 1))} \\
\text{(D 0 (C 1) (C 0)))} \\
\text{(D 1 (D 0 (C 0) (C 0))} \\
\text{(D 0 (C 0) (C 0)))))}
\]

The same applies to nat2bdd and its inverse bdd2nat.

\[
\text{bdds = map nat2bdd [0..]} \\
\text{genericTake 6 bdds} \\
\text{[BDD 0 (C 0), BDD 0 (C 1), BDD 1 (D 0 (C 0) (C 1))]} \\
\text{BDD 1 (D 0 (C 1) (C 0)), BDD 1 (C 1)}
\]

8. Multi-Terminal Binary Decision Diagrams (MTBDD)

MTBDDs (Fujita et al. 1997; Ciesinski et al. 2008) 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 \( \text{MTBDDs} \) that can be extended to ranking/unranking functions, in a way similar to \( \text{BDDs} \) as shown in section 7.

Our \( \text{MTBDD} \) data type is a binary tree like the one used for \( \text{BDDs} \), parameterized by two integers \( m \) and \( n \), indicating that an MTBDD represents a function from \( [0..n-1] \) to \( [0..2^m-1] \), or equivalently, an \( n \)-input/\( m \)-output boolean function.

\[
\text{data MTBDD a = MTBDD a (BT a) deriving (Show,Eq)}
\]

The function to_mtbdd creates, from a natural number \( \text{tt} \) representing a truth table, an MTBDD representing functions of type \( N \rightarrow M \) 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 bitmerge_unpair recursively up to depth n,
where leaves in range [0..mlimit − 1] are created.

convert_mtbdd_ mlimit n tt| (n<1)&&(tt<mlimit) = C tt
convert_mtbdd_ mlimit n tt = (D k l r) where
  (x,y)=bitmerge_unpair 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 (C tt)| (n<1)&&(tt<mlimit)=tt
from_mtbdd_ mlimit n (D _ l r) = tt where
  k=pred n
  x=from_mtbdd_ mlimit k l
  y=from_mtbdd_ mlimit k r
  tt=bitmerge_pair (x,y)

The following examples show that to_mtbdd and from_mtbdd
are indeed inverses values in [0..2^n − 1] × [0..2^m − 1].

>to_mtbdd 3 3 2008
MTBDD 3 3
(D 2
   (D 1
     (D 0 (C) (C 1))
     (D 0 (C) (C 1)))
   (D 1
     (D 0 (C) (C 0))
     (D 0 (C) (C 1))))

>from_mtbdd it
2008

>mprint (to_mtbdd 2 2) [0..3]
MTBDD 2 2
(D 1
   (D 0 (C) (C 0))
   (D 0 (C) (C 0)))

>from_mtbdd (MTBDD 2 2) [0..3]
MTBDD 2 2
(D 1
   (D 0 (C) (C 0))
   (D 0 (C) (C 0)))

9. Related work

Pairing functions have been used for work on decision problems as early as (Pepis 1938; Kalmar 1939; Robinson 1950).
BDDs are the dominant boolean function representation in the field of circuit design automation (Meinel and Theobald 1999; Drechsler et al. 2004).

Besides their uses in circuit design automation, MTBDDs have been used in model-checking and verification of arithmetic circuits (Fujita et al. 1997; Ciesinski et al. 2008).

BDDs have also been used in a Genetic Programming context (Sakanashi et al. 1996; Rothlauf et al. 2006; Chen et al. 2004) as a representation of evolving individuals subject to crossovers and mutations expressed as structural transformations.

10. Conclusion and Future Work

Our new pairing/unpairing functions and their surprising connection to BDDs, have been 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 (Tarau and Luderman 2008). We have found them also interesting as uniform blocks for Genetic Programming applications. In a Genetic Programming context (Koza 1992; Poli et al.), the bijections between bitvectors/natural numbers on one side, and tree/ 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


Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-terminal binary decision diagrams: An efficient data struc-


---

## Appendix

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

### Bitvector Boolean Operation Definitions

```haskell
data Nat = Integer

{- operation codes -}
opcode m 0 = nand_ m
opcode m 1 = nor_ m
opcode m 2 = impl_ m
opcode m 3 = less_ m
opcode _ 4 = xor
opcode _ n = error (“unexpected opcode:”++(show n))
```

### Boolean Operation Encodings and Names

```haskell
-- operation names
opname 0 = “nand”
opname 1 = “nor”
opname 2 = “impl”
opname 3 = “less”
opname 4 = “xor”
opname n = error (“no such opcode:”++(show n))
```

### A Few Interesting Libraries

**mixops** = [0,2]
**symops** = [0,1]
**asympms** = [2,3]

### Tests for the Circuit Synthesizer

```haskell
t0 = findFirstGood symops 3 8 71
t1 = syn asymops 3 71
t2 = mapM_ print (synall mixops 2)
t3 = syn asymops 3 83 -- ite

t4 = syn symops 3 83
t5 = syn [0..4] 3 83 -- ite with all ops
-- x xor y xor z -- cpu intensive

t6 = syn asymops 3 105
```

### Bit crunching functions

This function splits a natural number in a set of natural numbers indicating the positions of its 1 bits in its right to left binary representation.

```haskell
nat2set n = nat2exps n 0 where
nat2exps 0 _ = []
nat2exps n x =
  if (even n) then xs else (x:xs) where
  xs=nat2exps (div n 2) (succ x)
```
This function aggregates a set of natural numbers indicating positions of 1 bits into the corresponding natural number.

\[
\text{set2nat } ns = \text{sum } (\text{map } (2^\cdot) \cdot ns)
\]

**I/O functions**

These functions print out the BDDs of all the \(2^{2^k}\) truth tables associated to \(k\) variables.

\[
\begin{align*}
\text{print\_plain } k = \text{mapM\_} \\
& (\text{print } . \ (\text{plain\_bdd } k)) \ [0..(\text{bigone } k)] \\
\text{print\_reduced } k = \text{mapM\_} \\
& (\text{print } . \ (\text{bdd } k)) \ [0..(\text{bigone } k)]
\end{align*}
\]

This function applies \(f\) to a list of objects and prints the results on successive lines.

\[
\text{mprint } f = (\text{mapM\_ } \text{print}) \cdot \ (\text{map } f)
\]