Declarative Combinatorics: Boolean Functions, Circuit Synthesis and BDDs in Haskell

– unpublished draft –

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cs.unt.edu

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.

The paper is organized as a self-contained literate Haskell program, available at `http://logic.csci.unt.edu/tarau/research/2008/fBDD.zip`.

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`

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

resulting in

```
> [([x,y,z],if_then_else x y z)|
      x←[0,1],y←[0,1],z←[0,1]]
  [([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 (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:*

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

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`:

```
-- the k-th, out of n bitvector boolean variables
var_n n k = var_mn (bigone n) n k

-- the k-th, out of n boolean variables w.r.t mask
var_mn mask n k = mask `div` (2^(2^(n-k-1))+1)

-- represents constant 1 as 11...1
bigone nvars = 2^2^nvars - 1
```

We have used in `var_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 $2^{2^n} - 1$, corresponding to a column in the truth table containing ones exclusively.

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.

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. $2^{2^n} - 1$, passed as the precomputed parameter `m` to avoid costly recomputation).

```
encode_var m n k | k==n = m
encode_var m n k | k==n+1 = 0
encode_var m n k = var_mn m n k
```

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

```
init_inputs n =
  0:m:(map (encode_var m n) [0..n-1]) where
  m=bigone n

>init_inputs 3
  [0,15,3,5]
>init_inputs 3
  [0,255,15,51,85]
```

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

```
decode_var nvars v | v==(bigone nvars) = nvars
decode_var nvars 0 = nvars+1
decode_var nvars v = head
  [k|k<-[0..nvars-1],(encode_var m nvars k)==v]
    where m=bigone nvars

>map (decode_var 2) (init_inputs 2)
  [3,2,0,1]
>map (decode_var 3) (init_inputs 3)
  [4,3,0,1,2]
```

We can now connect the inputs to their future occurrences as leaves in the tree representing the circuit. This means simply finding all the functions from the set of inputs to the set of occurrences, represented as a list (with possibly repeated) values of the inputs.

```
bindings 0 us = [[]]
bindings n us =
  [zs|ys<-bindings (n-1) us,zs<-map (:ys) us]

>bindings 2 [0,3,5]
  [[0,0],[3,0],[5,0],[0,3],[3,3],
   [5,3],[0,5],[3,5],[5,5]]
```

For fast lookup, we place the precomputed value combinations in a list of arrays.

```
generateVarMap occs vs  =
  map (listArray (0,occs-1)) (bindings occs vs)

>generateVarMap 2 [3,5]
  [array (0,1) [(0,3),(1,3)],
   array (0,1) [(0,5),(1,3)],
   array (0,1) [(0,3),(1,5)],
   array (0,1) [(0,5),(1,5)]]
```

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.

```
data T a = V a | F a (T a) (T a) deriving (Show, Eq)
```

Generating all trees is a variant of an `unfold` operation (*anamorphism*).

```
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.

```
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:

```
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:

```
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:

```
>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):

```
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++")"

> showT 2 (F 4 (V 0) (F 1 (V 1) (V 0)))
  "xor(x0,nor(x1,x0))"
```

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`:

```
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`.

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

> findFirstGood [1] 2 8 1
  (F 1 (F 1 (V 0) (V 1)) (F 1 (V 2) (V 3)),
    array (0,3) [(0,5),(1,0),(2,3),(3,0)],1)
```

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).

```
synthesize_from lib nvars maxleaves ttn =
  decodeResult nvars candidate where
  candidate=findFirstGood lib nvars maxleaves ttn

synthesize_with lib nvars ttn =
  synthesize_from lib nvars (bigone nvars) ttn

-- synthesizes an shows a circuit
syn lib nvars ttn =
  (show ttn)++":"++
  (showT nvars (synthesize_with lib nvars ttn))
```

```
-- shows all circuits synthesized
-- for functions with nvars inputs
synall lib nvars =
  map (syn lib nvars) [0..(bigone nvars)]
```

The following example shows a minimal circuit for the 2 variable boolean function with truth table 6 (xor) in terms of the library with opcodes in [0] i.e. containing only the operator nand. Note that codes for functions represent their truth tables i.e. 6 stands for [0,1,1,0].

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

The following examples show circuits synthetized 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 $[0..2^{2^n}-1]$ 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*, $f[x \leftarrow 0]$ 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:

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

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 cofactors's truth tables) to a natural number (the function's truth table), i.e. it works as a pairing function*.*

5. Pairing Functions

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

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

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

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

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

We refer to (Cégielski 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

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

and the Pepis-Kalmar-Robinson function

$$f(x, y) = 2^x * (2 * y + 1) - 1 \tag{8}$$

bijections from $Nat \times Nat$ to Nat have been used for various proofs and constructions of mathematical objects (Pepis 1938; Kalmar 1939; Robinson 1950, 1955, 1968; Cégielski 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 in 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 $Nat \times Nat$ to 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)

bitmerge_unpair n = (f xs,f ys) where
  (xs,ys) = partition even (nat2set n)
  f = set2nat . (map (`div` 2))
```

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

PROPOSITION 2. *The following function equivalences hold:*

$$bitmerge\_pair \circ bitmerge\_unpair \equiv id \tag{9}$$

$$bitmerge\_unpair \circ bitmerge\_pair \equiv id \tag{10}$$

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` decision points, 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 tt = BDD n bt 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 (shf f k tt1) (shf f k tt2) where
  k=pred n
  (tt1,tt2)=f tt
```

The following examples show the results returned by `plain_bdd` for all 2^{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 $n \in Nat$.

```
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 (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_pair`. 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 1 (D 0 (C 1) (C 1))
           (D 0 (C 1) (C 0))))

plain_inverse_bdd 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.*

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

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

PROPOSITION 3. *The complete binary tree of depth n, obtained by recursive applications of* `bitmerge_unpair` *on a truth table tt computes an (unreduced) BDD, that, when evaluated, returns the truth table, i.e.:*

$$plain\_inverse\_bdd\ (plain\_bdd\ n\ tt)) \equiv id \tag{11}$$

$$ev\ n\ (plain\_bdd\ n\ tt)) \equiv id \tag{12}$$

Moreover, `ev` *also acts as a left inverse of* `bdd`*, i.e.*

$$ev\ n\ (bdd\ n\ tt)) \equiv id \tag{13}$$

Proof sketch: The function `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 (Shannon 1993) 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_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. 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 (Bryant 1986).

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

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 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.

```
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[1]:

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

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

What we are really interested into, 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 `plain_bdd` and `bdd`.

```
nat2plain_bdd n = plain_bdd k n_m
  where (k,n_m)=to_bsum n

nat2bdd n = bdd 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.

```
plain_bdd2nat bdd@(BDD nv _) =
  (bsum nv)+(plain_inverse_bdd bdd)

bdd2nat bdd@(BDD nv _) = (bsum nv)+(ev bdd)
```

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

```
>nat2plain_bdd 42
  BDD 3
     (D 2
       (D 1
         (D 0 (C 0) (C 1))
         (D 0 (C 1) (C 0)))
       (D 1 (D 0 (C 0) (C 0))
            (D 0 (C 0) (C 0))))
>plain_bdd2nat it
  42
```

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

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

```
>nat2bdd 42
  BDD 3
     (D 2
       (D 1
         (D 0 (C 0) (C 1))
         (D 0 (C 1) (C 0)))
       (C 0))
>bdd2nat it
  42
```

We can now generate infinite streams of BDDs as follows:

```
plain_bdds = map nat2plain_bdd [0..]

bdds = map nat2bdd [0..]

>genericTake 4 plain_bdds
  [
    BDD 0 (C 0),
    BDD 0 (C 1),
    BDD 1 (D 0 (C 0) (C 0)),
    BDD 1 (D 0 (C 1) (C 0))
  ]
genericTake 6 bdds
  [
   BDD 0 (C 0),
   BDD 0 (C 1),
   BDD 1 (C 0),
   BDD 1 (D 0 (C 1) (C 0)),
   BDD 1 (D 0 (C 0) (C 1)),
   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 $MTBDDs$ that can be extended to ranking/unranking functions, in a way similar to $BDDs$ as shown in section 7.

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 MTBDD a = MTBDD a a (BT 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 bitmerge_unpair recursively up to depth n, where leaves in range $[0..mlimit-1]$ are created.

```
to_mtbdd_ mlimit n tt|(n<1)&&(tt<mlimit) = C tt
to_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 $[0..2^n-1] \times [0..2^m-1]$.

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

>from_mtbdd it
2008

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

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

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.

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.

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.

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.

Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

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. ISSN 0022-4812.

Donald Knuth. The Art of Computer Programming, Volume 4, draft, 2006. http://www-cs-faculty.stanford.edu/∼/knuth/taocp.html.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

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.

Jozef Pepis. Ein verfahren der mathematischen logik. *The Journal of Symbolic Logic*, 3(2):61–76, jun 1938. ISSN 0022-4812.

Stephen Pigeon. *Contributions à la compression de données.* Ph.d. thesis, Université de Montréal, Montréal, 2001.

Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. *A Field Guide to Genetic Programming*. URL `http://www.gp-field-guide.org.uk`. e-book.

Julia Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950. ISSN 0002-9939.

Julia Robinson. A note on primitive recursive functions. *Proceedings of the American Mathematical Society*, 6(4):667–670, aug 1955. ISSN 0002-9939.

Julia Robinson. Finite generation of recursively enumerable sets. *Proceedings of the American Mathematical Society*, 19(6): 1480–1486, dec 1968. ISSN 0002-9939.

Arnold L. Rosenberg. Efficient pairing functions - and why you should care. In *IPDPS*. IEEE Computer Society, 2002. ISBN 0-7695-1573-8.

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, EvoHOT, EvoIASP, EvoINTERACTION, EvoMUSART, and EvoSTOC, Budapest, Hungary, April 10-12, 2006, Proceedings*, volume 3907 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-33237-5.

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. ISBN 3-540-63173-9.

Claude E. Shannon. *Claude Elwood Shannon: collected papers*. IEEE Press, Piscataway, NJ, USA, 1993. ISBN 0-7803-0434-9.

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.

Appendix

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

Bitvector Boolean Operation Definitions

```
type Nat=Integer

nand_ :: Nat→Nat→Nat→Nat
nor_ :: Nat→Nat→Nat→Nat
impl_ :: Nat→Nat→Nat→Nat
less_ :: Nat→Nat→Nat→Nat

nand_ mask x y = mask .&. (complement (x .&. y))
nor_ mask x y = mask .&. (complement (x .|. y))
impl_ mask x y = (mask .&. (complement x)) .|. y
less_ _ x y = x .&. (complement y)
```

Boolean Operation Encodings and Names

```
-- 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))

-- 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]
asymops = [2,3]
```

Tests for the Circuit Synthesizer

```
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.

```
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.

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

I/O functions

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

```
print_plain k = mapM_
  (print . (plain_bdd k)) [0..(bigone k)]
print_reduced k = mapM_
  (print . (bdd k)) [0..(bigone k)]
```

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

```
mprint f = (mapM_ print) . (map f)
```