

Experimental Mathematics in Haskell: on Pairing/Unpairing Functions and Boolean Evaluation

Paul Tarau¹ Brenda Luderman²

University of North Texas¹

ACES CAD²

TFP 2010, May 2010

Outline

- by using **pairing functions** (bijections $N \times N \rightarrow N$) on natural number representations of truth tables, we derive an encoding for Ordered Binary Decision Trees (OBDTs)
- **boolean evaluation of a OBDT mimics its structural conversion to a natural number through recursive application of a matching pairing function**
- also: we derive *ranking* and *unranking* functions for OBDTs, generalize to arbitrary variable order and multi-terminal OBDTs
- literate Haskell program, code at `http://logic.csci.unt.edu/tarau/research/2009/fOBDT.hs`

Pairing functions

“pairing function”: a bijection $J : \text{Nat} \times \text{Nat} \rightarrow \text{Nat}$

$$K(J(x, y)) = x,$$

$$L(J(x, y)) = y$$

$$J(K(z), L(z)) = z$$

examples:

- Cantor's pairing function: geometrically inspired (100++ years ago - possibly also known to Cauchy - early 19-th century)
- the Peis-Kalmar Pairing Function (1938):

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

a pairing/unpairing function based on boolean operations

```
type Nat = Integer
```

```
type Nat2 = (Nat,Nat)
```

```
bitpair :: Nat2 → Nat
```

```
bitunpair :: Nat → Nat2
```

```
bitpair (x,y) = inflate x .|. inflate' y
```

```
bitunpair z = (deflate z, deflate' z)
```

```
inflate : abcde-> a0b0c0d0e
```

```
inflate' : abcde-> 0a0b0c0d0e
```

inflate/deflate in terms of boolean operations

```
inflate 0 = 0
```

```
inflate n = (twice . twice . inflate . half) n .|. parity n
```

```
deflate 0 = 0
```

```
deflate n = (twice . deflate . half . half) n .|. parity n
```

```
deflate' = half . deflate . twice
```

```
inflate' = twice . inflate
```

```
half n = shiftR n 1 :: Nat
```

```
twice n = shiftL n 1 :: Nat
```

```
parity n = n .&. 1 :: Nat
```

bitpair/bitunpair: an example

the transformation of the bitlists – with bitstrings aligned:

```
*OBDT> bitunpair 2012  
(62,26)
```

```
-- 2012:[0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1]  
--   62:[0,   1,   1,   1,   1,   1]  
--   26:[ 0,   1,   0,   1,   1 ]
```

Note that we represent numbers with bits in reverse order.

Also, some simple algebraic properties:

```
bitpair (x,0) =   inflate x  
bitpair (0,x) = 2 * (inflate x)  
bitpair (x,x) = 3 * (inflate x)
```

Visualizing the pairing/unpairing functions

- Given that unpairing functions are bijections from $N \rightarrow N \times N$ they will progressively cover all points having natural number coordinates in the plan.
- Pairing can be seen as a function $z=f(x,y)$, thus it can be displayed as a 3D surface.
- Recursive application – the unpairing tree can be represented as a DAG – by merging shared nodes.

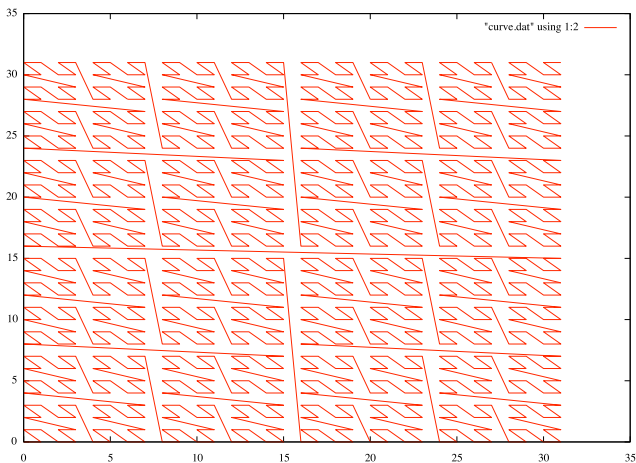


Figure: 2D curve connecting values of `bitunpair n` for $n \in [0..2^{10} - 1]$

"curve.dat" using 1:2:3

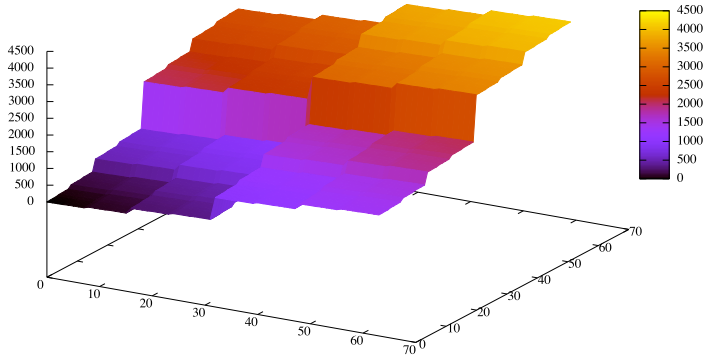


Figure: Values of bitpair $x y$ with x, y in $[0..63]$

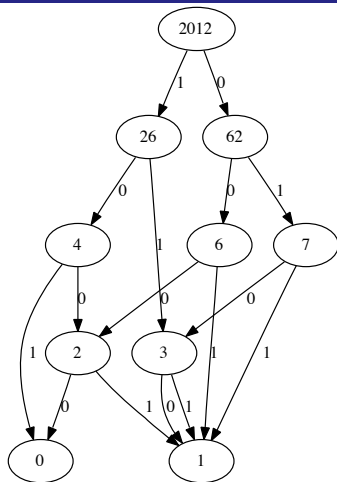


Figure: Graph obtained by recursive application of `bitunpair` for 2012

Unpairing Trees: seen as OBDTs

```
data OBDT a = OBDT a (BT a)
data BT a = B0 | B1 | D a (BT a) (BT a)
```

```
unfold_obdt :: Nat2 → OBDT Nat
unfold_obdt (n,tt) | tt < 2^2^n = OBDT n bt where
  bt = unfold_with bitunpair n tt
```

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

Folding back Trees to Natural Numbers

```
fold_obdt :: OBDT Nat → Nat2
fold_obdt (OBDT n bt) = (n, fold_with bitpair bt) where
  fold_with rf B0 = 0
  fold_with rf B1 = 1
  fold_with rf (D _ l r) = rf (fold_with rf l, fold_with rf r)
```

This is a purely structural operation - no boolean evaluation involved!

```
*OBDT>unfold_obdt (3,42)
  OBDT 3 (D 2 (D 1 (D 0 B0 B0)
              (D 0 B0 B0) )
        (D 1 (D 0 B1 B1)
              (D 0 B1 B0) ))
```

```
*OBDT>fold_obdt it
  42
```

Truth tables as natural numbers

```
x y z → f x y z
(0, [0, 0, 0]) → 0
(1, [0, 0, 1]) → 1
(2, [0, 1, 0]) → 0
(3, [0, 1, 1]) → 1
(4, [1, 0, 0]) → 0
(5, [1, 0, 1]) → 1
(6, [1, 1, 0]) → 1
(7, [1, 1, 1]) → 0
      ::
{1, 3, 5, 6} :: 106 = 2^1 + 2^3 + 2^5 + 2^6 = 2 + 8 + 32 + 64
01010110 (right to left)
```

Computing all Values of a Boolean Function with Bitvector Operations (Knuth 2009 - Bitwise Tricks and Techniques)

Proposition

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

$$v_k = (2^{2^n} - 1) / (2^{2^{n-k-1}} + 1) \quad (2)$$

For instance, if $n = 2$, the formula computes $v_0 = 3 = [0, 0, 1, 1]$ and $v_1 = 5 = [0, 1, 0, 1]$.

we can express v_n with boolean operations + bitpair!

The function v_n , working with arbitrary length bitstrings are used to evaluate the $[0..n-1]$ *projection variables* v_k representing encodings of columns of a truth table, while v_m maps the constant 1 to the bitstring of length 2^n , $111\dots 1$:

$$v_n \ 1 \ 0 = 1$$

$$v_n \ n \ q \mid q = \text{pred } n = \text{bitpair } (v_n \ n \ 0, 0)$$

$$v_n \ n \ q \mid q \geq 0 \ \&\& \ q < n' = \text{bitpair } (q', q')$$
 where

$$n' = \text{pred } n$$

$$q' = v_n \ n' \ q$$

$$v_m \ n = v_n \ (\text{succ } n) \ 0$$

OBDTs

- an ordered binary decision diagram (OBDT) is a rooted ordered binary tree obtained from a boolean function, by assigning its variables, one at a time, to 0 (left branch) and 1 (right branch).
- deriving a OBDT of a boolean function f : repeated Shannon expansion

$$f(x) = (\bar{x} \wedge f[x \leftarrow 0]) \vee (x \wedge f[x \leftarrow 1]) \quad (3)$$

with a more familiar notation:

$$f(x) = \text{if } x \text{ then } f[x \leftarrow 1] \text{ else } f[x \leftarrow 0] \quad (4)$$

Boolean Evaluation of OBDDs

- OBDDs \Rightarrow ROBDDs by sharing nodes + dropping identical branches
- `fold_obdt` might give a different result as it computes different pairing operations!
- **however, we obtain a truth table if we evaluate the OBDD tree as a boolean function – it would be nice if we could relate this to the original truth table from which we unfolded the OBDD!**

```
eval_obdt (OBDD n bt) = eval_with_mask (vm n) n bt where
  eval_with_mask m _ B0 = 0
  eval_with_mask m _ B1 = m
  eval_with_mask m n (D x l r) = ite_ (vn n x)
    (eval_with_mask m n l) (eval_with_mask m n r)
```

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

The Equivalence of boolean evaluation and recursive pairing

SURPRISINGLY, it turns out that:

- boolean evaluation `eval_obdt` faithfully emulates `fold_obdt`
- and it also works on reduced OBDTs, ROBDDs, BDDs as they **represent the same boolean formula**

```
*OBDT> unfold_obdt (3, 42)
OBDT 3 (D 2 (D 1 (D 0 B0 B0) (D 0 B0 B0))
        (D 1 (D 0 B1 B1) (D 0 B1 B0)))
*OBDT> eval_obdt it
42
```

The Equivalence

Proposition

The complete binary tree of depth n , obtained by recursive applications of `bitunpair` on a truth table computes an (unreduced) OBDT, that, when evaluated (reduced or not) returns the truth table, i.e.

$$\text{fold_obdt} \circ \text{unfold_obdt} \equiv \text{id} \quad (5)$$

$$\text{eval_obdt} \circ \text{unfold_obdt} \equiv \text{id} \quad (6)$$

Ranking and Unranking of OBDTs

Ranking/unranking: bijection to/from *Nat*

- one more step is needed to extend the mapping between *OBDTs* with N variables to a bijective mapping from/to *Nat*:
- we will have to “shift toward infinity” the starting point of each new block of OBDTs in *Nat* as OBDTs of larger and larger sizes are enumerated
- we need to know by how much - so we compute the sum of the counts of boolean functions with up to N variables.

Ranking/unranking of OBDTs

$\text{bsum } 0 = 0$

$\text{bsum } n \mid n > 0 = \text{bsum1 } (n-1)$ where

$\text{bsum1 } 0 = 2$

$\text{bsum1 } n \mid n > 0 = \text{bsum1 } (n-1) + 2^{2^n}$

```
*OBDT> map bsum [0..6]
```

```
[0, 2, 6, 22, 278, 65814, 4295033110]
```

A060803 in the Online Encyclopedia of Integer Sequences

```
*OBDT> nat2bdd 42
```

```
OBDT 3 (D 2 (D 1 (D 0 B0 B1) (D 0 B1 B0))  
        (D 1 (D 0 B0 B0) (D 0 B0 B0)))
```

```
*OBDT> bdd2nat it
```

```
42
```

Generalizations

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_obdt vs tt | 0 ≤ tt && tt ≤ m =  
  OBDT n (to_obdt_mn vs tt m n) where  
    n = genericLength vs  
    m = vm n
```

```
to_obdt_mn []      0 _ _ = B0
```

```
to_obdt_mn []      _ _ _ = B1
```

```
to_obdt_mn (v:vs) tt m n = D v l r where
```

```
  cond = vn n v
```

```
  f0 = (m `xor` cond) .&. tt
```

```
  f1 = cond .&. tt
```

```
  l = to_obdt_mn vs f1 m n
```



Applications

- possible applications to (RO)BDDs: circuit synthesis/verification
- BDD minimization using our generalization to arbitrary variable order
- combinatorial enumeration and random generation of circuits
- succinct data representations derived from our OBDT encodings
- an interesting “mutation”: use integers/bitstrings as genotypes, OBDTs as phenotypes in Genetic Algorithms

Conclusion

- **NEW:** the connection of pairing/unpairing functions and boolean evaluation of OBDTs
- synergy between concepts borrowed from *foundation of mathematics, combinatorics, boolean logic, circuits*
- **Haskell as sandbox for experimental mathematics: type inference helps clarifying complex dependencies between concepts quite nicely - moving to a functional subset of Mathematica, after that, is routine.**