Ranking and Unranking Functions for Ordered Decision Trees with Applications to Circuit Synthesis

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. In the form of a self-contained literate Haskell program (available at `http://logic.csci.unt.edu/tarau/research/2009/fOBDT.hs`), we explore natural number encodings of boolean functions and logic circuit representations.
Using *pairing* and *unpairing* functions on natural number representations of truth tables, we derive an encoding for Ordered Binary Decision Trees (OBDTs) with the unique property that its boolean evaluation faithfully mimics its structural conversion to a natural number through recursive application of an unpairing function.
We then use this result to derive *ranking* and *unranking* functions for OBDTs and reduced OBDTs.
Finally, a generalization of the encoding techniques to Multi-Terminal OBDTs and an application to circuit synthesis are described.
Keywords: *ordered binary decision trees (OBDTs), encodings of boolean functions, pairing/unpairing functions, ranking/unranking functions for OBDTs, computational mathematics in Haskell*

1 Introduction

This paper is an exploration with functional programming tools of the relation between *pairing/unpairing* and *ranking/unranking* functions and Ordered Binary Decision Trees, as well as their connection to boolean evaluation.

The paper is organized as follows:

Section 2 overviews Ordered Binary Decision Trees (OBDTs). Section 3 introduces pairing/unpairing functions acting directly on bitlists. Section 4 introduces a novel OBDT encoding (based on *unpairing* functions) and discusses the surprising equivalence between boolean evaluation of OBDTs and the inverse of our encoding. Section 5 describes *ranking* and *unranking* functions for OBDTs and reduced OBDTs. Section 6 extends our techniques to OBDTs with arbitrary

variable order and Multi-Terminal OBDTs and applications to OBDT-based circuit minimization and generation of random test data. Sections 7 and 8 discuss related work, future work and conclusions.

2 Ordered Binary Decision Trees

Natural numbers in $[0..2^{2^n}-1]$ can be used as representations of truth tables defining n-variable boolean functions. An ordered binary decision tree (OBDT) 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). When a directed acyclic graph is used to share subtrees in an OBDT, they lead to Binary Decision Diagrams (BDDs) which in turn lead to Reduced Ordered Binary Decision Diagrams (ROBDDs) providing a compressed canonical form directly supporting various boolean operations [1].

The construction is known as Shannon expansion [2], and is expressed as a decomposition of a function in two *cofactors*, $f[x \leftarrow 0]$ and $f[x \leftarrow 1]$ where $f[x \leftarrow a]$ is computed by uniformly substituting a for x in f. Using the familiar boolean `if-the-else` function the Shannon expansion can be described as:

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

Alternatively, we observe that the Shannon expansion can be directly derived from a 2^n size truth table, in terms of bitstring operations on integer 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.

3 Pairing/Unpairing

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

We will now introduce an unusually simple pairing function (also mentioned in [3], p.142 and similar to Misra's *zip* function [4] in the *powerlist* algebra). We start with a few imports needed across the paper for bitstring, list operations and random number generators.

```
module OBDT where
import Data.List
import Data.Bits
import Random
```

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
type Nat2 = (Nat,Nat)

bitpair ::  Nat2 → Nat
bitunpair :: Nat→Nat2

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

They can be expressed in terms of simple bitstring operations by first defining operations that insert 0s after every binary digit (`inflate`) and then operations that select every odd/even bit in a bitstring (`deflate`).

```
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' z = half (deflate (twice z))
inflate' = twice . inflate
```

which in turn are expressed in terms of bitshifts and boolean operations.

```
half n = shiftR n 1 :: Nat
twice n = shiftL n 1 :: Nat
parity n = n .&. 1 :: Nat
```

Fig. 1 shows the digraph obtained by applying the `bitunpair` operation recursively, starting with `2010`, with labels 0 and 1 on the edges indicating the order in the resulting pair at each step.

4 Pairing Functions and Encodings of Ordered Binary Decision Trees

We show in this section that an Ordered Binary Decision Tree ($OBDT$) 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 $OBDT$ 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 `OBDT` wraps together a tree of type `BT` and the number of logic variables occurring in it.

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

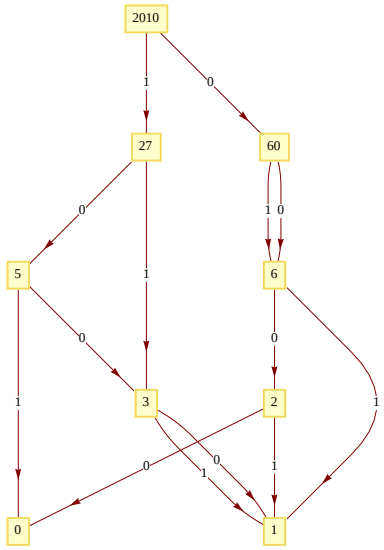


Fig. 1: Recursive application of bitunpair

4.1 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 `OBDT` data type.

```
unfold_obdt :: Nat2 → OBDT Nat
unfold_obdt (n,tt) = OBDT n bt where
  bt=if tt<max then shf bitunpair n tt
     else error ("unfold_obdt: 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
    (tt1,tt2)=f tt
```

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

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

```
 ...
 OBDT 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 OBDTs associated to truth tables.

4.2 Folding binary trees to natural numbers with `bitpair`

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

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

```
*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
```

4.3 Evaluation of Boolean Functions in Terms of Pairing Functions

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. We will introduce here an alternate representation, expressed directly in terms of pairing functions, uses integer encodings of 2^n bits for each boolean variable $v_0, \ldots, v_{n-1}$. Our representation is similar to the one used in [5], based on arithmetic operations.

Bitvector operations are used to evaluate all value combinations at once.

The function `vn`, working with arbitrary length bitstrings are used to evaluate the [0..n-1] variables v_k while `vm` maps the constant `1` to the bitstring of length 2^n, `111..1`:

```
vn 1 0 = 1
vn n q | q==pred n = bitpair (vn n 0,0)
vn n q  | q≥0 && q < n' =
    bitpair (q',q') where
      n'=pred n
      q'=vn n' q

vm n = vn (succ n) 0
```

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.

4.4 Boolean Evaluation of OBDTs

Practical uses of OBDTs involve reducing them to Reduced Ordered Binary Decision Diagrams (ROBDDs) by sharing nodes and eliminating identical branches [1]. Note that in this case `obdt2nat` 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_obdt` describes the *OBDT* evaluator:

```
eval_obdt (OBDT n bt) = eval_with_mask (vm 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_ (vn n x)
         (eval_with_mask m n l)
         (eval_with_mask m n r)
```

The *projection functions* `vn` defined in section 4.3 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_obdt` *faithfully emulates* `fold_obdt`*!*

```
*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
```

4.5 The Equivalence

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

Proposition 1 *The complete binary tree of depth n, obtained by recursive applications of* `bitunpair` *on a truth table tt computes an (unreduced) OBDT, that, when evaluated, returns the truth table, i.e.*

$$fold\_obdt \circ unfold\_obdt \equiv id \tag{2}$$

$$eval\_obdt \circ unfold\_obdt \equiv id \tag{3}$$

Proof. The function `unfold_obdt` 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 [2] 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 `vn 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 OBDT data type defined in section 4 actually represents OBDTs mapped one-to-one to truth tables given as natural numbers. Note also that the equivalence also holds after further reduction of OBDT to Binary Decision Diagrams (BDDs) and ROBDDs given that they represent the same boolean function.

5 Ranking and Unranking of OBDTs

One more step is needed to extend the mapping between $OBDTs$ with n variables to a bijective mapping from/to $\mathbb{N}$: we will have to "shift towards infinity" the starting point of each new block[3] of OBDTs in $\mathbb{N}$ as OBDTs 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
```

[3] defined by the same number of variables

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

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

*OBDT> 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 OBDT is now easy - the index `k` determines the number of variables and `n-m` determines the rank. Together they select the right OBDT with `unfold_obdt` and `obdt`.

```
nat2obdt n = unfold_obdt (k,n_m)
  where (k,n_m)=to_bsum n
```

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

```
obdt2nat obdt@(OBDT nv _) =  ((bsum nv)+tt) where
   (_,tt) =fold_obdt obdt
```

As the following example shows, `obdt2nat` implements the inverse of `nat2obdt`.

```
*OBDT> nat2obdt 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> obdt2nat it
42
```

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

```
ev_obdt2nat obdt@(OBDT nv _) = (bsum nv)+(eval_obdt obdt)
```

We can confirm that `ev_obdt2nat` also acts as an inverse to `nat2obdt`:

```
*OBDT> ev_obdt2nat (nat2obdt 42)
42
```

5.1 Reducing the *OBDTs*

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

The function `obdt_reduce` reduces a *OBDT* by collapsing identical left and right subtrees, and the function `obdt` associates this reduced form to $n \in \mathbb{N}$.

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

```
obdt_reduce (OBDT n bt) = OBDT 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_robdt = obdt_reduce . unfold_obdt
```

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

```
*OBDT> mapM_ (λtt→print (unfold_robdt (2,tt))) [0..15]

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

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

```
nat2robdt = obdt_reduce . nat2obdt
```

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

```
obdt_size (OBDT _ t) = 1+(size t) where
  size B0 = 1
  size B1 = 1
  size (D _ l r) = 1+(size l)+(size r)
```

6 Generalizing OBDT ranking/unranking functions

6.1 Encoding OBDTs with Arbitrary Variable Order

While the encoding built around the equivalence described in Prop. 1 between bitwise pairing/unpairing operations and boolean decomposition is arguably as simple and elegant as possible, it is useful to parametrize OBDT generation with respect to an arbitrary variable order. This is of particular importance when using OBDTs 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_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 _ tt = error ("bad arg in to_obdt⇒" ++ (show tt))
```

where the function `to_obdt_mn` recurses over the list of variables `vs` and applies Shannon expansion [2], expressed as bitvector operations. This computes the cofactor functions $f1$ and $f0$, to be used as `then` and `else` branches, when evaluating back the OBDT to a truth table with if-the-else functions.

```
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
  r=to_obdt_mn vs f0 m n
```

Proposition 2 *The function* `to_obdt` *builds an (unreduced) OBDT corresponding to a truth table* `tt` *for variable order* `vs` *that returns* `tt`*, when evaluated as a boolean function.*

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

```
to_robdt vs tt = obdt_reduce (to_obdt vs tt)
from_obdt obdt = eval_obdt obdt
```

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

```
search_obdt minORmax n tt = snd $ foldl1 minORmax
 (map (sized_robdt tt) (all_permutations n)) where
    sized_robdt tt vs = (obdt_size b,b) where
      b=to_robdt 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_obdt min` can be used for multilevel boolean formula minimization on functions with up to 6-7 arguments.

```
*OBDT> search_obdt min 3 42
OBDT 3 (D 2 B0 (D 0 B1 (D 1 B1 B0)))
*OBDT> search_obdt min 4 2008
OBDT 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)))
*OBDT> search_obdt min 3 2008
OBDT 7 (D 1 (D 2 (D 4 (D 3 (D 0 (D 5 ...
      ... (D 0 (D 5 B0 B1) B0)))))
*OBDT> obdt_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 OBDT minimization algorithm compares with "perfect circuits" provided by an exact synthesizer as the one described in [6, 7]. 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 `search_obdt min` on the 3-variable function 22 representing as a natural number the truth table of our half adder, we obtain:

```
*OBDT> search_obdt min 3 22
OBDT 3 (D 0 (D 1 (D 2 B0 B1) (D 2 B1 B0))
            (D 1 (D 2 B1 B0) B0))
```

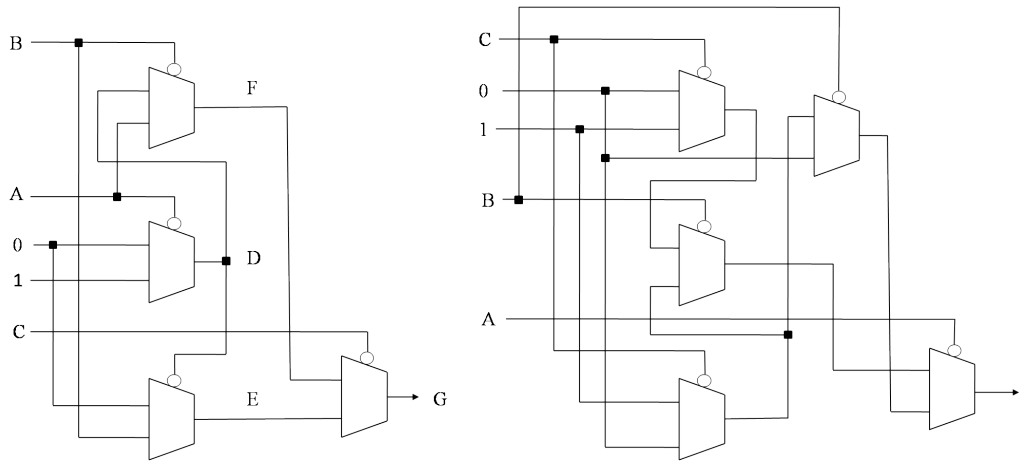
Assuming the sharing of the two `(D 2 B1 B0)` nodes we can see that while we do not obtain the exact minimum of 4 in this case, we get close enough (5 gates). The diagrams in Fig. 2 show the actual circuits, with variables `0,1,2` renamed as `A,B,C` for easier comparison.

6.2 Multi-Terminal Ordered OBDTs (MTOBDT)

MTOBDTs [8, 9] are a natural generalization of OBDTs 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 $MTOBDTs$ that can be extended to ranking/unranking functions, in a way similar to $OBDTs$ as shown in section 5.

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



(a) Exact synthesis (b) As minimal ROBDTs

Fig. 2: Binary decision diagrams associated to $ITE(A \oplus B, A \wedge B)$

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

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

```
to_mtobdt m n tt = MTOBDT m n r where
  mlimit=2^m
  nlimit=2^n
  ttlimit=mlimit^nlimit
  r=if tt<ttlimit
    then (to_mtobdt_ 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_mtobdt_` applies `bitunpair` recursively up to depth n, where leaves in range $[0..mlimit-1]$ are created.

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

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

```
from_mtobdt (MTOBDT m n b) = from_mtobdt_ (2^m) n b

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

The following example shows that `to_mtobdt` and `from_mtobdt` are indeed inverses.

```
*OBDT> to_mtobdt 3 3 2010
MTOBDT 3 3 (M 2 (M 1 (M 0 (L 2) (L 1)) (M 0 (L 2) (L 1)))
                (M 1 (M 0 (L 3) (L 0)) (M 0 (L 1) (L 1))))
>from_mtobdt it
2010
```

6.3 Generating Random OBDTs and MTOBDTs

Random generation of OBDTs and MTOBDTs 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 $\mathbb{N}$ 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 OBDT mapped to natural numbers from 10 to 20 one can write:

```
*OBDT> nrandom nat2robdt 10 20 3 77
[ OBDT 2 (D 1 (D 0 B1 B0) B1),
  OBDT 2 (D 1 (D 0 B0 B1) B1),
  OBDT 2 (D 0 B0 B1)]
```

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

```
*OBDT> head $ nrandom (to_mtobdt 3 3) 1000 2000 1 1
MTOBDT 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 OBDTs to reduced OBDTs with something like:

```
*OBDT> sum $ map obdt_size $ nrandom nat2obdt 1000 2000 10 7
320
*OBDT> sum $ map obdt_size $ nrandom nat2robdt 1000 2000 10 7
194
```

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

```
*OBDT> sum $ map obdt_size (nrandom (search_obdt max 4) 0 (2^2^4-1) 100 77)
2384
*OBDT> sum $ map obdt_size (nrandom (search_obdt min 4) 0 (2^2^4-1) 100 77)
1744
*OBDT> sum $ map obdt_size (nrandom (search_obdt max 5) 0 (2^2^5-1) 100 77)
4812
*OBDT> sum $ map obdt_size (nrandom (search_obdt 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.

7 Related work

Preliminary work related to this paper, has been presented at the CICLOPS'08 workshop and at Calculemus 2009 - Emerging trends (both with informal online proceedings only).

Pairing functions have been used for work on decision problems as early as [10].

ROBDDs derived from OBDTs by implementing node sharing are the dominant boolean function representation in the field of circuit design automation [11].

Besides their uses in circuit design automation, multi-terminal BDDs have been used in model-checking and verification of arithmetic circuits [8, 9].

ROBDDs have also been used in a Genetic Programming context [12] as a representation of evolving individuals subject to crossovers and mutations expressed as structural transformations.

8 Conclusion and Future Work

The surprising connection of bitstring based pairing/unpairing functions and to OBDTs 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 [6, 7] and ongoing work on the use of genetic programming algorithms in circuit synthesis.

We have found such encodings interesting as uniform building blocks for Genetic Programming applications. In a Genetic Programming context [13], the bijections between bitvectors/natural numbers on one side, and trees/graphs representing OBDTs on the other side, suggest exploring the mapping and its action on various transformations as a phenotype-genotype connection. Given the connection between OBDTs and BDDs to boolean and finite domain constraint solvers it would be interesting to explore in that context, efficient succinct data representations derived from our OBDT encodings.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691
2. Shannon, C.E.: Claude Elwood Shannon: collected papers. IEEE Press, Piscataway, NJ, USA (1993)
3. Pigeon, S.: Contributions á la compression de données. (2001)
4. Misra, J.: Powerlist: a structure for parallel recursion. ACM Transactions on Programming Languages and Systems **16** (1994) 1737–1767
5. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
6. Tarau, P., Luderman, B.: A Logic Programming Framework for Combinational Circuit Synthesis. In: 23rd International Conference on Logic Programming (ICLP), LNCS 4670, Porto, Portugal, Springer (September 2007) 180–194
7. Tarau, P., Luderman, B.: Exact combinational logic synthesis and non-standard circuit design. In: CF '08: Proceedings of the 2008 conference on Computing frontiers, New York, NY, USA, ACM (2008) 179–188
8. Fujita, M., McGeer, P.C., Yang, J.C.Y.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in System Design **10**(2/3) (1997) 149–169
9. Ciesinski, F., Baier, C., Groesser, M., Parker, D.: Generating compact MTBDD-representations from Probmela specifications. In: Proc. 15th International SPIN Workshop on Model Checking of Software (SPIN'08). (2008)
10. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
11. Meinel, C., Theobald, T.: Ordered binary decision diagrams and their significance in computer-aided design of vlsi circuits. Journal of Circuits, Systems, and Computers **9**(3-4) (1999) 181–198
12. Sakanashi, H., Higuchi, T., Iba, H., Kakazu, Y.: Evolution of binary decision diagrams for digital circuit design using genetic programming. In Higuchi, T., Iwata, M., Liu, W., eds.: ICES. Volume 1259 of Lecture Notes in Computer Science., Springer (1996) 470–481
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)