Boolean Evaluation with a Pairing and Unpairing Function

Paul Tarau
Department of Computer Science and Engineering
Univ. of North Texas
Denton, USA
Email: tarau@cse.unt.edu

Brenda Luderman
Analog Design Services
Texas Instruments Inc.[*]
Dallas, USA
Email: brenda.luderman@gmail.com

***Abstract*—A *pairing function* is a bijection $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. Its inverse is called an *unpairing function*. We show that boolean logic on bitvector variables can be expressed as compositions of pairing/unpairing operations which can emulate boolean evaluation of ordered binary decision trees (OBDTs) of a canonical form. Applications to enumeration and random generation of OBDTs and a generalization to Multi-Terminal Ordered OBDTs (MTOBDT) are also described. The paper is organized as a literate Haskell program (code available at http://logic.csci.unt.edu/tarau/research/2012/hOBDT.hs).**

***Keywords*-encodings of boolean functions; pairing/unpairing functions; ordered binary decision trees; bitvector operations; functional programming.**

I. Introduction

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

Pairing functions have been used in foundations of set theory since its origins, starting with G. Cantor's geometrically inspired pairing function. They are also related to 2D-space filling curves (Z-order, Gray-code and Hilbert curves) [1], [2], [3], [4]. Such curves are obtained by connecting pairs of coordinates corresponding to successive natural numbers. They have applications in spatial and multi-dimensional database indexing.

As part of an effort to organize transformations between fundamental data types used in computer science and discrete mathematics as a groupoid of isomorphisms [5], [6], [7] and as type class instances sharing polymorphic operations [8], [9], we have studied various pairing functions and observed their important role in ranking/unranking algorithms and encodings of sequences, trees and various graph types.

This paper focuses on a specific use of a specific pairing function in emulating some simple but fundamental properties of boolean functions, including boolean evaluation and a canonical representation that connects ordered binary decision trees and trees obtained by recursive applications of an unpairing operation.

This unusual application of pairing functions originates in our experiments with optimal exact circuit synthesis where the use of pairing functions facilitates exhaustive enumeration of candidate circuits matching a given specification.

Another motivation for this work is inspired by applications of the powerlist algebra of [10] to parallelize arithmetic operations, involving tree representations similar to the representations we obtain through the use of recursive application of unpairing functions.

We will use a subset of the non-strict functional language Haskell (seen as a notation for the theory of recursive functions) to provide a self-contained executable specification of all our computations. We mention, for the benefit of the reader unfamiliar with the language, that a notation like `f x y` stands for $f(x,y)$, `[t]` represents sequences of type `t` and a type declaration like `f :: s -> t -> u` stands for a function $f : s \times t \rightarrow u$ (modulo Haskell's "currying" operation, given the isomorphism between the function spaces $s \times t \rightarrow u$ and $s \rightarrow t \rightarrow u$). Our Haskell functions are always represented as sets of recursive equations guided by pattern matching, conditional to constraints (simple arithmetic relations following `|` and before the = symbol). Locally scoped helper functions are defined in Haskell after the `where` keyword, using the same equational style. The composition of functions `f` and `g` is denoted `f . g`. It is also customary in Haskell, when defining functions in an equational style (using =) to write $f = g$ instead of $f\ x = g\ x$ ("point-free" notation). The use of Haskell's "call-by-need" evaluation allows us to work with infinite sequences, like the `[0..]` infinite list notation, corresponding to the set $\mathbb{N}$ itself.

[*]Brenda Luderman joined Texas Instruments in October 2011. This work was completed prior to 2011 and is an independent research project with the co-author.

II. Pairing/Unpairing as Bitvector Operations

We will start by defining a "bitwise" pairing function used in 2D Z-order encodings [1], (also known as Morton encoding, independently rediscovered in [11], p.142, [6] and similar to Misra's *zip* function [10] in the *powerlist* algebra). The function `bitunpair` works by splitting a natural number's big endian bitstring representation into odd and even bits, while its inverse `bitpair` blends the odd and even bits back together.

```
type N = Integer

bitunpair :: N→(N,N)
bitpair ::  (N,N) → N

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

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

```
inflate,deflate :: N → N

inflate 0 = 0
inflate n =
  (twice . twice . inflate . half) n .|. firstBit n

deflate 0 = 0
deflate n =
  (twice . deflate . half . half) n .|. firstBit n

deflate′ = half . deflate . twice
inflate′ = twice . inflate
```

This operations are expressed in terms of bitshifts and boolean operations.

```
half n = shiftR n 1 :: N
twice n = shiftL n 1 :: N
firstBit n = n .&. 1 :: N
```

Clearly, the following holds:

Proposition 1: bitpair and bitunpair are inverses.
The transformation is illustrated in the following example with bitstrings (least significant bit first) aligned:

```
*BP> 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 parallel hardware can compute `bitpair` and `bitunpair` on registers of arbitrary size in constant time by communicating $2n$ bits independently. One can also see these bits simply as $2n$ wires in a combinational circuit.

Proposition 2: The following relations hold:

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

Definition 1: A set M together with a binary operation $f : M \times M \to M$ is called a magma. A free magma is an initial object in the category of magmas.

Seen as a binary operation, a pairing function provides only a *magma* structure on $\mathbb{N}$ as it is non-associative, non-commutative etc. However, the following holds, as an immediate consequence of bijectivity:

Proposition 3: A pairing function f is left and right *cancellative* i.e. f(a,b)=f(a',b') implies a=a' and b=b'.

We will denote $x \uparrow y$ the result of `bitpair (x,y)` and $\swarrow z$ and $z \searrow$ the first and second projection of `bitunpair z`.

Proposition 4: Every natural number is uniquely represented as an element of the free magma generated by $\uparrow$ on 0,1.

Proof: Observe that if $z \notin \{0,1\}$, then $\swarrow z < z$ and $z \searrow < z$. This implies that applying bitunpair repeatedly stops after a finite number of steps on either 0 or 1. Uniqueness of the representation follows from $\uparrow$ being left and right cancellative. ■

Fig. 1 shows a DAG representation of the binary tree obtained by applying the `bitunpair` operation recursively, starting with `2012`, with labels 0 and 1 on the edges indicating the order in the resulting pair at each step.

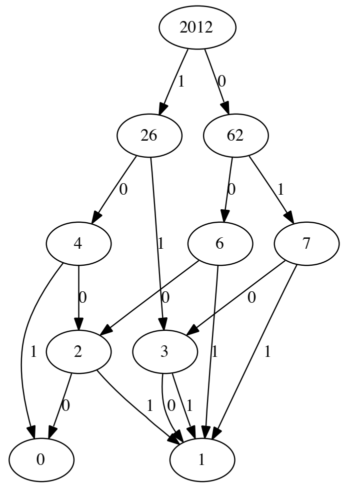

Fig. 1. Recursive application of bitunpair

III. Unfolding and Folding with Pairing Functions

We define a binary tree type `BT` that has the constants `O` and `I` as leaves representing the boolean values 0 and 1 and internal nodes marked with `D`.

```
data BT = O | I | D BT BT deriving (Eq,Ord,Read,Show)
```

The function `bdepth` computes the depth of a complete binary tree of type `BT`

```
bdepth O = 0
bdepth I = 0
bdepth (D x _) = 1 + (bdepth x)
```

A. *Unfolding Natural Numbers to Binary Trees with* `bitunpair`

The following functions apply `bitunpair` recursively, on a natural number `tt`, (to be seen later as a n-variable 2^n bit truth table), to build a complete binary tree of depth n, that we represent using the BT data type.

```
unfold_bt :: (N,N) → BT
unfold_bt (n,tt) = if tt<2^2^n
  then unfold_with bitunpair n tt
  else undefined where
    unfold_with _ n 0 | n<1 =  O
    unfold_with _ n 1 | n<1 =  I
    unfold_with f n tt =
      D (unfold_with f k tt1) (unfold_with f k tt2) where
        k=n-1
        (tt1,tt2)=f tt
```

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

```
D (D O O) (D O O)
D (D I O) (D O O)
.................
D (D O I) (D I I)
D (D I I) (D I I)
```

B. *Folding binary trees to natural numbers with* bitpair

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

```
fold_bt :: BT → (N,N)
fold_bt bt = (n,fold_with bitpair bt) where
    n = bdepth bt
    fold_with f O = 0
    fold_with f I = 1
    fold_with f (D l r) = f (fold_with f l,fold_with f r)
```

Clearly, the following holds:

Proposition 5: The functions fold_bt and unfold_bt are inverses.

The two bijections work as follows:

```
*BP> unfold_bt (3,42)
D (D (D O O) (D O O)) (D (D I I) (D I O))
*BP>fold_bt it
(3,42)
```

IV. Boolean Evaluation in Terms of a Pairing Function

We will overview boolean evaluation using bitvectors, followed by algorithms to express projection variables and boolean operations in terms of pairing functions.

A. *The Arithmetic of Bitvector Operations*

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

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

Unfortunately this does not take advantage of the ability of modern hardware to perform such operations one word a time - with the instant benefit of a speed-up proportional to the word size. We will use here an alternative representation, based on integer encodings of 2^n bits for each boolean variable $v_0, \ldots, v_{n-1}$. Bitvector operations are used to evaluate all value combinations at once.

Boolean functions with n variables can be seen as truth tables having as columns 2^n-bit vectors corresponding to natural numbers in the interval $[0, 2^{2^n} - 1]$.

Definition 2: Column k, for $0 \leq k < n$ of the truth table is called *bitvector variable* x_k.

Donald Knuth, in the section on Boolean Evaluation of [12] (Algorithm L, section 7.1.2) observes that

Proposition 6 (Knuth): Bitvector variables x_k can be computed with integer operations as $x_k = (2^{2^n} - 1)/(2^{2^{n-k}} + 1)$.

Proof: The proposition follows from the fact that x_k is obtained as a concatenation of alternating blocks of 2^{n-k} 0s and 1s, and therefore, the product $x_k(1 + 2^{2^{n-k}})$ gives a bitstring consisting all of 1s representing $2^{2^n} - 1$. ■

B. *Expressing Bitvector Variables in Terms of a Pairing Function*

We show first that bitvector variables can be expressed directly in terms of bitpair. The following hold:

Proposition 7: Repeated application of bitpair(x,x), n times, starting with x=1 produces $2^{2^n} - 1$

Proof: Obvious - we start with a string containing a 1 and then obtain 11, 1111, 11111111 etc. ■

Proposition 8: The bitpair operation applied to a bitvector variable of bitlength n, with blocks of k alternating 0s and 1s, and itself, produces a bitvector variable of bitlength 2n with blocks of 2k alternating 0s and 1s.

Proof: It follows from the definition of bitpair - given that they are aligned, the blocks of 0s and 1s double in size each. ■

These propositions justify the following definitions, covering the generation of arbitrary bitvector variables using bitpair.

```
vn :: N→N→N
vn 1 0 = 1
vn n q | q == n-1 = bitpair (vn n 0,0)
vn n q  | q≥0 && q < n' = bitpair (q',q') where
  n' = n-1
  q' = vn n' q

vm :: N→N
vm n = vn (n+1) 0
```

The function vn, working with arbitrary length bitstrings, is used to evaluate, for $0 \leq k \leq n-1$ the *bitvector variables* v_k representing encodings of columns of a truth table. Note that, based on Prop. 2, vn could also be expressed directly in terms of inflate. Note also that the constant 0 is represented as 0 while the constant 1 is represented as vm n$=2^{2^n} - 1$, corresponding to a column in the truth table containing 2^n ones 111..1. The following hold:

Proposition 9: Bitvector variable $vn\ n\ (n-1)$, an alternation of 1s and 0s, is obtained by applying bitpair to a bitvector of 2^n 1s and bitvector 0.

Proposition 10: A bitvector variable of bitsize n is either 0,1 or of the form bitpair (0,x) or bitpair (x,x) where x is a bitvector variable of bitsize (n-1).

Bitvector variables can be combined with the usual bitwise integer operators, to obtain arbitrary bitvector representations of truth tables, encoding all possible boolean value combinations of their arguments.

Our representation for the bitvector variables `vn` is essentially equivalent to the one used in Donald Knuth's chapter on boolean evaluation [12], except that he provides there a formula based on arithmetic operations while we use predecessor/successor and recursion on pairing/unpairing functions. The following holds:

Proposition 11: `vn n k` $= x_{k-1} = (2^{2^n}-1)/(2^{2^{n-(k-1)}}+1)$.

Proof: It follows from Prop. 6 as `vn n k` corresponds to the same blocks of 0s and 1s as x_{k-1}. ∎

C. Boolean Evaluation of OBDTs

Ordered Reduced Binary Trees (OBDTs) are complete binary trees with internal nodes interpreted as `if-then-else` gates controlled by the same sequence of n distinct boolean variables on each path from the root to a leaf.

Practical uses of OBDTs involve transforming them to Reduced Ordered Binary Decision Diagrams (ROBDDs) by sharing nodes and eliminating identical branches [13].

The function `eval_obdt_with` describes the $OBDT$ evaluator parametrized by a permutation of n variables and a complete binary tree of depth n:

```
eval_obdt_with :: [N]→BT→N
eval_obdt_with vs bt =
  eval_with_mask (vm n) (map (vn n) vs) bt where
    n = genericLength vs

eval_with_mask m _ O = 0
eval_with_mask m _ I = m
eval_with_mask m (v:vs) (D l r) =
  ite_ v (eval_with_mask m vs l) (eval_with_mask m vs r)
```

The bitvector variables `vn` 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 $2^{2^n}-1$.

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

```
ite_ :: N→N→N→N
ite_ x t e = ((t `xor` e).&.x) `xor` e
```

Definition 3: We call *canonical OBDT* the OBDT specified by n variables in decreasing order and a complete binary tree of depth n.

One can specialize `eval_obdt_with` for a choice of variables in decreasing order as:

```
eval_obdt :: BT → N
eval_obdt bt =
  eval_obdt_with (reverse [0..(bdepth bt)-1]) bt
```

V. The Equivalence between canonical OBDTs and unpairing trees

Given the bijection between the set of trees generated by an unpairing function and the set of natural numbers, one can view such trees as a generator of boolean expressions, provided that a boolean operation is associated to each node of the tree and the resulting natural number encodes the bitvector representing the truth table of a boolean function.

We will show that a canonical Ordered Binary Decision Tree ($OBDT$) representing the same logic function as an n-variable 2^n bit truth table (seen as an integer `tt`), 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 an $OBDT$ such that there exists a variable ordering for which its boolean evaluation returns `tt`.

Proposition 12: The complete binary tree of depth n, obtained by recursive applications of `bitunpair` on a truth table tt computes a binary tree, that, when evaluated as a canonical OBDT (i.e. with variable order $[n-1,...,0]$), returns the truth table. More precisely, if

$$fold\_bt \circ unfold\_bt \equiv id \tag{1}$$

then

$$eval\_obdt \circ unfold\_bt \equiv id \tag{2}$$

Proof: The function `unfold_bt` builds a binary tree by splitting the bitstring $tt \in [0..2^{2^n}-1]$ up to depth n. This corresponds to the repeated Shannon expansion [14] 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 bitsize 2^n is the double of bitsize 2^{n-1}, the same invariant holds at each step, as the bitsize of the truth table reduces to half. ∎

Note that boolean evaluation can also happen after further reduction of OBDTs to Binary Decision Diagrams (BDDs) and ROBDDs, given that they represent the same boolean function.

The following examples illustrate the equivalence between the `fold_bt` *and* `eval_obdt`*, which acts as an inverse of* `unfold_bt` *in the same way as* `fold_bt`*.*

```
*BP> unfold_bt (3,42)
D (D (D O O) (D O O)) (D (D I I) (D I O))
*BP> eval_obdt it
42
*BP> unfold_bt (4,2012)
D (D (D (D O I) (D I O)) (D (D I I) (D I O)))
  (D (D (D O I) (D O O)) (D (D I O) (D I O)))
*BP> eval_obdt it
2012
```

This result creates a bridge between pairing/unpairing functions, originating in the foundation of mathematics, and bitvector operations implementing boolean logic. Potential practical applications include parallel execution of boolean operations using the binary tree structure induced by repeated application of an unpairing function and the design of boolean circuits for reversible computing taking advantage of the fact that pairing functions are bijections.

One might wonder at this point why one cannot turn the well-known Shannon expansion into a pairing/unpairing bijection, implemented as concatenation of two bitvectors. The reason why this does not lead to a pairing / unpairing bijection is a subtle difference with `bitpair/bitunpair`: undoing the concatenation of two bitstrings A and B to reverse the Shannon expansion is ambiguous, unless information about the bitlengths of A and B is kept. However, assuming that this is information is implicit in the depth of the tree, a similar bijection can be built. More generally, as we will show in subsection VII-A, such a bijection can be associated to an arbitrary variable order.

VI. Ranking and Unranking of canonical OBDTs

One more step is needed to extend the mapping between canonical $OBDTs$ with n variables to a bijective mapping to $\mathbb{N}$ (a *ranking/unranking* function): we will have to "shift towards infinity" the starting point of each new block[1] 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 :: N→N
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[2]:

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

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

We are now able to decompose `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 canonical 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_bt (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 =  (bsum nv) + tt where
  nv = bdepth obdt
  (_,tt) = fold_bt obdt
```

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

```
*BP> nat2obdt 42
D (D (D O I) (D I O)) (D (D O O) (D O O))
*BP> obdt2nat it
42
```

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

[1]defined by the same number of variables

[2]`bsums` turns out to be the sequence A060803 in The On-Line Encyclopedia of Integer Sequences, http://www.research.att.com/~njas/sequences

```
ev_obdt2nat obdt = (bsum nv)+(eval_obdt obdt) where
  nv = bdepth obdt
```

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

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

VII. Generalizing OBDT ranking/unranking functions

While the encoding built around the equivalence described in Prop. 12 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 [13].

A. Encoding OBDTs with Arbitrary Variable Order

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 =
  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 [14], 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 _ _ = O
to_obdt_mn []      _ _ _ = I
to_obdt_mn (v:vs) tt m n = D 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 13: 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.

B. Multi-Terminal Ordered OBDTs (MTOBDT)

MTOBDTs [15] 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 describe an encoding of $MTOBDTs$ that can be extended to ranking/unranking functions, in a way similar to $OBDTs$ as shown in section VI.

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.

```
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, a canonical MTOBDT representing functions of type $N \to M$ with $M = [0..2^m-1], N = [0..2^n-1]$. Similarly to a canonical OBDT, it is represented as binary tree of n levels, except that its leaves are in $[0..2^m-1]$, rather than $\{0,1\}$.

Note that, after validation, the actual work is performed by the function `to_mtobdt_` that applies the `bitunpair` function recursively.

```
to_mtobdt :: N → N → N → MTOBDT N
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_ :: N → N → N → MT N
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=n-1
   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 N → N
from_mtobdt (MTOBDT m n b) = from_mtobdt_ (2^m) n b

from_mtobdt_ :: N → N → MT N → N
from_mtobdt_ mlimit n (L tt)|(n<1)&&(tt<mlimit)=tt
from_mtobdt_ mlimit n (M _ l r) = tt where
   k=n-1
   x=from_mtobdt_ mlimit k l
   y=from_mtobdt_ mlimit k r
   tt=bitpair (x,y)
```

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

```
*BP> 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
```

C. 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 random natural numbers in a given interval. After customizing Haskell's library random generator

```
nrandom_nats :: (Random a) ⇒ a → a → N → Int → [a]
nrandom_nats smallest largest n seed = genericTake n
    (randomRs (smallest,largest) (mkStdGen seed))
```

one can define:

```
nrandom :: (Random a) ⇒
  (a → b) → a → a → N → Int → [b]
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:

```
*BP> nrandom nat2obdt 10 20 3 77
[D (D I O) (D I I),D (D O I) (D I I),
 D (D O I) (D O I)]
```

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

```
*BP> 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))))
```

VIII. Related work

This paper uses the functional programming language Haskell as a catalyst for exploring analogies in experimental mathematics, derived from bijective mappings between datatypes described in [6], [7], [16].

Preliminary work related to this paper, has been presented at Calculemus 2009 - Emerging trends (with informal online proceedings only).

Pairing functions have been used in work on decision problems as early as [17], [18]. A typical modern use in the foundations of mathematics is [19]. An extensive study of various pairing functions and their computational properties is presented in [20].

Boolean function evaluation, encoding and synthesis are extensively studied in Donald Knuth's recent work [12], [21].

A number of papers of J. Vuillemin develop techniques aiming to unify various data types, with focus on theories of boolean functions and arithmetic [22].

Reduced OBDTs (also called ROBDDs) derived from OBDTs by implementing node sharing are the dominant boolean function representation in the field of circuit design automation [23]. Various applications to program analysis are discussed in [24], [25]. ROBDDs have also been used in a Genetic Programming context [26] as a representation of evolving individuals subject to crossovers and mutations expressed as structural transformations.

IX. Conclusion

We have provided a bridge between connecting pairing/unpairing bijections and boolean function evaluation, enumeration and random generation. Traditionally, such pairing functions have been used in the foundation of mathematics to provide constructive proofs that the infinite sets $\mathbb{N}\times\mathbb{N}$ and $\mathbb{N}$ have the same cardinality, and in recursion theory to reduce multiple argument functions on $\mathbb{N}$ to single argument functions. The somewhat surprising connection with boolean evaluation hints towards applications to the theory of of boolean functions as well as boolean circuit complexity and circuit synthesis. Our initial interest in the connection of bitvector based pairing/unpairing functions and OBDTs has been triggered by applications of the encodings to combinational circuit synthesis [27], [28]. In [8] we have shown that various tree-like data types can perform symbolically arbitrary length arithmetic operations with efficiency comparable with conventional bitstring implementations. This suggests exploring the possibility to use the connection between the pairing function discussed in this paper and arbitrary length arithmetic operations, of special interest also given that similar tree representations based on powerlists [10] have been shown useful in parallelization of various algorithms.

Acknowledgments

We thank the anonymous reviewers of SYNASC'2012 for their thoughtful comments and suggestions. Paul Tarau's work has been supported by NSF research grant 1018172.

References

[1] J. Lawder, "The application of space-filling curves to the storage and retrieval of multi-dimensional data," 2000, phD Thesis, University of London, UK.

[2] J. Lawder and P. King, "Using space-filling curves for multi-dimensional indexing," in *Advances in Databases*, ser. Lecture Notes in Computer Science, B. Lings and K. Jeffery, Eds. Springer Berlin / Heidelberg, 2000, vol. 1832, pp. 20–35.

[3] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data indexed using the hilbert space-filling curve," *SIGMOD Rec.*, vol. 30, pp. 19–24, March 2001.

[4] B. Moon, H. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, pp. 124–141, 2001.

[5] P. Tarau, "Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell," Jan. 2009, unpublished draft, http://arXiv.org/abs/0808.2953, updated version at http://logic.cse.unt.edu/tarau/research/2010/ISO.pdf, 150 pages.

[6] ——, ""Everything Is Everything" Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms," *Complex Systems*, no. 18, pp. 475–493, 2010.

[7] ——, "A Groupoid of Isomorphic Data Transformations," in *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009* , J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, Eds. Grand Bend, Canada: Springer, LNAI 5625, Jul. 2009, pp. 170–185.

[8] ——, "Declarative modeling of finite mathematics," in *PPDP '10: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. New York, NY, USA: ACM, 2010, pp. 131–142.

[9] ——, "On Arithmetic Computations with Hereditarily Finite Sets, Functions and Types," in *Proceedings of 7th International Colloquium on Theoretical Aspects of Computing, ICTAC 2010*, A. Cavalcanti, D. Deharbe, M. C. Gaudel, and J. Woodcock, Eds. Natal, Brazil: Springer, LNCS 6255, Sep. 2010, pp. 367–381.

[10] J. Misra, "Powerlist: a structure for parallel recursion," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1737–1767, 1994.

[11] S. Pigeon, "Contributions à la compression de données." Ph.D. thesis, Université de Montréal, Montréal, 2001.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 0: Introduction to Combinatorial Algorithms and Boolean Functions*. Addison-Wesley Professional, 2009.

[13] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[14] C. E. Shannon, *Claude Elwood Shannon: collected papers*, N. J. A. Sloane and A. D. Wyner, Eds. Piscataway, NJ, USA: IEEE Press, 1993.

[15] M. Fujita, P. C. McGeer, and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *Formal Methods in System Design*, vol. 10, no. 2/3, pp. 149–169, 1997.

[16] P. Tarau, "An Embedded Declarative Data Transformation Language," in *Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009*. Coimbra, Portugal: ACM, Sep. 2009, pp. 171–182.

[17] J. Pepis, "Ein verfahren der mathematischen logik," *The Journal of Symbolic Logic*, vol. 3, no. 2, pp. 61–76, jun 1938.

[18] J. Robinson, "General recursive functions," *Proceedings of the American Mathematical Society*, vol. 1, no. 6, pp. 703–718, dec 1950.

[19] P. Cégielski and D. Richard, "Decidability of the Theory of the Natural Integers with the Cantor Pairing Function and the Successor," *Theor. Comput. Sci.*, vol. 257, no. 1-2, pp. 51–77, 2001.

[20] A. L. Rosenberg, "Efficient pairing functions - and why you should care," *International Journal of Foundations of Computer Science*, vol. 14, no. 1, pp. 3–17, 2003.

[21] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.

[22] J. Vuillemin, "On circuits and numbers," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 868–879, 1994.

[23] C. Meinel and T. Theobald, "Ordered binary decision diagrams and their significance in computer-aided design of vlsi circuits," *Journal of Circuits, Systems, and Computers*, vol. 9, no. 3-4, pp. 181–198, 1999.

[24] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDDs," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Inplementation*. ACM Press, 2003, pp. 103–114.

[25] G. Price and M. Vachharajani, "A Case for Compressing Traces with BDDs," *IEEE Comput. Archit. Lett.*, vol. 5, no. 2, pp. 18–18, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1109/L-CA.2006.17

[26] H. Sakanashi, T. Higuchi, H. Iba, and Y. Kakazu, "Evolution of binary decision diagrams for digital circuit design using genetic programming," in *ICES*, ser. Lecture Notes in Computer Science, T. Higuchi, M. Iwata, and W. Liu, Eds., vol. 1259. Springer, 1996, pp. 470–481.

[27] P. Tarau and B. Luderman, "A Logic Programming Framework for Combinational Circuit Synthesis," in *23rd International Conference on Logic Programming (ICLP), LNCS 4670*. Berlin Heidelberg: Springer, Sep. 2007, pp. 180–194.

[28] ——, "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, pp. 179–188.