NOTES ON MATRIX TECHNIQUES FOR LOGIC

William Bricken
March 1997

Tables, matrices, trees, and graphs all lay out data spatially, providing convenient processing models for the abstract concept of possibility space (ie state space, ensemble).

Tables come with a database calculus for joining and intersecting collections of information. Etter uses a database calculus to implement operations in matrix algebra that are used in quantum mechanics. Matrices come with an efficient matrix calculus, and understood techniques for matrix transformation. The distinction between computation over tables or matrices is moot when a problem is phrased in terms of abstract data structures.

Relational algebra and matrix algebra are equivalent in expressibility, and are both quite similar in computational style, the major difference being that the ordering imposed by a matrix formulation provides structured transformation tools, while the lack of ordering of elements in a relational formulation requires database search during computation.

A central issue for representation systems is whether or not they provide useful intermediary tools for phrasing computational problems. Matrices provide some excellent tools for

* decomposition of operations (factoring for symmetries and invariants),
* unifying objects and operators, and
* selection over evaluation orderings.

As well the process model for matrices is *multiplicative* rather than additive, providing 0 as a powerful annihilator and avoiding the necessity of the carry operation associated with additive models.

At the silicon level, the implementation advantages of multiplicative models are compelling. Silicon addition (mod 2) is naturally expressed as XOR logic, the most complex binary logic operation. In contrast, silicon multiplication is expressed in a simpler AND logic.

Matrices also provide a generalized notation for negation as the complement, **1** - M. This concept of negation easily maps to boundary notation, providing the algebraic transformations of boundary mathematics within a context of matrix computation.

In the following, the basic characteristics of doing logic (and thus circuit evaluation) with matrices are explored. Highlights include the concepts of a generalized negation as compliment, the natural occurrence of imaginary logical values, and the introduction of negative logical states.

Matrix Bit Operations

Binary addition, implemented by XOR:

```
x  +  y  =  z
0  +  0  =  0
0  +  1  =  1
1  +  0  =  1
1  +  1  =  0     plus complexity of carry
```

The matrix logic XOR operator takes the form of the generalized complement:

```
     y=0 y=1
x=0 [ 0   1 ]
x=1 [ 1   0 ]
```

Binary multiplication, implemented by AND:

```
x  *  y  =  z
0  *  0  =  0
0  *  1  =  0
1  *  0  =  0
1  *  1  =  1
```

Matrix logic AND has the form:

```
     y=0 y=1
x=0 [ 0   0 ]
x=1 [ 0   1 ]
```

Place Notation

For computation, we encode large numbers by expressing them as a power series. Binary notation treats a string of bits as defining an additive sequence of powers of two.

```
decimal 11  ==binary==>  1011  =  1*2^3 + 0*2^2 + 1*2^1 + 1*2^0
```

Power notation can be expressed in matrix form, decoupling the power series from the characteristic bits.

```
[1 0 1 1]   [ 2^3 ]
            [ 2^2 ]
            [ 2^1 ]
            [ 2^0 ]
```

or in bra-ket notation:

```
< characteristic-bits | power-series >  =  sum[ bit[i]*power[i] ]
```

Logics as Matrices

The shape of a logic function of two variables is a 2x2 square table of resultant bits. There are 16 (2^4) possible varieties, all of the form:

```
[ bit1  bit2 ]
[ bit3  bit4 ]
```

Logical ground values (ie True and False) are two-bit vectors. The two orthogonal truth values are then

```
False = [ 1 0 ]  =  < 0 |
True =  [ 0 1 ]  =  < 1 |
```

In general, an evaluation of a boolean operator B over two boolean ground values, x and y, can be expressed as:

```
< x | B | y >
```

We can examine the 16 binary logic operators for complements, symmetries, transposes, etc. This is equivalent to putting them into a boolean lattice.

Name	*Distinction*	*Flat-matrix*	*Bra-ket*
tautology	( )	1111	< x | 1 | y >
nand	(a)(b)	1110	1 - < x | 1 >< 1 | y >
implies	(a) b	1101	1 - < x | 1 >< 0 | y >
conv-implies	a (b)	1011	1 - < x | 0 >< 1 | y >
or	a b	0111	1 - < x | 0 >< 0 | y >
var	a	0011	< x | 1 >
conv-var	b	0101	< 1 | y >
equiv	(a b)((a)(b))	1001	< x | y >
xor/negation	((a b)((a)(b)))	0110	1 - < x | y >
conv-var-neg	(b)	1010	< 0 | y >
var-neg	(a)	1100	< x | 0 >
nor	(a b)	1000	< x | 0 >< 0 | y >
conv-inhibit	(a (b))	0100	< x | 0 >< 1 | y >
inhibit	((a) b)	0010	< x | 1 >< 0 | y >
and	((a)(b))	0001	< x | 1 >< 1 | y >
empty		0000	< x | 0 | y >

Generalized Negation as Complement

We use the mark of distinction to represent negation/complementation for all matrix orders. The mark provides a singular notational extension, equivalent to any kind of overbar.

```
scalar      (x)    =  1 - x    =  not x

vector      |(x)>  =  1 - |x>  =  (  )|x>

            <(x)|  =  1 - <x|  =  <x|(  )

operator    (L)    =  1 - L    ≠  (  ) L
```

Generalized negation is the complement of identity (logical XOR):

```
      [1 1]  -  [1 0]  =  [0 1]
      [1 1]     [0 1]     [1 0]
```

We can now express the complementation/negation operation of matrix algebra in mixed boundary-bra-ket form:

```
< x | y >  =def=  < x | equiv | y >
< x | x >  =def=  I
< x | 1 >    =    x
< x | 0 >    =   (x)

(   | x > ) =     | (x) >  =  1 - |x>  =  not |x>  =  (  ) |x>
( B | x > ) = (B) | x >
(< x | B  ) = < x | (B)

< x | y >         =  < (x) | (y) > = (( < x | y > ))
< x | (  ) | y >  =  ( < x | y > ) = < (x) | y > = < x | (y) >

  | x >< y |   = B
  | y >< x |   = B-transpose
( | x >< y | ) = 1 - B = (B)

  < x | B | y >   =  < y | B-transpose | x >
  < x | B | y >   =    ( < x | (B) | y > )
( < x | B | y > ) =      < x | (B) | y >
```

Name	*Distinction*	*Flat-matrix*	*Boundary-Bra-ket*
tautology	( )	1111	< x | 1 | y >
nand	(a)(b)	1110	(< x | 1 >< 1 | y >)
implies	(a) b	1101	(< x | 1 >< 0 | y >)
conv-implies	a (b)	1011	(< x | 0 >< 1 | y >)
or	a b	0111	(< x | 0 >< 0 | y >)
var	a	0011	< x | 1 >
conv-var	b	0101	< 1 | y >
equiv	(a b)((a)(b))	1001	< x | y >
xor/negation	((a b)((a)(b)))	0110	(< x | y >)
conv-var-neg	(b)	1010	< 0 | y >
var-neg	(a)	1100	< x | 0 >
nor	(a b)	1000	< x | 0 >< 0 | y >
conv-inhibit	(a (b))	0100	< x | 0 >< 1 | y >
inhibit	((a) b)	0010	< x | 1 >< 0 | y >
and	((a)(b))	0001	< x | 1 >< 1 | y >
empty		0000	< x | 0 | y >

Note also:

1 = (0), 0 = (1)

( ) | 0 > = | 1 >

<x|0><0|y> = <x|(1)><(1)|y> = <(x)|1><1|(y)>

Symmetry

Matrix logic operators expose the symmetries in logic:

Operator	*Complement*	
implies	inhibit	
conv-imp	conv-inhibit	*Asymmetric*
a	not-a	
b	not-b	
or	nor	
nand	and	*Symmetric*
equiv	xor	
empty	taut	

Imaginary Logical Values

During the course of a matrix evaluation, intermediate vectors which have no traditional logical interpretation occur.

```
[1 0] [1 1]  =  [1 1]
      [0 0]
```

while

```
[1 0] [1 1] [1]  =  1
      [0 0] [0]
```

These can be interpreted as imaginary values,

```
[1 1]    underconstrained,  both True and False
[0 0]    overconstrained, neither True nor False
```

Only matrix equivalence and complementation are free of these imaginaries as intermediate computational forms.

Decomposition of Logical Operators

Addition and multiplication of operators is defined in matrix notation, so these operations can be used to convert between sets of logical operators, for instance in converting to selected basis operators. Egs:

"not + and = or"

```
[0 1] + [0 0]  =  [0 1]
[1 0]   [0 1]     [1 1]
```

"nor squared = nor"

```
[1 0] * [1 0]  =  [1 0]
[0 0]   [0 0]     [0 0]
```

"square-root of equal" (compliment is its own inverse)

```
[0 1] * [0 1]  =  [1 0]
[1 0]   [1 0]     [0 1]
```

Some operator combinations introduce "illegal" logic matrices.

"or + or = ?"

```
[0 1] + [0 1]  =  [0 2]  =  2 [0 1]
[1 1]   [1 1]     [2 2]       [1 1]
```

"or * or = ?"

```
[0 1] * [0 1]  =  [1 1]
[1 1]   [1 1]     [1 2]
```

Parsing between Boundary Notation and Bra-ket Notation

( ) = <1|

x = <x|1> = (<x|0>) = <x|( )|0>

x y = <x|(0><0)|y> = (<x|0><0|y>) = ((<x|1>) (<1|y>))

((x)(y)) = <x|1><1|y>

[NOTE: composition rules not yet fully understood]

Negative Logical Operations

The inverse of a 2x2 logical operator represents a functional reversal of input and output. Most of the logical operators do not have an inverse.

Self-inverse

equivalent
xor/negation/complement

Inverse with negative cell

or-inverse	[-1 1] [1 0]	X Y
implies-inverse	[1 -1] [0 1]	(X) Y
conv-implies-inverse	[1 0] [-1 1]	(Y) X
nand-inverse	[0 1] [1 -1]	(X)(Y)

No inverse (determinant is equal to zero)

tautology, empty
var, conv-var, conv-var-neg, var-neg
nor, conv-inhibit, inhibit, and

The interpretation of negative logical operations is an area for exploration. Several techniques might be considered:

1. *Negative case counts:*

Interpret the entries in a logical operator matrix B as case counts of events given possible value assignments.

2. *Logical inverses:*

Six logical operators have inverses (are reversible). Reversing an operator generates an inverse-truth table:

x	OR-inv	y	=	z
0	or-1	0	=	-1
0	or-1	1	=	1
1	or-1	0	=	1
1	or-1	1	=	0

The scalar result of an inverse operation can be a new scalar logical value, -1. As well, new logical vectors occur as intermediate results:

(-1 1), (1 -1), (-1 -1), (0 0)

3. *Logical vectors:*

The domain of logical constants can be expanded to include the new logical vectors. If logic is interpreted as a vector space, then vector calculus itself introduces negative logical vectors.

neg-True	-(0 1)	=	(0 -1)
neg-False	-(1 0)	=	(-1 0)

4. *Multiple valued logic:*

We could introduce a three valued scalar logic:

{-1, 0, 1}

5. *Subtraction of matrices:*

Since complementaton is defined in terms of matrix subtraction,

1 - B = (B)

we could permit unconstrained subtraction as a generator of operators with negative entries. For example:

"equivalence - negation"

```
[1 0]  -  [0 1]  =  [1 -1]
[0 1]     [1 0]     [-1 1]
```

6. *Non-commutativity:*

Non-commutativity is defined over operators by

Bi*Bj - Bj*Bi ≠ 0

The non-zeroness defined by non-commuting logic matrices is expressed by operators with negative entries. For example:

"and*or - or *and"

```
[0 0] [0 1] - [0 1] [0 0]  =  [0 0] - [0 1]  =  [0 -1]
[0 1] [1 1]   [1 1] [0 1]     [1 1]   [0 1]     [1  0]
```