CWP-108
April 1991

Mathematica Packages for Logic and Set Theory

by

Jack K. Cohen

Center for Wave Phenomena
Colorado School of Mines
Golden, Colorado 80401
(303)273-3557

ABSTRACT

The evaluation of two-valued boolean algebraic expressions can be reduced to straight forward calculations. Furthermore, Mathematica supplies tools that make an automated implementation concise and elegant: the Mathematica package is only 10 simple lines of active code. Specific implementations are given for the cases of set theory and logic.

THE CALCULUS

A calculus for the evaluation of expressions over a two-valued boolean algebra with applications to both logic and set theory may be elegantly implemented with Mathematica. To describe the calculus [Watts and Cohen, 1970, Sawyer, 1955], it is helpful to consider a specific boolean algebra, say, set theory. The usual characteristic function of a set A is defined as:

$$\hat{A} = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases} \tag{1}$$

Theorem 1 *Two sets are equal if and only if their characteristic functions are equal.*

This theorem is easily established by considering the four cases of an element being in one, both, or neither of the sets.

Because the characteristic functions have only the values 0 and 1, it is natural to think of combining them by arithmetic modulo the base 2. Indeed, the common binary operation of intersection maps into multiplication modulo 2 for the characteristic functions. The operation corresponding to addition modulo 2 is the symmetric difference of two sets:

$$A \triangle B = (A - B) \cup (B - A) \tag{2}$$

In terms of these two operations, the commonly used set operations are described by a simple table of of characteristic function transforms, Table 1. Some readers might prefer to use minus signs in some of the transform results—and this is acceptable because all operations on characteristic functions are done modulo 2. Here, the symbol, $'$, denotes the complement operation. The remaining part of the proposed calculus is embodied in the following two simple results.

Theorem 2 $\hat{A}\hat{A} = \hat{A}$.

Theorem 3 $\hat{A} + \hat{A} = 0$.

Table 1. Set Theory Transforms

$\emptyset$	0
universe	1
A'	$1 + \hat{A}$
$A \triangle B$	$\hat{A} + \hat{B}$
$A \cap B$	$\hat{A}\hat{B}$
$A \cup B$	$\hat{A} + \hat{B} + \hat{A}\hat{B}$
$A - B$	$\hat{A} + \hat{A}\hat{B}$

These simplification theorems can also be established by considering cases: $x \in A$ and $x \notin A$. Alternatively, recognize that the theorems correspond to well known set identities.

The calculus consists of transforming a given set expression using the above table and then applying Theorems 2 and 3 repeatedly. This yields a canonical expression in terms of the symmetric difference, intersection and complementation operations. Moreover, to establish set identities simply combine the proposed equivalents with the symmetric difference operation and apply the calculus—an actual identity will simplify to 0. In general, the canonical form is a representation as a *disjoint* union of intersections. Often a simpler form can be recognized from the right side of the transform table.

Before describing the applications of the characteristic calculus to logic, a brief summary of logic symbolism is needed. In logic, the transform corresponds to truth value. Further, the complement operation is denoted by $\neg$ (not) and the multiplication operation by $\wedge$ (and). The addition operation is "exclusive or" which we denote by the symbol $\not\equiv$ [Massey, 1970]. Though the commonly used operations of logic differ from those of set theory, the transform table for logic (Table 2) is similar.

The simplification Theorems 2 and 3 and their application are unchanged. In this context, the canonical form is akin to the classical disjunctive normal form, but the disjunction is with the "exclusive or" instead of the "inclusive or".

Mathematica provides an excellent environment for implementing Theorems 2 and 3 to produce either set theory or logic transforms in canonical form. An example from logic shows how a verification of DeMorgan's first law appears in Mathematica. In the Mathematica implementation, the hat notation of Tables 1 and 2 is dropped in favor of an intuitive infix notation for the binary operations together with a function "not[ ]" for logical negation.

Compare the canonical representations forms of the two sides of DeMorgan's first law produced by the Mathematica logic implementation:

```
d1 = reduce[not[a] ~and~ not[b]]
1 + a + b + a*b

e1 = reduce[not[a ~or~ b]]
1 + a + b + a*b
```

Because the two forms are the same the law is established. Alternatively, one could form the "exclusive or" of the two sides to completely automate the verification:

```
reduce[d1 ~xor~ e1]
0
```

The "0" returned above shows that the two sides of DeMorgan's law have the same truth value. Similarly, we can verify the other DeMorgan law:

```
reduce[(not[a] ~or~ not[b]) ~xor~ not[a ~and~ b]]
0
```

The "reduce[ ]" function used above to obtain the canonical form of logic expressions will be defined in the next section. This is followed by a section with additional examples. The complete Mathematica packages for logic and for set theory are given in appendices A and B respectively.

THE MATHEMATICA IMPLEMENTATION

The basic idea is to implement Theorems 2 and 3 and then hand the simplification chore to Mathematica's "Simplify" function. A first attempt might be to directly implement Theorems 2 and 3 by altering the built-in Plus and Times functions:

Table 2. Logic Transforms

absurdity	0
tautology	1
$\neg p$	$1 + \hat{p}$
$p \not\equiv q$	$\hat{p} + \hat{q}$
$p \wedge q$	$\hat{p}\hat{q}$
$p \vee q$	$\hat{p} + \hat{q} + \hat{p}\hat{q}$
$p \rightarrow q$	$1 + \hat{p} + \hat{p}\hat{q}$
$p \leftrightarrow q$	$1 + \hat{p} + \hat{q}$

```
Unprotect[Plus]
        Plus[a_, a_] := 0
Protect[Plus]
Unprotect[Times]
        Times[a_, a_] := a
Protect[Times]
```

Unfortunately it soon becomes apparent that applying "Simplify" leaves terms like $2pq$ or p^2q^3 and, worst of all, the atom 2. All but the last problem can be overcome by further tinkering with the arithmetic operations—however a better approach is to implement Theorem 2 by using Mathematica's built-in "Mod" function to manipulate the polynomials in the logical variables. Finally, to obtain a disjunctive canonical form, the built-in "Expand" is applied—this decision is admittedly debatable.

These considerations lead to the definition of the "reduce[ ]" function:

```
        reduce[x_] := Expand[Simplify[Mod[x, 2]]].
```

With Theorem 2 implemented in terms of Mathematica primitives, only *one* alteration in the standard arithmetic is needed to implement Theorem 3:

```
Unprotect[Power]
        Power[a_, n_Integer] := a   /; n > 1
Protect[Power]
```

The remainder of the Mathematica packages merely express the right sides of the transform tables—see the Appendices.

EXAMPLES

As a first example, consider the construction of "or" from "xor" and "and". This follows as:

```
reduce[(p ~xor~ q) ~xor~ (p ~and~ q)]
p + q + p*q
```

From the transform table, recognize that the construction is correct.

Next, verify some of the fundamental laws of reasoning. First treat the syllogism $(p \to r,\ r \to q) \Rightarrow (p \to q)$:

```
reduce[ ((p ~implies~ r) ~and~ (r ~implies~ q)) ~implies~
          (p ~implies~ q) ]

1
```

The result "1" indicates that the proposition is, indeed, a tautology.

Similarly verify two additional well-known tautologies. Consider the statement, $(\neg q \rightarrow \neg p) \Rightarrow (p \rightarrow q)$:

```
reduce[ (not[q] ~implies~ not[p]) ~implies~
        (p ~implies~ q) ]

1
```

and finally consider $(p \rightarrow q,\ q \rightarrow p) \Rightarrow (p \leftrightarrow q)$:

```
reduce[ ((p ~implies~ q) ~and~ (q ~implies~ p)) ~implies~
        (p ~iff~ q) ]
1
```

The Mathematica solutions of some typical elementary exercises found in textbooks on logic or discrete mathematics are shown next. In each case, the comment gives the method suggested in the textbook.

```
(* Construct a truth table for a given compound statement *)
f[p_, q_, r_, s_] :=
         reduce[ (p ~implies~ q) ~or~ (r ~iff~ s) ]

Do [
    Do [
        Do [
            Do [
                Print[p, "        ",
                      q, "        ",
                      r, "        ",
                      s, "                ",
                      f[p,q,r,s]],
                {s, 0, 1}],
        {r, 0, 1}],
    {q, 0, 1}],
{p, 0, 1}]
0 0 0 0  1
0 0 0 1  1
0 0 1 0  1
0 0 1 1  1
0 1 0 0  1
0 1 0 1  1
```

```
0 1 1 0  1
0 1 1 1  1
1 0 0 0  1
1 0 0 1  0
1 0 1 0  0
1 0 1 1  1
1 1 0 0  1
1 1 0 1  1
1 1 1 0  1
1 1 1 1  1


(* Decide if Tautology (1), Absurdity (0) or Contingency *)
reduce[ not[p] ~or~ p ]
1

reduce[ not[p] ~and~ p ]
0

reduce[ not[p ~implies~ q] ~implies~ (q ~implies~ not[p]) ]
1

reduce[ ((not[p] ~and~ not[q]) ~or~ (p ~and~ q)) ~iff~
         (p ~iff~ q) ]
1

reduce[ (p ~iff~ q) ~implies~ (p ~or~ q) ]
p + q + p*q

reduce[ (p ~iff~ q) ~implies~ (not[p] ~iff~ not[q]) ]
1


(* Prove with truth tables *)
reduce[((p ~or~ q) ~and~ not[q]) ~implies~ p]
1

reduce[ ((p ~implies~ q) ~and~ not[q]) ~implies~ not[p] ]
1


(* Give direct and indirect proofs *)
reduce[ ( (a ~implies~ b) ~and~ (c ~implies~ b) ~and~
```

```
        (d ~implies~ (a ~or~ c)) ~and~ d ) ~implies~ b ]
1

reduce[( not[q] ~and~ (p ~implies~ q) ~and~ (p ~or~ t) ) ~implies~ t]
1


(* Simplify by using a Karnaugh Map *)
reduce[
    (not[x] ~and~ not[y] ~and~ not[z]) ~or~
    (x ~and~ y ~and~ z) ~or~
    (x ~and~ not[y] ~and~ z) ~or~
    (not[x] ~and~ not[y] ~and~ z) ~or~
    (not[x] ~and~ y ~and~ z) ~or~
    (not[x] ~and~ y ~and~ not[z])
]

1 + x + x*z
```

The final result was given in the text as $\neg x \vee z$ which is easily seen to be equivalent to the Mathematica result albeit in a different normal form—from the transform Table 2, the expression $x \rightarrow z$ is seen to be another and more terse equivalent.

Finally, here are some results for the set theory implementation. Note that one letter mnemonics are used: "u" is union, "i" is intersection, etc.

```
(* Check DeMorgan's Laws *)
d1 = reduce[c[a] ~i~ c[b]]
1 + a + b + a*b

e1 = reduce[c[a ~u~ b]]
1 + a + b + a*b

reduce[d1 ~s~ e1]
0

d2 = reduce[c[a] ~u~ c[b]]
1 + a*b

e2 = reduce[c[a ~i~ b]]
1 + a*b

reduce[d2 ~s~ e2]
0
```

CONCLUSIONS

Mathematica implementations of both set theory and logic have been presented. It is a tribute to the design of Mathematica that the packages are only a few lines long and yet automate many of the typical tasks in these fields. Indeed, I was motivated to write this paper because of the stunning contrast between the ease of the Mathematica implementation and the difficulties Dale Watts and I encountered in our procedural language implementations (Algol and Fortran) a decade ago.

ACKNOWLEDGMENTS

I wish to acknowledge stimulating discussions with both John Stockwell and Jerry Dozoretz that enhanced the text appreciably. I am also grateful to Stockwell for a superb editing job on the manuscript.

REFERENCES

Massey, G. J., 1970, Understanding Symbolic Logic: Harper & Row Pub., New York.

Sawyer, W. W., 1955, Prelude to Mathematics: Pelican Books, Baltimore.

Watts, D., and J. K. Cohen, 1980, Computer-Implemented Set Theory: Amer. Math. Monthly, Vol. 87, No. 7, 557-560.

APPENDIX A: MATHEMATICA LOGIC IMPLEMENTATION

```
BeginPackage["logic`"];

(* Declaration of public function names in logic package *)

not::usage = "not[x]"
and::usage = "x ~and~ y"
xor::usage = "x ~xor~ y  (exclusive or)"
or::usage = "x ~or~ y"
implies::usage = "x ~implies~ y"
iff::usage = "x ~iff~ y "
reduce::usage = "reduce[x]  (simplifies expressions)"

Begin["`private`"];
        Unprotect[Power]
                Power[a_, n_Integer] := a   /; n > 1
        Protect[Power]

        reduce[x_] := Expand[Simplify[Mod[x, 2]]]

        not[x_] := 1 + x
        and[x_, y_] := x y
        xor[x_, y_] := x + y
        or[x_, y_] := x + y + x y
        implies[x_, y_] := 1 + x + x y
        iff[x_, y_] := 1 + x + y
End[];
Endpackage[];
```

APPENDIX B: MATHEMATICA SET THEORY IMPLEMENTATION

```
BeginPackage["set`"];

(* Declaration of public function names in set theory package *)

c::usage = "c[x]   (complement of x)"
i::usage = "x ~i~ y   (intersection of x and y)"
s::usage = "x ~s~ y   (symmetric difference: (x-y) U (y-x))"
u::usage = "x ~u~ y   (union of x and y)"
d::usage = "x ~d~ y   (difference of x and y:  (x - y))"
reduce::usage = "reduce[x]   (simplifies expressions)"

Begin["`private`"];

        Unprotect[Power]
                Power[a_, n_Integer] := a   /; n > 1
        Protect[Power]

        reduce[x_] := Expand[Simplify[Mod[x, 2]]]

        c[x_] := 1 + x
        i[x_, y_] := x y
        s[x_, y_] := x + y
        u[x_, y_] := x + y + x y
        d[x_, y_] := x + x y
End[];
Endpackage[];
```