Efficiency in Boolean Expressions

by Winfried Gerum

Winfried Gerum

Binary logic is supposed to be a big strength for computers (unfortunately, the same is not true for humans). It is much simpler than addition and multiplication. A simple *truthtable* in principle tells you everything to be told about the main Boolean operations:

A	B	A AND B	A OR B
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE

in M

A	B	A&B	A!B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Occasionally programmers pick the wrong one of these operators, because in common parlance **and** or **or** are not used as precisely as **And** or **Or** in formal logic. A quick glance at the above table tells you that writing an AND instead of an OR gives you the same result in half the cases. This is the reason that in a test run of a routine you may get the proper result despite a programming error. So be careful when using these operators.

In M, all binary operators evaluate both operands, even if the first operand alone determines the result:

```
0*A     always 0
1**N    always 1
0&A     always 0
0'&A    always 1
1!A     always 1
0'!B    always 0
```

With the multiplication or the exponentiation the exception seems to be a rare case that does not need special attention. But with the Boolean operators, the first operand alone determines the result in 50 percent of the cases. That situation indeed warrants some attention. Most programming languages do not pay special attention to this. The C programming language actually avoids evaluation of the second argument in Boolean operations. Programmers who are not aware of this may get funny results when writing code such as:

```
if ((*a <-'A')||(*a++>='Z'))(...);
```

the pointer a is not incremented (a++) unless the right side of the OR (||) is evaluated.

In standard M, as stated above, both sides always are evaluated. This is more reasonable if one wants to reduce side effects. But it is not without surprise either:

```
IF $D(A)&(A>5)
```

this will give an error in standard M, if $D(A)[0 (i.e., unsubscripted A is undefined). It will work under GTM (Greystone Technology), however.

There is no point in arguing for a change in the language standard in this regard. Such a change would not be backward compatible due to error processing and the notorious naked reference.

If A and B in the above examples stand just for simple local variables, we need not make much fuss about it. But they stand for expressions of arbitrary complexity, which might be worthwhile not to evaluate.

Logical AND can be done efficiently in IF arguments in a straightforward way:

```
IF A&B ;Mcode
```

may be replaced by

```
IF A,B ;Mcode
```

The therorem of deMorgan

```
A AND B = NOT ((NOT A) OR (NOT B))
```

tells us that for every theorem on AND there is a dual theorem for OR (and vice versa). Applying this to

```
IF A!B ;Mcode
```

you get

```
IF 'A'&'B ;Mcode
```

which may be rewritten as

```
IF 'A&'B
ELSE ;Mcode
```

and, finally:

```
IF 'A,'B
ELSE  ;Mcode
```

But be careful. $TEST is different after this replacement code (use IF '$TEST instead of ELSE to get the same $TEST value).

If you think IF 'A,'B is confusing, then replace IF A!B by

```
IF A
ELSE IF B
IF  ;Mcode
```

These methods can be used to restate IF arguments of great complexity without & or !. For example,

```
IF A&B!C!D
```

goes to

```
IF A,B
ELSE IF C
ELSE IF D
IF ;Mcode
```

Logical AND in the postconditional of a DO, GOTO, or XECUTE is as simple as splitting the expressions between command and argument.

```
DO:A&B LABEL
```

is to be replaced by

```
DO:A LABEL:B
```

If you cannot refrain from using GOTO you might see the situation of GOTO

```
GOTO:A!B LABEL
```

We possibly avoid the evaluation of B by repeating the argument with both conditions:

```
GOTO LABEL:A,LABEL:B
```

Using the same replacement for DO or XECUTE is improper:

```
DO LABEL:A,LABEL:B
```

might result in two calls to LABEL.

In the situation of QUIT

```
QUIT:A!B
```

you simply repeat the command with both conditions:

```
QUIT:A  QUIT:B
```

In arbitrary situations it is always possible to avoid the evaluation of B with the help of the $SELECT function:

A&B may be replaced by

```
$SELECT('A:0,'B:0,1:1)
```

or by

```
$SELECT(A:B,1:0)
```

(if B is Boolean or if the result is subject to Boolean interpretation).

A!B may be replaced by

```
$SELECT(A:1,B:1,1:0)
```

or by

```
$SELECT(A:1,1:B)
```

(if B is Boolean or if the result is subject to Boolean interpretation).

A'&B may be replaced by

```
$SELECT('A:1,'B:1,1:0)
```

or by

```
$SELECT('A:1,1:'B)
```

A'!B may be replaced by

```
$SELECT(A:0,B:0,1:1)
```

or by

```
$SELECT(A:0,1:'B)
```

That trick may be generalized to multiple ANDs or ORs:

```
A&B&C  -> $S('A:0,'B:0,'C:0,1:1)
A!B!C  -> $S(A:1,B:1,C:1,1:0)
A&B!C  -> $S(C:1,'A:0,'B:0,1:1)
A!B&C  -> $S('C:0,A:1,B:1,1:0)
...
```

While the replacements in these examples look more complex, in many real-world problems they may be even more readable. A typical mind does not grasp complex Boolean expressions very efficiently. Therefore we frequently see an improvement in readability if any expression with many ANDs or ORs is split up into convenient morsels.

If the evaluation of subexpressions can be side-stepped, look carefully for which one to avoid. Among the criteria are:

- Complexity of subexpression (e.g., extrinsic functions);
- Evaluation time (extrinsic function, global); and
- Among equally "expensive" subexpressions, place the expression first in $SELECT, which evaluates TRUE more often than the other subexpression.

Readability and reliability are intimately connected. So apply these tricks only in a way that does not produce obscure code.

Winfried Gerum is with Winner Software GmbH located in Röttenbach, Germany. Contact him by phone at 49-9195-940022 or by fax at 49-9195-940030. His column appears regularly in *M Computing* and in European publications on M.