**Multi-Valued Logic**

David Kantor

[Kantor.d@att.net](mailto:Kantor.d@att.net)

2017jun23, rev 2017jun26

Several years ago, I gave a presentation on three-valued logic and a suite of programs that effect such calculations in Stata. I’d now like to discuss further developments of those ideas; we will start with a brief synopsis of the original three-valued logic presentation.

In Stata, expressions such as

a | b

a & b

have a risk that we are all warned about: they operate as if missing values were equivalent to *true* (or 1).

a | b yields *true* if either a or b is true or missing.

a & b yields *true* if both a and b are true or missing.

In fact, in a logical expression, any missing value functions as if it were *true*. This is probably not what you want to happen. (Note also that even an expression with no operations – a – is interpreted as true for missing values in a conditional context: … if a … but we will not focus on that matter here.)

Before going further, note that there are often situations in which you are fortunate enough that you can be certain that the operands are not missing; you can assert that fact before using them:

assert ~mi(a)

assert ~mi(b)

gen byte c = a | b

This is applicable in some situations; indeed it may be used often. But it is not always applicable.

If there are potentially missing values present, one solution is to safeguard the use of such expressions with conditions:

gen byte c = a | b if ~mi(a) & ~mi(b)

Thus, the result is missing if any of the operands are missing, thereby guarding against erroneous (false positive – or “falsely true”) results. But this may be too restrictive; it is the Draconian solution. It solved the correctness problem, but it introduces inefficiency. You can be more inclusive in evaluating such expressions, by observing that, in two-valued logic,

* for disjunction (|), if one operand is *true*, then the result is *true*, regardless of what the other value is;
* for conjunction (&), if one operand is *false*, then the result is *false*, regardless of what the other value is.

Thus, even if some missing values are present in the operands, you can sometimes derive a nonmissing result. Thus, we can derive “classic” three-valued logic.

|  |  |  |  |
| --- | --- | --- | --- |
| or | 0 | 1 | . |
| 0 | 0 | 1 | . |
| 1 | 1 | 1 | 1 |
| . | . | 1 | . |

|  |  |  |  |
| --- | --- | --- | --- |
| and | 0 | 1 | . |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | . |
| . | 0 | . | . |

|  |  |
| --- | --- |
| A | Not a |
| 0 | 1 |
| 1 | 0 |
| . | . |

But there is another possibility; you can also evaluate expressions using a “liberal” protocol, in which a missing value stands for nothing; it lets the other non-missing values prevail:

|  |  |  |  |
| --- | --- | --- | --- |
| or | 0 | 1 | . |
| 0 | 0 | 1 | 0. |
| 1 | 1 | 1 | 1 |
| . | 0 | 1 | . |

|  |  |  |  |
| --- | --- | --- | --- |
| and | 0 | 1 | . |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| . | 0 | 1 | . |

Note that, under the liberal protocol, a missing value in a binary logical operation serves as a universal identity element. (This presents some problems, which we will return to, later.)

So we have seen three ways of dealing with the possible presence of missing values in logical operations:

* Draconian
* Standard three-valued logic operations
* Liberal three-valued logic operations

But having these choices, you might wonder which protocol to use in any situation. Thus, the problem has become murkier, rather than clearer. To resolve this situation, it may help to ask, how *should* such expressions be evaluated?

* 1 | .
* 0 | .
* 1 & .
* 0 & .

Classic three-valued logic seems ostensibly right, but it is actually based on an assumption that we have, until now, elided. We stated earlier…

* for disjunction (|), if one operand is *true*, then the result is *true*, regardless of what the other value is;
* for conjunction (&), if one operand is *false*, then the result is *false*, regardless of what the other value is.

And when we extended this to missing values, we assumed that a missing value stands for an actual classical value (true or false); we just don’t know what the value is. You can call it a hidden or unknown value. (Suppose it is true; suppose it is false. What results do you get? Are they consistent? If so, then use that result.)

The liberal protocol is just another, different assumption about missing values. It supposes that a missing value is something that “doesn’t exist”; it will disappear when combined with other things; it is vacuous.

And the Draconian protocol: while we may not like it, we should realize that it is yet another interpretation of missing values: there is something so wrong with this operand that it infects the whole expression.

Having established these notions, we can now answer the question: how *should* such expressions be evaluated?

* 1 | .
* 0 | .
* 1 & .
* 0 & .

The answer is: it depends on what you mean by missing.

The next step is to realize that each of the meanings can correspond to a distinct missing value. That is, rather than specifying the missing-value protocol for an operation, we can let the missing value itself carry that information; this is facilitated by Stata allowing extended missing values. (Note that I wrote the original three-valued logic programs prior to the introduction of extended missing values.)

Furthermore, these various species of missing values can coexist in one computational system.

Thus, suppose that we agree to use the following:

* Draconian: .b (for a bad value)
* Unknown: .u
* Liberal: .v (for vacuous)

Then we can construct more complex expressions, such as (a | b) & c, and evaluate them accordingly. For example,

* (1 | .u) & .v = 1
* (1 | .u) & .u = .u
* (1 | .b) & .u = .b

Here are the complete operational tables:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| or | 0 | 1 | .u | .v | .b |
| 0 | 0 | 1 | .u | 0 | .b |
| 1 | 1 | 1 | 1 | 1 | .b |
| .u | .u | 1 | .u | .u | .b |
| .v | 0 | 1 | .u | .v | .b |
| .b | .b | .b | .b | .b | .b |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| and | 0 | 1 | .u | .v | .b |
| 0 | 0 | 0 | 0 | 0 | .b |
| 1 | 0 | 1 | .u | 1 | .b |
| .u | 0 | .u | .u | .u | .b |
| .v | 0 | 1 | .u | .v | .b |
| .b | .b | .b | .b | .b | .b |

|  |  |
| --- | --- |
| A | not a |
| 0 | 1 |
| 1 | 0 |
| .u | .u |
| .v | .v |
| .b | .b |

How would you use these operations? Suppose you have information on the gender of children in a household. Suppose this is in wide form, in variables f1, f2, f3, f4, f5, f6, f7, f8; f1 indicates that child 1is female, and so on. Sysmis can stand for an unknown gender; let .v be the value used when there is no child in the given position. Thus, if there are only three children, f1, f2, and f3 are given either 0 or 1, or sysmis if the gender is unknown; f4 through f8 would be given a value of .v. Then we can express the following calculations.

* At least one child is female: f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8
* Every child is female: f1 & f2 & f3 & f4 & f5 & f6 & f7 & f8
* At least one child is male: ~f1 | ~f2 | ~f3 | ~f4 | ~f5 | ~f6 | ~f7 | ~f8
* Every child is male: ~f1 & ~f2 & ~f3 & ~f4 & ~f5 & ~f6 & ~f7 & ~f8

Stata does not work this way, but you can use egen functions to accomplish effect these calculations. And of course, expressions with both & and | need to be done in stages.

Some notes on these operations: The binary operations are associative; thus, the expressions shown above make sense. They preserve DeMorgan’s Laws. The do *not* preserve the distributivity properties; the vacuous values get in the way of that. The vacuous value is the culprit there. For example, 1&(.v|0) is 0, while (1&.v) | (1&0) is 1. This does not mean that you cannot evaluate expressions that combine & and |; it just means that some of the familiar algebraic transformations should not be applied.

**Beyond Logic**

These ideas originated in the context of logical operations, but they apply to arithmetic as well. Just as we asked how to evaluate logical operations involving missing values, so too must we ask how to evaluate such expressions as…

* 3 + .
* 7 \* .
* 0 \* .
* 12 / .
* sum(4, 17, 30, 12, .)
* mean(4, 17, 30, 12, .)

Again, the answer is that it depends on what you mean by missing.

If you mean “unknown’, then most of those expressions should yield unknown. (There are two exceptions, which we will get to soon.)

If you mean, vacuous, then the results are 3, 7, 0, 12, 63, and 15.75.

If you mean the bad/Draconian interpretation, then they all evaluate to the “bad” value.

Again, you can use specific extended missing values to signify different meanings. If we adopt those same standards, then we see...

* 3 + .u = .u
* 3 + .v = 3
* 3 + .b = .b
* mean(4, 17, 30, 12, .u) = .u
* mean(4, 17, 30, 12, .v) = 15.75
* mean(4, 17, 30, 12, .b) = .b

It is worth noting two special situations:

* 0 \* .u = 0
* 12 / .u = .b

The first is clear, but demands some explanation. If you had written a generic missing value, without being specific about what it means, then it is not clear. But with the unknown value, then the product is 0. With a generic / nonspecific value, it might be the result of division by 0, in which case it is really a bad value; you can’t assert that multiplying it by 0 will yield 0. I had once asked Stata Tech Support why 0\*. wasn’t evaluated as 0; the reply was that same story about division by zero; that was implicitly, though maybe not explicitly about different meanings of missing values.

For the second situation: 12 divided by an unknown value would usually be unknown – unless that unknown value happens to be 0. Since it might be 0, the result might be bad.

Another point worth noting is that when Stata evaluates sums and means – in the sum() function, or the egen sum, rowtotal, mean, and rowmean functions, or summarize – it applies the Liberal protocol; it treats missing values as vacuous. Documentation states “treating missing values as zero” in the case of sums, or “ignoring missing values”, in the case of means. What this really means more generally, is that it treats missing values as vacuous.

One might imagine creating a computational system that embodies these ideas. Then we could evaluate such expressions as…

* (12 + 3 +.v) \* 2 = 30
* (12 + 3 +.v) \* .u = .u
* ((7 + .v) \* 2 - 14) \* .u = 0

Of course, Stata doesn’t operate this way. This is what I would do if I were constructing my own system.

In the case of logical operations, I have created egen functions to do this work; again, they can only handle one species of operation at a time. I might imagine doing the same for arithmetic; I have not tried that yet.

**Abstraction**

What we have seen is the extension of classic operations by mixing in additional special values. Let us refer to the standard or classical values as “normal”. Then we extend the operation by establishing that some of the additional values as “superior” to the normal values, and some are “inferior”. By “superior” and “inferior”, we mean that they have higher or lower precedence in the evaluation process.

* If a superior value is present, then it prevails.
* If an inferior value is present, ignore it, unless it the only value present (there are no higher-ranking values).

Under this scheme of understanding, we have.

* .b is superior
* .u is also superior, but not as much as .b
* .v is inferior

We can depict this in a lattice:

|  |
| --- |
| .b |
| .u |
| Normal values |
| .v |

Thus, we have that, not only are the extra values superior of inferior to the normal values, but they also have a ranking among themselves: .b superior to ..

Thus, we can generalize this to many levels:

|  |
| --- |
| sup n |
| sup\_n-1 |
|  |
| Sup 2 |
| Sup 1 |
| Normal values |
| Inf 1 |
| Inf 2 |
|  |
| Inf k |

The system we have explored here is a special case: two superior and one inferior value.

This abstraction is just that. I can’t think of a practical use beyond the three extra values discussed here. But I hope that this has given you a system of understanding – of things you may have already known, but may have not though about in such explicit terms.