Correcting A Widespread Error in Unification Algorithms

PETER NORVIG
Computer Science Division, University of California, Berkeley, CA 94720, USA

Summary

The unification of two patterns both containing variables is an ubiquitous operation in Logic Programming and in many Artificial Intelligence applications. Thus, many texts present unification algorithms. Unfortunately, at least seven of these presentations are incorrect. The common error occurs when logic variables are represented as binding lists; implementations that destructively update variable cells do not manifest the error. This note gives the examples that uncover the error and presents a correction.
KEY WORDS Unification Logic Programming Prolog Lisp

The Problem

Correct unification algorithms have been known and employed at least since Robinson's algorithm[1,2] was published in 1965. The problem with the Robinson algorithm is that it applies substitutions, replacing variables with their bindings, at each intermediate step of the algorithm. While correct, the algorithm creates many copies of intermediate structures, generating unnecessary garbage. There are two common solutions to this problem. The first, used by most PROLOG implementors, is to represent logic variables as cells that can be destructively updated. When a variable is unified with a value, the variable's cell is modified to point at the value. A trail of these modifications must be kept so that they can be undone later, if the computation must backtrack.

The other solution, taken by the authors of at least six texts[3,4,5,6,7,8], is to build up a substitution list, and only apply the substitution at the end of the algorithm. This approach doesn't generate intermediate garbage, but it may take more time, because it is necessary to search the substitution list for the values of variables. The trade-off is justified when patterns are large relative to the number of variables in them. It is not just text books that follow this model; the reasoning system MRS[9,10] also uses the same algorithm.

The following code is representative of the algorithm used in six of the sources[3,4,5,7,8,9]. The seventh[6] is similar, except that it, like Prolog, chooses not to handle the occurs check. To be concrete, COMMON LISP code is given rather than pseudo-code. Logical variables are restricted to the symbols `X`, `Y` and `Z`, and the result of a unification is either a valid substitution or the constant `fail`. Substitutions are implemented as association lists of variable/value pairs. The empty list represents a successful unification with no bound variables.

```
(defun unify (x y &optional subst)
  (cond ((equal x y) subst)
        ((equal subst 'fail) 'fail)
        ((var? x) (unify-variable x y subst))
        ((var? y) (unify-variable y x subst))
        ((or (atom x) (atom y)) 'fail)
        (t (unify (rest x) (rest y)
                  (unify (first x) (first y) subst)))))

(defun unify-variable (var val subst)
  "Unify var with val, using (and possibly extending) subst."
```

```
  (cond ((equal var val) subst)
        ((bound? var subst)
         (unify (lookup var subst) val subst))
        ((occurs-in? var val subst) 'fail)
        (t (extend-subst var val subst))))

(defun occurs-in? (var x subst)
  "Does var occur anywhere inside x?"
  (cond ((equal var x) t)
        ((bound? x subst)
         (occurs-in? var (lookup x subst) subst))
        ((consp x) (or (occurs-in? var (first x) subst)
                       (occurs-in? var (rest x) subst)))
        (t nil)))

(defun var? (x) "Is x a variable?" (member x '(X Y Z)))
(defun bound? (x subst) "Is x a bound variable?" (assoc x subst))
(defun lookup (var subst) (cdr (assoc var subst)))
(defun extend-subst (var val subst) (cons (cons var val) subst))
```

(Of course, the routines could be restructured to avoid the redundant searching of the subst list entailed by following a `bound?` check with a `lookup`. Similarly, some of the calls to `equal` might be replaced with `eq` or `eql`. The implementation is intended to be clear, but not optimal.)

Unfortunately, this algorithm (and each of the seven implementations) is incorrect. In unifying the variable `X` against an expression `y`, this code correctly checks for the cases where `X` equals `y`, and where `X` occurs within `y`, but if `y` is a variable that is bound to `X`, the function fails, as in the following example:

```
> (unify '(p X Y) '(p Y X))
FAIL
```

Here `p` would first match `p` creating the empty substitution; then `X` would be bound to `Y`, and with that substitution the algorithm would next check if `Y` occurs in `X`. Since `X` is bound to `Y`, the occurs check would return true, and the unification would fail.

It has been argued[5] that the unifier is not intended to be called with the same variables on both sides, but the same problem can arise even without sharing of variables between the two arguments:

```
> (unify '(q (p X Y) (p Y X)) '(q Z Z))
FAIL
```

In implementations that do not do the occurs check, the example above returns successfully:

```
> (unify '(p X Y) '(p Y X))
((Y . X) (X . Y))
```

While valid under the interpretation of non-iterated substitution, in practice the substitution list is used as a partial map, where substitutions must be done repeatedly until all bound variables are eliminated. Thus, while the call to `unify` terminates, an application of the resulting substitution would not. Another example makes this clear:

```
(unify '(p X Y a) '(p Y X X))
```

This example results in an infinite loop without the occurs check, and failure with the check, even though there is a valid unification where both variables are bound to `a`.

The Solution

Fortunately, there is a simple way to avoid these difficulties. The following version of `unify-variable` adds a single condition to eliminate circular binding lists:

```
(defun unify-variable (var val subst)
  "Unify var with val, using (and possibly extending) subst."
  (cond ((equal var val) subst)
        ((bound? var subst)
         (unify (lookup var subst) val subst))
        ;; New condition: dereference val when it is a variable
        ((and (var? val) (bound? val subst))
         (unify var (lookup val subst) subst))
        ((occurs-in? var val subst) 'fail)
        (t (extend-subst var val subst))))
```

This operation is called *dereferencing* because it replaces a variable with its value. Notice the symmetry of the second and third clause: if either `var` or `val` is a bound variable, then it is replaced by its value. Together, these two clauses enforce the policy that no substitution contains circular bindings (because `extend-subst` will not be called with a bound value), and no call to `occurs-in?` references a bound variable.

This modification leads to the correct result in the problematic cases:

```
> (unify '(p X Y) '(p Y X))
((X . Y))

> (unify '(p X Y a) '(p Y X X))
((Y . A) (X . Y))
```

Conclusions

This note uncovers a widespread error in unification algorithms, an error that shows up, among other places, in one of the best-regarded books on introductory programming, one of the standard Introductions to AI, and several highly-regarded books on COMMON LISP. It is somewhat surprising that an algorithm that has been studied in such detail[11,12] should still be presented incorrectly. The publication of seven erroneous versions of the algorithm does show that the error is not easily uncovered by testing. This may be because all of the texts present the unifier in conjunction with a backward chaining logic-programming system. These systems rename variables before each backward chaining step, so there can be no common variables between the two arguments to `unify`. The only way the error can come up is when higher order predicates, like `q` below, are used.

```
> (unify '(q (p X Y) (p Y X)) '(q Z Z))
```

Apparently, such higher-order predicates are used infrequently enough that casual testing did not reveal the bug. It is interesting to speculate why standard PROLOG-based unification algorithms[13,14] do not have this bug. In these algorithms it is more obvious that dereferencing is needed because they explicitly deal with the destructive manipulation of pointers; in writing the loop that follows pointers one is immediately confronted with the possibility of a circular chain, and therefore can see how to avoid constructing one.

Functional Programming advocates have claimed that assignment (to a variable or data structure) is dangerous because it violates referential transparency. Curiously, this note shows that for unification, the functional approach is error-prone, while the procedural, state-modification approach tends to lead to a correct solution.

Another reason why the PROLOG-based algorithms are correct may be that they do not handle the occurs check. Thus, there are more inputs that can put these algorithms into an infinite loop, forcing their authors to find and correct the bug. The LISP-based unifiers terminate on such examples, so the bug is harder to detect.

In conclusion, the LISP-based unifiers implement variable binding with an association list rather than with pointers. With this implementation, it is tempting to use the built-in function `assoc` instead of doing a proper

dereferencing. It appears that the seven texts in question all succumbed to this temptation. They all failed to either test the resulting code sufficiently or attempt even an informal proof of correctness. Either approach could have uncovered the bug that has remained hidden until now.

References

[1] J. A. Robinson, 'A machine-oriented logic based on the resolution principle,' *Journal of the ACM*, **12**, 1, 1965.

[2] J. Slage and M. Gini, 'Unification', in S. Shapiro ed., *The Encyclopedia of Artificial Intelligence*, Wiley, NY, 1988.

[3] H. Abelson and G. Sussman, *The Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[4] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*, Addison-Wesley, 1985.

[5] E. Charniak, C. K. Riesbeck, D. V. McDermott and J. R. Meehan, *Artificial Intelligence Programming*, 2nd edn., Erlbaum, Hillsdale, NJ, 1987 (1st edn. 1980).

[6] W. L. Hennessey, *Common Lisp*, McGraw-Hill, 1989.

[7] R. Wilensky, *Common LispCraft*, Norton, NY, 1986.

[8] P. H. Winston and B. K. P. Horn, *Lisp*, 3rd edn., Addison-Wesley, 1989.

[9] M. R. Genesereth, 'MRS source code, version 7.24, function `unifyp`,' copyright 1985.

[10] S. Russell, 'The Compleat Guide to MRS,' Stanford University Computer Science Dept. Report No. STAN-CS-85-1080, 1985.

[11] J-L. Lassez, M.J. Maher and K. Marriott, 'Unification Revisited,' in: J. Minker (ed.), *Foundations of deductive Databases and Logic Programming*, Morgan Kaufman, 1988, pp. 587-625.

[12] K. Knight, 'Unification: A Multidisciplinary Survey,' ACM Computing Surveys, Vol. 21, No. 1, March 1989, pp. 93-121.

[13] D. Maier and D. S. Warren, *Computing with Logic*, Benjamin Cummings, Menlo Park, CA, 1988.

[14] A. Ramsey and R. Barrett, *AI in Practice: examples in Pop-11*, Ellis Horwood, Chichester, England, 1987.