Detecting Looping Simplifications

by

Paul W. Purdom, Jr.

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 208

Detecting Looping Simplifications

by

Paul W. Purdom, Jr.

December, 1986

Detecting Looping Simplifications

Abstract: A generalization of tree matching and unification algorithms is presented. Given the equation $s = t$, this algorithm can often quickly determine that the rewrite rule $s \rightarrow t$ leads to an infinite sequence of "simplifications". The rule $t \rightarrow s$ can be tested in the same way. Rules leading to infinite simplifications should not be included in a rewrite system. In general, the problem of deciding whether a set of rewrite rules leads to infinite simplifications is undecidable. The algorithm that is used for this problem is a cross between a unification algorithm for terms with overlapping variables and a matching algorithm. In the simplest case it attempts to find a, σ_M and σ_U such that $\sigma_M \sigma_U s = \sigma_U t/a$. The substitution σ_M is used like the substitution that occurs in matching problems and the substitution σ_U is used like the one that occurs in unification problems. If $\sigma_M \sigma_U s = \sigma_U t/a$ has a solution, then the rule $s \rightarrow t$ leads to infinite simplifications. The same basic algorithm can be used to test more complex cases of looping involving the interaction of several rules, but it is limited to those cases where each application of a rule occurs inside of the previous rule application. Experiments suggest that the simplest form of the algorithm is about 80 percent effective in eliminating bad orientations of rules. The algorithm never rules out a good orientation of a rule, and so it is most useful when one wants to consider all possible rule orientations.

Keywords: Knuth-Bendix, Looping, Matching, Rewrite rules, Simplification, Unification.

1. Introduction

One basic problem that arises in the Knuth-Bendix procedure [6, 10] is to orient equations: convert an equation $s = t$ into the rule $s \rightarrow t$ or the rule $t \rightarrow s$. The decision of how to orient each rule can be critical to the operation of the procedure. Two variations of the procedure obtain the same final answer if they use the same method for orienting rules [11]. On the other hand, one orientation method may lead to a finite set of rewrite rules, while another may lead to infinite calculations.

Often when a person runs the Knuth-Bendix system, he wants to obtain a finite set of rewrite rules so long as it is not too much trouble. Various termination orderings have been devised for such people. Dershowitz [2] gives a survey of such orderings. Sometimes, however, a person wishes to obtain a finite set of rewrite rules if at all possible. Then he may wish to try every possible ordering until he finds one that works. He will have to be careful not to spend too much time on any one ordering because some of them probably lead to infinite computations. At such times he would like an orientation method that eliminates those orientations which are *known* to lead to infinite computations but that does not eliminate any which might lead to finite computations. A perfect system would eliminate *all* orientations that lead to infinite computations and no others, but there is no such algorithm — the problem is undecidable.

This paper gives an algorithm for the elimination of most orientations that lead to infinite simplifications while not eliminating any that do not. In what follows infinite

simplifications will be called *looping* to save space. The Knuth-Bendix procedure can diverge due to looping or due to starting to generate an infinite set of rules. The current paper attacks the looping problem. Herman and Prívara [4] attack the infinite number of rules problem. They eliminate some orientations that are known to lead to an infinite number of rules. Thus, their algorithm complements the current one; the Knuth-Bendix procedure needs to consider only those rules that pass both test.

The present algorithm is apparently the first algorithm to reject (some) looping rules while not ever rejecting non-looping rules. Plaisted [13] gives an algorithm based on a similar idea: he eliminates orientations that are not consistent with any simplification ordering while not eliminating any that are consistent. His method does not allow rules such as $f(f(X)) \rightarrow f(g(f(X)))$, which do not loop, because such rules are not consistent with a simplification ordering.

I recommend that you first use the Knuth-Bendix procedure with some termination ordering. If the termination ordering leads to a finite set of rules, you have obtained your answer the easy way. If it does not seem to be leading to a finite set of rules, then you can use the current method and order the rules whichever way seems best. Eventually you will orient your rules correctly or give up on the problem. The method does not prevent you from trying out any promising orientation. So far I have not followed my own advice; the loop detection algorithm is easy to program and is fairly effective, so I have used it alone until I get around to building a simplification ordering routine. Before I had any assistance with orientation, looping was a major problem. After I added the loop testing algorithm, looping has seldom been a problem.

It is undecidable whether or not a set of rewrite rules loops [7]. The problem remains undecidable for sets with 2 rules [2]; the complexity of one rule systems is not known. As a result, no ordering method is completely satisfactory if one wishes to exhaustively consider possible orientations for rules. Compared to other published orientation methods the current algorithm gives the most freedom but the least help in solving orientation problems. The user is left with the problem of proving that his final set of rules do not loop.

2. Some Simple Loops

The rewrite rule $*(e, X) \rightarrow *(X, e)$ loops on the term $*(e, e)$. The reason for this is that the left and right sides of the rule unify with each other — for $X = e$ the two sides are the same. This generalizes to: $s \rightarrow t$ loops when s unifies with any part of t.

For many applications of unification, one first modifies one of the terms so that they have distinct variables. This is incorrect for this problem; the original variables must be retained. For example, the rule $+(X) \rightarrow X$ does not loop even though X' unifies with $+(X)$.

The rule $*(X, Y) \rightarrow *(i(i(X)), Y)$ loops. In this case the two sides do not unify. Here the reason for looping is that the left side matches the right — if on the left side X is replaced with $i(i(X))$ then the left and right sides are the same. This generalizes to: $s \rightarrow t$

loops when s matches part of t. For this case, the variables of t can be considered as distinct from those of s.

These two observations can be generalized. Let σ be a substitution of values for variables. The individual substitutions in σ will be written in the form $X \leftarrow t$, where X is a variable and t is a term. The term t may contain variables. Those variables that are not assigned values in σ retain their original value. I will say $\sigma[X]$ is *empty* in such cases. The notation σt means replace each variable in the term t with its value in σ. Do not apply σ to itself, i.e., do not assign values to variables that appear in the right side of the assignments in σ. The notation $\sigma^2 t$ means $\sigma(\sigma t)$ and t/a means the subtree at position a in t. The position is usually represented as a number where the first digit says which child of the root to follow, the second digit says which child of that node to follow, etc. The notation a^i means to follow the path consisting of i a's.

Theorem 1. The rewrite rule $s \rightarrow t$ loops if there exists a position a in t and substitutions σ_M and σ_U such that

$$\sigma_M \sigma_U s = \sigma_U(t/a).$$

(The substitution σ_M is similar to matching, and the substitution σ_U is similar to unification.)

Proof: Let

$$u_i = (\sigma_U t)[a \leftarrow \sigma_M \sigma_U(t)] \cdots [a^i \leftarrow \sigma_M^i \sigma_U(t)].$$

Applying the rule $s \rightarrow t$ to u_i at a^{i+1} gives u_{i+1}, leading to an infinite sequence of simplifications.

Theorem 1 provides a way to test a single rule for looping, but it does not always detect looping when it is present. Dershowitz [2] gives a number of cases where it fails. In practice, however, it does quite well. For example, when the Knuth-Bendix procedure is run on the standard 3 rule representation of group theory [10], theorem 1 suggests a unique orientation for all equations that arise except for two: $*(*(X,Y),Z) = *(X,*(Y,Z))$ and $i(*(X,Y)) = *(i(Y),i(X))$. These two equations can be oriented either way without causing looping, although the second rule must be oriented properly to avoid generating an infinite number of rules.

The following table summarizes the results of a few tests based on theorem 1.

	number of orientations		
Initial rules	0	1	2
groups (three rules) [10]	0	17	2
dihedral group of order 8	0	22	24
central groupoid (second version) [10]	0	43	13
groups (one rule) [5]	9	137	14

The three columns of numbers show the number of rules for which it permitted zero orientations, one orientation, and two orientations. In most cases the algorithm was able

to rule out one of the two possible orientations. Even when just one orientation passes the test there is no guarantee that the rule does not loop, but this trouble seldom occurs in practice. In most cases the algorithm was able to suggest the correct orientation by considering a rule in isolation. When the Knuth-Bendix procedure was run on the one rule version of group theory, some equations were generated where neither orientation was satisfactory. Some of these equations were eliminated by introducing new operators. Others were simplified away by later rules.

During the tests many rules had one orientation eliminated for one of the following simple reasons: (1) one side was a variable, (2) one side contained a variable that the other side did not, or (3) one side was a subpart of the other side. A number of more complex cases were also correctly oriented. Any method of testing each rule in isolation will have trouble when there are a large number of equations relating terms with two different main operators. In such cases it is often important to orient all the equations for a pair of main operators the same way.

Theorem 1 can be generalized as follows.

Theorem 2. The set of rewrite rules $s_j \rightarrow t_j$ for $1 \leq j \leq m$ loops there exists position a_i in t_{r_i} and rule numbers r_i where $0 \leq i < k$ for some k and there exists substitutions σ_M and σ_U such that

$$\sigma_M \sigma_U s_{r_{(i+1) \bmod k}} = \sigma_U(t_{r_i}/a_i) \qquad \text{for } 0 \leq i < k.$$

For $i \neq j$, the rules $s_{r_i} \rightarrow t_{r_i}$ and $s_{r_j} \rightarrow t_{r_j}$ use distinct variables (even when $r_i = r_j$).

Proof: Let

$$u_{nk+j} = \sigma_U(t_{r_0})[a_0 \leftarrow \sigma_M \sigma_U(t_{r_1})][a_1 \leftarrow \sigma_M^2 \sigma_U(t_{r_2})] \cdots [(a_0 \ldots a_{k-1})^n a_0 \ldots a_{j-1} \leftarrow \sigma_M^{nk+j} \sigma_U(t_{r_j})]$$

Applying the rule $s_{r_i} \rightarrow t_{r_i}$ to u_{i-1} gives u_i, leading to an infinite sequence of simplifications.

The time for testing with theorem 2 appears to increase rapidly with k ($O(n^k)$ times the time for solving equations of the form $\sigma_M \sigma_U s = \sigma_U t$, where n is the total size of the rules). I have not tried to use it with $k \geq 2$. Like Theorem 1, Theorem 2 does not find all cases of looping.

3. The U-match Algorithm

To solve for σ_M and σ_U that satisfy the equations

$$\sigma_M \sigma_U s_{r_{(i+1) \bmod k}} = \sigma_U(t_{r_i}/a_i)$$

for $0 \leq i < k$, one can try each combination of possible values for the a's and r's. For each set of a's and r's, one obtains a set of equations of the form

$$\sigma_M \sigma_U p_i = \sigma_U q_i.$$

This set of equations can be solved using decomposition and merging [9]. If some p has the form $f(p_1, \ldots, p_b)$ and the corresponding q has the form $g(q_1, \ldots, q_c)$ with $f = g$ and $b = c$, then the equation $\sigma_M \sigma_U p = \sigma_U q$ can be replaced with the set of equations $\sigma_M \sigma_U p_i = \sigma_U q_i$ for $1 \leq i \leq b$. If some equation has this form with $f \neq g$ or with $b \neq c$, then the original set of equations does not have a solution [9, 10]. This replacement of equations is called decomposition.

After decomposition, each equation will have one of the following three forms:

1. $\sigma_M \sigma_U t = \sigma_U X$,
2. $\sigma_M \sigma_U X = \sigma_U t$,
3. $\sigma_M \sigma_U X = \sigma_U Y$,

where t is a term that is not a single variable and where X and Y are variables.

If there is any equation of form 1, then it is necessary that t does not contain X and that the assignment $X \leftarrow \sigma_M \sigma_U t$ be in σ_U. If $\sigma_M \sigma_U[X]$ is empty (as it will be the first time) then the assignment is $X \leftarrow t$. This assignment can now be applied to all remaining equations to eliminate X without changing the set of solutions. This process may lead to more decomposition and convert equations of form 3 to form 1 or 2, but repeated application will eventually eliminate all equations of form 1. Solving equations of form 1 is a unification problem.

If two equations have the same left side variable, they can be satisfied if and only if their right sides unify, because only σ_U affects the right sides. If the right sides do unify, then the unifier can be applied to the equations and the two original equations will be the same. This process may generate some more equations of form 1, but those can be eliminated as before. Eventually all equations will be of forms 2 and 3, and there will be at most one equation for each variable. These remaining equations can be solved setting σ_M so that $\sigma_M(X)$ is the right side of the equation for X.

In all of the unifications, it is important to check that the variable X does not occur (either directly or indirectly) in the value of X. When computing σ_M no such occur check is made.

The above explanation of the algorithm was chosen for clarity. The actual implementation was selected to be reasonably fast without being overly complex. It is a variation of an exponential time unification algorithm, but in practice it is "fast enough" — loop testing takes a small fraction of the total time used during a Knuth-Bendix calculation. It would be interesting to know how fast the calculation can be done. The linear time algorithm of Paterson and Wegman [1, 12] does not appear to apply because their algorithm requires that the initial terms have distinct variables. (It uses this when avoiding explicit occur-checks.) Probably some of the unification algorithms that run in just over linear time can be applied to this problem, but that possibility has not yet been investigated. The current approach can take up to n times longer than the underlying unification algorithm, but perhaps a faster approach exists. A fast version of the algorithm would be useful as a

filter on a termination ordering algorithm. The loop test could eliminate about half of the possible orderings, potentially saving about half the time used by the termination ordering algorithm.

Here is the algorithm that I actually use.

Algorithm *U-match*(s, t, c). The parameters s and t are terms. The parameter c has the value *unify*, *match*, or *both*. The algorithm also uses two global variables, substitutions σ_M and σ_U. The function *Occur*(X, t) returns *true* if the variable X occurs in $\sigma_U^* t$ (the results of repeated applying σ_U to t) and *false* if it does not. On entry to the algorithm σ_U must be such that, for each variable X in σ_U, *Occur*(X, X) must be *false* (otherwise the algorithm may fail to terminate). Also for each variable X either $\sigma_M[X]$ or $\sigma_U[X]$ is empty. These conditions are also true when exiting the algorithm. When $c =$ *both* the algorithm attempts to set σ_M and σ_U so that $\sigma_M \sigma_U^i s = \sigma_U^i t$ for some i, i.e. σ_M and σ_U^i solve the equation $\sigma_M \sigma_U' s = \sigma_U' t$ with $\sigma_U' = \sigma_U^i$. The algorithm is allowed to modify σ_U by adding assignments for variables that do not yet have a value in σ_U. It is allowed to modify σ_M only by adding assignments for variables that do not yet have a value in either σ and by moving an assignment from σ_M to σ_U (leaving an empty assignment in σ_M). Thus, this algorithm finds a solution to the equation $\sigma_M \sigma_U s = \sigma_U t$ that is consistent with the current substitutions, if such a solution exists. The algorithm returns the value *true* if it finds a solution and *false* otherwise. If $c =$ *match* then the algorithm only changes σ_M, i.e. it attempts to match s to t, and if $c =$ *unify* then it changes σ_U, i.e. it attempts to unify s and t. When $c =$ *unify* and σ_M is not empty, the algorithm can also change σ_M. In effect, it is unifying $\sigma_M s$ and t, and it may need to modify σ_M to do this.

Step 1 If s is a variable then

Step 1.1 while t is a variable and $\sigma_U t \neq t$ set $t \leftarrow \sigma_U t$;

Step 1.2 if $s = t$ then return *true*,

Step 1.3 otherwise

Step 1.3.1 if $(\sigma_M \sigma_U)[s]$ is empty then

Step 1.3.1.1 if $c =$ *unify* then

Step 1.3.1.1.1 if *Occur*(s, t) then return *false*,

Step 1.3.1.1.2 otherwise add $s \leftarrow t$ to σ_U and return *true*;

Step 1.3.1.2 if $c \neq$ *unify* then add $s \leftarrow t$ to σ_M and return *true*,

Step 1.3.2 otherwise if $\sigma_M[s]$ is empty then

Step 1.3.2.1.1 return *U-match*$(\sigma_U s, t,$ *unify*$)$,

Step 1.3.2.2 otherwise set s' to be the assignment to s in σ_M,

change σ_M to have $s \leftarrow t$, and

return *U-match*$(s', t,$ *unify*$)$.

Step 2 If s is not a variable then

Step 2.1 if t is a variable then

Step 2.1.1 if $c = \textit{match}$ then return *false*;

Step 2.1.2 if $\textit{Occur}(t, s)$ then return *false*;

Step 2.1.3 if $(\sigma_M \sigma_U)[t]$ is empty then add $t \leftarrow s$ to σ_U and return *true*;

Step 2.1.4 if $\sigma_M[t]$ is not empty then

Step 2.1.4.1 move $t \leftarrow \sigma_M t$ to σ_U leaving $\sigma_M[t]$ empty;

Step 2.1.5 return $\textit{U-match}(s, \sigma_U t, \textit{unify})$;

Step 2.2 if t is not a variable then (decompose)

let s have the form $f(s_1, \ldots, s_m)$ and t have the form $g(t_1, \ldots, t_n)$,

constants have this form with zero for the number of children;

Step 2.2.1 if $f \neq g$ or $m \neq n$ then return *false*;

Step 2.2.2 for $i = 1$ to m if not $\textit{U-match}(s_i, t_i, c)$ then return *false*,

Step 2.2.3 otherwise return *true*.

4. Conclusions

The U-match algorithm gives a practical way to test single rewrite rules for looping. If a rule fails the test, then it definitely causes looping. If it passes, it may or may not cause looping. The most important use of this test is for eliminating many of the possibly bad orientations of rewrite rules when exhaustively considering various orderings.

The algorithm may also be useful as a filter on slow termination ordering algorithms. It can eliminate nearly half of the possible orientations from further consideration. Unification and matching are special cases of the computation done by the algorithm, so a fast implementation could be considered as a replacement of both the matching and unification algorithms of a Knuth-Bendix program.

It is interesting to note that a local test, such as theorem 1, can often show that a rule is oriented incorrectly, but that their are no useful local tests to show that a rule is oriented correctly. Whether or not an orientation is correct depends on the complete set of rules.

Acknowledgement: I wish to thank Professors Cynthia Brown and Edward Robertson for discussions that contributed to this work. I wish to thank Bruce Smith for carefully reading the manuscript.

References

1. Dennis De Champeaux, *About the Paterson-Wegman Linear Unification Algorithm*, Journal of Computer and System Sciences **32** (1986), pp 79–90.

2. Nachum Dershowitz, *Termination*, Rewriting Techniques and Applications (Jean-Pierre Jouannaud ed.), Lecture Notes in Computer Science **202**, Springer-Verlag New York (1985), pp 180–224.

3. Nachum Dershowitz, *Computing with Rewrite Systems*, Information and Control **65** (1985), pp 122–157.

4. Mikuláš Herman and Igor Prívara, *On Nontermination of Knuth-Bendix Algorithm*, 13-th Colloq. Automata, Languages and Programming, Lecture Notes in Computer Science, Springer-Verlag, New York (1986), pp 146–156.

5. G. Higman and B. H. Neumann, *Groups as Groupoids with One Law*, Publ. Math. Debrecen **2** (1952), pp 215–227.

6. Gérard Huet, *A Complete Proof of the Correctness of the Knuth-Bendix Completion Algorithm*, J. Computer and Systems Sciences **23** (1981), pp 11–21.

7. G. Huet and D. S. Landford, *On the Uniform Halting Problem for Term Rewriting Systems*, Rapport Laboria 283, IRIA (1978).

8. J. P. Jouannaud and H. Kirchner, *Construction d'un plus petit order de simplification*, Report 82-R-033, Centre de Recherche en Informatique de Nancy, Nancy, France (1982).

9. Claude Kirchner, *Computing Unification Algorithms*, Symposium on Logic in Computer Science, IEEE Computer Society (1986), pp 208–216.

10. Donald E. Knuth and Peter B. Bendix, *Simple Word Problems in Universal Algebras*, Computational Problems in Abstract Algebra (J. Leech ed.) Pergamon Press, Oxford (1970), pp 263–297.

11. Yves Metivier, *About the Rewriting Systems Produced by the Knuth-Bendix Completion Algorithm*, Information Processing Letters **16** (1983), pp 31–34.

12. M. S. Paterson and M. N. Wegman, *Linear Unification*, Journal of Computer and System Sciences, **16** (1978), pp 158–167.

13. David A. Plaisted, *A Simple Non-Termination Test for the Knuth-Bendix Method*, 8-th International Conference on Automated Deduction, Lecture Notes in Computer Science **230**, Springer-Verlag, New York (1986), pp 79–88.