A Practical Unification Algorithm

by

Paul W. Purdom

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT No. 242

A Practical Unification Algorithm

By

Paul W. Purdom

February, 1988

A Practical Unification Algorithm

Abstract: By refining Robinson's algorithm, Corbin and Bidoit obtained a unification algorithm that is fast for typical unification problems. The worst-case time of their algorithm, however, is $O(n^2)$. This paper gives two additional refinements that improve the worst-case time to $O(n\alpha(n,n))$, where $\alpha(n,n)$ is an extremely slowly growing function of n. The resulting algorithm is an efficient implementation of the Martelli and Montanari algorithm. Measurements show it to be quite fast.

Introduction

A substitution assigns terms as values to the variables in its argument. Given two terms, s and t, a unification algorithm returns true if and only if there exists a substitution σ such that $\sigma(s)$ is identical to $\sigma(t)$. In addition, if such a substitution exists, the most general such substitution is produced. The algorithms being considered represent terms using directed acyclic graphs.

Corbin and Bidoit [1] refined Robinson's algorithm [2] for this problem to produce an algorithm that is fast for small problems. The worst-case time for Robinson's algorithm is exponential in the size of the input, while the worst-case time for their algorithm is quadratic. The unification algorithm of Paterson and Wegman [3, 4] runs in linear time, but it is complex and not fast for small problems. The algorithm of Martelli and Montanari [5] has a worst-case time that is nearly linear ($O(n\alpha(n,n))$, where $\alpha(n,n)$ is an extremely slowly growing function [6]). The measurements of Corbin and Bidoit [1] showed their algorithm to be faster than that of Martelli and Montanari for reasonable size problems. As shown in the next section, adding two refinements to the Corbin and Bidoit algorithm produces a nearly linear algorithm that is fast for all reasonable problem sizes. This algorithm is the same as the algorithm of Martelli and Montanari, except for the data structures and the order of calculation.

The algorithm

There are two reasons why the algorithm of Corbin and Bidoit requires time $O(n^2)$ in the worst case. First, they do an Occur Check as each variable is set. Second, each time they find two nodes to be equivalent, they replace all pointers to one of the nodes with pointers to the other.

The first modification to their algorithm is to remove the Occur Check (acting as though each variable passes the omitted Occur Check call). This algorithm still makes at most $n-1$ recursive calls, because each successful call results in one node being replaced. The answer from this modified algorithm is not always correct, because some variable may be defined in terms of itself, but such incorrect answers are rejected by following the algorithm with a single Occur Check call. The Occur Check is done in linear time using depth-first search.

The second modification to the algorithm of Corbin and Bidoit consists of using pointers instead of replacing nodes. Those nodes that have been merged form an equivalence class. The equivalence class is represented by a tree, where each node except the root has a *match* field that points to an equivalent node. The root has a *nil match* field. For any node in the class, the representative node can be found by following *match* fields until a node with a *nil match* field is found. If each chain is collapsed as it is traversed, then doing m finds on a structure with n nodes takes time no more than $O(n + m\log_{2+m/n} n)$ [7]. If the node to be replaced is selected in a way to keep chains short (by selecting the node from the smaller of the two equivalence classes, for example) then the time is reduced to $O(n + m\alpha(m+n,n))$ [7]. (Prohibiting the *match* field of a variable node from pointing to a nonvariable node does not increase this bound.)

After these two modifications, plus one more to avoid forming cyclic chains of *match* fields, the unification algorithm of Corbin and Bidoit becomes the algorithm shown in Figure 1.

```
Function Unify(s, t)
  Set $s \leftarrow Find(s)$ and $t \leftarrow Find(t)$. (follow match fields and collapse chains)
  If $s = t$ then $Unify \leftarrow true$.
  Otherwise if $s$ is a variable then set $s\uparrow.match \leftarrow t$ and $Unify \leftarrow true$.
  Otherwise if $t$ is a variable then set $t\uparrow.match \leftarrow s$ and $Unify \leftarrow true$.
  Otherwise if $label(s) = label(t)$ then
    call $Union(s, t)$; (set match of $s$ or $t$ to point the other one)
    let $s_i$ be the $i^{\text{th}}$ immediate subterm of $s$
    and let $t_i$ be the $i^{\text{th}}$ immediate subterm of $t$;
    for each $i$ if not $Unify(s_i, t_i)$ then set $Unify \leftarrow false$ and exit immediately;
    set $Unify \leftarrow true$.
  Otherwise set $Unify \leftarrow false$.
```

Figure 1. A unification algorithm.

The routine *Find*(t) sets t to the last node on the chain of *match* fields, and collapses the chain. In the version of the program that was tested, *Union*(s, t) sets the *match* field of s to point to t. This gives an algorithm with a worst-case time of $O(n \ln n)$. If the *Union* procedure keeps chains small by selecting which *match* field to set, then the running time for small problems is increased by a small constant factor but the worst-case time is reduce to $O(n\alpha(n, n))$. For the test problems, measurements show that there is no advantage in using a more complex *Union* algorithm.

Comparison

The algorithm of this paper is a variation of the algorithm of Martelli and Montanari, but it differs in the data structures and the order of the calculations. To unify s and t, Martelli and Montanari form the equation $s = t$. They compute the nonvariable *common parts* of s and t. From the *frontier* of the *common parts* they form new equations by equating the variable from one term with the corresponding part of the other term. Equations with common terms are combined to form multiequations (a set of terms that are all equal). The process is continued by forming the *common parts* and *frontier* of some multiequation. If the Martelli and Montanari algorithm is programmed to conform with their description, then there is a good bit of overhead associated with forming and processing multiequations. This is clear from the measurements that Corbin and Bidoit [1] did on the algorithm.

Each equivalence class formed by the algorithm in this paper corresponds to a multiequation. Unifying the children of two terms is similar to computing the *common part* of the two terms. For successful unifications, each comparison that Martelli and Montanari do is also done in this algorithm. This algorithm is faster (by a constant factor) because it builds less intermediate structure. For unsuccessful unifications, the time depends on the details of the two terms, because this algorithm compares the terms in a strictly depth first order, while the order of the Martelli and Montanari algorithm is similar to breadth first search.

Measurements

Table 1 shows the running time for this algorithm and for an algorithm that is identical to this algorithm except that it calls the Occur Check routine each time a variable is set. The time for this second algorithm should be similar to but less than that for the algorithm of Corbin and Bidoit. The Find and Union routines were coded in line. The problem sets, ex 2 and ex 3, are taken from [1]. The programs were written in Web [8] and compiled with the Berkeley Pascal compiler, and run on a VAX8800. For each problem, the time is the result of averaging 100,000 executions of the problem. (Similar runs on a VAX780 gave times about 7 times larger.)

problem	ex 2						ex 3						
v	2	4	8	16	20	40	2	4	8	16	20	40	80
n	10	18	34	66	82	162	12	28	60	124	156	316	636
T_1	0.17	0.30	0.55	1.06	1.31	2.95	0.24	0.59	1.28	3.10	3.74	8.08	18.02
T_2	0.20	0.50	1.45	4.87	7.69	32.81	0.24	0.77	2.59	9.62	15.02	61.10	249.98

Table 1. CPU time in milliseconds for the new algorithm (T_1) and the time for the new algorithm with extra Occur Checks (T_2) is shown for problems with various numbers of variables (v) and sizes (n).

The times in Table 1 include the time used for unification, for Occur Checks, and for resetting the data structure. For the new algorithm about half the time is for unification and half for the Occur Check. Of the time for unification, about half is for the actual unification and half for resetting the data structure. For ex 2 the measured time is approximately $0.017\,n$ milliseconds, while for ex 3 it is about $0.027\,n$ milliseconds.

References.

1. Jacques Corbin and Michel Bidoit, *A Rehabilitation of Robinson's Unification Algorithm*, Information Processing 83 (1983), pp 909–914.
2. J. A. Robinson, *A Machine-Oriented Logic Based on the Resolution Principle*, J. ACM **12** (1965), pp 23–41.
3. M. S. Paterson and M. N. Wegman, *Linear Unification*, J. of Computer and System Sciences **16** (1978), pp 158–167.
4. Dennis de Champeaux, *About the Paterson-Wegman Linear Unification Algorithm*, J. of Computer and System Sciences **32** (1986), pp 79–90.
5. Alberto Martelli and Ugo Montanari, *An Efficient Unification Algorithm*, ACM Trans. Prog. Languages and Systems 4 (1982), pp 258–282.
6. Robert Endre Tarjan, *Efficiency of a Good But Not Linear Set Union Algorithm*, J. ACM **22** (1975), pp 215–281.
7. Robert E. Tarjan and Jan van Leeuwen, *Worst-Case Analysis of Set Union Algorithms*, J. ACM **31** (1984), pp 245–281.
8. Donald E. Knuth, *Literate Programming*, The Computer Journal **27** (1984), pp 97–111.