JOURNAL OF COMPUTER AND SYSTEM SCIENCES 32, 79–90 (1986)

About the Paterson–Wegman Linear Unification Algorithm

DENNIS DE CHAMPEAUX*

Computer Science, Tulane University, New Orleans, Louisiana 70118

Received May 16, 1984; revised October 9, 1984

The Paterson–Wegman unification algorithm is investigated. An error is removed. The proper input format is clarified. A pre-processor is described that associates with each node a list of parent nodes, as required by the core algorithm. A post-processor is described that transforms an ordered substitution, produced by the core algorithm, into a regular substitution, making use of structure sharing when possible. The combination of pre-processor, core algorithm, and post-processor is still a linear algorithm (with respect to the sum of nodes and edges in the input graph). © 1986 Academic Press, Inc.

1. INTRODUCTION

This paper assumes that the reader is familiar with the concept of unification, see [7, 1, 2]. In 1976, Paterson and Wegman published a unification algorithm that was claimed to be linear [4]. Two years later a revised version, differing substantially, appeared [6]. The theorem proving community somehow has not caught on to this algorithm. The monumental text [2] does not even mention it, while the textbook [3] gives only a reference to a report [4].

A plausible explanation is that the presentation of their algorithm is somewhat alien to the tradition in the deduction community. A serious source of confusion is the data structure on which this algorithm operates: directed acyclic graphs (DAGs) so they claim, which makes one wonder whether input has to be pre-processed first to that format. Another source of confusion is the output format produced by this algorithm. When a match is found, the algorithm produces an ordered substitution in which a variable, for which a replacement prescription exists, may still occur in another replacement prescription. Thus one may have to do post-processing when an ordinary substitution is required. This makes it hard to decide whether this algorithm is really linear and/or useful.

In this paper, we clarify these issues by giving a better and more elaborate description of the algorithm as well as of a pre-processor and a post-processor. At least one error that still slumbered in the formulation of [6] is removed while more commitment is made to a feasible data representation also. Whether the algorithm is useful in practice—where one deals only with "small" terms and thus where linearity may be offset by high initialization costs—we do not address.

* Present address: ADAC Laboratories, 4747 Hellyer Ave., San Jose, CA 95138.

0022-0000/86 $3.00

571/32/1-6

2. The Original Formulation

In this section, we repeat the description of the algorithm given in [6]. We have added line numbers to facilitate the discussion.

WARNING: The reader should not worry about the opaqueness of the following code, which is in fact erroneous; the balance of the paper is devoted to its clarification.

ALGORITHM C. Comment: to test $\langle u, v \rangle$ for unifiability.

```
1  Begin
2    Create undirected edge (u, v).
3    While there is a function node r, Finish(r)
4    While there is a variable node r, Finish(r).
5    Print("UNIFIED") and halt.
6  End of Algorithm C.

7  Procedure Finish(r)
8  Begin
9    If pointer(r) defined then print ("FAIL:LOOP") and halt
10                          else pointer(r) := r.
11   Create new pushdown stack with operations Push(*) and Pop.
12   Push(r).
13   While stack nonempty do
14   begin
15     s := Pop.
16     If r, s have different function symbols
       then print ("FAIL : CLASH") and halt.
17     While s has some father t do Finish(t).
18     While there is an undirected edge (s, t) do
19     begin
20       If pointer(t) undefined then pointer(t) := r
21       If pointer(t) ≠ t then print ("FAIL : LOOP") and halt.
22       Delete undirected edge (s, t).
23       Push(t).
24     end.
25     If s ≠ r then
26     begin
27       If s is variable node, print(s, "←", r).
28       If s is a function node with outdegree q > 0
29       then create undirected edges {jth son(r), jth son(s)) | 1 ≤ j ≤ q}.
30       Delete s and directed arcs out of s.
31     end.
32   end.
33   Delete node r and all directed arcs out of r.
34 End of Finish.
```

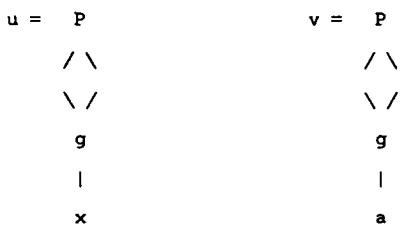


FIG. 1. We depict here a possible representation, with structure sharing of two terms u and v to be unified. The term u stands, unwound, for the formula $P(g(x), g(x))$ and similarly v stands for $P(g(a), g(a))$. The classical unification algorithm will follow the right-hand branches out of P which, in fact, is not necessary.

Before opening up this code, we will describe a pre-processor whose main task is adding back-pointers from descendant nodes to parent nodes, see Section 4. In addition, this pre-processor will associate with each node in the input a value under the indicator "pointer" with initial value NIL. Thus we can replace tests in the code whether a pointer value is defined by tests that check whether a pointer value is unequal to NIL. This leads to the following cosmetic changes:

Replace line 9 by

If pointer(r) $\neq$ NIL then print("FAIL : LOOP") and halt.

Replace line 20 by

If pointer(r) = NIL then pointer(t) := r else (Yes, "else" has been added on purpose).

3. THE KEY IDEA IN THE PATERSON–WEGMAN UNIFICATION ALGORITHM

The classical unification algorithm [7] is exponential for two reasons:

(1) it does not recognize structure sharing in input data, see Fig. 1 (observe that we do *not* say that the input should share structure when possible);

(2) it generates exponentially large structures that need to be traversed, by indiscriminate left-to-right processing of the input structures, see Fig. 2.

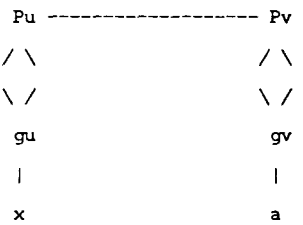
```
u = P( h(x1, x1), h(x2, x2),     y2   ,     y3   , x3 )
v = P(     x2   ,     x3   , h(y1, y1), h(y2, y2), y3 )
```

FIG. 2. The pair $\langle u, v \rangle$, when processes by the Robinson [7] algorithm, generates the task to unify $\langle x3, y3 \rangle$, which, due to the left-to-right processing, will have been blown up to

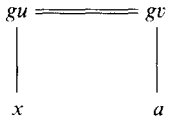
$$\langle h(h(x1, x1), h(x1, x1)), h(h(y1, y1), h(y1, y1)) \rangle.$$

These pitfalls are being avoided by superimposing "parent"-pointers on the input data structures, thus allowing one to schedule the matching of nodes. Building up of additional structures thus can be avoided. A crucial observation by [4] is that processing a node, say N, should be postponed until all its parents "higher up" have been processed. When processing of N resumes, all nodes with which N has to be "confronted" will have been gathered. We will illustrate this scheduling by sketching the solution of the example given in Fig. 1.

Solution 1. The nodes P and g in u are referred to respectively by Pu and gu, and similarly Pv and gv in v. The task of unifying u and v is translated as matching Pu and Pv in



Suppose, we start processing node a. According to the rule that parent nodes must be investigated first, we freeze the processing of a and focus on gv. Similarly, we freeze gv and look at Pv (following say the left branch from gv to Pv). Since Pv has no parent, we match Pv with any node with which it has a horizontal edge. The nodes Pu and Pv have the same predicate P and thus this match succeeds. This generates two (!) new subtasks by making a horizontal edge between the first argument of Pu and the first argument of Pv, producing gu ——— gv, and similarly for the second arguments, which, accidentally, produces again a horizontal link between gu and gv, resulting in gu ═══ gv. We return to gv and recognize that the other parent of gv, along the right-hand branch, need not be followed, since Pv has been investigated already. The situation thus has become



Matching gv and gu succeeds, since they both have the same function symbol g, and generates the task of matching x and a. Following up the other horizontal edge between gu and gv is aborted since gu will have been marked completed. A crucial observation is that we have generated only one link between x and a. The Robinson [7] algorithm has to face two tasks at this level. As is to be expected, matching x—a finally produces the substitution $\{x \leftarrow a\}$.

Another consequence of the scheduling via parent-pointers is that we can avoid

with little effort the so-called occurrence check. A substitution of the form $x \leftarrow u$, for an arbitrary expression u, is only acceptable when the expression u does not contain x. Descending into u to verify that it does not contain x can be avoided as follows. Each node N will be checked before we enter it to see whether N is already being explored. If so, then we know that we have obtained a cycle, and therefore a non-allowed occurrence. Otherwise, we will mark N right away that it is being investigated. For example, suppose that we face the task of matching the non-terminal node N with the variable x. Before performing that match, the parents of x will be investigated and this happens recursively. If in this process we stumble on N, which is marked as being explored, then we know that x occurs in N and we can abort the match.

At this stage the reader is invited to peruse the algorithm given in Section 2 to obtain a first grasp of its structure.

4. Pre-processor for Generating Parent-Pointers

We assume that the input of the algorithm is formulated as a directed graph, without cycles and with the important restriction that formula terminals are represented uniquely. Thus we accept the representation of $f(x, x)$ in Fig. 3a, but not the representation in Fig. 3b.

The pre-processor that we describe here presupposes that internal nodes and terminal nodes can be distinguished by an operation of unit cost. We assume also that the test which decides whether a terminal node is a variable or a constant consumes only one unit.

The algorithm ASSIGN-PARENT-POINTER works in depth-first mode. Associating parent pointers inside, for example, u is done by the call ASSIGN-PARENT-POINTER(u, TOP), where TOP is a hypothetical parent of u and thus an ancestor of all nodes in u. With the call ASSIGN-PARENT-POINTER(v, TOP), TOP will become also the ancestor of all nodes in v. The node TOP is marked completed, preventing it being explored, see the code of Finish in the Appendix.

FIG. 3. (a) is an acceptable graph representation of the formula $f(x, x)$ while (b) is not acceptable.

```
PROCEDURE ASSIGN-PARENT-POINTER (Node, Parent).

IF Leaf(Node)
THEN IF Parents-of(Node) ≠ NIL
  THEN Parents-of(Node) := Cons(Parent, Parents-of(Node))
  ELSE Parents-of(Node) := {Parent};
    Pointer-of(Node) := NIL;
    IF Variable(Node)
    THEN VNODES := Cons(Node, VNODES)
    ELSE FNODES := Cons(Node, FNODES);
    ENDIF;
  ENDIF
ELSE IF Parents-of(Node) ≠ NIL
  THEN Parents-of(Node) := Cons(Parent, Parents-of(Node))
  ELSE Parents-of(Node) := {Parent};
    Pointer-of(Node) := NIL;
    FNODES := Cons(Node, FNODES);
    FOR-EACH argument Ai of Node DO
      ASSIGN-PARENT-POINTER(Ai, Node);
    OD;
  ENDIF;
ENDIF.
```

Remark. The function Cons gets as first argument a node and as second argument a pointer to a list of nodes and will return a pointer to a list of nodes with the first argument in front of the second argument.

As a side-effect of constructing the parent-pointers this procedure collects in the global variables VNODES all terminals that are variables. All other nodes are collected in the global variable FNODES. This algorithm is certainly linear with respect to the sum of the nodes and the edges in the input.

5. The Core Algorithm

In this section, we criticize the formulation given in Section 2 and formulate improvements.

Line 1. If we take seriously the notion that the input is a DAG, and thus a graph, then the phrase "Create undirected edge (u, v)" dictates that we should install a unique connection between u and v. Since this is the first "horizontal" edge that is being constructed the uniqueness is here unproblematic. In the body of the "procedure Finish," line 22, we encounter again the requirement to construct

horizontal edges. We may wonder whether it is crucial for the algorithm that duplicate horizontal links between node pairs are avoided. Let us consider

$$u = P(x, x, \ldots, x, y1, \ldots, yn, x),$$
$$v = P(y, x1, \ldots, xn, y, \ldots, y, y).$$

If we insist on the graph-interpretation of lines 1 and 29 concerning these horizontal edges then we get into problems when processing line 29. It will take more than linear time to prevent a double link between x and y (unless we unrealistically assume available a square matrix M of booleans, initialized with false values, where $M(i, j)$ indicates whether there is an edge between node i and j). In fact, we do not have to, i.e., we can tolerate multiple links between a pair of nodes, provided we make small changes elsewhere that we point out below.

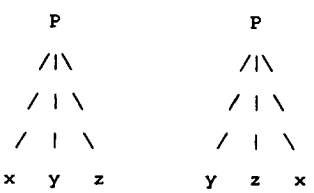
The solution we sketched in Section 3 for the problem given in Fig. 1 indicated already that we do not follow the graph interpretation of the horizontal edges.

Line 5. Halting the computation is not really what we want. Instead, when control reaches this point, the algorithm should return a substitution. This is accomplished by gathering in a variable, say SIGMA, which is local to Algorithm C and global for "Finish," the substitution prescriptions produced by line 27. In the next section, we describe a post-processor that takes the orderd substitution that is build-up in SIGMA and produces an ordinary substitution.

Lines 18–24. As we have argued above we cannot prevent double links between nodes. Therefore, we have to prevent pushing a node at the "other side" twice. The test in line 21 is wrong. As it stands, if a node t, at the "other side," has a pointer that does not point to t then the algorithm reports a failure. This test should be replaced by

$$\text{if pointer}(t) \neq r \text{ then } \cdots,$$

since the node r is the representative of the equivalence class of nodes that should match. For example, consider the terms



When processing P, we will generate the following "horizontal" links between x, y, and z:

Assume that x is the next node to be processed by "Finish" and that y and z gets pushed on x's stack. Thus, while still in x, we will enter z. Pushing x can be prevented easily, see below. According to line 21, the link between z–y leads to failure, because y has its pointer value directed at x.

From this example, we see also that when a node has a pointer value equal to r, then it need not be pushed again. Thus we should replace lines 20–23 by

```
IF t = r
THEN Ignore t (since matching r against r is needless)
ELSE IF Pointer(t) = NIL
  THEN Pointer(t) := r;
    Push(t)
  ELSE IF Pointer(t) ≠ r
    THEN Exit with failure
    ELSE Ignore t (since t is already on the stack);
    ENDIF;
  ENDIF;
ENDIF;
Delete link(s, t).
```

Deleting the horizontal "edges," the last line in the code above, is in fact optional. It can be done in constant time with a fairly hairy data structure (to prevent a linear search at the "other side"). An alternative is to keep the horizontal links (and also the vertical edges) and to mark instead a node, in line 30 and in line 33, as being completed. If we follow this alternative, which leads to an easy implementation, then we have to make two more modifications:

— the procedure "Finish" should start out investigating whether its input node is completed, and if so return immediately;

— when a link (s, t) is considered for pushing t, it should investigate t only when it is not completed.

See the Appendix for the code.

6. Post-processor for Generating Non-ordered Substitutions

A substitution $\{x1 \leftarrow u1,..., xn \leftarrow un\}$ is non-ordered when xi does not occur in uj for all i and j. The algorithm as it stands produces ordered substitutions. For exam-

ple, when x, y, and z are variables and we unify $P(x, x, x)$ and $P(g(g(a)), g(g(z)), g(y))$ then we obtain, with right-to-left selection of the non-terminal nodes, the ordered substitution $\{x \leftarrow g(y), y \leftarrow g(z), z \leftarrow a\}$. This example shows that it is easy to produce the non-ordered substitution $\{x \leftarrow g(g(a)), y \leftarrow g(a), z \leftarrow a\}$. This transformation, as is observed in [4–6], may lead to an exponential blow up when the final substitution is linearized.

However, since we admit structure sharing in the input, we may wonder whether we can produce, in linear time, non-ordered substitutions, provided we represent the output in graph form. A requirement is that the input may not be affected. For instance, in our example given above, it is not permitted to replace z in $g(z)$ by a, since that would affect the input formula $P(g(g(a)), g(g(z)), g(y))$. The following code, which copies only structure when necessary, assumes that a value is associated with an internal node and with a variable under the indicator Ready, that has initially the value NIL. A NIL value indicates that the node has not yet been investigated, a non-NIL value stands for the final value associated with this node, where original variable leaf nodes possibly have been replaced (recursively, since control is essentially depth-first). We assume also that when a node is encountered with a Ready value non-NIL, we can determine in constant time whether its final value deviates from its original value. When a value has been changed this leads to copying "above" this node.

Thus we have the following cross-recursive set of procedures/functions.

In the procedure BUILD-SIGMA, its argument stands for a list of variables that have been associated, in line 27 of the procedure Finish, with a substitution. We assume that the substitution can be found under the indicator Subs at the variable.

```
PROCEDURE BUILD-SIGMA (list-of-variables);
FOR-EACH variable xi in list-of-variables DO
  Add to final substitution: xi ← EXPLORE-VARIABLE(xi);
OD.
```

The function EXPLORE-VARIABLE takes as argument a variable, which is not necessarily a variable that has obtained a value in the procedure Finish.

```
Function EXPLORE-VARIABLE(xi);
IF Ready(xi) ≠ NIL
THEN Exit with Ready(xi)
ELSE out := DESCEND(Subs(xi));
  IF out = NIL
  THEN out := xi (the case that xi is not associated with a value)
  ENDIF;
  Ready(xi) := out;
  Exit with out;
ENDIF.
```

The function DESCEND will investigate a value associated with a variable, if there is any. In case any argument of a function node gets changed, copying will start "upwards," which is achieved by the function Cons, which behaves as explained in Section 4.

```
Function DESCEND(ui);
IF ui = NIL
THEN Exit with NIL
ELSE IF Variable(ui)
  THEN Exit with EXPLORE-VARIABLE(ui)
  ELSE IF Constant(ui)
    THEN Exit with ui
    ELSE IF Ready(ui)
      THEN Exit with Ready(ui)
      ELSE out := EXPLORE-ARGUMENTS(Arguments-of(ui));
        IF out = Arguments-of(ui)
        THEN Ready(ui) := ui
        ELSE Ready(ui) := Cons(Head-of(ui), out);
        ENDIF;
        Exit with Ready(ui);
      ENDIF;
    ENDIF;
  ENDIF;
ENDIF.
```

The function EXPLORE-ARGUMENTS gets a list of arguments and will explore them in a depth-first mode.

```
Function EXPLORE-ARGUMENTS(list-of-arguments);
IF list-of-arguments = NIL
THEN Exit with NIL
ELSE 1st-new := DESCEND(1st(list-of-arguments));
  tail-new := EXPLORE-ARGUMENTS(Tail(list-of-arguments));
  IF 1st-new ≠ 1st(list-of-arguments) OR
    tail-new ≠ Tail(list-of-arguments)
  THEN Exit with Cons(1st-new, tail-new)
  ELSE Exit with list-of-arguments;
  ENDIF;
ENDIF.
```

The functions 1st and Tail have the standard list-processing meaning and correspond in LISP respectively with CAR and CDR.

The total number of invocations of EXPLORE-VARIABLE, DESCEND, and EXPLORE-ARGUMENTS is limited by the number of nodes and the number of edges due to the marking with "Ready."

7. Conclusion

The Paterson–Wegman algorithm requires only that identical leaf nodes are shared, i.e., are represented uniquely (but accepts and works faster when other identical structure is shared). The Paterson-Wegman algorithm can be extended with a post-processor, also working in linear time, which will transform an ordered substitution into a regular non-ordered substitution.

Appendix

We give here the code of the core procedure Finish, using the terminology of the pre- and post-processor, and integrating the correction and modifications, as described in Section 5.

```
Procedure Finish(r);
IF Complete(r)
THEN Exit
ELSE IF Pointer(r) ≠ NIL
  THEN Exit with failure
  ELSE Create new Stack with operations Push(*) and Pop;
    Pointer(r) := r;
    Push(r);
    WHILE Stack ≠ NIL DO
      s := Pop;
      IF s, r have different function symbols
      THEN Exit with failure;
      ENDIF;
      FOR-EACH parent t of s DO
        Finish(t);
      OD;
      FOR-EACH link (s, t) DO
        IF Complete(t) or t = r
        THEN Ignore t
        ELSE IF Pointer(t) = NIL
          THEN Pointer(t) := r;
            Push(t)
          ELSE IF Pointer(t) ≠ r
            THEN Exit with failure
            ELSE Ignore t; (since t is already on STACK)
            ENDIF;
          ENDIF;
        ENDIF;
```

```
      OD;
      IF s ≠ r
      THEN IF Variable(s)
        THEN Subs(s) := r;
          Add s to SIGMA (input to BUILD-SIGMA)
        ELSE Create links{ jth son(r), jth son(s) | 1 ≤ j ≤ q};
        ENDIF;
      ENDIF;
      Complete(s) := true;
    OD;
    Complete(r) := true;
    Exit;
  ENDIF;
ENDIF.
```

Remarks for an implementation in LISP. Realization of ASSIGN-PARENT-POINTER in LISP is somewhat subtle, because when Node is non-terminal then we cannot store the Parents-of(Node) values on the property list of the predicate/function symbol associated with Node. A feasible way is to generate with GENSYM an atom, say G, replace, with RPLACA, the predicate/function symbol in Node by G, associate the predicate/function symbol, the Node and the Parents-of list with G, and subsequently add G instead of Node to FNODES.

In case, the pre-processor has modified the input with RPLACA then just before returning with a failure or with a substitution the original predicate/function should be restored. This can be done easily by scanning the list of nodes FNODES. When a node on FNODES, say x, has Ready(x) $\neq$ NIL then the "head" in Ready(x) needs to be replaced also by the original predicate/function.

REFERENCES

1. C. L. WANG AND R. C. T. LEE, "Symbolic Logic and Mechanical Theorem Proving," Academic Press, New York, 1973.
2. D. W. LOVELAND, "Automated Theorem Proving: A Logical Basis," North-Holland, Amsterdam, 1978.
3. N. J. NILSSON, "Principles of Artificial Intelligence," Tioga, Palo Alto, Calif., 1980.
4. M. S. PATTERN AND M. N. WEGMAN, "Linear Unification," IBM Research Report 5304, IBM, 1976.
5. M. S. PATERSON AND M. N. WEGMAN, Linear unification, *in* "Conference Record of the 8th Annual ACM Symposium on the Theory of Computing," May 1976, pp. 181–186.
6. M. S. PATERSON AND M. N. WEGMAN, Linear unification, *J. Comput. System Sci.* **16**, No. 2 (1978), 158–167.
7. J. A. ROBINSON, A machine-oriented logic based on the resolution principle, *J. Assoc. Comput. Mach.* **12**, No. 1 (1965), 23–41.