IMPROVING THE EFFICIENCY OF HIGHER ORDER UNIFICATION

Jared L. D rlington
Institut fur Informatik
University of : onn, Germany

Abstract

The sources of inefficiency in currently existing higher order unification algorithms are investigated. Aside from such theoretical difficulties as the undecidability of unification in third order logic, and the existence of infinite unifiers and the lack of a polynomial bound on the number of applications of the "imitation" rule even in the monadic subcase of second order unification, the current algorithms suffer from a built-in inefficiency due to their introduction and subsequent elimination of many auxiliary functional variables, and to the nondirected nature of the substitutions made by the "projection" rule. It is argued that a procedure based on attempting to match the argument or arguments of a functional or predicate variable with the subterms of the other formula in the unification can decide the possibility of unification and generate the resulting unifiers much more directly than the theoreticallv comolete algorithm.

Descriptive Terms

Higher order logic, resolution, theorem proving, unification

The recent interest in developing linear and near-linear unification algorithms for first order languages (see for example Huet 1976, Paterson and Wegman 1976, and work referred to by them) has, with few exceptions, not been matched by a corresponding effort to improve the efficiency of higher order unification. A linear unification algorithm is of course-out of the question for higher order logic in general, for not only is unification known to be undecidable in third order logic (Huet 1973; Lucchesi 1972), but even in the monadic subcase of second order logic it has been shown (by Winterstein 1976) that there exists no polynomial upper bound on the number of applications of the "imitation" rule which together with the "projection" rule plays an essential role in a complete higher order unification algorithm. Although linear bounds on the number of their applications do exist in some cases (Winterstein 1976), the two above-mentioned rules are inherently inefficient, (a) because of their introduction and subsequent elimination of many auxiliary functional variables, and (b) because of the nondirected nature of the substitutions made by the projection rule. The new function symbols referred to in (a) appear to play an essential role in the unification of certain pairs of formulae -- for example, where the head of one formula is a higher order variable 'f' which occurs also within the other formula but not within an argument of another variable 'g' -- but there are many cases in which they merely delay the unification process. For example, suppose one is unifying

$$fA \quad \text{with} \quad K^nA \qquad \text{(Example 1)}$$

in second order logic, where

$$\text{type}(A) = i \quad \text{(individual)}$$

$$\text{type}(f) = \text{type}(K) = (i \rightarrow i)$$

f is a variable, A and K are constants, and n is any positive integer. The usual unification procedure based on imitation and projection (see for example Huet 1975, Winterstein 1976, and Jensen and Pietrzykowski 1976, the last of whom also use other rules) introduces and eliminates for this example n new functional variables of type $(i \rightarrow i)$, and after n+1 imitations and an equal number of projections generates the two unifiers

$$\langle f, \ \lambda u.K^nA \rangle \qquad \text{(i)}$$

$$\langle f, \ \lambda u.K^nu \rangle \qquad \text{(ii)}$$

In general, if one is unifying e_1 and e_2 in second order logic, where

$$e_1 = f(a_1, \ldots, a_m)$$

$$e_2 = P(b_1, \ldots, b_n)$$

$\text{type}(e_1) = \text{type}(e_2)$, and f is a functional or predicate variable with

$$\text{type}(f) = (i_1, \ldots, i_m \rightarrow i) \quad \text{or}$$

$$(i_1, \ldots, i_m \rightarrow o)$$

(o = "truth value"), the

Imitation rule

yields for f

$\langle f, \lambda u_1 \dots u_m.$

$P(q_1(u_1, \dots, u_m), \dots, q_n(u_1, \dots, u_m))\rangle$

where the q_j are new variables with

type$(q_j) = (i_1, \dots, i_m \to i)$

type$(u_k) = i$

resulting in the "successor node"

$$\{\langle q_1(a_1, \dots, a_m), b_1\rangle \dots \langle q_n(a_1, \dots, a_m), b_n\rangle\}$$

while the

<u>Projection rule</u>

generates m substitutions for f

$$\langle f, \lambda u_1 \dots u_m. u_k\rangle \quad 1 \le k \le m$$

with type$(u_k) = i$,

resulting in m successor nodes

$$\{\langle a_1, e_2\rangle\}, \dots\dots, \{\langle a_m, e_2\rangle\} .$$

If n=3 in Example 1, imitation-cum-projection generates the tree shown in Figure 1 below. It is clear that each increase of 1 in n, the number of K's, adds one new function symbol q_j and therefore one imitation and one projection, without adding any unifiers. One reason for this waste of effort is that imitation-cum-projection as it is ordinarily used does not take account of the fact that projecting f onto one of its arguments a_k, in the unification of e_1 and e_2 in second order logic, succeeds if and only if the resulting successor node $\{\langle a_k, e_2\rangle\}$ is unifiable. Imitation-cum-projection, however, first makes the projection substitutions for f and then compares the results with e_2. In comparison with imitation, projection is in this respect less like unification, which should ideally make only substitutions that are in some way <u>directed by</u> the other formula in the unification, and more like some of the pre-resolution first order Herbrand proof procedures that, instead of unification, employed substitution for variables followed by comparison of the resulting formulae.

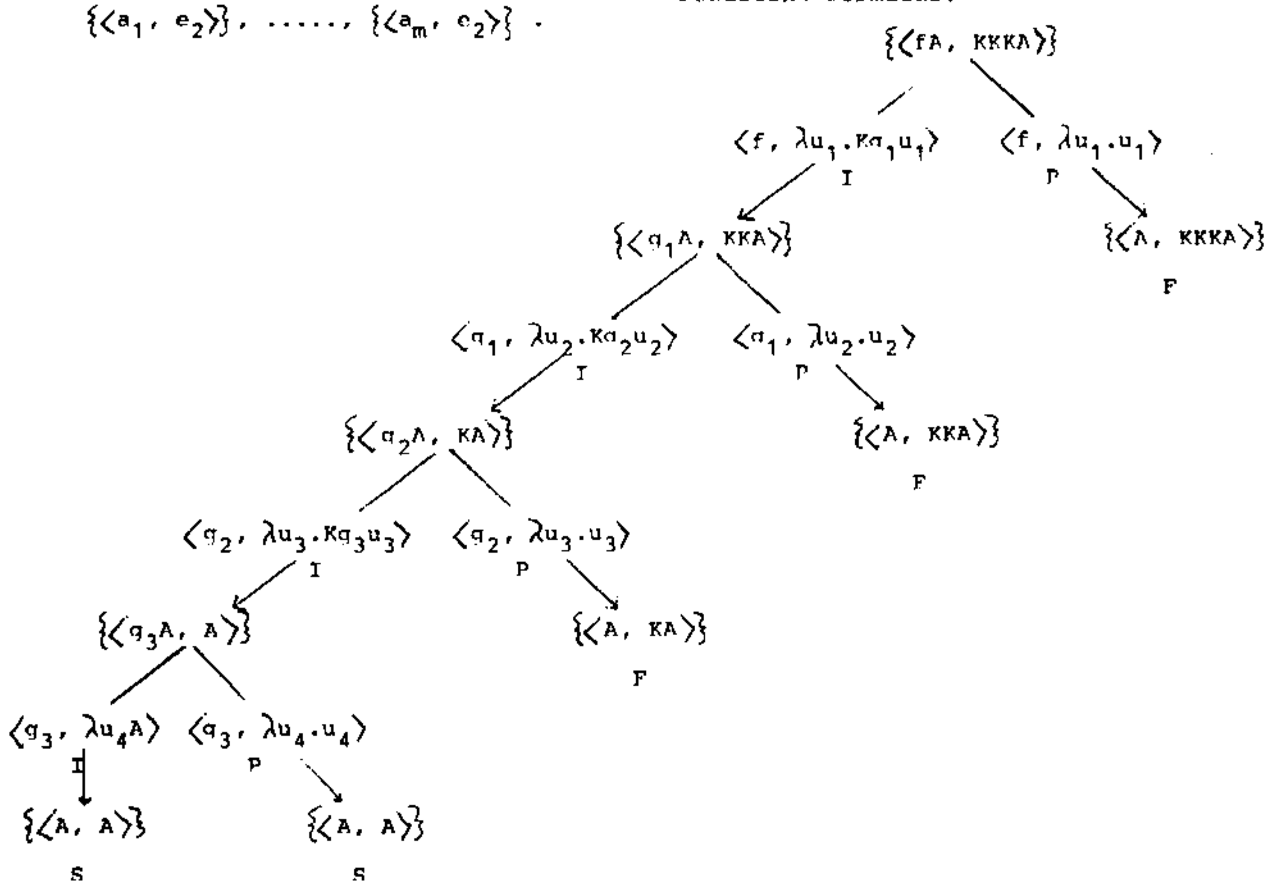

Fig. 1. Unification tree for Example 1 (n=3) given bv imitation~cum-projection.

To see how unification mav be performed more efficientlv in examples like the preceding, let us consider the unification of

$$\{\langle f(a_1),\ e_2\rangle\}$$

where the types are as before, and where f is "monadic" and does not occur in e_2. For every subformula t of e_2, the unification tree includes some node containing the pair

$$\langle q_j(a_1),\ t\rangle$$

where q. is either f (if t is e_2) or is a variable introduced into the tree bv imitation (see for example Figure 1). Applying imitation to every q_j oduces the unifier

$$\langle f,\ \lambda u.e_2\rangle$$

as in unifier (i) of Example 1, but applying projection to a g. leads to a successful unification if and only if

$$\langle a_1,\ t\rangle$$

leads to a successful unification. This suggests that one may "screen out" in advance projections that are bound to fail, bv first checking the unifiabilitv of a- with the various subterms t in e2. Tn Example 1, there is clearlv just one projection that can succeed, since there is just one subterm of K A, namelv 'A', that the arqument 'A' of f matches, leading to the unifier (ii). Here, the match of A with A is trivial, but in other cases a fair amount of work mav be reauired completely to unifv an argument a- of f with a subterm t of e~, especially If either or both contain functional variables. It is therefore more efficient to set UP a "search pattern" corresponding to a-, but based on less than complete unification, that will "match" just those subterms of e~ that are potentially unifiable with a. wnile ignoring the obviously impossible cases', such as those in which a1 and t begin with opposing constants. specifically:

P1 : If a_1 is <u>flexible</u> (i.e. has a variable head), then

PATTERN = any term of type i.

P2: If not, then

PATTERN = anv flexible term or any term with the same head as a1 .

A program based on this idea and coded in SN0B0L4 is being experimented with on the IBM 360/50 at the GMD/Bonn. Earlier programs performed the pattern matching bv means of complete unification of a1 with each subterm t of e2, a technicme that was called "f-matchlna", and the current method is a refinement of this technicme. Moreover, it can be incorporated into anv proaram that uses imitation and projection. In cases where it is applicable, it has the advantages of introducing no new functional variables and of screening out a priori impossible projections in advanc.e.

To explain the method in greater detail, we start with a node

$$\{\langle A_1,B_1\rangle\ \ldots\ldots\ \langle A_n,B_n\rangle\} \qquad \text{(N)}$$

of a unification tree, containing one or more unification pairs <A.,B > where each A, is of the same tvpe as B.. Imitation-cum-projection generates successor nodes to N by choosinq a pair <A.,B.> from N accordinq to some criterion and applvinq imitation and projection to A. and B. in all nossible wavs. The unification[1] substitutions resultinq from each application are then applied to all the nairs in N, thereby generating a successor node after all possible "lambda normalisations" and simplifications have been made. If no unification substitutions are applicable to anv pair in N, it is labelled either 's' for "success" or 'F' for "failure", as the case mav be. If the pair sinaled out from N

$$\langle f(a_1),\ e_2\rangle \qquad \text{(P)}$$

the simnlified nrocedure first checks whether the nair P bears a subscript u.. If so, it proceeds directlv to the aeneration of nodes N" described below, but if not it adds a subscript u. to P, where u. is a variable not occurriita in N or its predecessors, and will serve as the variable to be used in Kubseauent lambda abstractions. It then qenerates a successor node N from P, based on the substitution

$$\langle f,\ \lambda u_j.e_2\rangle \qquad (f_0)$$

that results from imitation alone (in Example 1, this is unifier (i)). For each subterm t of e_2 that PATTERN matches, it- next aenerates a node N. that results from replacing P in N $\langle a_1,\ t\rangle$,1 d a unification substitution

$$\langle f,\ \lambda u_j.e_2^t\rangle \qquad (f_1^t)$$

where e_2 results from replacing the matched term t by u.. In Example 1, there is just one subterm t that PATTERN matches, namelv 'A', producinq the successor node $\{\langle A,\ A\rangle\}$ and the corresponding unification substitution (ii).

At the time of generating each N_1^t, the procedure examines the lambda expression for f_1^t to see whether a node N_2^t can be generated, as follows: if there is some subterm appearing before the first lambda-bound variable u_j in e_2^t (this cannot happen if e_2^t is monadic), then a successor node N_2^t is generated by replacing P in N by

$$\langle f(t),\ e_2^t\rangle_{u_j}\ \langle a_1,\ t\rangle$$

if a_1 is flexible, or

$$\langle f(a_1),\ e_2^t\rangle_{u_j}\ \langle a_1,\ t\rangle$$

otherwise. The nodes N_2^t correspond to applying more than one projection in the unification of P, in case a_1 is simultaneously unifiable with two or more subterms of e_2 and may therefore be replaced by the same u_j in the lambda expression. The restriction on the generation of nodes N_2^t (don't scan past a u_j) is necessary to prevent multiple derivations of the same unifier. Without it, for example, the unifier

$$\langle g_1,\ \lambda u_1.P(u_1,u_1,u_1,u_1)\rangle$$

for $\{\langle g_1(A),\ P(A,A,A,A)\rangle\}$

would be generated many times over.

The procedure for generating successor nodes just described may be compared with imitation-cum-projection in the unification of

$$f(g(A)) \quad \text{with} \quad P(K(A),\ B) \quad \text{(Ex. 2)}$$

where $\text{type}(A) = \text{type}(B) = i$

$\text{type}(f) = (i \rightarrow o)$

$\text{type}(g) = \text{type}(K) = (i \rightarrow i)$

$\text{type}(P) = (i, i \rightarrow o)$

It may be seen from Figures 2 and 3 that both procedures generate the same six unifiers for Example 2, namely

$$\langle f,\ \lambda u_1.P(KA,B)\rangle \quad \text{(i)}$$
$$\langle f,\ \lambda u_1.P(u_1,B)\rangle\langle g,\ \lambda u_2.KA\rangle \quad \text{(ii)}$$
$$\langle f,\ \lambda u_1.P(u_1,B)\rangle\langle g,\ \lambda u_2.Ku_2\rangle \quad \text{(iii)}$$
$$\langle f,\ \lambda u_1.P(Ku_1,B)\rangle\langle g,\ \lambda u_2.A\rangle \quad \text{(iv)}$$
$$\langle f,\ \lambda u_1.P(Ku_1,B)\rangle\langle g,\ \lambda u_2.u_2\rangle \quad \text{(v)}$$
$$\langle f,\ \lambda u_1.P(KA,u_1)\rangle\langle g,\ \lambda u_2.B\rangle \quad \text{(vi)}$$

but the simplified procedure gets them considerable more directly, generating a tree of depth two instead of depth five.

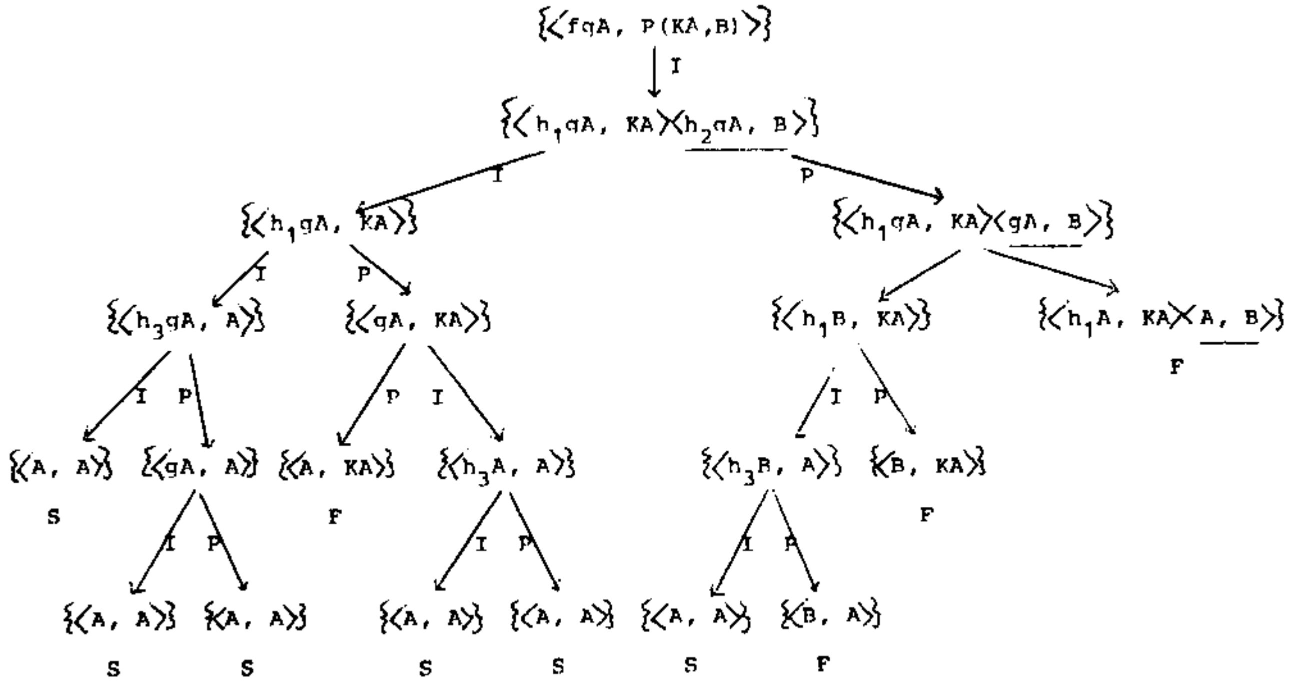

Fig. 2. Imitation-cum-projection unification tree for Example 2.

$$\begin{tikzcd}
 & & & \{\langle fgA,\ P(KA,B)\rangle_{u_1}\} \arrow[dlll, dash] \arrow[dll, dash, dashed] \arrow[d, dash] \arrow[drr, dash] & & & \\
\langle f,\ \lambda u_1.P(KA,B)\rangle \arrow[d, "N_0"] & \langle f,\ \lambda u_1.P(u_1,B)\rangle \arrow[d, "N_1"] & & \langle f,\ \lambda u_1.P(Ku_1,B)\rangle \arrow[d, "N_1"] & & \langle f,\ \lambda u_1.P(KA,\ u_1)\rangle \arrow[d, "N_1"'] \arrow[dr, "N_2"] & \\
\{\langle P(KA,B)\ ,\ P(KA,B)\rangle\} & \{\langle gA,\ KA\rangle_{u_2}\} \arrow[d, dash] \arrow[dr, dash] & & \{\langle gA,\ A\rangle_{u_2}\} \arrow[d, dash] \arrow[dr, dash] & & \{\langle gA,\ B\rangle_{u_2}\} \arrow[d, dash] & \dfrac{\{\langle fB,\ P(KA,u_1)\rangle_{u_1}}{\langle gA,\ B\rangle\}} \\
S & \langle g,\ \lambda u_2.KA\rangle \arrow[d, "N_0"] & \langle g,\ \lambda u_2.Ku_2\rangle \arrow[d, "N_1"] & \langle g,\ \lambda u_2.A\rangle \arrow[d, "N_0"] & \langle g,\ \lambda u_2.u_2\rangle \arrow[d, "N_1"] & \langle g,\ \lambda u_2.B\rangle \arrow[d, "N_0"] & F \\
 & \{\langle KA,\ KA\rangle\} & \{\langle A,\ A\rangle\} & \{\langle A,\ A\rangle\} & \{\langle A,\ A\rangle\} & \{\langle B,\ B\rangle\} & \\
 & S & S & S & S & S &
\end{tikzcd}$$

Fig. 3. Simplified unification tree for Example 2.

(The apparent equality in size of the diagrams in Figures 2 and 3 is due to the fact that, in order to save space, the unification substitutions have been omitted from Figure 2.) After generating the first success node, N_0, in Figure 3 above, the simplified procedure then searches in '$P(KA,B)$' for possible matches for 'gA'; since gA is a flexible term, the search pattern will match any term of type i, namely 'KA', 'A' or 'B', generating the three N_1-nodes $\{\langle gA,\ KA\rangle\}$, $\{\langle gA,\ A\rangle\}$ and $\{\langle gA,\ B\rangle\}$, with their corresponding substitutions for f. Since one of these f's, namely

$$\langle f,\ \lambda u_1.P(KA,\ u_1)\rangle$$

has a term 'KA' appearing before u_1, a node N_2 is generated in an attempt to get a value for f containing two or more occurrences of u_1, namely

$$\langle f,\ \lambda u_1.P(u_1,\ u_1)\rangle \quad \text{or}$$

$$\langle f,\ \lambda u_1.P(Ku_1,\ u_1)\rangle$$

but this is not possible since 'B' will not match any part of 'KA': in other words, 'gA' will not simultaneously match 'B' and some part of 'KA'. The three N_1-nodes are then processed in the same way as the root node, resulting in five success nodes. The procedure may also be applied to cases where f is nonmonadic, namely

$$\langle f(a_1,\ \ldots,\ a_m),\ e_2\rangle \qquad m > 1$$

Here, however, it is necessary to apply the procedure m times, searching separately in e_2 for a match for each argument a_i. It is fair to say that, as the degree of f increases, the advantages of our procedure vis-a-vis imitation-cum-projection become progressively less. Furthermore, it will not handle all cases in which the same functional or predicate variable occurs in both terms to be unified, such as

$$\{\langle fKA,\ KfA\rangle\} \qquad \text{(Example 3)}$$

in second order logic, which generates infinite unifiers, but even here our procedure generates two unifiers, namely

$$\langle f,\ \lambda u_1.u_1\rangle \quad \text{and}$$

$$\langle f,\ \lambda u_1.Ku_1\rangle\ .$$

This type of case, however, does not arise in the applications of second (and higher) order theorem proving that we have been making, such as to automatic program synthesis (Darlington 1976) and to proofs of theorems in topology. The simplified procedure has in fact been extended to perform certain unifications in third and higher order logic, though we do not at present have a general characterisation of the limits of its applicability beyond second order logic. Within second order logic itself, it appears to be equivalent to imitation-cum-projection for unifications whose trees contain no pairs of the sort found in Example 3, where the same higher order variable occurs in both e. and e_2. The argument in outline is that each successful path containing n projections (n * 0) in an imitation-cum-projection tree corresponds uniquely to a successful path containing n N - and N_2~nodes in the simplified tree.

A practical result of the simplified procedure is, in its application to "constrained resolution" (Huet 1973a), to decide more quickly that only one unifier is possible in a given case, and therefore to reduce the number of constraints that need be generated. For example, if a particular resolvent is based on the unification of 'fA' with a complex e_2, constrained resolution would normally decide that there exists no "most general unifier" in this case and therefore generate only a skeletal resolvent with {fA,e_2} attached to it as a "constraint" to be unified later. If, however, e_2 contains nothing that 'A' will match, then there is only one possible unifier, namely *4.i,* Au.e_2^ , and no constraint need be generated. Alternatively, if 'A^1 matches only one term t of e_2, then ^f, >u.e_2> may be taken as "the^ unifier, leaving out <f, ^lu.e_2> , since unifiers of this sort, based on imitation alone, seldom if ever lead to "useful" inferences — for example, imitation alone will not permit the derivation by resolution of P(B) from P(A), A = B, and x + y V ^f(A) V f(B), which requires the unifier <f, ^u.Pu>. Similar heuristics are employed by BUlow (1976) , who does a certain amount of "look-ahead" during higher order resolution in order to rule out branches resulting from nonproductive or impossible imitations or projections, thereby reducing the number of constraints. Another procedure related to ours is Bledsoe's (1977) method for finding values of set variables, equivalently monadic predicate variables, in topology, program verification and other theorem proving domains, where the full power of imitation-cum-projection is not required. In view of the theoretical difficulties in achieving significant improvements in higher order inference in general, research of this sort into improving its efficiency in important special cases is particularly vital if the inclusion of higher order features in automatic theorem provers, be they based on resolution or natural deduction, is to become a practical proposition.

References

Bledsoe, W. W. (1977). A Maximal Method for Set Variables in Automatic Theorem Proving. Report ATP-33, Departments of Mathematics and Computer Science, University of Texas.

BUlow, R. (1976). Resolutlonsverfahren fUr die Typentheorie. Diplomarbeit, University of Bonn, Institut fur Informatik.

Darlington, J. L. (1976). Automatic synthesis of SN0B0L programs. In Computer Oriented Learning Processes, J. C. Simon ed., Noordhoff-Leyden.

Huet, G. P. (1973). The undecidability of unification in third order logic, Information and Control 22(3), 257-267.

Huet, G. P. (1973a). A mechanization of type theory. In Proceedings of IJCAI-3, Stanford, California.

Huet, G. P. (1975). A unification algorithm for typed lambda calculus, Theoretical Computer Science 1(1), 27-57-

Huet, G. P. (1976). Resolution d*Equations dans des Langages d'Ordre 1,2, .., *xjj* . These de doctorat d'etat, Universite Paris VII.

Jensen, D. C. and Pietrzykowski, T. (1976). Mechanizing w-order type theory through unification, Theoretical Computer Science 3(2), 123-171.

Lucchesi, C. L. (1972). The Undecidability of the Unification Problem for Third Order Language's. Report CSRR 2059 Department of Applied Analysis and Computer Science, University of Waterloo.

Paterson, M. S. and Wegman, M. N. (1976). Linear unification, Proceedings of 8th Annual ACM Symposium on Theory of Computing, 181-186.

Winterstein, G. (1976). Unification in Second Order Logic. FB Informatik, University of Kaiserslautern.