TR-402

On Structures for Efficient Unification Join and Select Operations

by
L. Henschen. S. Lee(North western Univ.)
M. Murakami and Y. Morita

June. 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

On Structures for Efficient Unification Join and Select Operations[1]

Lawrence J. Henschen and Sanggoo Lee
Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, Illinois 60201

M. Murakami and Y. Morita
Institute for New Generation Computing Technology
Tokyo, Japan

Abstract

Recent research at ICOT has focused on extending the concept of a relational database to that of a relational knowledge base; one that allows more general terms as elements of tuples in relations. This requires a more sophisticated treatment of the relational operations, in particular, join and select. We propose in this paper some techniques for reducing the amount of work to compute unification joins and selects. The main idea is to take advantage of common structure between terms when computing unifiers. This technique allows a faster computation of certain additional pairs of terms when the Most General Unifier (MGU) of one pair is found.

1. Introduction

Recent research at ICOT has focused on extending the concept of a relational database to that of a relational knowledge base [MURAS5, MORI86]. One aspect of this research is to allow general *terms* as elements of tuples in relations. This, in turn, requires a more sophisticated treatment of the relational operations, in particular, join and selection. The natural extension of the *equi-join* and *select* operators is to require corresponding attributes in the relations to *unify* as opposed to just being equal. Thus, for example, to *unify-join* two relations R_1 and R_2 on attributes a_1 and a_2, all pairs of terms (t_1,t_2) with t_i as attribute a_i in some tuple of R_i must be tested for unification. For any pair that is unifiable, the join relation will have a tuple constructed from the pairs of tuples from R_1 and R_2 with the Most General Unifier (MGU) applied.

The key point here is that many unification tests need to be performed. It is to be expected in a rich knowledge base that a relation may have thousands or even

[1] This work was supported in part by the National Science Foundation under Grant DCR-860-8311

millions of tuples. Joining two such relations by unification could require a prohibitive number of unification tests. For example, if each relation has 1,000 tuples, then one million pairs of terms would need to be tested. If the average time per unification could be made as low as 100 μsec by the use of specially designed hardware, the join operation will require 100 seconds *just for* the unification tests; there will, of course, be additional time for retrieving the terms from long term storage, generating the pairs, formulating the new tuples and applying the MGUs. For two relations with 100,000 tuples each, the time for unification goes up to 10,000 seconds. It is, therefore, crucial to develop methods for processing unification joins that reduce the amount of unification necessary to compute a join and optimize the computation of MGUs for those pairs of terms that *do* unify.

Such reduction and optimization can be accomplished by a combination of algorithms, data structures, and hardware specially designed for knowledge bases. An important first step in this direction was taken in [MORI86]. They introduce the idea of filters to eliminate pairs of terms that can readily be seen not to unify. They also propose data structures and a hardware system based on their filtering mechanism. They give an example in which the number of unification tests required is reduced by about 50% and in which the number of unification tests that succeed is about 67%.

We propose in this paper some alternative techniques for further reducing the amount of work to compute unification joins and selects. The main idea is to take advantage of common structure between terms when computing unifiers. This technique allows a faster computation of certain pairs of terms for the initial unification test. A system can maximize the number of pairs to which this faster computation can be applied. This technique requires a modification to the storage scheme proposed previously as discussed in section 5. We also mention some other filtering mechanisms which may have a higher efficiency than the *outer-most* total ordering of [MORI86], perhaps as high as 90–95% in typical cases. Such a scheme would reduce the number of unification attempts and insure that almost every attempt resulted in success.

In section 2, we discuss the main advantage of using the *generality partial order*. In section 3, we give the algorithm for computing MGUs of less general terms from more

general terms (top-down), and in section 4, the algorithm for computing MGUs of more general terms from less general terms (bottom-up). In section 5, a comparison of storage requirements is given, and in section 6, we cite some additional filtering mechanisms.

2. Advantages of Generality Ordering

In [MORI86], the partial ordering given by generality of terms is extended to a total ordering in two different ways, *left-most* and *outer-most*. These total orderings are useful in filtering out pairs of terms that cannot unify and therefore should not be sent to the unification unit of the unification engine. The total ordering is especially useful for this filtering purpose because it allows terms to be sorted, which in turn allows for an efficient pair generation algorithm.

However, the major effort in unification join and unification selection is the actual unification of pairs of terms. For this effort, the following two properties, which hold for the partial order determined by generality but not for either total order, will be most useful :

1. if s is more general than t and θ is the minimal substitution such that $s\theta = t$, then s and t are unifiable and θ is a MGU of s and t.
2. if s_1 and s_2 are unifiable with MGU σ and if t_i is more general than s_i for $i=1,2$ and if θ_i are the minimal substitutions mapping t_i to s_i, then t_1 and t_2 are unifiable and their MGU can be computed directly from σ, θ_1 and θ_2 with complexity no worse than and normally much better than computing the MGU of t_1 and t_2 with the normal unification algorithm.

We propose maintaining the partial order lattice for all terms in the knowledge base along with the θs by which less general terms are instances of more general ones. When a unification join, for example, is attempted, the sublattices for the two lists of terms should be formed; these sublattices may very well be small enough to fit in the core memory of the unification engine of [MORI86]. We may then proceed either top down or bottom up.

In the top down scheme, we find the highest level pairs that unify and compute unifiers for pairs below directly from the top unifier and the θ substitutions of the

lattice. The downward process stops when a non-unifiable pair is found. This is justified by the contrapositive of remark 2 above. It is to be expected that the θ substitutions in the lattice will be much simpler than the actual terms themselves. Thus, computing from θs should be much faster than computing from the terms directly.

As for the bottom up method, we propose to form a list of candidate pairs as in [MORI86]. However, there will now be three distinct processes for finding the set of all unifiable pairs. First, any pair in which one term is more general than the other requires no real computation; they are unifiable, and the MGU is already available in the lattice. Second, in order to take advantage of the second remark, we should attempt to unify pairs which occur as low as possible in the lattice; when such a pair is found, all pairs above it in the lattice are removed from the list of pairs to send to the normal unification unit. Third, we directly compute the MGUs for all pairs of terms above two such unifiable terms. If there is a great deal of common structure among the terms in the knowledge base, then the first and third processes may be expected to provide a large percentage of the unifiers; this is desirable because each of these processes is more efficient than general unification. On the other hand, if there is little or no common structure, most pairs will require the normal unification test, a situation no worse than that proposed in [MORI86].

Concerning storage, the lattice specifying the partial order and the corresponding substitutions should be a permanent part of the knowledge base storage and can be stored on disk. It is possible that an efficient storage scheme for the actual terms of the knowledge base itself can be derived from this lattice, so the storage for the lattice may in fact reduce the storage required for the knowledge base itself. This will be examined in section 5. During an actual join or select operation, it will very likely be necessary to store the relevant sublattice in the local memory of the unification engine. This will require a somewhat larger local memory, perhaps, than in [MORI86]. However, the kind of storage scheme discussed in section 5 may again offset these storage requirements.

As for computation, we need an efficient algorithm for deriving the unifiers of less general terms from those of more general ones, or vice versa. These are provided in sections 3 and 4. In the top down process, when two terms, s_1 and s_2, are found to be

unifiable, we process all pairs of less general terms in a special order down the lattice; first process s_1 with the immediate descendants of s_2, then with their immediate descendants, then s_2 with the immediate descendants of s_1, etc. That is, each new pair is obtained from a given pair by taking an immediate descendant of one of the terms. In the process of doing so, we would want to avoid visiting the same term (which may be a descendant of several terms) more than once. The procedure for generating the set of all pairs of descendants in this manner is quite straightforward. The bottom up process is quite similar.

Finally, there is the overhead of maintaining the lattice and its related substitution when terms are added to or deleted from the knowledge base. Deleting a term is fairly easy — just delete the node in the lattice, reattach the links and adjust the substitutions. For example, if we had

$$\underset{f\,(x\,,h\,(y\,))}{t_1} \xrightarrow{\theta_1} \underset{f\,(x_1,h\,(a\,))}{t_2} \xrightarrow{\theta_2} \underset{f\,(a\,,h\,(a\,))}{t_3}$$

and t_2 is deleted, then the lattice must be adjusted as follows.

$$\underset{f\,(x\,,h\,(y\,))}{t_1} \xrightarrow{\theta} \underset{f\,(a\,,h\,(a\,))}{t_2}$$

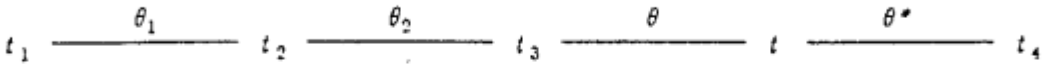
The new substitution θ is obtained from the composition of θ_1 and θ_2, more specifically, $\theta = \theta_1\theta_2 - \theta_2$, assuming t_1 and t_2 have no variables in common.

When a new term, t is added, the computation is more difficult. We must find the lowest position(s) where t is an instance of an existing term. We must then detach existing links and reformulate the substitutions. For example, if the knowledge base looked like

$$t_1 \xrightarrow{\theta_1} t_2 \xrightarrow{\theta_2} t_3 \xrightarrow{\theta_3} t_4$$

and a term t was added which was an instance of t_3, then t is also an instance of t_1 and t_2, but its proper position is below t_3. If t_4 is an instance of t, then t_4 will no longer be a *direct* subordinate of t_3, in which case the substitutions should be modified

as follows.



where $t_3\theta \equiv t$ and $t\theta^* \equiv t_4$. If t_4 is not an instance of t, then θ_3 remains as well as its link, and the link to t from t_3 forms a branch.

Of course, the storage scheme is a lattice, and there may be several links into and out of a term that is being deleted or inserted, so the problem of updating is not quite so simple. The update overhead may be reduced by proper indexing schemes. We discuss this issue in section 6.

3. Computing Unifiers for Less General Terms (Top-down)

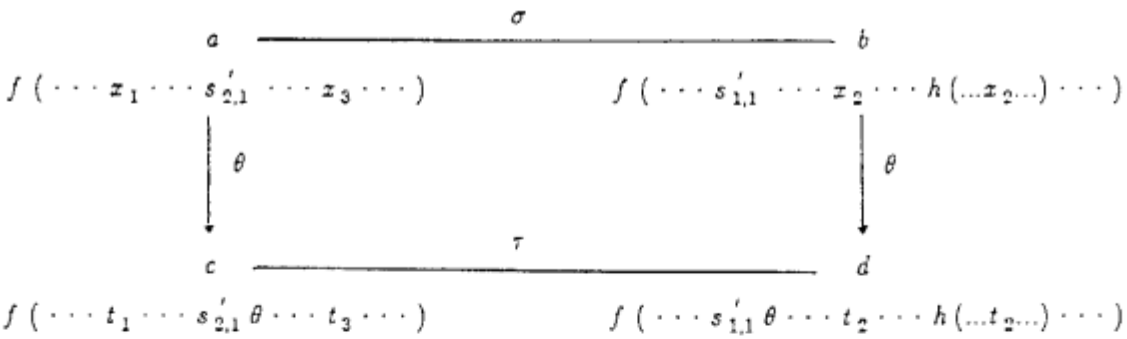


Figure 1

In this section we discuss the computation of unifiers in a top-down fashion. For this discussion let us assume the following notation:

a, b, c and d are terms with no variables in common;

a and b are unifiable with MGU σ;

c and d are instances of a and b by the substitution θ;

let the variables occurring in a and b be denoted by $x_1, \cdots, x_n$, and let those

occurring in c and d be $y_1, \cdots, y_m$; for this discussion, it is not necessary to be able to refer to the separate variables of a and b or c and d; note that $\{x_i\}$ and $\{y_j\}$ are disjoint;

suppose c and d are unifiable with MGU τ.

Pictorially, we have Figure 1. Because no variable in a or b occurs in c and d, θ must substitute a term for every x_i even if that term is just one of the variables y_j. We may assume that θ is minimal so that it does not substitute for any variable other than $x_1, \cdots, x_n$. On the other hand, σ need not substitute a term for every x_i; for example, $f(x_1)$ vs. $f(g(x_2))$. Similarly, τ need not substitute for every y_j. Let us number the x and y variables so that the variables appearing in σ and τ are the first of their respective sets. Then we have the following;

$$\sigma = \{s_i/x_i \mid i = 1, \cdots, n^*\}$$
$$\theta = \{t_i/x_i \mid i = 1, \cdots, n\}$$
$$\tau = \{u_i/y_i \mid i = 1, \cdots, m^*\}$$

where $n^* \leq n$ and $m^* \leq m$.

The t_i terms in Figure 1 are those from θ substituted for x_i. The $s'_{i,j}$ terms are those that occur at positions corresponding to the variable x_i. For example, suppose we had two terms,

$$f(x_1, g(x_2, x_1), a) \quad \text{and} \quad f(h(x_4), g(h(x_5), x_3), x_4),$$

Since $h(x_4)$ and x_3 appear in positions corresponding to variable x_1, they both are s'_1 terms ($s'_{1,1} = h(x_4)$, $s'_{1,2} = x_3$). Likewise, $s'_{2,1} = h(x_5)$, $s'_{3,1} = x_1$, and $s'_{4,1} = a$. Since x_5 does not have a term corresponding to its position, there are no s'_5 terms. Clearly, there is at least one term s'_i for each $i \leq n^*$. As noted above, there may be many.

Much of the theory developed in this section depends on some facts about unifiers. The reader is recommended to refer to [CHAN73] for definitions of *substitutions, composition* of substitutions, and other terminologies that we use without formal definitions. We define here the *combination* of substitutions and present the Unification Algorithm from [HAYE73]. The algorithm differs from the Robinson's Algorithm [ROBI63] only in restricting the direction of substitution for variable–variable pairs (step 3). We also present four basic lemmas without proofs, which are quite straightforward.

Definition Let $\theta = \{t_1/x_1, \ldots, t_n/x_n\}$, $\lambda = \{u_1/y_1, \ldots, u_m/y_m\}$ be substitutions where the y_i's need not be distinct from the x_j's. From θ and λ, we define

$E_1 = P(x_1, \ldots, x_n, y_1, \ldots, y_m)$ and $E_2 = P(t_1, \ldots, t_n, u_1, \ldots, u_m)$.

Then θ and λ are said to be *consistent* if and only if E_1 and E_2 are unifiable. A MGU for E_1 and E_2 is called a *combination* of θ and λ.

The Unification Algorithm

Unify (E_1, E_2)

Step 1. Set $k=0$, $T_k = E_1$, $T_k' = E_2$, $\sigma_k = \phi$

Step 2. If $T_k = T_k'$ then stop; σ_k is a MGU of E_1 and E_2.
Otherwise, let D_k = the set containing the leftmost (or rightmost) subexpressions from T_k and T_k' which do not agree. Go to Step 3.

Step 3. If D_k contains a term t_k which is not a variable and a variable v_k which does not occur in t_k, go to Step 4.
If D_k contains two variables, let t_k = the variable from T_k and v_k = the variable from T_k', and go to Step 4.
Otherwise, E_1 and E_2 are not unifiable; stop.

Step 4. Set $\sigma_{k+1} = \sigma_k(t_k/v_k)$, $T_{k+1} = T_k(t_k/v_k)$, $T_{k+1}' = T_k'(t_k/v_k)$, and $k = k+1$.
Go to Step 2.

Lemma 1 If σ is a MGU produced by the Unification Algorithm, then $\sigma\sigma = \sigma$.

Lemma 2 Let $\sigma = \{g_i/v_i \mid i=1, \cdots, n\}$ be a substitution, and suppose that no v_i occurs in any g_j. Then σ is the MGU of $P(v_1, \cdots, v_n)$ and $P(g_1, \cdots, g_n)$ produced by the Unification Algorithm.

Lemma 3 If σ is the output of the Unification Algorithm applied to terms A and B, then no variable or term part of σ contains any variable not occurring in either A or B.

Lemma 4 Let $\sigma = \{g_i/v_i \mid i=1, \ldots, n\}$ be the output of the Unification Algorithm. Then $g_i\sigma = g_i$.

We now begin to show the main result, namely, that a MGU of c and d of Figure 1 can be computed directly from σ and θ by

1. forming the *combination* of σ and θ, and

2. deleting from it the components over the x variables.

Recall the definitions of σ, θ, and τ :

$$\sigma = \{s_i / x_i \mid i = 1, \ldots, n^*\}$$
$$\theta = \{t_i / x_i \mid i = 1, \ldots, n\}$$
$$\tau = \{u_i / y_i \mid i = 1, \ldots, m^*\}$$

Let us assume σ and τ are computed by the Unification Algorithm.

Theorem 1 $\sigma\sigma = \sigma$, $\theta\theta = \theta$, and $\tau\tau = \tau$.

Proof. The result for σ and τ follow directly from Lemma 1, since these are MGUs. θ maps a and b onto c and d which have no x variables, so no t_i can contain any x_j. By Lemma 2, θ is also an MGU produced by the Unification Algorithm. QED

Lemma 5 $\theta\tau = \sigma\lambda$ for some λ.

Proof. $a(\theta\tau) = (a\theta)\tau = c\tau = d\tau = (b\theta)\tau = b(\theta\tau)$.

Therefore, $\theta\tau$ unifies a and b. σ is the MGU. QED

Lemma 6 Let $\gamma^* = \theta\tau = \sigma\lambda$. Then $\gamma^* = \theta^* \div \tau$,
where $\theta^* = \{(t_i \tau)/x_i \mid i = 1, \ldots, n\}$ and '$\div$' indicates set union.

Proof. $\gamma^* = \theta\tau$. Note that no $t_i\tau$ can be the same as x_i because t_i and τ come from c and d which contain no x variables. So, no $(t_i\tau)/x_i$ will be deleted in the computation of the composition of θ and τ. Further, the variable parts of τ, $y_1, \ldots, y_{m^*}$, and the variable parts of θ, $x_1, \ldots, x_n$ are distinct, which allows every component of τ to be added to the resulting composition. Therefore, there are exactly $n \div m^*$ components in γ^*; θ^* substitutes for the x variables and τ for the y variables. QED.

Theorem 2 γ^* unifies

$$P(x_1, \ldots, x_{n^*}, x_1, \ldots, x_n) \text{ and } P(s_1, \ldots, s_{n^*}, t_1, \ldots, t_n).$$

Proof. For $i \leq n^*$, we have

$$x_i\gamma^* = x_i(\sigma\lambda) = (x_i\sigma)\lambda = s_i\lambda \text{ and}$$
$$s_i\gamma^* = s_i(\sigma\lambda) = (s_i\sigma)\lambda = s_i\lambda \text{ and}$$
$$t_i\gamma^* = t_i(\theta\tau) = (t_i\theta)\tau = t_i\tau$$

$= (s_{i,1}^{'}\theta)\tau$ since τ unifies c and d, it unifies corresponding terms in c and d

$= s_{i,1}^{'}(\theta\tau) \quad = s_{i,1}^{'}(\sigma\lambda) \quad = (s_{i,1}^{'}\sigma)\lambda$

$= s_i\lambda$ since σ unifies corresponding terms in a and b.

For $i > n^*$ $\quad t_i\gamma^* = t_i(\theta\tau) = (t_i\theta)\tau = t_i\tau,$

$\quad x_i\gamma^* = x_i(\theta\tau) = (x_i\theta)\tau = t_i\tau.$

Corollary. The *combination* of σ and θ exists. Call it γ. Then $\gamma^* = \gamma\lambda^*$ for some λ^*.

Proof. Since the two terms in the above theorem are unifiable, the MGU is γ, and γ^* is a unifier. QED

Lemma 7 $\gamma = \sigma\lambda_1 = \theta\lambda_2$ for some λ_1 and λ_2.

Proof. γ is computed by the Unification Algorithm from

$$P(x_1, \ldots, x_{n^*}, x_1, \ldots, x_n) \text{ and } P(s_1, \ldots, s_{n^*}, t_1, \ldots, t_n).$$

If we proceed left to right, after n^* steps, σ_{n^*} of the Unification Algorithm will just be σ. The remaining n steps unify

$$P(s_1, \ldots, s_{n^*}, s_1, \ldots, s_{n^*}, x_{n^*+1}, \ldots, x_n) \text{ and}$$

$$P(s_1, \ldots, s_{n^*}, t_1, \ldots, t_{n^*}, t_{n^*+1}, \ldots, t_n).$$

Note, no t_j contains an x_i, so the t_j terms are unchanged so far. Thus, the last steps of unification just compose more components to $\sigma_{n^*}(=\sigma)$. Similarly, if we proceed right to left, after n steps, we have θ, etc. QED

Thus, under the assumption that c and d are also unifiable, we proved that the combination γ of σ and θ exists and that it is more general than γ^*. We now show that γ unifies c and d, and a subset $\tilde{\tau}$ of γ is a MGU.

Theorem 3 $\gamma = \tau\hat{\lambda}$ for some $\hat{\lambda}$.

Proof. Consider $c\gamma$ and $d\gamma$. Let the position vector $r_1.r_2...r_k$ denote a maximal (i.e., highest) position of c and d that are different (the position vector $r_1.r_2...r_k$ is a vector

of integers and indicates the r_k th argument of the r_{k-1}th argument $\cdots$ of the r_1th argument of c or d). One of c and d contains a variable in that position, since they are unifiable. Without loss of generality, we may assume that it is c. Thus c contains y_l, for some $l \leq m^*$, in position $r_1.r_2...r_k$. Since c is an instance of a, some superposition $r_1...r_{k^*}$, $k^* \leq k$ of a contains a variable, say x_i. If the position of b corresponding to $r_1...r_{k^*}$ does not exist, then b also has a variable x_j at a higher position, say $r_1...r_{k'}$ with $k' < k^*$. Then the term in position $r_1...r_{k'}$ of a is an s_j'. If position $r_1...r_{k^*}$ of b has a non-variable term g, then g is an s_i'. If a and b both have variables in position $r_1...r_{k^*}$, then one of them is an s' for the other. *In any case,* a and b differ in some superposition of $r_1...r_{k^*}$ with a corresponding x_i and $s_{i,l}'$. Let that superposition be $r_1...r_p$, $p \leq k$. Then the chosen difference position of c and d occurs as a subposition of $r_1...r_p$, and the differing terms of c and d occur as subterms of $t_i (=x_i\theta)$ and $s_{i,l}'\theta$. Note that i of x_i must satisfy $i \leq n^*$ because a and b *must* disagree at that point (if a and b were identical at that point, they would have the same variable, and c and d would not have disagreed at that position). Then we have $t_i\gamma = s_i\gamma = x_i\gamma$ by the definition of γ and the fact that $i \leq n^*$. Further,

$$
\begin{aligned}
(s_{i,l}'\theta)\gamma &= (s_{i,l}'\theta)\theta\lambda_2 && && \text{by Lemma 7}\\
&= s_{i,l}'(\theta\theta\lambda_2) && = s_{i,l}'(\theta\lambda_2) && = s_{i,l}'\gamma\\
&= s_{i,l}'(\sigma\lambda_1) && = s_{i,l}'(\sigma\sigma\lambda_1) && = (s_{i,l}'\sigma)\sigma\lambda_1\\
&= s_i(\sigma\lambda_1) && && \text{because } \sigma \text{ unifies } a \text{ and } b\\
&= s_i\gamma. &&
\end{aligned}
$$

The difference position in c and d are unified by γ, and thus, γ unifies c and d. Since τ is a MGU of c and d, the result follows. QED

Note that γ contains only variables over $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$. However, it could contain substitutions for variables y_j with $j > m^*$. For example, τ might contain a component y_m/y_1 while γ has y_1/y_m. In any case, γ consists of components whose variable parts are among $\{x_i\}$ and components whose variable parts are among $\{y_i\}$. Let us write γ as

$$\gamma = \hat{\theta}\dot{+}\hat{\tau}$$

where $\hat{\theta}$ substitutes for x's and $\hat{\tau}$ for y's. Further, let

$$\hat{\tau} = \{ v_i / z_i \mid i = 1, \ldots, \hat{m} \}$$

where each z_i is a y_j. As above, a z_i may be a y_j with $j > m^*$. Also, $\hat{m}$ need not be the same as m^*.

Theorem 4 No x_j occurs in a v_i.

Proof. In computing γ, we may proceed right to left. In the process, even if any t_k is a single variable y_l, the Unification Algorithm presented earlier guarantees that x_k gets substituted by $y_l (= t_k)$. After n iterations of the algorithm, we get σ_n equal to θ. Further, the formulas remaining to unify are

$$P(t_1, \ldots, t_{n^*}, t_1, \ldots, t_n) \text{ and } P(s_1\theta, \ldots, s_{n^*}\theta, t_1, \ldots, t_n).$$

In these formulas, no x_j occurs. Thus, all the components leading to $\hat{\theta}$ have already been introduced and all the components leading to $\hat{\tau}$ are produced in the remaining steps. In these steps, no variable x_j can participate either as the variable of a substitution or a part of the term replacing it. QED

Theorem 5 $\hat{\tau} = \tau\delta$ for some δ.

Proof.

$c\hat{\tau}$	$= c(\hat{\theta} \div \hat{\tau})$		since no x_j occur in c
	$= c\gamma$	$= d\gamma$	by Theorem 3
	$= d(\hat{\theta} \div \hat{\tau})$	$= d\hat{\tau}$	since no x_j occur in d.

Thus, $\hat{\tau}$ unifies c and d, and τ is a MGU. QED

Theorem 6 $\hat{\tau}$ is a MGU of $P(z_1, \ldots, z_{\hat{m}})$ and $P(v_1, \ldots, v_{\hat{m}})$.

Proof. γ satisfies that none of its variable parts occur in any of its term parts, and $\hat{\tau}$ is a subset of γ. The result follows from Lemma 2. QED

Theorem 7 $\tau = \hat{\tau}\delta'$ for some δ'.

Proof

$z_i\tau$	$= z_i(\theta^* \div \tau)$		since θ^* only affects x's
	$= z_i\gamma^*$	$= z_i\gamma\lambda^*$	$= z_i(\hat{\theta} \div \hat{\tau})\lambda^*$
	$= z_i\hat{\tau}\lambda^*$		since $\hat{\theta}$ only affects x's
	$= v_i\lambda^*$.		
$v_i\tau$	$= v_i(\theta^* \div \tau)$		since no x occurs in v_i

$$= v_i \gamma^* \qquad = v_i \gamma \lambda^* \qquad = v_i (\hat{\theta} + \hat{\tau}) \lambda^*$$
$$= v_i \hat{\tau} \lambda^* \qquad \text{since no } x \text{ occurs in } v_i$$
$$= v_i \lambda^* .$$

Thus, τ unifies $P(z_1, \ldots, z_{\hat{m}})$ and $P(v_1, \ldots, v_{\hat{m}})$, and $\hat{\tau}$ is an MGU. QED

Theorem 8 $\hat{\tau}$ is a MGU of c and d.

Proof. We have shown that $\hat{\tau}$ unifies c and d, and $\tau = \hat{\tau}\delta'$ for some δ'. τ is a MGU of c and d, thus $\hat{\tau}$ is also. QED

Thus, we may compute unifiers of pairs of terms in a unification join or select in a top down fashion. This has several advantages over the bottom up method of the next section. First, the top down method uses only standard unification. As will be seen below, the bottom up method requires a second operation to be performed on the lower unifier, requiring considerably more hardware/software. A second advantage is that more parallelism can be used. The system may start computing unifiers as soon as the initial parts of the sublattice of terms for the two joining relations are formed. (These sublattices will surely be formed top down.) In order to take fullest advantage of the technique, the bottom up method needs to unify the lowest possible terms first, and thus has to wait until the full sublattices for the two lists of terms are formed. Finally, the top down method unifies terms at the top first, and then computes unifiers for lower pairs. These top most pairs will also be the simplest terms. The bottom up method must unify very complicated terms first.

The top down method has the advantage over simply applying unification directly to all pairs because the top down method need never actually generate the terms of the lower pairs. It only uses the top level unifiers and the θ substitutions from the lattice. The terms unified in the composition operation can be expected to be much simpler than the actual terms in the relations. For example, the actual terms may have several occurrences of a variable x for which a complicated substitution is made for the next lower term. Computing with the terms themselves would require the common instance term to be examined several times while the top down method only looks at it once in computing the combination.

We envision a system somewhat like the following. When the join command is issued, two extraction engines work in parallel to construct the two sublattices of terms. As soon as there are terms in these two lattices, a unification engine (or engines) begins working in parallel to find top level unifiers. When these are found, a third set of engines works in parallel to compute combinations for the lower pairs. Of course, the unification engines are all the same and can be shared between the top level unification tests and the lower ones. At any point where a unification test fails (either top level or combination), no pairs of terms below the failing pair are tested. This eliminates the need to sort terms and changes the pair generation engine from what is described in [MORI86].

4. Computing Unifiers for More General Terms (Bottom-up)

While we believe the top down method is the best, for the sake of completeness, we show in this section how to calculate MGUs for more general terms from unifiers of their instances. In particular, we will present an algorithm for terms t_1 and t_3 given a MGU σ of t_1 and t_2 and a substitution θ such that $t_3\theta = t_2$ (Figure 2). To this end, we describe a modified representation for unifiers, after which we describe the algorithm itself.

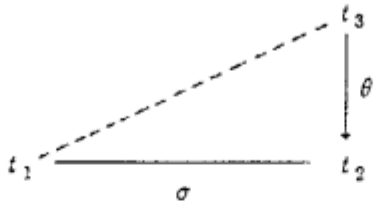


Figure 2

4.1 Representation of Unifiers

We use a modified representation of unifiers in which a substitution is treated as a set of term blocks, each block containing all the terms that must be alike. In addition to that, each term in a block must have a list of its occurrences in t_1 or t_2; because we assume the variables are separated in the original terms, it is not necessary, except for

ground terms, to indicate from which of t_1 or t_2 the occurrence arises. Occurrences are represented by position vectors used in the previous section. The use of the position vectors will be explained in the next section; they are not necessary to simply represent the unifier, but will be used to compute the unifier for the more general terms. Notice that each block contains a list of terms exactly as they occur in t_1 and t_2; no substitutions are *applied*, although the actual unifying substitution can easily be constructed from the set of blocks. We present some examples to illustrate the representation.

Example 1 Let $t_1 =$ $f\,(i(g\,(h\,(b\,,x\,),y\,)),g\,(w\,,z\,),u\,,v\,,s\,,t\,)$
and $t_2 =$ $f\,(i(g\,(\quad n\quad ,\quad e\,)),\ g\,(n\,,e\,),n\,,n\,,e\,,e\,)$.

Here, b and e are constants, and x, y, z, u, v, w, s, t, n are variables. The MGU is represented by the two blocks

$[n\ \{1.1.1,\ 2.1,\ 3,\ 4\},\ h\,(b\,,x\,)\{1.1.1\},\ w\ \{2.1\},\ u\ \{3\},\ v\ \{4\}]$, and
$[y\ \{1.1.2\},\ z\ \{2.2\},\ e\ \{1.1.2,\ 2.2,\ 5,\ 6\},\ s\ \{5\},\ t\ \{6\}]$.

The first block indicates that the variables n, w, u and v must all be replaced by the term $h\,(b\,,x\,)$. □

Example 2 Let $t_1 = f\,(x\,,g\,(x\,,y\,),g\,(x\,,b\,))$ and $t_2 = f\,(a\,,u\,,u\,)$. The unifier is represented by the blocks

$[x\ \{1,2.2,3.1\},\ a\ \{1\}]$
$[y\ \{2.2\},\ b\ \{3.2\}]$, and
$[u\ \{2,3\},\ g\,(x\,,y\,)\{2\},\ g\,(x\,,b\,)\{3\}]$.

Notice that the terms $g\,(x\,,y\,)$ and $g\,(x\,,b\,)$ occur in the last block as opposed to their instance $g\,(a\,,b\,)$. The meaning of this block is that the terms u, $g\,(x\,,y\,)$, and $g\,(x\,,b\,)$ must be alike for t_1 and t_2 to be unified. The first two blocks contain the substitutions required for $g\,(x\,,y\,)$ and $g\,(x\,,b\,)$ to be made alike. Thus, the first two blocks are subordinate to the last one. If the substitutions in the lowest level blocks are applied to the higher blocks, the actual unifying substitution can be obtained. In the present case, applying the first two blocks to the last one yields $[u\,,\ g\,(a\,,b\,),\ g\,(a\,,b\,)]$ which is the proper substitution for u. Of course, the actual substitution for u would be written $g\,(a\,,b\,)/u$. □

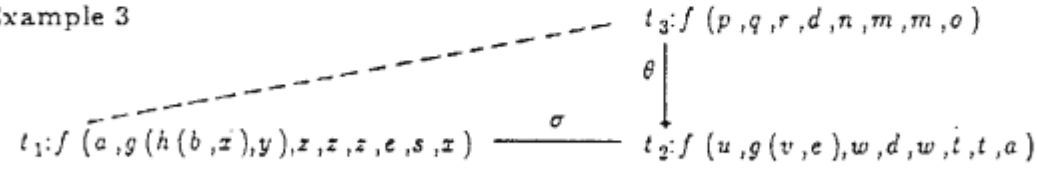
It should be clear that the normal unification process can easily be modified to

produce a unifier in the above representation.

4.2 Constructing the MGU

We start this section with some motivating examples. We assume at the outset that the variables in t_1, t_2, and t_3 are all separated. Further, we assume that θ includes the position vectors for the variables of t_3. Note that the separation of variables in t_2 from variables of t_3 will require a substitution component for every variable of t_3 in θ.

Example 3



σ is $[u\ \{1\},\ a\ \{t_1{:}1\}]$
$[v\ \{2.1\},\ h\ (b\ ,x\ )\{2.1\}]$
$[y\ \{2.2\},\ e\ \{t_2{:}2.2\}]$
$[z\ \{3,4,5\},\ w\ \{3,5\},\ d\ \{t_2{:}4\}]$
$[s\ \{7\},\ t\ \{6,7\},\ e\ \{t_1{:}6\}]$
$[x\ \{8\},\ a\ \{t_2{:}8\}]$.

θ is $(p\ \{1\}/u\ ,\ q\ \{2\}/g\ (v\ ,e\ ),\ r\ \{3\}/w\ ,\ n\ \{5\}/w\ ,\ m\ \{6,7\}/t\ ,\ o\ \{8\}/a\ )$.

Note that we are using a reverse notation for substitutions for θ. Consider each block of σ in turn. The first block references u which is in a position affected by θ (position 1). Thus, the term in position 1 of t_2 arises from the corresponding term in position 1 of t_3. Obviously, for t_1 and t_3 to unify, the terms in position 1 of t_1 and t_3 must be in the same block. Therefore, we replace u in block 1 by p. Moving on to the second block, the one containing $h\ (b\ ,x\ )$, we see that it occurs as a *subposition* of a substitution component of θ. That is, in t_3 the variable q occurs in a position which includes $h\ (b\ ,x\ )$. In fact, the same is true for the block containing y. Since q occurs in position 2 of t_3, we must form a new block containing q and the corresponding term of t_1, namely, $g\ (h\ (b\ ,x\ ),y\ )$. The position vector attached to q in θ tells us immediately which t_1 term to use. The new block is

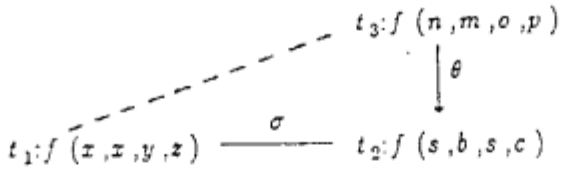
$$[q\ \{2\},\ g\ (h\ (b\ ,x\ ),y\ )\{2\}]$$

and in this case we may delete the two blocks corresponding to positions 2.1 and 2.2. For the block containing z, w, and d, we see that w occurs in two positions for which there are substitutions in θ. Thus, $w\{3\}$ in σ must be replaced by $r\{3\}$. Similarly, $w\{5\}$ is replaced by $n\{5\}$. Because m occurs in positions 3,4 and 5, z, r, n and d will again occur in the same block in the unifier of t_1 and t_3. Similarly, the occurrences of t are replaced by occurrences of m from θ. Finally, the occurrence of a in position 8 of t_2 also has a component in θ. Therefore, $a\{8\}$ is replaced by $o\{8\}$. Note that in this case, the non-variable term in σ occurs in exactly the same position as the variable in t_3, in contrast to the situation of the variable q which was a *proper* superterm. The resulting set of blocks is

$[p\{1\}, a\{1\}]$
$[q\{2\}, g(h(b,x),y)\{2\}]$
$[z\{3,4,5\}, r\{3\}, n\{5\}, d\{4\}]$
$[s\{7\}, m\{6,7\}, e\{t_1{:}6\}]$
$[x\{8\}, o\{8\}]$.

Note again that the although $g(h(b,x),y)$ occurs in the second block, the last block indicates that x and o are identified. □

Example 4



σ is $[z\{1,2\}, y\{3\}, s\{1,3\}, b\{2\}]\,[x\{4\}, c\{4\}]$

θ is $(n\{1\}/s, m\{2\}/b, o\{3\}/s, p\{4\}/c)$.

As before, we begin by replacing occurrences of terms in σ coming from t_2 by corresponding terms from t_3 as determined by θ. In this case, there are no proper superterm replacements, so the replacement is fairly simple. Note however, that the replacement is based on position. Therefore, $s\{1,3\}$ is replaced by two separate items, $n\{1\}$ and $o\{3\}$. The resulting blocks are

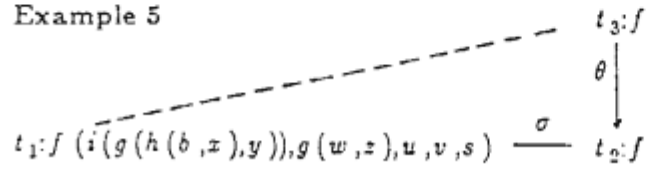
$[x\{1,2\}, y\{3\}, n\{1\}, o\{3\}, m\{2\}]\;\;[x\{4\}, p\{4\}]$.

However, these two blocks do not represent a MGU of t_1 and t_3 because too much substitution is indicated. The problem is that positions in t_2 that were the same are not

the same in t_3, but rather distinct variables. Thus, in unifying t_1 and t_3, positions 1 and 2 must be the same because of x, but position 3 is now independent. Thus, the first block above needs to be split. This can be accomplished easily by a kind of *orbit analysis*. Consider the variable x. It occurs in position 1 and 2 of its item. Any other items in that block with positions 1 or 2 must be kept with x. Thus, m and n must stay in the same block as x. Since m and n do not occur in any positions other than 1 and 2 which are already in the orbit, one block in the unifier of t_1 and t_3 is $[x\ \{1,2\},\ n\ \{1\},\ m\ \{2\}]$. The remaining items of the above block occur only in position 3 and form a second block, $[y\ \{3\},\ o\ \{3\}]$. The unifier of t_1 and t_3 is therefore given by

$$[x\ \{1,2\},\ n\ \{1\},\ m\ \{2\}]\quad[y\ \{3\},\ o\ \{3\}]\quad[z\ \{4\},\ p\ \{4\}].\quad \square$$

Example 5



σ is $[n\ \{1.1.1,2.1,3,4\},\ h\ (b\ ,x\ )\{1.1.1\},\ w\ \{2.1\},\ u\ \{3\},\ v\ \{4\}]$
$[y\ \{1.1.2\},\ e\ \{t_2{:}1.1.2,2.2,5\},\ s\ \{5\},\ z\ \{2.2\}]$

θ is $(q\ \{1.1.2\}/g\ (n\ ,e\ ),\ o\ \{3\}/n\ ,\ p\ \{4\}/n\ ).$

The first two occurrences of n, 1.1.1 and 2.1, are deleted because θ contains a component replacing a proper superterm of these positions in t_3. This also requires a new block with q and the corresponding two terms from t_1. Similarly, the first two position vectors for e are deleted. The remaining replacements are straightforward. The result is

$[q\ \{1.1.2\},\ g\ (h\ (b\ ,x\ ),y\ )\{1.1\},\ g\ (w,z)\{2\}]$ the new block
$[o\ \{3\},\ p\ \{4\},\ h\ (b\ ,x\ )\{1.1.1\},\ w\ \{2.1\},\ u\ \{3\},\ v\ \{4\}]$
$[y\ \{1.1.2\},\ s\ \{5\},\ e\ \{t_2{:}5\},\ z\ \{2.2\}].$

Simple orbiting produces

$[q\ \{1.1.2\},\ g\ (h\ (b\ ,x\ ),y\ )\{1.1\},\ g\ (w\ ,z\ )\{2\}]$
$[o\ \{3\},\ u\ \{3\}]\quad[p\ \{4\},\ v\ \{4\}]$
$[h\ (b\ ,x\ )\{1.1.1\}]\qquad[w\ \{2.1\}]\qquad[y\ \{1.1.2\}]$
$[z\ \{2.2\}]\qquad[s\ \{5\},\ e\ \{5\}]$

with four degenerate blocks. We cannot simply delete these four blocks since $h\ (b\ ,x\ )$

and w as well as y and z must match. Positions 1.1.1 and 2.1 are in fact in the same orbit because there is another block in which proper superpositions are in the same orbit, namely the first block above requires positions 1.1 and 2 to be in the same orbit. Therefore, all corresponding pairs of subpositions of 1.1 and 2 must be in the same orbits. Correct orbiting, then, will produce

$$[h(b,x)\{1.1.1\},\ w\{2.1\}] \qquad [y\{1.1.1\},\ z\{2.2\}]. \quad \square$$

The algorithm can now be stated. Its statement is considerably simpler than the examples that led us to it.

Algorithm for Computing MGU for More General Terms

1. Use θ to replace items in σ and possibly create new blocks.
2. Perform orbit analysis : two positions $a_1.a_2 \cdots a_i$ and $b_1.b_2 \cdots b_j$ are in the same orbit if
   i. they occur in the same position list ({} bracket) of a term, or
   ii. $a_1 \cdots a_{i'}$ and $b_1 \cdots b_{j'}$ is in the same orbit and $a_{i'+1} \cdots a_i = b_{j'+1} \cdots b_j$.
3. Delete any degenerate blocks.

5. Storage Schemes for Terms

Our suggestions for storage terms arise from the following two remarks. First, each term in the knowledge base has at least one occurrence in the generality lattice. If the maximal terms in the lattice are stored explicitly, then any term, t, appearing at a lower level can be reconstructed by iteratively applying the substitutions leading from any maximal superior term down to t. This might be required in the bottom up strategy or when printing a result, for example. Similarly, the initial non-variable strings used in the total orders can be constructed from the lattice if those orderings are to be used (see Section 6). Thus storing both the lattice and explicit representations of terms is redundant.

Moreover, as the terms become more complex, the simple term representations, like character string or tree representations, repeat large numbers of common structures. For example, if the term $t_1 = f(g(h(b,x),y))$ occurs and if there are n instances of t_1 through n different substitutions for x, the initial structure $f(g(h(b$ will be repeated n times. Similarly, if the term $k(x,x,x,x)$ has an instance obtained by substituting

the above t_1 for x, the entire term may be repeated in the storage of the knowledge base. On the other hand, the lattice representation will not repeat any of these structures, but rather store them only once. In the case of n different instances of t_1, each one will be represented by a link in the lattice labeled with only the substitution for x; the structure of t_1 will not be repeated. Similarly, in the case of the instance of $k(x,x,x,x)$, there will again be a labeled link on which t_1 will occur once as the replacement term for x. Thus, storage requirements may actually be reduced by storing terms only indirectly through the lattice.

Of course, such a scheme requires more computation to retrieve a term, which leads to the second remark. Typically, the only accesses to terms are for printing answers and performing unification joins and selections. The whole purpose of the lattice is to help speed up unification join and selection, so there should be a net gain in computational speed for those operations. As for the output of terms as answers, it is clearly less efficient when the terms must be reconstructed than when the terms already exist explicitly. However, I/O is a much slower process anyway. Further, the lattice storage method lends itself to a very fast, stream-oriented scheme for generating the lower level terms. One main objection still remains, namely, if the term to be output occurs many levels deep in the lattice, the system has to generate all the terms in between the top and the desired term. This could be avoided by storing the transitive closure of the lattice, but of course at a much larger cost for storage and update computation. This point remains as a difficulty for the lattice storage method.

It should be noted that various other schemes exist for optimizing the storage terms. For example, a system may provide for only one occurrence of each term to be stored independent of the number of times that term occurs in formulas of the knowledge base (for example, [LUSK82] or [BOYE72]). These schemes could be used to optimize for both the original proposal of [MORI86] as well as the present proposal. The lattice representation is also well suited for storage as a relational database using a three-placed relation scheme

$$L(tag_1, tag_2, subst)$$

where tag_1 is the identifier of a term, say t, tag_2 is the identifier of a term s which lies immediately below t in the lattice, and *subst* is a pointer to a representation for the

substitution that maps t to s. Then traversing up or down the lattice is equivalent to selecting on the second or the first attribute, etc.

6. Term Indexing Schemes

Before discussing *outlines* as an alternative to the indexing schemes given in [MORI86], we point out yet another potential tradeoff between storage and response time. This tradeoff involves the use of the indexing scheme (any of *left-most, outer-most,* or *outlines*) as a hash function for accessing terms. In the *outer-most* scheme in [MORI86], the hashing function is just $prefv(t)$; that is, all terms with the same initial string of non-variable symbols are stored together in the same list. Just as in outlines, the set of prefixes can be compared when they are created once and for all to determine which pairs of prefixes are compatible for potential unification. For example, any term with prefix $f(g(a),b$ is potentially unifiable with any term whose prefix is $f($. We propose storing the prefixes in a network which links together compatible prefixes; prefix $f(g(a),b$ would be linked to $f($, $f(g$, etc. In fact, these prefixes also display a partial order relationship ($p_1 < p_2$ if p_1 is an initial substring of p_2). This structure is very easy to maintain as the knowledge base is updated. Further, it eliminates the need for the sort units in the unification engine.

Pair generation can be accomplished in the following way. We are given two lists of terms, L_1 and L_2, and are to find all pairs, t_1 and t_2, that are unifiable. Recall, all terms are stored in a single lattice. Therefore, in order to distinguish which list a term belongs to, we mark each term of L_i in the prefix network with the symbol m_i, for $i=1,2$. Then, starting with one prefix, say p_1, we form all pairs (t_1,t_2) such that t_1 is in p_1 and has the m_1 mark and t_2 is in a prefix linked to p_1 and has the mark m_2. Repeat this for each prefix. Of course, at the end the m_i marks must be removed. The advantage here is that pair generation requires no comparison of prefixes. Prefixes are compared only when the network is built and maintained. This speeds the computation of a unification join or select and simplifies the structure of the unification engine at the expense of storing the prefix network and some extra computation for maintaining it as the knowledge base is updated with new terms. An interesting question is whether or not the prefix network and the generality lattice could be incorporated into a single

structure in a way that would allow pair generation to automatically produce pairs lowest in the generality network first. Then, processes 1 and 3 in section 2 could be incorporated into the pair generation unit.

As for the actual unification filter, the total orderings proposed in [MORI86] are a simple and effective mechanisms, but not as effective as some other known schemes. The difficulty is that the filtering comparison stops at the first non-variable position in either of the terms. One scheme that avoids this problem is *outlines* [HENS83]. For the present, we will not give any background on *outlines* here, but only cite the advantages and disadvantages of it and possible fix for the major disadvantage. The disadvantage is that the *outline* structure requires a commitment to a maximum number of argument positions for any symbol in the language and assumes that every symbol (including variables and constants) has that many arguments. The filtering mechanism does not take into account any arguments of terms beyond the maximum. On the other hand, the larger the maximum number of arguments, the more space is required, and the length of an outline grows exponentially with the level of nesting of terms. A key point however is that the outline for a deeply nested term tends to have mostly blank space; further, the location and length of the blank space can be easily computed as the outline is being formed. Thus, while the storage required for outlines will in general be more than that required for prefixes, the efficiency in eliminating pairs to be sent to the unification unit will generally be more efficient in filtering out pairs of terms. Test cases have shown that outlines are usually 80–95% accurate in eliminating non-unifiable pairs. This is of particular importance in unification joins and selects where the major computational effort is in unifying pairs of terms. In addition, outlines can be used to filter pairs for generality, that is to determine if t_1 could be more general than t_2 (The outer-most and left-most schemes can also serve this purpose). As above, an interesting question is whether or not outlines could be incorporated into the generality lattice.

7. Conclusions

We have presented a number of alternatives that may improve the response time of unification joins and selects. The implementation of these techniques and the degree of improvement requires further study.

References

[MURA85] Murakami, M., Yokota, M., Itoh, H., "Formal Semantics of a Relational Knowledge Base," *ICOT Technical Report TR-149*, Dec. 1985.

[MORI86] Morita, Y., Yokota, H., Nishida, K., Itoh, M., "Retrieval- by-Unification Operation on a Relational Knowledge Base," *Proceedings of the 12th Int'l Conf. on VLDB*, Kyoto, Aug. 1986.

[ROBI65] Robinson, J.A., "A Machine-oriented Logic Based on the Resolution Principle," *JACM*, Vol 12,1 (Jan 1965).

[HAYN73] Haynes, G.A., "Completeness of Variable-Constrained Resolution," *MS thesis, Dept. of EECS, Northwestern Univ.*, Evanston, Ill., Aug. 1973.

[CHAN73] Chang, C.L., Lee, C.T., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, NY, 1973.

[HENS83] Henschen, L.J., Naqvi S., "An Improved Filter for Literal Indexing in Resolution Systems," *Proceedings of IJCAI*, Vancouver, BC, 1983.

[LUSK82] Lusk, E., Overbeek, R., McCune, W, "Logic Machine Architecture ; Kernal Functions," *Proceedings of CADES*, 1982.

[BOYE72] Boyer, R., Moore, J., "The Sharing of Structure in Theorem Proving Programs," *Machine Intelligence*, Vol. 7, Edinborough Univ. Press, 1972.