UNIFICATION AND PREDICATE LOCKING

Traci Ann Wheeler

March 1988
CSD-880023

UNIVERSITY OF CALIFORNIA

Los Angeles

Unification and Predicate Locking

A thesis submitted in partial satisfaction of the
requirement for the degree Master of Science
in Computer Science

by

Traci Ann Wheeler

1986

TABLE OF CONTENTS

1 Introduction 1

2 Unification 2

2.1 Terminology and Definitions 2

2.2 Basic Properties of Unifiers 5

2.3 Unification Algorithms 6

2.3.4 Robinson's Algorithm 7

2.3.5 Efficient Algorithms 10

2.3.5.1 Paterson and Wegman 11

2.3.5.2 Martelli and Montanari 16

2.3.5.3 Complexity of the Algorithms 19

3 Predicate Locking 21

3.1 Terminology 21

3.2 Consistency vs. Concurrency 22

3.3 Locking 23

3.3.1 Granularity of Locks 24

3.3.2 Phantom tuples 24

3.4 Predicate Locks 24

4 Unification and Predicate Locking 27

4.1 Limitations of Terms as Predicate Locks 28

5 Extended Unification 30

5.1 Extended Terms 30

5.2 Projections 33

5.3 Extended Unification 37

6 Extended Unification and Predicate Locking 44

7 Summary 48

Appendix I: Generic algorithm for extended unification 49

Appendix II: Constraints for Predicate Locking 53

References 57

ABSTRACT OF THE THESIS

Unification and Predicate Locking

by

Traci Ann Wheeler

Master of Science in Computer Science

University of California, Los Angeles, 1986

Professor S. Greibach, Chair

Predicate locking has been suggested as a way of ensuring consistency in shared databases while providing maximal concurrency. This paper explores the similarities between the central problems in predicate locking and the power of the pattern matching technique, unification, and suggests extensions to unification that allow a complete modeling of useful predicate locks.

1. Introduction

Locking is used to ensure consistency in shared database systems by requiring that users of a database indicate which data items they intend to modify before performing any modifications, that is, users *lock* every data item they intend to use. A central lock manager, in turn, prevents any other user from using these locked data items until the original user is finished; thus ensuring that users have exclusive access to any data they intend to modify. Eswaran et al. [Esw] have suggested that ensuring consistency requires that locks represent a logical (i.e., possible) subset of data items rather than a physical (i.e., existing) subset; they suggest the use of predicates on tuples to specify these locks. The use of predicate locks presents the problem of "matching" two arbitrary predicates, that is, determining if two predicates are satisfiable by the same tuple.

Unification provides powerful pattern matching. It is used in a variety of different applications, notably resolution theorem proving [Bun], interpreters for equation languages [Bur], type checkers [Mil] and computation of critical pairs in term rewriting systems [Knu] and the execution of the logic programming language, Prolog [War]. A great deal of literature has been devoted to efficient implementations of unification notably [Bax, Mar, Pat]. Linear time algorithms have been presented, and the complexity of the problem is well understood.

This paper demonstrates the similarities between unification and the matching of predicates. In the second chapter, the background of unification is discussed and a number of unification algorithms are presented. The third chapter presents the background and issues of predicate locking. Chapter four demonstrates the similarities between predicate locking and unification, and suggests extensions to unification that allow its use in an implementation of predicate locking. Chapter five defines a general framework for extended unification. Chapter six shows how the extended unification developed in chapter five can be used to implement a predicate lock manager. Finally, an appendix outlines algorithms solving unification for a particular class of extensions.

2. Unification

The unification problem can be informally stated as follows: given two "patterns", find a minimal set of substitutions of patterns for variables that will make the two patterns identical. As an example, consider the two patterns f(X,Y) and f(a,b), where the uppercase letters X and Y represent variables, and the lowercase letters f, a and b represent constant values. One set of substitutions that will make these patterns identical is (X→a) and (Y→b) (read "substitute X with a and Y with b"). In fact, this is the only such substitution. A more a formal definition of unification follows, along with a number of interesting unification algorithms.

2.1. Terminology and Definitions

Let F be a finite set of function symbols, where each symbol has a name and specific arity (functions of arity 0 are also known as constants) and let V be a set of variables. A *term* is defined recursively as follows:

(i) a ∈ F with arity 0 is a term (i.e., constants are terms)
(ii) v ∈ V is a term (i.e., variables are terms)
(iii) if t_1, t_2, ... t_n are terms and f ∈ F is a function symbol of arity n, then f(t_1, ..., t_n) is a term.

A *ground term* is defined to be a term having no variables, that is, any term constructible using only function symbols. A *simple substitution* of t for X is specified as (X → t) where t is a term that does not contain the variable X. A *substitution* of terms for variables is specified as the set or sequence

$$\{ (X_1 \rightarrow t_1), ..., (X_n \rightarrow t_n) \},$$

where X_i and X_j are distinct variables for i ≠ j. Such substitution may be considered to be a *simultaneous substitution* or a *sequential substitution*. In a simultaneous substitution, the ordering of the simple substitutions is not significant; whereas, in a sequential substitution, the ordering is significant. Substitutions will always be labeled as simultaneous or sequential in this text, unless their type is clear from context. The *application* of a simple substitution, (X → t), to a term u, written

$$(X \rightarrow t)u \text{ or } (X \rightarrow t)(u),$$

results in the replacement of all occurrences of the variable X in u by the term t. A simultaneous sub-

stitution σ is applied to term u by applying each simple substitution in σ u simultaneously without regard to order, while the sequential substitution σ is applied to term u by applying each simple substitution in σ in order from left to right. Applying the *composition* of two substitutions, ρ and σ, written $\rho\sigma$, to term u means first apply ρ to u and apply σ to the result, i.e.,

$$\rho\sigma u = \sigma(\rho(u)).$$

Thus the sequential substitution

$$(X_1 \to t_1, \ldots, X_n \to t_n)$$

is the composition of the n simple substitutions $X_i \text{->} t_i$ in order from 1 to n. Two substitutions, ρ and σ, are considered to be *equivalent with respect to term t* if $\rho t = \sigma t$. They are considered *equal* if for all terms, t, $\rho t = \sigma t$.

A *unifier* of two terms is any substitution that, when applied to the two terms, produces identical terms. Two terms are said to be *unifiable* if and only if such a unifier exists. For example, f(X,Y,g(X)) and f(a,g(b),g(a)) are unifiable, by the unifier { (X→a), (Y→g(b)) }, while f(X,a) and f(Y,b) are not. A *most general unifier* (mgu) of two terms, t and u, is defined to be any unifier, μ, of t and u, having the property that if λ is also a unifier of t and u, then $\mu\lambda = \lambda$. The unifier μ is considered to be "most general" since every other unifier, λ, can be decomposed into the composition $\mu\lambda$, that is, the result of applying any unifier, λ, to a term is equivalent to first applying the most general unifier, then applying λ. The standard unification problem, as defined by Robinson, is to determine whether two terms are unifiable and, if so, to find an mgu for the terms. The *resultant* of a unifier is the term that is produced when the unifier is applied to either of the terms it unifies.

In standard unification, the two terms X and f(X) are not unifiable since the only substitution that would seem to make them identical is (X → f(X)) which is disallowed by the definition of substitution, since X occurs in f(X). In some application areas it is useful to consider these terms to be unifiable with the substitution (X→f(X)), where the result of applying (X→f(X)) to X is considered to produce the infinite term f(f(f(...))). Certain variations of the unification problem specifically disallow such substitutions while others allow them in order to facilitate the building of infinite terms. The test for this

case is known as the *occurs check* since the resulting substitution, (V→t), would require that the variable, V, "occur" in the substituting term, t. If a unification algorithm includes an occurs check, then the terms X and f(X) are not unifiable, otherwise they are unifiable by the substitution (X→f(X)).

The *difference* of two terms t and u, diff(t,u), (defined in [Rob]) is a measure of "how far" t and u are from being identical. It is defined recursively as follows:

(i) If t and u are identical then diff(t,u) = ∅, the empty set.

(ii) If t and u have identical arity and function symbol, then their difference is the set of differences of their arguments, i.e., if t = f(t_1, ..., t_n) and u = f(u_1, .., u_n) then

diff(t,u) = {diff(t_1, u_1), ..., diff(t_n, u_n)}.

(iii) Otherwise, diff(t,u) = { (t,u) }.

For example,

diff(f(a, g(f(X),b), X), f(X, g(f(a),Y), X))	
= { diff(a,X), diff(g(f(X),b), g(f(a),Y)), diff(X,X) }	by rule (ii)
= { (a,X), diff(g(f(X),b), g(f(a),Y)), diff(X,X) }	by rule (iii)
= { (a,X), diff(g(f(X),b), g(f(a),Y)) }	by rule (i)
= { (a,X), diff(f(X), f(a)), diff(b,Y)}	by rule (ii)
= { (a,X), (X,a), (b,Y)}	by rules (ii) and (iii)

A difference diff(t,u) is comprised of a set of *disagreement pairs* each representing a point of difference or disagreement among the two input terms. Each of these differences must be resolvable in order for the input terms to be unifiable. Some pairs are obviously not resolvable, for example, the pair (a,b) represents a difference that can not be resolved since the constants a and b cannot be made to match. The measure of *negotiability* is intended to identify pairs that are obviously not resolvable. As we will see in a later section, negotiability is a necessary but not sufficient condition for unifiability. A difference, diff(t,u), is considered to be *negotiable* only if each disagreement pair (x,y) within diff(t,u) has the following properties:

(i) at least one of x,y is a variable, and

(ii) neither x nor y is a variable which occurs in the other (thus, verifying negotiability implies verification of the "occurs check" mentioned above). As we will see in a later section, negotiability is a necessary but not sufficient condition for unifiability.

The difference illustrated above, { (a,X), (X,a), (b,Y) } is negotiable since at least one element of each pair, (a,X), (X,a) and (b,Y), is a variable and no element of any pair occurs in its partner. In contrast, the difference { (g(a), f(a)) } is not negotiable since neither element of the pair (g(a), f(a)) is a variable. Furthermore, diff(X,f(X)) = { (X,f(X)) } is not negotiable since X occurs in f(X).

It is often convenient to talk about the *position* held by a term, t, in a larger term, u. The position is defined by the sequence of argument numbers that one would have to follow to find t in u. For example, if u=f(g(t)), then the position held by t in u is "the first argument of the first argument of u". Note that the elements of a disagreement pair must hold the same position in their respective terms.

2.2. Basic Properties of Unifiers

Two important well-known properties of unifiers are the following:

Property 1: If two terms are unifiable, then there exists at least one most general unifier for them. (The first property will be shown in the section on Robinson's algorithm by constructing an mgu for any two unifiable terms.)

Property 2: The second property, which will be proven here, is the uniqueness of mgus up to a renaming of variables. If two terms, t and u, are unifiable with mgu μ, then μ is unique with respect to its resultant up to a renaming of variables; that is, if μ and λ are both mgus of t and u then their resultants are equivalent up to a renaming of variables.

Proof: Assume that the terms t and u have two mgus, μ and λ, whose resultants are distinct (where the distinction is more than a simple renaming of variables). Since μ and λ produce distinct resultants, the difference between their resultants, diff(μt, λt), must be non-empty and contain some disagreement pair (x,y) such that $x \neq y$, and x and y are not both variables; if the difference does not contain such a disagreement pair, then the difference is only a renaming of variables. Furthermore, every disagreement pair in diff(μt, λt) must have been generated by applying the substitutions μ and λ to some

corresponding variable, V, in t, i.e., $x=\mu V$ and $y=\lambda V$.

The terms x and y can take one of four forms:

(i) Both x and y are ground terms.
(ii) one is a ground terms while the other is a non-ground term.
(iii) one is a non-variable (ground or non-ground) term while the other is a variable.
(iv) Both are non-ground terms (but not variables).

We know that

$$
\begin{aligned}
(1)\ x &= \mu V \\
&= \lambda\mu V && \text{since } \lambda \text{ is an mgu} \\
&= \mu y && \text{since } \lambda V = y.
\end{aligned}
$$

And,

$$
\begin{aligned}
(2)\ y &= \lambda V \\
&= \mu\lambda V && \text{since } \mu \text{ is an mgu} \\
&= \lambda x && \text{since } \mu V = x.
\end{aligned}
$$

If x is a ground term, then $\lambda x = x$, and $y = x$ (from (2)); but this contradicts our assumption that x and y are distinct. A similar argument holds if y is a ground term. If x is a variable, then y must not be a variable so $\mu y = y$; but, we know $\mu y = x$ from (1), so again $x = y$; but x is a variable and y is not, so we have a contradiction. A similar argument holds if y is a variable and x is not. Finally, if x and y are distinct non-ground, non-variable terms, then either their initial function symbols differ or their arity differs (from the definition of diff). But $x = \mu y$, and a substitution cannot change the initial function symbol or arity of a term, so again we reach a contradiction. So, our assumption that two terms, t and u, can have two mgus, μ and λ, which produce distinct resultants cannot be true. QED.

2.3. Unification Algorithms

A great deal of literature has been devoted to perfecting unification algorithms. The first algorithm presented here is due to Robinson. It is included because of its clear illustration of the unification process. The subsequent two algorithms improve upon this process by reordering some of the basic steps. Both result in linear behavior. Each of the unification algorithms that follow includes an occurs check, produces an mgu if their input terms are unifiable, and halts with failure otherwise.

2.3.1. Robinson's Algorithm

Robinson's initial algorithm for unification [Rob] builds a sequential substitution by recursively computing the difference between its two input terms, t and u, and trying to resolve the leftmost disagreement pair in the difference.

At each recursion, the difference must be either empty, nonempty and negotiable, or nonempty and nonnegotiable. If the difference is empty then t and u must be identical, by definition, and the algorithm halts with the mgu $\varnothing$. If diff(t,u) is nonempty and nonnegotiable, then, as Robinson shows, t and u are not unifiable and the algorithm halts with failure. If diff(t,u) is nonempty but negotiable then the algorithm creates the substitution (V→w) from the leftmost disagreement pair, $D_1 \in$ diff(t,u), in which V is the element of D_1 which is a variable (if both elements of D_1 are variables, then the choice is irrelevant), and w is the other element. Each such substitution is applied to both of the input terms and added to the sequence of substitutions being created. The entire process is repeated on the resulting terms beginning with the computation of the new difference. The algorithm halts when the new difference is empty or non-negotiable.

A proof of correctness is included after the actual algorithm.

Robinson's Algorithm

```
Unify(X, Y)
Begin
    Mgu ← ∅; Ans ← success
    While diff(X,Y) ≠ ∅ and Ans = success Do
            (a, b) ← the leftmost disagreement pair of diff(X,Y)
            If a is a variable that does not occur in b
            Then
                add (a → b) to Mgu
                apply (a → b) to X
                apply (a → b) to Y
            Else If b is a variable that does not occur in a
            Then
                add (b → a) to Mgu
                apply (b → a) to X
                apply (b → a) to Y
            Else Ans ← failure
            End If
    End of While
    If Ans = success, then Mgu is a most general unifier of X and Y
    Else halt with failure
End of Unify
```

The justification for Robinson's algorithm centers around the Negotiability Lemma.

Negotiability Lemma: Two terms, t and u, are unifiable only if diff(t,u) is negotiable and, if λ unifies t and u, then it also unifies every disagreement pair in diff(t,u). Given the definition of negotiability, this lemma conforms to our intuition about unifiers. A complete proof can be found in [Rob3] but will not be repeated here.

From this lemma, it follows that Robinson's algorithm produces an mgu if and only if its two input terms are unifiable and halts with failure otherwise.

Proof: Clearly, if the algorithm produces a result, call it σ, then $\sigma t = \sigma u$, so σ unifies t and u, and t and u are unifiable. In addition, the algorithm must halt since at each iteration of the algorithm, one of the following must be true:

(i) The difference is nonempty and negotiable, in which case the algorithm selects a substitution $(V \rightarrow w)$ from the difference, eliminates the variable V from both input terms and continues.

(ii) The difference is empty and the algorithm halts with success (by definition, the difference of two terms is empty only if the two terms are identical, i.e., unified).

(iii) The difference is non-empty and non-negotiable and the algorithm halts with failure.

Eventually, some iteration must produce either an empty or nonnegotiable difference since as long as there is a variable in the difference, the algorithm will try to eliminate it. (If the variable cannot be eliminated then it must violate the occurs check; let us call such a variable nonnegotiable.) Since there are a finite number of variables in each initial input term, and so a finite number of variables in the first difference set, and since one variable is eliminated during each iteration in which the algorithm does not halt, and no variables are ever added, the algorithm must eventually produce either an empty difference set or a difference set containing no negotiable variables. A non-empty difference set containing no negotiable variables is itself non-negotiable. So the algorithm must eventually halt either with success (in the case of producing an empty difference set) or failure (in the case of a producing a nonnegotiable difference set).

It remains to be shown that if the input terms, t and u, are unifiable then the algorithm halts with success, producing a result μ, and μ is a most general unifier of t and u. The proof proceeds by showing that the substitution built satisfies the requirement of being most general during each iteration, that is, during each iteration, the following property holds (μ is the current substitution being built by the algorithm):

(p1) for all unifiers, λ, of t and u, $\lambda = \mu\lambda$.

Once (p1) is shown to be true, it follows immediately that if the algorithm halts with success and produces μ then μ is a most general unifier. In addition, if t and u are unifiable and if (p1) is true at each iteration of the algorithm, then the algorithm must halt with success since it must eventually produce an empty difference set. To see this recall that the algorithm must always halt with either

(i) diff(μt, μu) = $\varnothing$, or
(ii) diff(μt, μu) nonnegotiable.

Assume diff(μt, μu) is nonnegotiable. From the negotiability lemma, μt and μu are not unifiable. But, from (p1), if λ is a unifier of t then

$$\lambda = \mu\lambda$$

so $\lambda t = \mu\lambda t = \lambda u = \mu\lambda u$

and μt and μu are unifiable by λ

which contradicts our assumption that diff(μt, μu) is nonnegotiable. So, if (p1) is true, then whenever t and u are unifiable the algorithm must produce an empty difference set and halt with success. It remains to be shown that (p1) is true.

(p1) is trivially true at the beginning of the first iteration since $\mu = \varnothing$. The remainder of the proof proceeds by induction on the number of iterations. Assume that (p1) holds for the first i steps; that is, after i steps, for all unifiers, λ, of t and u, $\lambda = \mu\lambda$. If diff(t,u) is non-empty at step i+1 then it must be negotiable (since we are assuming that t and u are unifiable), and μ is replaced by $\mu(V \rightarrow w)$, where $(V \rightarrow w)$ is formed from some disagreement pair in diff(t,u). There must be such a pair since t and u are unifiable. For all unifiers, λ, of t and u, $\lambda = (V \rightarrow w)\lambda$ since

$(V \rightarrow w)\lambda V \quad = \lambda w \quad$ since $(V \rightarrow w)V = w$

$\qquad\qquad = \lambda V \quad$ since, by the negotiability lemma, λ must unify V and w

And, for all other variables, $X \neq V$,

$(V \rightarrow w)\lambda X \quad = \lambda X \quad$ since $(V\text{->}w)X = X$.

Based on our assumption that $\mu\lambda = \lambda$ at step i, and our result that $(V \rightarrow w)\lambda = \lambda$, it must be that

$\mu(V \rightarrow w)\lambda = \lambda$.

So, after step i+1, (p1) still holds.

By induction on i, (p1) must hold at the end of each iteration in which diff(t,u) is non-empty. If diff(t,u) is empty, then the algorithm halts with the substitution from the previous iteration, so the result must still satisfy $\lambda = \mu\lambda$ for all unifiers λ, so μ must be a most general unifier. QED.

2.3.2. Efficient Algorithms

Consider the unification of the terms $g(f(X_1,X_1),...,f(X_{n-1},X_{n-1}),X_n)$ and $g(X_2,...,X_n,X_n)$. Robinson's algorithm generates the mgu

$$\{ (X_2 \rightarrow f(X_1, X_1)),$$
$$(X_3 \rightarrow f(f(X_1, X_1), f(X_1, X_1))),$$
$$\text{... etc ...}$$
$$\}$$

ending with a final substitution containing 2^{n-1} occurrences of X_1. The final substitution requires building a term whose length is exponential in the number of variables contained in the initial input terms. The more efficient algorithms that follow notice that the above mgu can be rewritten as the sequential substitution

$$\{ (X_n \rightarrow f(X_{n-1}, X_{n-1})), \ldots, (X_2 \rightarrow f(X_1, X_1)) \}.$$

Note that this substitution is linear with respect to the number of distinct variables in the initial input terms. The efficient unification algorithms that follow minimize the time required for unification by producing a minimal length sequence of substitutions representing the mgu. The minimal representation is achieved by carefully ordering the resolution of disagreement pairs.

2.3.2.1. Paterson and Wegman

Paterson and Wegman [Pat] present an interesting correlation between unification and the computation of an equivalence relation on the elements of the input terms. They show that if terms are represented as ordered directed acyclic graphs (dags), then the unifiability of terms is equivalent to finding a *valid equivalence relation* (defined below) on the dags representing the terms.

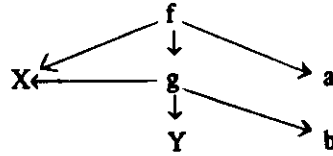
The mapping of terms to ordered dags is as follows:

(i) A function symbol, f, of arity n > 0 is representing by a node labeled f, with outdegree n such that the sons represents the arguments of the function f. The ordering of the sons represents the ordering of the arguments.

(ii) Each occurrence of a constant, b, is represented by a unique node of outdegree 0 labeled b.

(iii) Each distinct variable, X, is represented by a single node of outdegree 0 labeled X.

For example, the term f(X,g(X,Y,b),a) is represented by the graph



An equivalence relation on two dags is considered valid if it has the following three properties:

(i) If two function nodes are equivalent, then they have the same outdegree (arity) and their sons are pairwise equivalent in order (that is son_1 of $function_1$ must be equivalent to son_1 of $function_2$, etc...).

(ii) Each equivalence class is *homogeneous* in that it does not contain two distinct function symbols.

(iii) The equivalence classes can be partially ordered, and this ordering can be determined by the edges of the dags, i.e. two nodes, f and g, cannot be equivalent if f is an ancestor or descendant of g, an ancestor of f is equivalent to a descendant of g, or if a descendant of f is equivalent to an ancestor of g. This requirement ensures that the equivalence classes are still acyclic, implying a built-in occurs check.

In the following discussion, if t is a term, then t' refers to the root node of the dag representing t; if Σ is an equivalence relation, then $x' \equiv_\Sigma y'$ indicates that x' and y' are equivalent under the relation Σ. Patterson and Wegman's algorithm centers around the following lemma.

Lemma: Let x and y be two terms represented by dags having root nodes x' and y' respectively. x and y are unifiable if and only if there is a valid equivalence relation under which x' is equivalent to y'. If x and y are unifiable, then there exists a unique minimal valid equivalence relation under which x' is equivalent to y'.

Proof: If Σ is a valid equivalence relation and $x' \equiv_\Sigma y'$ then x and y must be unifiable since we can construct a unifier, σ, for them in the following way. Let $\sigma = \emptyset$. For each equivalence class C defined by Σ create a substitution in the following way. If C contains a non-variable node, w', (and by homogeneity, it can contain at most one), then for each variable node, V', in C add the substitution $(V \rightarrow w)$ to σ; otherwise, select one of the variable nodes in C, call it W', and for every other variable, V', in C

add the substitution $(V \rightarrow W)$ to σ.

Clearly, σ is a unifier of x and y since it pairwise and recursively equates the function symbol and arguments of x and y by pairwise and recursively equating the labels and sons of x′ and y′. So if there is a valid equivalence relation under which x′ is equivalent to y′, then we can build a unifier of x and y from it, and so x and y are unifiable.

Conversely, if x and y are unifiable, then we can define a valid equivalence relation, Σ, in the following way: Let σ be any unifier of x and y. For every pair of nodes a′ and b′ in the dags representing x and y, if $\sigma a = \sigma b$ then $a' \equiv_\Sigma b'$. To see that Σ is a valid equivalence relation note:

(i) If nodes a′ and b′ are equivalent under Σ, then their sons are pairwise equivalent under Σ.

Proof: Let $a = f(f_1, ..., f_m)$ and $b = g(g_1, ..., g_n)$ be the terms represented by the two equivalent nodes a′ and b′. Since a and b are unifiable (by definition of $a' \equiv_\Sigma b'$), we know f=g and m=n. Furthermore, since σ is a unifier of a and b, the negotiability lemma states that σ must unify f_i and g_i, so $f_i' \equiv_\Sigma g_i'$, $0 < i \leq m$. QED.

(ii) Each equivalence class is homogeneous.

Proof: For each pair of function nodes a′ and b′ in each equivalence class, C, defined by Σ, the terms represented by a′ and b′ must be unifiable since $a' \equiv_\Sigma b'$. Since the terms are unifiable, they must have the same function symbol, hence C must be homogeneous. QED.

(iii) The equivalence classes of Σ can be partially ordered by the partial ordering on the original dags.

Proof: Label the original input dags so that each member of each equivalence class is labeled with the term resulting from the application of the unifier σ to the class (since σ unifies the class, each member in a class must have the same label). Reduce the graph by combining nodes having a common label into a single node. The resulting graph depicts the father and son relationships between a set of finite (i.e., acyclic) terms, and so must be acyclic. Note: the original dags were acyclic, and σ can not introduce any cycles, so the result of applying σ to the input dags must be a dag. QED.

So, x and y are unifiable, if and only if there exists a valid equivalence relation under which x' is equivalent to y'.

Finally, if x and y are unifiable, then we can construct a unique minimal valid equivalence relation by equating nodes only as required by validity condition (i). That is, construct an equivalence relation that recursively equates the initial function symbols and corresponding sons of each term. This relation must be contained within every other valid relation (since every valid relation must comply with condition (i)), so it must be valid and minimal. It also must be unique (up to a renaming of variables) since there cannot be two ways to equate function symbols and sons. QED.

The following algorithm solves the unification problem by first finding the minimal valid equivalence relation for the dags representing the input terms, and then deriving a unifier for the terms from that relation as outlined above. Since the relation is minimal, the unifier must also be most general.

In the algorithm that follows, an equivalence between two nodes has been *propagated* when the pairwise equivalence of the sons of the nodes has been hypothesized. A *hypothesized* equivalence is one that has not been verified. An equivalence class is *complete* when all of its elements are known, *processed* when it has been checked for validity (i.e., its equivalences have been propagated and its homogeneity has been verified).

The algorithm decides if two terms are unifiable by hypothesizing that the root nodes of the two dags are equivalent under some relation and attempting to build an equivalence relation in which the root nodes are in a single class. The relation is built by propagating any known equivalences (i.e., if two fathers are equivalent then their sons must be) and verifying the resulting set of classes. New equivalence classes are introduced by the propagation of a known (or hypothesized) equivalences. Hypothesized classes can only grow through propagation from other classes. The algorithm processes classes until either all classes are complete and have been processed (i.e., a valid equivalence relation has been found), or until a clash is detected.

Paterson and Wegman point out that if the homogeneity of a hypothesized equivalence class is

verified each time a new member is added, it is possible to perform a number of comparisons that is nonlinear in the size of the class. If however, the processing of an equivalence class is deferred until the class is complete, then a linear check for homogeneity is possible. The linear behavior of the algorithm depends on the ability to determine when a class is complete.

Completed classes can be identified by exploiting the partial ordering on the nodes of the input dags. Since a class can only grow through propagation from father to son, a class is complete when all of the ancestors of each of the nodes within the class have been processed. Such a class is referred to as a *root class*. Note that a cycle can be detected by determining that there are unprocessed equivalence classes but no root class. That is, at least one node in each of the remaining classes has an unprocessed ancestor. This cycle detection provides a built-in occurs check.

The hypothesized equivalence of two nodes A and B is indicated during execution by adding a new undirected arc between A and B. When a class is processed, all of the nodes in the class are deleted from the graph. A list of fathers for each node is maintained. A root node is found by choosing any node and traversing the ancestor links until a node having no fathers is found. A complete root class can be found by finding a root node belonging to a hypothesized class populated entirely by root nodes.

Since there is a partial ordering on the nodes of the dags, a root class can be found by choosing any node, following its ancestor list to a root, R. If there is no such root then a cycle has been detected and unification fails. If there is a non-root node in the hypothesized class containing R, then ancestor list is followed until a root node is found. This process is repeated until a cycle is detected or a root class is found. The partial ordering on the nodes of dags guarantees that this process will find either a root class or a cycle.

Patterson and Wegman's Algorithm

```
Unify(X, Y)
Begin
        Set X equivalent to Y
        Mgu ← ∅; Ans = success
        While there is a root class R and Ans = success
        Begin
                If there is more than one distinct symbol in R
                        Then Ans ← failure
                End If
                Let R = {r1, r2,...,rn} where
                        if R contains a function node then r1 is a function node
                For i = 2 To n Do
                        If ri is a variable
                         Then add (ri → r1) to Mgu
                        End If
                        If ri is a function node with arity m
                        Then For j = 1 To m Do
                         Set jth son of r1 ≡ jth son of ri
                        End of For
                        End If
                End of For
                Delete the nodes of R
        End of While
        If Ans = success and there are no nodes remaining
                Then Halt with success -- Mgu contains the most general unifier
        Else Halt with failure
        End If
End of Unify
```

2.3.2.2. Martelli and Montanari

Martelli and Montanari represent the input terms as a set of multi-equations. Their multi-equations are basically equivalence classes written as an equation with all of the variables in the equivalence class on the left hand side of the equation and the remaining terms on the right. The initial multi-equation for the unification of two terms t and u is

$$\{ NewVar \} = \{ t, u \}$$

where NewVar is some new variable that does not appear in the input terms. In a multi-equation, each of the variables on the left hand side and each of the terms on the right hand side must be pairwise equivalent. There are three operations performed on the set of multi-equations; (i) reduction, (ii) compaction and (iii) common part calculation. Reduction is the propagation of Paterson and Wegman's algorithm. All together, the terms on the left and right hand side of any multi-equation represent one

equivalence class. Reduction propagates the equivalence by placing the arguments (sons) of the functions on the right hand side in the same equivalence class. Example: the result of applying reduction to the multi-equation

{ NewVar } = { f(g(Z),b), f(X,Y) }

is the set of multi-equations

{X} = { g(Z) }
{Y} = { b }

Compaction is the process of merging two multi-equations when they contain a variable in common on their left hand side. Example: the result of compacting the two multi-equations

{X} = { f(a) }
{X} = { f(Z) }

is

{X} = { f(a), f(Z) }

Common part calculation for a set M is a process of computing the structure that is common to each term in M. If the terms in a set are represented using Paterson and Wegman's dag representation described above, then the common part can be calculated by superimposing the dags on one another, taking the arcs the dags have in common as the arcs of the common part and labeling the nodes with the "most general" labeling from among the individual labelings. (Variables are considered more general than constants.) For example, the common part of { f(a,g(b,Z)), f(X,g(b,a)) } is f(X,g(b,Z)). If a node is labeled with two distinct function symbols, the terms do not have a common part. Note that implicit in the common part calculation is a test for homogeneity of function symbols within a multi-equation.

Initially, the input multi-equation set consists of one multi-equation with each of the input terms on the right hand side and a dummy variable on the left. A multi-equation can only be *processed* if the variables in its left hand side do not occur elsewhere. This requirement parallels the "root class" identification of Paterson and Wegman's representation. A multi-equation whose variables do not occur elsewhere is the "top" of the partially ordered equivalence relation represented by the current set of multi-equations. The processing of a multi-equation is similar to the processing of a root class in the

previous algorithm, i.e., equivalences are propagated (through reduction) and the multi-equation is marked "processed" (by its removal from the set). The algorithm halts when all multi-equations have been processed (i.e., removed) or there is no equation that can be processed (i.e., there is a cycle involving the variables in the remaining equations). Again, this represents a built-in occurs check.

Martelli and Montanari's Algorithm

```
Unify(X, Y)
Begin
   Mgu ← Ø
   Let U = the multiequation  {V} = {X, Y}
   While there is a multiequation S = M in U such that
          the variables in S do not occur elsewhere in U
   Do
          Let S = {V1,...,Vn}
          If M is empty
          Then
                        For i = 2 To n
                                    add the substitution (Vi → V1) to Mgu
          Else
             reduce M
             compact U
             For i = 2 To n
                        add the substitution (Xi → CommonPart(M)) to Mgu
          End If
          Remove S = M from U
   End of While
   If U is not empty
          Then halt with failure
          Else halt with success -- Mgu is the most general unifier
   End If
End of Unify
```

The implementations of reduction, compaction and finding the common part of a set do not add any insight to the algorithm and are omitted here.

The exponential complexity of Robinson's algorithm is avoided by only selecting a multi-equation for variable elimination if each of the variables on the left hand side of the multi-equation do not appear in any other equation. In effect, this is similar to the strategy of Paterson and Wegman's algorithm, i.e., the partial ordering of equivalence classes is exploited.

The right hand side of a multi-equation is checked for validity each time it is merged with another multi-equation (due to compaction), so some equations may be checked more often then necessary. The inefficiency due to this is justified, according to Martelli and Montanari, since inconsistencies

(or clashes) are detected much more quickly than in Paterson and Wegman's algorithm where the check for validity is postponed until a class is complete. In applications such as theorem proving or data base searching where there is a high probability of detecting a clash (i.e., non-unifiable terms), this inefficiency in the non-clash case does appear to be justifiable.

An additional algorithm due to Huet [Hue] and, independently, Robinson [Rob2] uses the Union-Find algorithm for maintaining equivalence classes.

2.3.2.3. Complexity of the Algorithms

Paterson and Wegman's algorithm has worst case complexity linear in the size of the input terms. Martelli and Montanari's is O(n log n) in the worst case, where n is the number of distinct variables in the input terms. The Union-Find algorithms of Huet and Robinson each have worst case complexity of O(m G(m)) where G is the inverse Ackerman function and m is the number of variable occurrences (not necessarily distinct variables) in the input terms.

Martelli and Montanari offer the following comments on the worst case complexity of each of the above algorithms for three separate cases: unification with (i) a high probability of success, (ii) a high probability of detecting a cycle, and (iii) a high probability of failure due to detecting a clash.

(i) **High probability of success.** Paterson and Wegman's algorithm performs best for the worst case of input terms, but all have good performance. (In an experimental test of linear and "non-linear but good" unification algorithms, Murray [Mur], determined that, for the average case, this distinction is negligible.)

(ii) **High probability of detecting a cycle.** Paterson and Wegman's algorithm is again best. Martelli and Montanari's algorithm is also good but might do some unnecessary merging (due to compaction), that is a set undergoes compaction before cycle detection. The Union-Find algorithms perform cycle detection as a final step and have very poor performance in this case.

(iii) **High probability of detecting a clash.** The Union-Find algorithms are best since no occurs check overhead is present. Martelli and Montanari's algorithm performs better than Paterson and

Wegman since multi-equations are checked for consistency whenever they are merged. Paterson and Wegman do not check for consistency until an equivalence is complete.

3. Predicate Locking

Predicate locking was first suggested by Eswaran et al. [Esw] as a mechanism for ensuring "consistency" in shared access database systems. A database is considered to be in a "consistent" state if the data within it satisfies some set of implicit or explicit "consistency constraints".

Informally, a consistency constraint is some statement about the values in a database that users expect to be true. For example, in a database containing the two entities A and B, users may want to require that the values of A and B always be equal.

It is often impossible to avoid temporarily violating some consistency constraint during the update of a database. For example, to increment the values of A and B in the database above by some non-zero value, Y, two separate actions need to be performed, namely (i) "increment A by Y", and (ii) "increment B by Y". Between these actions, the database will temporarily violate the constraint imposed, i.e., it will be in an inconsistent state.

3.1. Terminology

Sequences of operations on a database are often grouped into a single *transaction*. For purposes of this discussion, a transaction can be viewed as an atomic action that preserves consistency (for example, the sequence "increment A by Y", "increment B by Y" represents a consistency preserving transaction). As a further simplification, it will be assumed that transactions always modify any item they access.

For the remainder of this discussion, the relational database model [Cod] will be used to represent data base values. That is, a database will be viewed as a set of "tables" or "relations" comprised of "columns" or "fields". A single instance of a relation (i.e., a single row in the table) will be represented as a row of field values. As an illustration of this model consider a database comprised of the following two tables, a table of branch assets and a table of account balances for a particular bank.

Table 1: ASSETS

BranchName	TotalAssets
b1	a1
b2	a2
b3	a3

Table 2: BALANCES

AccntName	BranchName	Balance
c1	b1	x1
c2	b2	x2
c3	b3	x3
c4	b1	x4

Table 1 represents assets of branches within the bank. Table 2 represents account balances within the bank. A single tuple from table 2 will be written as the tuple (Name, Branch, Balance) and so on.

3.2. Consistency vs. Concurrency

In systems allowing shared access to a database, there are two conflicting goals: ensuring consistency and maximizing performance. To achieve maximal performance, maximal concurrency of user processes must be provided (here a process can be viewed as a sequence of consistency preserving transactions). To ensure that each user obtains a consistent view of the database, concurrency of transactions that access the same data must be regulated. To see why this is necessary, consider the following two processes.

Process 1: Update the balance of account A within branch B by X dollars:

P1.1: balance of A ← balance of A + X
P1.2: assets of B ← assets of B + X

Process 2: Audit the balance of branch B:

P2.1 Sum ← the sum of all account balances in branch B
P2.2 Verify that Sum = assets of branch B

Concurrency of these two processes could result in the failure of process 2 due to a temporary inconsistency introduced by process 1. For example, the sequence of the individual actions

{ P1.1, P2.1, P2.2, P1.2 }

results in the failure of process 2.

Two processes are said to be "in conflict" if they access the same data (in reality two processes can read the same data without being in conflict, i.e., without affecting each other, but for simplicity we won't consider this case).

Consistency can be ensured by introducing a mechanism that provides mutual exclusion of conflicting transactions. To provide mutual exclusion, the mechanism must be able to detect conflicts. If it can determine that two transactions are in conflict, then it can ensure mutual exclusion by scheduling the two processes to run at separate times. However, it is not generally possible to determine efficiently when two transactions will conflict due to the dynamic nature of most processes. For example, assume there are two processes P1 and P2. P1 has two branches, B1 and B2, only one of which is executed in a single invocation of P1. Selection of the branch to be executed is determined by the state of the database at some point in the execution of P1. The second branch, B2, is in conflict with process P2, i.e., accesses the same data, and must not be run concurrently. The first branch, B1, does not conflict with P2. So, P1 and P2 can be scheduled to run concurrently only if branch B1 of process P1 will be executed. Statically (as opposed to dynamically) determining if P1 and P2 can run concurrently would be costly. It would require simulating the execution of some portion of P1 to determine which branch will be executed. In this case, it may be more efficient to run P1 and P2 separately than to check for conflict; but this would eliminate some, possibly valid, concurrency. The purpose of predicate locking is to maximize concurrency under such conditions.

3.3. Locking

Locking is a mechanism that provides dynamic identification of conflicting transactions. Simply stated, a data item may have a "lock" placed on it (a data item is a table or set of records). The existence of a lock indicates that the data item is in use by a particular transaction. A lock request is made by a transaction for each data item that it needs to manipulate. It is the job of the lock manager to grant or deny lock requests. A lock request may be granted only if no other transaction has a lock on the data item in question. In addition, the lock manager must prevent manipulation of any item by a

transaction unless that transaction holds a lock on the item. Locks are released by the transaction whenever the data is no longer needed.

Locks guarantee mutual exclusion of conflicting transactions since no two transactions can receive a lock (and so manipulate) the same data item.

3.3.1. Granularity of Locks

A transaction may need to manipulate an entire table or only some small subset of tuples within a relation. If a transaction only needs a small group of records within a table, it would be wasteful to require that the transaction lock the entire table. To maximize concurrency, locks must be specified with the finest degree of granularity possible. For example, while running the above banking database audit process on branch b1, there is no reason to lock the entire ASSETS and BALANCE tables; such locking would prevent the concurrent execution of account balance updates running in the remaining branches even though there is no conflict between the two processes. To preserve consistency, it is sufficient to lock only those tuples of BALANCE having BranchName = b1. This would allow the concurrent execution of account balance updates for the remaining branches.

3.3.2. Phantom tuples

To really preserve consistency in the previous example, all currently existing tuples representing accounts in branch b1 must be locked in and the addition of new accounts in branch b1 must be blocked or locked out; the audit process would probably not run correctly if a new account was added to branch b1 during its execution. Non-existent tuples that satisfy an existing lock are referred to as *phantoms*. In order to lock the current state of the database, phantoms must be locked out and current tuples locked in.

3.4. Predicate Locks.

Eswaran et al. point out that the locking of phantom tuples can not be accomplished with the storage of physical locks (i.e., specifically storing which tuples in a relation are locked). They suggest that both existing and phantom tuples can be captured with "logical" locks, describing the set of all

possible (rather than existing) tuples to be locked. They specify logical locks by a predicate on the relation to be locked. In particular, a lock on the relation, R, is a predicate P on R that defines the set of locked tuples, S, in the following way:

$$\text{for a tuple } t,\ t \in S \text{ iff } P(t) \text{ is true.}$$

The truth or falsity of P(t) must be dependent only on the tuple t. So, the lock on branch b1 from above would be stored as

$$P(t) = (\ \text{BranchName}(t) = b1\ )$$

Recall that there are two functions associated with managing database locks: (i) granting new locks and (ii) verifying that a transaction holds a lock for each record that it accesses. The first function requires the ability to determine if two locks refer to the same piece of data (such locks are said to "conflict"). With locks specified as predicates, this implies the ability to determine if two predicates P and P′ "conflict" or, equivalently, if it is possible for there to exist a tuple t such that $P(t) \wedge P'(t)$. For arbitrarily complex predicates this determination is recursively unsolvable [Kle]. Eswaran et al. note that a decision procedure for predicate conflict exists for certain "simple predicates". Their simple predicates are any boolean combination of *atomic predicates* of the form

<field_name> OP <constant>, where OP can be $\leq$, $=$, $\neq$ or $>$

for alphabetic or numeric constants.

The decision procedure for satisfiability of two predicates P and P′ requires first forming the new predicate $P'' = P \wedge P'$, transforming P″ into Disjunctive Normal Form (DNF), i.e., disjunctions of conjuncts of atomic predicates, and determining if each of the disjuncts is satisfiable. Hunt and Rosenkrantz [Hun] point out that while it is possible to determine in polynomial time if the disjunctive normal form of P″ is satisfiable, transforming an arbitrary boolean combination of atomic predicates into DNF is NP-complete. Hunt and Rosenkrantz propose an alternative scheme for predicate locking which requires a separate algorithm for each possible size of table. Their algorithms are polynomial in the size of the input table, but have the obvious drawback that they only work for a fixed table size.

The following sections show how unification can be modified to provide a viable implementation

of predicate locking.

4. Unification and Predicate Locking

Predicate locking is essentially a problem of pattern matching. A complete lock manager can be implemented given the following two operations:

(i) Determine if lock A should be granted given that lock B exists; i.e., do A and B refer to any common record? Alternatively, can A and B be made to match?

(ii) Determine if record B can be accessed by transaction T given that transaction T holds lock A; i.e., is B an instance of A? Alternatively, can A and B be made to match?

If records or sets of records can be mapped into the terms of unification, then unification can be used to implement these two operations since pattern matching is precisely what unification does.

A single record can be mapped into a term in the following way:

- The table name of the record is mapped into the initial function symbol of the term.
- Each field of the record is mapped into the corresponding argument of the term (fields and arguments are both ordered left to right, i.e., field i is mapped into argument i so a record having n fields is represented by a term having arity n).
- Alphanumeric field values are mapped into corresponding alphanumeric constants.

For example, the assets and balances tables from the preceding section map into the following terms:

```
assets(b1,a1).
assets(b2,a2).
assets(b3,a3).
balances(c1,b1,x1).
balances(c2,b2,x2).
balances(c3,b3,x3).
balances(c4,b1,x4).
```

Certain sets of records can be mapped into a single term if they belong to the same table. Sets are mapped into a term in the following way:

- The table name containing the set is mapped into the initial function symbol of the term.
- Any single field for which each of the records in the set have the same value, v, is mapped into the corresponding argument of the term as the value v.

- Every other field is mapped into the corresponding argument of the term as a distinct variable.

For example, the set of all possible records in the assets table can be mapped into the term assets(X,Y). The set of all possible balance records for branch b1 can be represented by the term balances(A,b1,B). Note that these terms represent all possible records satisfying a set of constraints, not just all existing records. Terms containing variables can represent both current and phantom records. The term, balances(A,b1,B), represents precisely what the Eswaran lock (BranchName(t) = b1) represents.

This suggests that the terms of unification can be used as a natural model of limited predicate locks (a later section discusses the limitations of terms as predicate locks and how these limitations might be lifted). If a lock is expressed as a term, then the two principle functions of a lock manager can be handled efficiently by unification:

(i) Determine if lock A should be granted given that lock B exists, i.e., are A and B unifiable?

(ii) Determine if record B can be accessed by transaction T given that T holds lock A, i.e., are A and B unifiable?

4.1. Limitations of Terms as Predicate Locks

A single term can adequately model any satisfiable conjunction of tests for equality of field values (i.e., the '=' atomic predicate of Eswaran locks). The initial function symbol of the term gives the name of the table being locked. Since a single '=' predicate requires that the value of some field, i, equal a specific value, a term modeling this predicate must contain the value v as its ith argument. Conjunctions of '=' predicates can be modeled by placing each specified field value in the corresponding argument of t and placing a distinct variable in every other argument. As an example, the conjunction "(f1=v1) and (f2=v2) and (f3=v3)" for a table, t, having 4 fields can be represented by the term t(v1,v2,v3,X). Note that if there are two predicates

'field i = v1' and 'field i = v2', v1 ≠ v2,

then the predicate is unsatisfiable.

Disjunction and negation of '=' comparisons can also be modeled by simple terms by translating

disjunction and negation of comparisons into disjunction and negation of unification. For example, the lock "(f1=v1) and (f2=v2) or (f2=v3) and (f4=v4)" can be represented by the two terms

t(v1,v2,X,Y)
t(X,Y,v3,v4).

That is, an Eswaran lock in disjunctive normal form of n disjunctions (and limited to the '=' predicate) can be represented by n+1 separate term-locks. A slight modification needs to be made to the lock manager for this to be valid: the lock manager must grant a lock request even though it conflicts with an existing lock if both locks are held by the same transaction. This is easy to allow.

As an example of negation, the lock "(f1=v1) and not(f2=v2) or not((f2=v3) and (f4=v4))" can be represented by the combination of conjunction, negation and disjunction of terms

t(v1,X,Y,Z) and not t(U,v2,V,W)
not t(X,Y,v3,v3)

The negation of terms can be handled easily in the lock manager by reversing the sense of unification, that is by checking for "not unify" rather than "unify".

So, the '=' comparison can be modeled completely using standard unification, though this modeling still requires that locks be stated in (or translated to) disjunctive normal form. As mentioned above, this requirement can lead to exponential space requirements.

Modeling the remaining atomic predicates, '≠', '≤' and '>', will require some extensions to the terms of unification. The next section outlines a framework for extensions to unification that allows a complete modeling of Eswaran predicate locking.

5. Extended Unification

5.1. Extended Terms

If terms are extended to include the notions of inequality and of simple comparisons of argument values, then they can be used as a natural model for all of the predicate locks suggested by Eswaran et al. For example, consider the following *extended terms:*

(i) Any constant symbol or variable is an extended term.
(ii) If a is a constant, then each of the following constructs is an extended term:

not a
≤ a
> a

(iii) If $t_1, t_2, \ldots t_n$ are extended terms and $f \in F$ is a function symbol of arity n, then $f(t_1, \ldots, t_n)$ is an extended term.

For example, t(not v), where v is some constant, is a term. The field comparisons '≠', '≤' and '>' map directly onto the constructs 'not a', '≤a' and '>a', respectively. Although these extended terms can adequately model all of the Eswaran predicate locks, the unification of two of these terms is not easily defined, since the definition of unification is to produce a substitution of values for variables that makes two term identical. For example, intuitively, the terms

t(not a) and t(b), where $a \neq b$

should match, but there are no variables to be substituted, so it is not clear what unification should produce.

The substitution issue can be resolved by introducing variables wherever these new constructs are used, and placing a constraint representing the construct on the associated variable. For example,

t(not v)
is rewritten t(X:not v)
and read "t of X where X is subject to the constraint 'not v'"

A formal definition of constraints is given later.

Another shortcoming of terms as a model of predicate locks is the issue of the exponential space requirements arising from expanding predicates into disjunctive normal form. The above model does not remove this requirement. However, some of the most space consuming locks can be rewritten more efficiently by expanding the constraints allowed on variables to include disjunction and conjunction of

constraints. For example, if both comparisons of a disjunct deal with the same field, e.g., branch = b1 or branch = b2, then the disjunct can be modeled in a single term by allowing disjuncts of constraints to be placed on variables, that is by allowing constructs like 'X:1 or 2'. For example, without disjuncts of constraints, modeling the predicate

(branch = b1 or branch = b2) and (balance > 1000 or balance < 10)

for the balance table would require the four terms:

balance(A, b1, C : >1000)
balance(A, b1, C : <10)
balance(A, b2, C : >1000)
balance(A, b2, C : ><10).

However, by allowing disjuncts of constraints, the predicate can be modeled by the single term

balance(A,B:b1 or b2, C: >1000 or <10).

Similarly, the predicate

balance > 10 and balance < 1000

can be modeled as

balance(A,B,C: >10 and <1000).

These extensions do not resolve all of the space problems, but they do allow a large class of useful predicate locks to be modeled efficiently.

The following gives a more formal and more general definition of extended terms.

A *valid constraint specification* is defined to be any expression, E, that is defined to "match" a possibly empty set of constant values (i.e., a set of numeric constants or 0-arity function symbols) and that satisfies the following:

(i) If A is the set of all possible constants, then there is a procedure $M{:}A \times \{E\} \rightarrow \{\text{true},\text{false}\}$ such that for all $a \in A$, $M(a,E)$=true if a "matches" E otherwise $M(a,E)$=false (so for all $a \in A$, $M(a,E)$ is decidable).

(ii) The application of $(V \rightarrow t)$ to a constraint specification E, written $(V \rightarrow t)E$, is defined to be the result of replacing all occurrences of V in E by t. For all substitutions $(V \rightarrow t)$, $(V \rightarrow t)E = E'$ is a valid constraint specification, that is, if any variable in E is substituted by a term, then the result

is still a valid constraint spec. Furthermore, if a constant, a, matches E' then it matches E, that is, a substitution can not cause a constraint to match more then it originally matched.

There are no other restrictions on "match" or M.

A *valid set of constraint specifications* is defined to be any set, E, of valid constraint specifications such that the following properties hold:

(i) There is a procedure M:AxE→{true,false} such that for all a ∈ A, for all $E \in$ E, M(a,E)=true if a "matches" E, otherwise M(a,E)=false. For all substitutions (V→t), if $E \in$ E, then (V→t)$E \in$ E.

(ii) There is a procedure I:ExE→{true,false} such that for all $E,F \in$ E, I(E,F)=true if there is a constant a such that M(a,E)=M(a,F)=true, otherwise I(E,F)=false; that is I(E,F)=true only if E and F are simultaneously satisfiable (so I is decidable for all E, $F \in$ E). These are the only restrictions placed on valid constraint specifications. There is no restriction on the form that the "match" test can take. Note that the above definition allows constraint specifications to contain variables. For example, in addition to allowing comparisons with constants, such as balance(A,B,C: >100), comparisons between field values are allowed, such as balance(A,B,C: >A). The definition also allows more complicated expressions such as range expressions like balance(A,B,C:1-5).

An *Econstraint, C,* over a set E of valid constraint specifications is defined to be

(i) any $E_i \in$ E, or
(ii) the intersection of Econstraints $E_1,...,E_n$ (written $\{E_1, \ldots E_n\}$).

An *Econstrained variable,* written X:C, is defined to be a variable that may only be substituted by terms that "match" at least one of the Econstraints in C, where C is some set of Econstraints; the satisfaction of the constraint is only dependent on the properties of the set E containing C. That is, the actual "match" test is outside of the domain of the current discussion. (Recall that the only restriction that is placed on the match test is that it be decidable.)

An *Eterm* for some set E of valid constraint specifications is defined to be a pair (t, C) where t is a term and C is a set of Econstrained variables.

Intuitively, we want an Eterm to specify a term having constraints on some of its variables.

Furthermore, we want the unification of two Eterms to be equivalent to the unification of two standard terms with the additional requirement that all substitutions for a variable, X, satisfy all constraints on every member of the equivalence class containing X.

The goal of the remainder of this chapter is to define the unification of two Eterms. First, an alternative model of unification that is equivalent to the traditional approach is presented, and its equivalence to standard unification is demonstrated. Then, the unification of Eterms is defined as a natural extension of this model.

5.2. Projections

The *projection* of a standard term, t, written proj(t), is the set of all ground terms constructible by substitutions of terms for variables. For example, proj(f(a)) = {f(a)} since there are no variables to be substituted, while proj(f(X)) is the set of all ground terms having function symbol f (here, f is always of arity 1). Since the set of all possible function symbols is finite, for all terms, t, proj(t) is countably infinite.

The following are interesting and useful properties of projections which will be used in subsequent sections.

(p1) $\text{proj}(f(X_1, ..., X_n)) = \{ f(Y_1, ..., Y_n) \mid Y_i \in \text{proj}(X_i)\}$.

(p2) If proj(t) = proj(u) then either

- (i) t and u are both variables, or
- (ii) the initial function symbol of t = the initial function symbol of u, and
  the arity of t = the arity of u, and
  for each i=1,...,(arity of t), proj(ith argument of t) = proj(ith argument of u)

(p3) If a is a ground term then proj(a) = a.

(p4) If the projections of two terms, t and u, are non-empty and identical, then t and u must be identical up to a renaming of variables.

Proof: Assume this is not the case, that is, assume that there exist two distinct terms t and u that differ by more than a simple variable renaming, and proj(t) = proj(u). Let (x,y) be the leftmost disagreement pair in diff(t,u) such that x and y are not both variables. Such a pair must exist

since t and u are distinct. Furthermore, either

(i) x and y are both non-variable, or
(ii) one is a variable while the other is non-variable.

case (i): Assume both x and y are non-variable. Since (x,y) is a disagreement pair and both x and y are non-variable, either x and y have different initial function symbols or different arities.

(Recall, $\text{diff}(f(t_1, ..., t_n), f(u_1, ..., u_n)) = \{ \text{diff}(t_1, u_1), ..., \text{diff}(t_n, u_n)\}$.)

But, proj(x) = proj(y) from proj(t) = proj(u) and (p2), so x and y must have the same function symbol and arity (again from (p2)), so we reach a contradiction.

case (ii): Assume x is a variable and y is a non-variable having initial function symbol f and arity n.

Again, proj(x) = proj(y) from proj(t) = proj(u) and (p2). But, proj(y) is only the set of all ground terms of the form $f(t_1, ..., t_n)$, while proj(x) is the set of all ground terms constructible from a finite set of function symbols. So, again, we reach a contradiction.

So, our assumption that two distinct terms, t and u, can have equal projections can not be true. QED.

(p5) If two terms are identical up to a renaming of variables, then their projections are identical.

(p6) If two terms, t and u are unifiable, then proj(t) $\cap$ proj(u) is non-empty.

Proof: Since t and u are unifiable, there must be a substitution, μ, such that $\mu t = g = \mu u$, but then $g \in$ proj(t) and $g \in$ proj(u) so $g \in$ proj(t) $\cap$ proj(u). QED.

Projections give us a nice model of the unification of two terms. By (p6), two terms unify only if the intersection of their projections is non-empty. We would like to say that if the intersection is non-empty, then the two terms unify. That is we would like to say that two terms, t and u, have a most general unifier, μ, if and only if

(*) proj(t) $\cap$ proj(u) = proj(S)

where proj(S) is a non-empty set of terms, and

$\mu(t) = \mu(u) = S$.

In the next section, we will use this model of projections and their intersections to clarify the definition of unification of extended terms. But first, we need to determine under which conditions unification and the intersection of projections are equivalent.

If two terms, t and u, share variables, then (*) does not hold. As a counter-example, note that if t=f(X,Y) and u=f(Y,X), then $f(1,2) \in proj(t) \cap proj(u)$, but t and u are only unifiable by substitutions that make X and Y equal, such as $(X \rightarrow Y)$; that is, substitutions that make the first and second argument equal. The term f(1,2) can not be generated by such a substitution. However, as shown below, (*) does hold if t and u do not share any variables. Conveniently, given two terms t and u, the unification of t and u is equivalent to the unification of f(t,u) and f(X,X), where X is a variable that does not occur in t or u, since t and u must be simultaneously unifiable with X in order for the outer terms to be unifiable. So, the unification of two input terms that share variables can always be restated as the unification of two terms that do not. For the remainder of this paper, we will assume that the input terms do not share variables.

(p7): If σ is a unifier of terms t and u, then proj($\sigma(t)$) = proj($\sigma(u)$). Furthermore, if

proj($\sigma(t)$) = proj($\sigma(u)$) then there exists a simple variable renaming, ρ, such that $\sigma\rho$ is a unifier of t and u.

Proof: If σ is a unifier then $\sigma(t) = \sigma(u)$, since, by (p4), we know the projections of two identical terms are identical. Conversely, if the projections of $\sigma(t)$ and $\sigma(u)$ are identical, then by (p5), we know the terms $\sigma(t)$ and $\sigma(u)$ must be identical up to a renaming of variables, so there must be a simple variable renaming, ρ, such that $\sigma\rho$ is a unifier of t and u. QED.

(p8): If σ is a set of substitutions and t is a term, then proj($\sigma(t)$) $\subseteq$ proj(t).

Corollary of (p7) and (p8): σ is a unifier of t and u only if

$proj(\sigma t) = proj(\sigma u)$ and
$proj(\sigma t) \subseteq proj(t) \cap proj(u)$ and
$proj(\sigma u) \subseteq proj(t) \cap proj(u)$.

Furthermore, if

$proj(\sigma t) = proj(\sigma u)$ and
$proj(\sigma t) \subseteq proj(t) \cap proj(u)$ and
$proj(\sigma u) \subseteq proj(t) \cap proj(u)$

then there exists a simple variable renaming, ρ, such that $\sigma\rho$ is a unifier of t and u.

Projection Theorem: Let t and u be terms with disjoint variables. If μ is a most general unifier of t and u then

$$proj(\mu t) = proj(\mu u) = proj(t) \cap proj(u).$$

Furthermore, if

$$proj(\mu t) = proj(\mu u) = proj(t) \cap proj(u).$$

then there exists a simple variable renaming, ρ, such that $\mu\rho$ is a most general unifier of t and u.

Proof: Assume μ is an mgu of t and u. Since μ is a unifier, we know

$$proj(\mu t) = proj(\mu u) \subseteq proj(t) \cap proj(u)$$

We need to show that

$$proj(t) \cap proj(u) \subseteq proj(\mu t).$$

For every g such that $g \in proj(t) \cap proj(u)$ there exist substitutions ρ and σ such that

$$g = \rho t \text{ and } g = \sigma u \text{ (by definition of projections).}$$

Since t and u have disjoint variables, we may assume that ρ does not contain any variables in u, and σ does not contain any variables in t, so

$$\rho\sigma t = g = \rho\sigma u.$$

So the composition, $\rho\sigma$, is a unifier of t and u. But, by assumption, μ is a most general unifier of t and u so we must be able to decompose the unifier $\rho\sigma$ into the product $\mu\rho\sigma$ (by the definition of mgu). So, $g = \mu\rho\sigma t$ and $g \in proj(\mu t)$.

Conversely, assume $proj(\mu t) = proj(\mu u) = proj(t) \cap proj(u)$. We need to show that there exists a simple variable renaming, ρ, such that $\mu\rho$ is a most general unifier of t and u.

From (p4), we know that if $proj(\mu t) = proj(\mu u)$ then there exists a variable renaming, ρ, such that $\mu\rho$ is a unifier of t and u. If σ is a most general unifier of t and u, then $\sigma\mu\rho = \mu\rho$ (from the definition of mgu). So the application of $\mu\rho$ to t is equivalent to first applying σ to t and then applying $\mu\rho$ to the

result, so

$$proj(\mu\rho t) \subseteq proj(\sigma t)$$

which implies that $proj(\mu t) \subseteq proj(\sigma t)$. But, we also know that each term in $proj(\sigma t)$ is also in $proj(\mu t)$, since

$proj(\sigma t) \subseteq proj(t) \cap proj(u)$	since σ is a unifier
$= proj(\mu t)$	by assumption.

So, $proj(\mu t)$ must equal $proj(\sigma t)$. But this implies that $\mu t = \sigma t$ up to a renaming of variables, by (p4). Let λ be a renaming of variables such that $\mu\lambda = \sigma$, $\mu\lambda$ is an mgu. QED.

Hence unification and projection intersection are closely related in that terms (which do not share variables) are unifiable if and only if their projections are intersectable.

The projection model of unification allows a simpler proof of the uniqueness of mgus:

Uniqueness Theorem: If μ is an mgu of terms t and u then μ is unique with respect to the projection of t and u under μ.

Proof: Assume μ is an mgu. Then $proj(\mu t) = proj(\mu u) = proj(t) \cap proj(u)$, from the projection theorem. If σ is also an mgu, then $proj(\sigma t) = proj(t) \cap proj(u) = proj(\mu t)$. QED.

In the next section, the unification of extended terms is defined using the model of intersection of projections developed here.

5.3. Extended Unification

An *extended substitution*, γ, is written $\gamma = (\lambda, D)$, where λ is a standard substitution and D is a set of constrained variables. The *application* of the extended substitution, $\gamma = (\lambda, D)$, to an Eterm, $e = (t, C)$, is written γe and is defined by the Eterm $(\lambda t, \lambda(C \cup D))$.

The *application* of a simple substitution, $\lambda = (V \rightarrow t)$ to a set of constrained variables,

$$C = \{X_1{:}c_1, \ldots, X_n{:}c_n\},$$

written λC, is defined by

(i) If $\lambda = (V \rightarrow t)$, and t is non-variable, then

$$\lambda C = \{ \lambda(X_i):\lambda(c_i) \mid 0 \leq i \leq n \}$$

(ii) If $\lambda = (V \rightarrow t)$, t a variable, then

$$\lambda C = \lambda C' \cup \lambda C''$$

where

$C' = \{X_i:c_i \mid X_i \neq V \text{ and } X_i \neq t\}$, and
$C'' = \{X_i:c_i \mid X_i = V \text{ or } X_i = t\}$.
Note: $C' \cap C'' = \emptyset$.
$\lambda(C') = \{X_i:\lambda c_i \mid X_i:c_i \in C'\}$
$\lambda(C'') = \{t:\text{closure}(C'')\}$

where

$$\text{closure}(C) = \bigwedge_i \{ c_i \mid 0<i\leq|C|, X_i:c_i \in C\}$$

That is, t is constrained by the conjunction of all constraints initially placed on V or t. Note this conjunction may be unsatisfiable.

The *application of a simultaneous substitution*, σ, to a set of constrained variables is defined to be equivalent to the simultaneous application of each simple substitution in σ. The *application of a sequential substitution*, σ, to a set of constrained variables is defined to be equivalent to the sequential application of each simple substitution is σ.

Constraint sets C and D are defined to be *equivalent* if, for all simple substitutions, $(V \rightarrow t)$, $(V \rightarrow t)C$ is satisfiable iff $(V \rightarrow t)D$ is satisfiable. Alternatively, constraint sets C and D are defined to be equivalent if for all terms, t, Eproj((t,C)) = Eproj((t,D)). Eterms $e=(t,C)$ and $f=(u,D)$ are defined to be *equivalent* if $t=u$ and $C \equiv D$. Extended substitutions α and β are defined to be *equivalent* if, for all Eterms, e, $\alpha e \equiv \beta e$.

The following are interesting and useful properties of constraint sets.

(s0): $\{X:A\} \cup \{X:B\} \equiv \{X:A \cap B\}$.

(s1): If C and D are constraint sets $C = \{ X_i:c_i, ..., X_n:c_n\}$ and $D = \{ Y_i:d_i, ..., Y_n:d_n\}$ then

$(V \rightarrow t)(C \cup D) \equiv (V \rightarrow t)C \cup (V \rightarrow t)D$.

Proof: If t is non-variable, then

$(V \to t)(C \cup D)$
$= \{ (V\to t)X_i{:}(V\to t)c_i \mid X_i{:}c_i \in C\} \cup \{ (V\to t)Y_i{:}(V\to t)d_i \mid Y_i{:}d_i \in D\}$
$= (V\to t)C \cup (V\to t)D.$

If t is a variable, then

$(V\to t)C \cup (V\to t)D$
$= (V\to t)C' \cup (V\to t)C'' \cup (V\to t)D' \cup (V\to t)D''$

where,
$C' = \{X_i{:}c_i \mid X_i \neq V \text{ and } X_i \neq t\}$
$C'' = \{X_i{:}c_i \mid X_i = V \text{ or } X_i = t\}$
$D' = \{Y_i{:}d_i \mid Y_i \neq V \text{ and } Y_i \neq t\}$
and, $D'' = \{Y_i{:}d_i \mid Y_i = V \text{ or } Y_i = t\}$
so,
$(V\to t)C \cup (V\to t)D$
$= (V\to t)C' \cup \{t{:}\text{closure}(C'') \cup (V\to t)D' \cup \{t{:}\text{closure}(D'')\}$
$= (V\to t)C' \cup (V\to t)D' \cup \{t{:}\text{closure}(C'') \cap \text{closure}(D'')\},$ by (s0)
$= (V\to t)C' \cup (V\to t)D' \cup \{t{:}\text{closure}(C'' \cup D'')\}$
$= (V\to t)(C \cup D).$
QED.

Corollary to (s1): If μ is any substitution, then $\mu(C \cup D) = \mu C \cup \mu D$.

(s2): If C is a constraint set, then $(V\to t)(V\to t)C = (V\to t)C$.

Proof: $(V\to t)$ eliminates V from C, so a second application of $(V\to t)$ cannot modify the result of the first application. QED.

Corollary to (s2): If C is a constraint set and μ is any substitution, then $\mu\mu C = \mu C$.

(s3): If C and D are constraint sets, and μ is a substitution, then

$$\mu(C \cup \mu(C \cup D)) = \mu(C \cup D).$$

Proof:

$\mu(C \cup \mu(C \cup D)) = \mu C \cup \mu\mu C \cup \mu\mu D,$ from the corollary to (s1).
$= \mu C \cup \mu C \cup \mu D,$ from the corollary to (s2),
$= \mu C \cup \mu D$
$= \mu(C \cup D).$
QED.

The ***extended projection*** of an Eterm e, written Eproj(e) is the set of all ground terms constructible from e by the application of an extended substitution, that is, g $\in$ Eproj(e) if there exists $\rho = (\sigma, D)$ such that $\rho e = (g, C)$ where C is a satisfiable constraint set.

The following is an important property of extended projections.

(e1): If $e = (t,C)$ is an Eterm, then $\text{Eproj}(e) \subseteq \text{proj}(t)$.

Proof: By definition,

$$\text{Eproj}(e) = \{ \alpha t \mid \alpha t \text{ is a ground term and } \alpha C \text{ is satisfiable} \}$$
$$\text{proj}(t) = \{ \alpha t \mid \alpha t \text{ is a ground term} \}$$

Clearly, Eproj(e) cannot contain any ground terms that are not also in proj(t). QED.

An extended substitution, $\sigma = (\lambda, E)$ is defined to be a ***unifier*** of two Eterms, $e=(t,C)$ and $f=(u,D)$, if $\sigma e \equiv \sigma f$, that is,

$$\lambda t = \lambda u \text{ and}$$
$$\text{Eproj}(\sigma e) = \text{Eproj}(\sigma f) \neq \varnothing$$

The following are important properties of unifiers of Eterms.

(u1): If σ is a unifier of Eterms e and f, then $\text{Eproj}(\sigma e) \subseteq \text{Eproj}(e) \cap \text{Eproj}(f)$.

Proof:

$$\text{Eproj}(\sigma e) \subseteq \text{Eproj}(e)$$
$$\text{Eproj}(\sigma f) \subseteq \text{Eproj}(f) \qquad \text{from (e1)}$$
$$\text{so} \quad \text{Eproj}(\sigma e) \cap \text{Eproj}(\sigma f) \subseteq \text{Eproj}(e) \cap \text{Eproj}(f).$$

Furthermore, by definition of unifiers,

$$\text{Eproj}(\sigma e) = \text{Eproj}(\sigma f)$$
$$\text{so} \quad \text{Eproj}(\sigma e) = \text{Eproj}(\sigma e) \cap \text{Eproj}(\sigma f) \subseteq \text{Eproj}(e) \cap \text{Eproj}(f).$$

QED.

(u2): If $\sigma = (\lambda,E)$ is a unifier of $e=(t,C)$ and $f=(u,D)$, then t and u are unifiable and $\lambda(C \cup D)$ is satisfiable.

Proof: By definition, $\lambda t = \lambda u$, so t and u must be unifiable. Furthermore,

$$\text{Eproj}(\sigma e) = \{ \lambda\alpha t \mid \lambda\alpha t \text{ is a ground term and } \lambda\alpha C \text{ is satisfiable} \},$$
$$\text{Eproj}(\sigma f) = \{ \lambda\beta u \mid \lambda\beta u \text{ is a ground term and } \lambda\beta D \text{ is satisfiable} \},$$
$$\text{Eproj}(\sigma e) = \text{Eproj}(\sigma f) \neq \varnothing$$
$$\text{so} \quad \{ \lambda\alpha t \mid \lambda\alpha t \text{ is a ground term and } \lambda\alpha C \text{ is satisfiable} \}$$
$$= \{ \lambda\beta u \mid \lambda\beta u \text{ is a ground term and } \lambda\beta D \text{ is satisfiable} \}$$

and, since $\lambda t = \lambda u$,

$$= \{ \lambda\ \alpha t \mid \lambda\alpha t \text{ is a ground term and } \lambda\alpha C \wedge \lambda\alpha D \text{ is satisfiable} \} \neq \varnothing.$$

So there exists at least one ground term, g, such that $g=\lambda\alpha$ and $\lambda\alpha C$ and $\lambda\alpha D$ are satisfiable.

Hence

$$\lambda\alpha C \cup \lambda\alpha D \text{ is satisfiable,}$$
and by (u1), $\lambda\alpha(C \cup D)$ is satisfiable.

If $\lambda\alpha(C \cup D)$ is satisfiable then $\lambda(C \cup D)$ must be. QED.

A unifier μ of two Eterms, e and f, is defined to be a *most general unifier* if

$$\text{Eproj}(\ \sigma e) = \text{Eproj}(\ e\ ) \cap \text{Eproj}(\ f\ ) \neq \varnothing.$$

Theorem: Eterms $e=(t,C)$ and $f=(u,D)$ are unifiable iff t and u are unifiable by some mgu μ such that $\mu(C \cup D)$ is satisfiable. Furthermore, if t and u have mgu μ and $\mu(C \cup D)$ is satisfiable, then $(\mu, \mu(C \cup D))$ is an mgu of e and f.

Proof: Assume that e and f are unifiable by $\sigma=(\lambda,E)$. By property (u2), we know that t and u are unifiable by λ and $\lambda(C \cup D)$ is satisfiable. Since t and u are unifiable, there must be a most general unifier for them, call it μ, and $\lambda = \mu\lambda$. So $\mu\lambda(C \cup D)$ must be satisfiable, and $\mu(C \cup D)$ must be satisfiable.

Conversely, assume that μ is an mgu of t and u and $\mu(C \cup D)$ is satisfiable. To show that e and f are unifiable, we need to construct an extended substitution, $\sigma=(\lambda,E)$, such that

$\lambda t=\lambda u$ and
$\text{Eproj}(\ \sigma e\ ) = \text{Eproj}(\ \sigma f\ ) \neq \varnothing$.

Let $\sigma = (\mu, \mu(C \cup D))$. Clearly, $\mu t=\mu u$ since μ is an mgu of t and u. To see that

$$\text{Eproj}(\ \sigma e\ ) = \text{Eproj}(\ \sigma f\ ) \neq \varnothing,$$

note that

	$\text{Eproj}(\ \sigma e\ ) = \text{Eproj}(\ (\mu t, \mu(C \cup \mu(C \cup D)))\ )$
and	
	$\text{Eproj}(\ \sigma f\ ) = \text{Eproj}(\ (\mu u, \mu(D \cup \mu(C \cup D)))\ ).$
We know	
	$\mu t = \mu u.$
Furthermore,	$\mu(C \cup \mu(C \cup D)) \equiv \mu(C \cup D) \equiv \mu(D \cup \mu(C \cup D)),$
so	$\mu(C \cup \mu(C \cup D))$ is satisfiable iff $\mu(D \cup \mu(C \cup D))$ is satisfiable
hence	$\text{Eproj}(\ \sigma e\ ) = \text{Eproj}(\ \sigma f\ ).$

Finally, we need to show that if t and u have mgu μ and $\mu(C \cup D)$ is satisfiable, then $\sigma=(\mu, \mu(C \cup D))$ is an mgu of e and f. We know, from above, that σ is a unifier of e and f. We only need to show that

it is most general, that is Eproj(σe) = Eproj(e) $\cap$ Eproj(f). Since σ is a unifier, we know

Eproj(σe) $\subseteq$ Eproj(e) $\cup$ Eproj(f).

In addition,

Eproj(e) $\cap$ Eproj(f)
= { αt | αt is a ground term and αC is satisfiable}
$\cup$ { αu | αu is a ground term and αD is satisfiable}.

For all ground terms, g, in Eproj(e) $\cap$ Eproj(f), there exist substitutions α and β such that $\alpha t = g = \beta u$, αC is satisfiable and βD is satisfiable, α does not contain any variables in u, and β does not contain any variables in t (recall, we are assuming that t and u do not share any variables, and g is a ground term, so we must be able to construct α and β in such a way that they do not assign the same variables). Since α does not assign any variables in u, $\alpha u = u$, and since g is a ground term, $\beta g = g$, so

$\alpha\beta t = g = \alpha\beta u$,

and $\alpha\beta$ is a unifier of t and u. So,

Eproj(e) $\cap$ Eproj(f) = { $\alpha\beta$t | $\alpha\beta t = \alpha\beta u$ is a ground term and $\alpha\beta C \wedge \alpha\beta D$ is satisfiable}.

We know that μ is an mgu of t and u, so $\alpha\beta = \mu\alpha\beta$, and

Eproj(e) $\cap$ Eproj(f) = { $\mu\alpha\beta$t | $\mu\alpha\beta$t is a ground term and $\mu\alpha\beta C \wedge \mu\alpha\beta D$ is satisfiable}.
since $\mu\alpha\beta t = \mu\alpha\beta u$

$\mu\alpha\beta C \wedge \mu\alpha\beta D \equiv \mu\alpha\beta C \cup \mu\alpha\beta D$
$\equiv \mu\alpha\beta(C \cup D)$ by the corollary to (s1).

So,

Eproj(e) $\cap$ Eproj(f) = { $\mu\alpha\beta$t | $\mu\alpha\beta$t is a ground term and $\mu\alpha\beta(C \cup D)$ is satisfiable}
= Eproj((μt, $\mu(C \cup D)$))
= Eproj((μt, $\mu(C \cup \mu(C \cup D))$))
= Eproj((μ, $\mu(C \cup D)$)(t,C))
= Eproj(σe).
QED.

Corollary: If Eterms e and f are unifiable, then there is at least one mgu for them.

Corollary: If σ is an mgu for Eterms e and f, then σ is unique with respect to the Eproj of e and f under σ. That is, if γ is also an mgu for e and f, then

Eproj(σe) = Eproj(σf) = Eproj(γe) = Eproj(γf)

Proof: Since σ and γ are mgus for e and f, we know

$$\text{Eproj}(\sigma e) = \text{Eproj}(e) \cap \text{Eproj}(f) = \text{Eproj}(\gamma e). \quad \text{QED.}$$

Our definition of extended unification is an immediate and natural extension of standard unification under the projection model. In particular, the unification of two Eterms is equivalent to the unification of two standard terms except that every substitution for a variable, X, must satisfy all constraints on every member of the equivalence class containing X. Using this definition of extended unification, we can define the unification of two Eterms for any set E of expressions whose projections are sets of strings or numeric values. In an appendix, we develop a skeletal algorithm for computing the most general unifier for an arbitrary set of valid constraint specifications and a complete set of algorithms for unifying two interesting types of constraints.

6. Extended Unification and Predicate Locking

As mentioned above, the extensions to terms in the preceding chapter allow a complete modeling of Eswaran locks. This chapter illustrates how the principle functions of a lock manager can be implemented using extended unification.

Let R define a set of valid constraint specifications as follows:

(i) $a \in R$ if a is a constant
(ii) not $a \in R$ if a is a constant
(iii) [a-X] and [X-a] $\in$ R if a is a constant and X is a variable
(iv) [a-b] $\in$ R if a and b are constants.

Furthermore, define the operations match(c,e) for the constant c and $e \in R$, and intersect(e,f) for $e,f \in R$ as follows:

match(c,e):

If e is of the form	Then match(c,e) is equivalent to
a constant	c=e
not a	$c \neq a$
[a-X]	$c \geq a$
[X-a]	$c \leq a$
[a-b]	$a \leq c$ and $c \leq b$

intersect(e,f):

If e is of the form	If f is of the form	Then intersect(e,f) is equivalent to
a constant	-	match(e,f)
-	a constant	match(f,e)
not a	-	not match(a,f)
-	not a	not match(a,e)
[a-b]	[c-d]	"is there a constant, g, such that $a \leq g \leq b$ and $c \leq g \leq d$?"

The comparisons $a \leq X$ and $X \leq a$ are always true for constant a and variable X.

Match and intersect are clearly both decidable. An appendix outlines algorithms for match and intersect.

An *Rterm* is defined by the pair (t,*C*), where t is a standard term and *C* is a set of R constrained variables. For example, the following are Rterms:

(i) (balances(Accounts, b1, Balances), {})
(ii) (balances(Accounts, b1, Balances), {Balances : 1,000,000 - Infinity})
(iii) (balances(Name, b1, Balances), { Name: Anything - 'Doe'})

The extended projection of an Rterm, t, is intended to represent a set of possible tuples. Each ground term within the projection maps onto a possible tuple within a relation; the initial function symbol of t gives the name of the table or relation referred to by t; the arguments of a given ground term in proj(t) indicate the field values of the tuple being represented. So, the Rterms listed above represent respectively: (i) All records in the balances table having BranchName = b1. (ii) All records in the balances table having BranchName = b1 and Balances greater than 1,000,000 (note that by specifying an upper bound that is a unique variable, we are specifying an open ended upper bound). (iii) All records in the balances table having BranchName = b1 and AccntName = any string of characters lexicographically less than Doe.

Since nested records cannot be stored in most databases, the above definition of Rterms can be mapped into tuples that cannot exist. Allowing nesting of terms where such nesting cannot be mapped onto a single tuple allows meaningless predicate locks. However, it is easy to redefine standard terms so that no nesting is allowed. The complexity of our solution is not increased by allowing nested terms, and such nesting may be necessary for stating predicate locks for general knowledge base systems. The following implementation of predicate locks includes nesting. The implementation is also valid for non-nested locks.

An 'Rlock' is stated as an Rterm and has the following interpretation: the function symbol of an Rlock specifies the table or relation to which the lock refers. The arguments of the Rlock indicate the field value constraints specified by the lock. A variable argument in an Rlock indicates that the lock holds for all possible values in that field. For example, the Rlock

assets(b1, TotalAssets)

indicates a lock on tuples within the assets relation having first field value equal to b1 (e.g., BranchName = b1). A constrained variable argument in an Rlock indicates that the lock holds for all possible values that satisfy the constraint. For example, the Rlock

balances(AccntName, BranchName, Balance : 1,000,000 - Infinity)

specifies a lock on all tuples in the balances relation having Balance fields value greater than 1,000,000. More formally, the set of tuples specified by an Rlock, t, is the set of tuples specified by Eproj(t).

A predicate P over a relation R expressed as an Rlock, L, is considered satisfiable by the tuple t (e.g., P(t) is true) if the Rterm representing t is unifiable with L.

To see that this is a reasonable representation of predicate locks and tuples, consider the Rlock

L = balances(Accnt, b1, Balance)

for the balances relation defined above. The only tuples whose Rterm representation unifies with L are those having BranchName = b1, e.g., balances(c1,b1,x1) and balances(c4,b1,x4). Intuitively, these are the tuples that we expect to be locked. (Recall our notation that an argument in a term beginning with a capital letter is a variable, every other argument is a constant specification.)

Predicate locks of this form can model all of the locks in the Eswaran et al. model (though not necessarily as a single lock as mentioned above). In fact, they can specify some locks that are slightly more general, namely they can specify relationships between fields within a tuple. For example, given a relation of the form

employee

Name	Salary	Bonus
J.Doe	1000	500
J.Smith	500	1000

we can specify a lock on all tuples in the employee relation having bonus field greater than or equal to salary field by

employee(Name, Salary, Bonus : Salary - Infinity).

(Hunt and Rosenkrantz also present a lock specification language that allows comparisons of field values within a tuple.)

A lock manager for Rlocks needs to provide the two functions (i) testing two Rlocks for overlap (i.e., conflict) and (ii) testing if an access to a particular tuple in a relation by a transaction is allowable (i.e., testing that the transaction holds a lock that the access satisfies). Both functions are precisely the

problem of unification. That is, the statement that two predicate locks P and P′ conflict if and only if there exists a tuple t such that $P(t) \wedge P'(t)$ is valid is equivalent to stating that two Rlocks conflict if and only if they are unifiable. Similarly the statement that a tuple t satisfies a lock P if and only if P(t) is valid is equivalent to stating that a tuple t satisfies a lock P if and only if the Rterm representing t unifies with P.

As mentioned in a previous chapter, the Eswaran et al. algorithms are NP-complete for arbitrary locks. The Hunt and Rosenkrantz algorithms are polynomial for locks on tables of fixed length with the following cost:

$$O(\ (1/(k+1))\ (3(k+1)/k)^k\ n^{k+1})$$

where,

k = # of fields in the table
n = # of fields actually specified in the locks

An appendix outlines an algorithm that solves the unification of Rterms for most useful predicate locks with a worst case cost of

$$O(kr \log r)$$

where,

k = # of distinct variables in the locks
r = # of constraints in the locks.

Here, r maps roughly into the number of fields specified by the locks. Again, there are some predicate locks that cannot be modeled efficiently by Rterms, namely those that must be represented in disjunctive normal form. But there are fewer of these locks using extended terms then there are using Eswaran predicates. In fact, the only locks that must be represented in disjunctive normal form are those that require disjuncts on two or more distinct field values.

7. Summary

Extended unification provides a natural and efficient model for many useful predicate locks, including all of the locks suggested by Eswaran et al., and additional locks that allow comparisons between field values. Only a small class of these locks cannot be handled efficiently. The framework developed here allows further expansion of the types of constraints allowable in a predicate lock, limited only by the complexity of testing the constraints for nonempty intersection.

Appendix I: Generic algorithm for extended unification.

Extended unification is defined for an arbitrary set, E, of valid constraint specifications, as long as there are procedures for determining whether a constant matches an expression in E and whether two expressions in E have a nonempty intersection, and as long as E is closed under application of substitutions. That is E must satisfy the following properties:

(i) If A is the set of all possible constants, then there is a procedure M:AxE→{true,false} such that for all a ∈ A, for all *E* ∈ E, M(a,*E*) = true if a "matches" *E*, otherwise M(a,*E*) = false.
(ii) There is a procedure I:ExE→{true,false} such that for all *E*,*F* ∈ E, I(*E*,*F*) = true if there is an a ∈ A such that M(a,*E*) = M(a,*F*) = true, otherwise I(*E*,*F*) = false.
(iii) If *E* ∈ E then, for all substitutions, (V→t), (V→t)*E* ∈ E.

A previous theorem demonstrated that for any two Eterms e=(t,*C*) and f=(u,*D*) if μ is an mgu of t and u and μ(*C* ∪ *D*) is satisfiable, then (μ, μ(*C* ∪ *D*)) is an mgu of e and f, so an algorithm that performs standard unification followed by a satisfiability check for μ(*C* ∪ *D*) would be sufficient for extended unification. An alternative approach to implementing extended unification is to try to merge the satisfiability check into the standard unification by verifying each substitution as it is selected. But this approach can lead to complexity and inefficiency if constraints are allowed to contain embedded variables. For example, to unify the extended terms

(f(Z,Y),{Z:(1-Y)}) and (f(7,5),{})

one of Y or Z must be eliminated first. If Y is eliminated first, then unification must do the following:

Eliminate Y: mgu ← mgu ∪ (Y→ 5).
Apply (Y→5) to all constraints.
Verify any affected constraints, i.e., { Z:(1-5) }.
Eliminate Z: mgu ← mgu ∪ (Z→ 7).
Apply (Z→7) to all constraints.
Verify any affected constraints, i.e., { 7:(1-5) }.

So the constraint on Z is verified twice. Alternatively, if Z is eliminated first, then unification must do the following:

Eliminate Z: mgu ← mgu ∪ (Z→ 7).
Apply (Z→7) to all constraints.
Verify any affected constraints, i.e., { 7:(1-Y) }.
Eliminate Y: mgu ← mgu ∪ (Y→ 5).
Apply (Y→5) to all constraints.
Verify any affected constraints, i.e., { 7:(1-5) }.

In other words, each time a variable is eliminated, each constraint that contains that variable must be verified. Since constraints can contain disjunctions and conjunctions, this can be very costly.

For any set E, the following algorithm will yield a most general unifier of any two Eterms. The algorithm assumes the existence of a standard unification routine, UNIFY, that solves the unification problem for standard terms by supplying the mgu if the input terms are unifiable or indicating that they are not unifiable. The mgu produced must supply complete equivalence classes for variables. That is the mgu must be comprised of substitutions of the form $(\{X_1,\ldots,X_n\} \rightarrow t)$, where the $(X_i \cup t)$ denote the complete equivalence class containing the Xs and t. (Note: each of the algorithms described in this paper conform to this requirement or can be easily made to conform.)

Extended Unification

```
EUnify((t, C), (u, D))
Begin
        If UNIFY( t, u, μ )
        Then
                If Satisfy(C ∪ D, μ, ResultingConstraints)
                        Then print (μ, ResultingConstraints)
                        Else print "Not unifiable"
                        Else    print "Not unifiable"
        End If
End of EUnify
```

The Satisfy procedure must return NO if $\mu(C \cup D)$ is not satisfiable, otherwise it must set ResultingConstraints to $\mu(C \cup D)$ and return YES. The following algorithm will satisfy this requirement for an arbitrary set of valid constraints, E, given the following functions for E.

(i) A function Ematch(Symbol, Constraints) that takes a constant, Symbol, and a set of constraints, Constraints = $\{C_1, \ldots C_n\}$, $C_i \in$ E, and returns YES if Symbol satisfies each of the C_i, otherwise returns NO.

(ii) A function Econflict(Constraints) that takes a set of constraints, Constraints = $\{C_1, ..., C_n\}$, $C_i \in$ E, and returns YES if $\{C_1, ..., C_n\}$ is not satisfiable, otherwise it returns NO.

(iii) A function Eapply(Substitution, Constraint) that applies Substitution to any embedded variables in the constraints contained in Constraint (note: Eapply may have no affect if E doesn't allow embedded variables).

The algorithm also relies on the following function for extracting individual constraints from constrained variable sets.

SetOfCons(EConstraints, VariableList) returns the set of constraints in EConstraints that pertain to the variables in VariableList (e.g. SetOfCons({X:C1, Y:C2}, {X}) is {C1}).

```
Satisfy( C, μ, ResultingConstraint)
Begin
        ResultingConstraint ← ∅
                /* eliminate embedded variables in C where possible */
        For each substitution S=({Y1,...,Yn}→t) ∈ μ
                For each Xi:ci ∈ C
                        Eapply( S, ci)
                End of For
        End of For
                /* verify that each substitutions satisfies C */
        For each substitution S = ({X1, ..., Xn} → t) ∈ μ
                If t is a constant symbol
                Then
                        If not Ematch( t, SetOfCons( C, {X1, ..., Xn} ))
                                Then return NO
                        End If
                Else If t is a variable Then
                        If Econflict( SetOfCons( C, {X1, ..., Xn, t}))
                                Then return NO
                        Else add (t : SetOfCons( C, {X1,...Xn, t} ) ) to ResultingConstraint
                        End If
                /* Otherwise, t is of the form f(t1, ..., tn),
                        but we don't allow non constant constraints        */
                Else If SetOfCons( C, {X1, ..., Xn} ) is nonempty
                                Then return NO
        End of For
        return YES
End of Satisfy
```

Claim: Satisfy returns YES if only if $\mu(C \cup D)$ is satisfiable.

Proof: In each of the unification algorithms that we have reviewed, each distinct variable in the initial input terms appears alone, either on the left or right hand side of exactly one substitution in the mgu. A

variable may appear as part of a complex term, as in f(X), on the right hand side of more than one substitution. We will require that the Unify algorithm selected above conform to this format. So each substitution in the μ is of the form ($\{X_1, ..., X_n\}$ -> t), where no X_i on the left hand side appears in any other substitution and if t is a variable, it does not appear alone in any other substitution.

To see that Satisfy fails if μ produces a substitution that conflicts with C, note that each of the explicit substitutions in μ (i.e., (V→w) ∈ μ) satisfy C, since these are explicitly verified by Satisfy. We need to show that there are no implicit substitutions that violate C. An implicit substitution might occur if a variable X inherits the substitutions of some other variable to which X has been unified. In fact, there are no additional substitutions implied by μ since the only substitutions possibly implied by each

$$(\{X_1, ..., X_n\} \rightarrow t)$$

are additional substitutions inherited by the X_i's. The set of such variables is exactly the union of the sets of variables appearing in any substitution with any one of the X_i's. Since no X_i can appear in any other substitution in μ, the set of variables to which each X_i has been unified is exactly

$$\{X_1, ..., X_n\} \cup \{t\}.$$

If t is a variable and it appears in another substitution in μ, then it must appear on the right hand side and it must not appear alone (i.e. it must be part of a more complex term, such as f(t)).

To show that Satisfy succeeds if μ satisfies the constraints in C, we need to show that Satisfy succeeds if for each substitution (V->t) in μ, where V=$\{X_1, ..., X_n\}$, all of the constraints on the X_i's and all of the constraints on t (if t is a variable) are simultaneously satisfiable by t. To see that this is true, note that for each substitution V->t in μ, there are only three possible forms that t can take: constant symbol, variable or n-ary term, n ≥ 1. If t is a constant symbol, then every Econstraint on each of the X_i's in V (=SetOfCons(C,V)) is satisfied by t if and only if Ematch(t, SetOfCons(C,V)) by definition of Ematch. If t is an n-ary term, then it is not possible for t to satisfy any constant symbol constraints, so the variables in V must be unconstrained which is true if and only if SetOfCons(C,V) is empty, by definition of SetOfCons. Finally, if t is a variable, then the set of the constraints implied by the substitu-

tion

$V \to t$ (= SetOfCons(C, $V \cup t$))

are mutually satisfiable if and only if there are no conflicts among the constraints which is true if and only if Econflict(SetOfCons(C, $V \cup t$)) fails. QED.

The cost of EUnify is equal to the cost of the Unification algorithm chosen plus the cost of Satisfy. If j = the number of constraints and k = the number of distinct variables in the input terms, then the worst case cost of Satisfy is

```
O(j * k * cost(Eapply)
        + k * max( cost(Ematch for j constraints and k vars),
                cost(Econflict for j constraints and k vars))
        + k * cost(SetOfCons) ).
```

Since Eapply is invoked j * k times, and either Ematch or Econflict is invoked once for each substitution. A representation of constraints that allows efficient retrieval of constraints on a particular set of variables is possible, so k * cost(SetOfCons) is negligible. Of the remaining cost, the routine Econflict is probably the most costly operation, since it involves a test for non-empty intersection of a number of expressions in E. So, the cost of finding an mgu will probably most often be determined by the cost of Econflict for the particular set of constraints in question.

The following sections provide Ematch, Eapply and Econflict for a set of expressions suitable for solving the predicate locking problem.

Appendix II: Constraints for Predicate Locking

The complete set of predicate locks suggested by Eswaran, et al., can be modeled given the following set, R, of constraints.

(i) If c is a number or function symbol, then $c \in R$.
(ii) If X is a variable, then $X \in R$ and matches any value.
(iii) If a and $b \in R$, then the range $a\text{-}b \in R$.
(iv) If $a \in R$, then 'not a' $\in R$.

The atomic predicate '=' is completely modeled by (i); predicates '≤' and '≥' are modeled by a combination of (ii) and (iii); predicates '≠', '<' and '>' can be modeled by a combination of (i) - (iii) and (iv).

The functions Rapply and Rmatch are trivial: Rapply((V→t), R) simply replaces all occurrences of V in R by t. Rmatch has the following behavior:

Rmatch(a,i) = true,	if i is a constant and c=i.
Rmatch(a,X) = true,	if X is a variable.
Rmatch(a,b-c) = true,	if a,b and c are all numbers or all function symbols and $b \le a \le c$.
Rmatch(a, not c) = true,	if Rmatch(a, c) = false

Recall that Satisfy calls Econflict with the conjunction of all of the constraints on some set of variables. Each of these individual constraints is itself a disjunction of constraints. All constraints can be rewritten as a single range or disjunct of ranges:

a	can be rewritten	a-a
X	can be rewritten	X-Y, (where Y is a variable that doesn't occur elsewhere)
not a	can be rewritten	X-(a-1) or (a+1)-X

Since numbers and strings are ordered, a-1 and a+1 are well defined. So the function Rconflict reduces to a problem of determining if m sets have a nonempty intersection, where each set is a disjunct of some number of ranges. If any single range is unsatisfiable (i.e., upperbound < lowerbound) and it appears alone in a set, then that set is unsatisfiable and we are done. If the unsatisfiable range appears in a disjunction, then we can discard it, since the satisfiability of the disjunct will depend on the remaining ranges.

The test for nonempty intersection can be solved using a simple sorting algorithm in the following way: Let r denote the total number of ranges. Number the ranges, in any order, from 1 through r. Let

$l_{i,j}$ and $u_{i,j}$, $1 \le i \le m$, $1 \le j \le r$.

denote the lower and upper bounds of the jth range, which happens to be in set i. Sort the $l_{i,j}$ and $u_{i,j}$, in increasing order, into a single list, L, retaining position information about each value (i.e., whether it is a lower or upper bound and the set and range number it refers to). For the purposes of ordering, if a lower bound and an upper bound of a single range have the same value, then consider the lower bound to be smaller, that is if $l_{i,j} = u_{k,l}$ for any i,j,k,l, then consider $l_{i,j}$ to be less than $u_{k,l}$. Scanning L from left to right, let the range r_j be considered "unencountered" if its lower bound has not been

encountered in the scan, "open" if its lower bound has been encountered but its upper bound has not, and "closed" if its upper bound has been found. At the beginning of the scan, all ranges are unencountered. If, during the scan, there are m open ranges belonging to m distinct sets (that is, there is at least one open range in each of the m sets) then the intersection is nonempty, since it at least contains the largest lower bound from the open ranges. To keep track of open ranges, maintain an array, M such that |M| = m, and M[i] = j if there are j open ranges in set i. Also maintain a counter, C = the number of M[i]'s equal to 0; initially C = m. If at any point, C = 0 (i.e., each of the M[i]'s is >0), then the intersection is nonempty. To maintain the array: Set each M[i] = 0; increment and decrement M[i] as lower and upper bounds from set i are found. (Recall, for all ranges, i,j, $l_{i,j} \le u_{i,j}$, by assumption.) When a lower bound, $l_{i,j}$, is encountered in the sorted list, range j must be unencountered, so we can safely increment M[i] by 1. When an upper bound, $u_{i,j}$, is encountered, range j must be open, so we can safely decrement M[i] by 1.

The sorting of the r ranges,can be accomplished in time O(r log r). The scanning and maintenance of the M array requires time O(r). So, the entire test for intersection requires time O(r log r), and the cost of Satisfy for these extended terms is upper bounded by kr log r, where k is the number of distinct variables and r is the total number of constraints.

Useful extensions for Predicate Locks

The predicate locks possible with extended terms are limited only by the complexity of the test for non-empty intersection. The initial target of this paper was to allow constraints of numeric ranges and arbitrary *regular expressions.*

However, since we can define a direct mapping between a regular expression and a non deterministic finite state automaton, solving the test for nonempty intersection of some number, n, of regular expressions is equivalent to solving the test for nonempty intersection of n ndfsa's. In 1977, Kozen [Koz] showed that the test for nonempty intersection of n deterministic fsa's is PSPACE-complete through a reduction of the Linear Space Acceptance problem. If the test is PSPACE-complete for dfsa's then it is at least PSPACE hard for ndfsa's and regular expressions.

As a direction for future research, it would be interesting to classify the subclasses of regular expressions with respect to the computational difficulty of the test for nonempty intersection.

8. References

[Bax] Baxter, L.D., A practically linear unification algorithm, Research Report CS-76-13, Department of Computer Science, University of Waterloo (1976).

[Bun] Bundy, A., "The Computer Modeling of Mathematical Reasoning", Academic Press, 237-255 (1983).

[Cod] Codd, E.F., A relational model for large shared databanks, CACM, 14:6, 377-387 (1970).

[Esw] Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L., The notions of consistency and predicate locks in a database system, CACM, 19:11, 624-633, (1976).

[Hue] Huet, G., Resolution d'equations dans les langages d'order 1,2,...,w. These d'etat, Specialite Mathematiques, Universite Paris VII, (1976).

[Hun] Hunt, H.B. and Rosenkrantz, D.J., The complexity of testing predicate locks, Proc. 1979 ACM SIGMOD, 127-133 (1979).

[Kle] Kleene, S.C. "Introduction to Metamathematics", Van Nostrand, Princeton, N.J., p.204 (1952).

[Koz] Kozen, D., Lower bounds for natural proof systems, Proc 18th Ann. Symp. on Foundations of Computer Science, IEEE Computer Society, Long Beach, 254-266 (1977).

[Knu] Knuth, D. and Bendix, P., Simple word problems in universal algebras, in "Computational Problems in Abstract Algebra", Pergamon, Oxford, 263-297 (1970).

[Mar] Martelli, A. and Montanari, U., An efficient unification algorithm, ACM Transactions on Programming Languages and Systems, 4:2 258-282 (1982).

[Mur] Murray, N.V., Comparing linear and almost-linear methods for unification, Technical Report, School of Computer and Information Science, Syracuse University (1979).

[Pat] Paterson, M.S. and Wegman, M., Linear unification, J. Computer and System Sciences, 16, 158-167 (1978).

[Rob] Robinson, J.A., A machine-oriented logic based on the resolution principle, JACM 12:1, 23-41 (1965).

[Rob2] Robinson, J.A., Fast unification, Theorem Proving Workshop. Oberwolfach, W. Germany, Jan 1976.

[Rob3] Robinson, J.A., "Logic: Form and Function", North-Holland, Artificial Intelligence Series, 182-198 (1979).

[Ven] Venturini Zilli,M., Complexity of the unification algorithm for first- order expressions. Calcolo, 12:4, 361-372 (1975).

[War] Warren, D.H.D., Pereira, L.M., and Pereira, F., PROLOG -- The language and its implementation compared with LISP. Proc. of Symposium on Artificial Intelligence and Programming Languages, Univ. of Rochester, Aug 15-17 (1977).