CORRECTNESS OF UNIFICATION WITHOUT OCCUR CHECK IN PROLOG

RITU CHADHA AND DAVID A. PLAISTED

▷ For efficiency reasons, most Prolog implementations do not include an occur check in their unification algorithms and thus do not conform to the semantic model of first-order logic. We present a simple test that guarantees that unification without occur check is sound in programs satisfying the conditions of the test. We designate each argument position of every predicate as either an input or an output position and then describe a sufficient condition in terms of this designation for unification without occur check to be sound. Unification with occur check can be performed in places in the program where this condition is not satisfied. Two algorithms for implementing this test are described and compared. ◁

1. INTRODUCTION

In the recent past, Prolog has emerged as the most popular logic programming language. For efficiency reasons, however, most implementations of Prolog do not conform to the semantic model of first-order logic. This is illustrated by the fact that Prolog uses unification [10] without occur check (i.e., when unifying a variable X with a term t, t is not checked for occurrences of X). It has been argued that in most of the programs encountered in practice, unification without occur check is sound. However, as shown in [8], it is possible to derive meaningless results such as $3 < 2$ from formally correct Prolog programs when unification without occur check is used. Thus the results of a computation may not be compatible with the declarative semantics of the given logic program. Performing the occur check during unification requires time linear in the size of the literals being unified. However, because in practice most unifications do not require the occur check, an attractive approach is to look for sufficient conditions that guarantee that unifica-

Address correspondence to Ritu Chadha, Bell Communications Research, MRE 2A-246, 445 South Street, Morristown, NJ 07962-1910.
Received September 1990; revised February 1993.

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

tion without occur check is sound for given clauses. Then only those clauses that do not satisfy these sufficient conditions would be subject to occur check.

Several ways of tackling the occur check problem have been suggested in the past. Plaisted [8] developed a family of compile time tests for looping, i.e., for detecting infinite terms caused by unification of a variable with a term in which it occurs. This method consists of iterating to obtain a representation of the set of instances of each clause that can be generated by any execution of a Prolog program, in the process detecting loops that may occur. A bipartite graph is created from this set of clauses that represents the calling literal–called literal pairs for which loops may be created, and appropriate tests are inserted in the Prolog program. The algorithm is rather complicated and no analysis of the time complexity of the method is given. Sondergaard [11] draws on Plaisted's method, but his approach differs in that the abstract interpretation principle is applied to logic programs. A logic program is interpreted as a computation in the universe of substitutions that computes all the possible substitutions that may occur at different points in the program. Again no analysis of the algorithm's time complexity is given, although it is conjectured to be exponential in the largest number of variables in a clause of a program. The method is simpler than Plaisted's owing to lack of generality. Beer [2] suggested a method for dealing with the occur check problem that could be incorporated into a Prolog compiler. His method depends on a differentiation of the context in which variables occur. Prolog implementations use data tags or descriptors to identify the objects to be unified, and unbound variables are usually tagged just as "unbound variable." In this scheme, further tags are used to identify unbound variables and the context in which they occur, differentiating between variables that occur once or more than once in a given goal, and this information is used to avoid unnecessary occur checks. Deransart and Maluszynski [6] develop a relationship between attribute grammars and logic programs and then apply the methods of attribute grammars to develop and study properties of logic programs. They show that a logic program can be associated with a relational attribute grammar and its data dependencies can be modeled by attribute dependency schemata, and they prove that if the attribute dependency schemata associated with the program has certain properties, then unification without occur check is sound. However, the emphasis of their study is to illustrate the potential usefulness of the relationship between logic programs and attribute grammars rather than to develop specific practical applications. Colmerauer [3] has described a semantics of Prolog using infinite trees. He considers terms with loops as infinite trees and allows them to be used. However, this presents a problem because it deviates from the first-order semantics of Prolog.

Others have developed methods for inferring predicate modes in logic programs for different purposes. Reddy [9] presents a framework for transforming logic programs into functional programs based on the predicate modes "input" and "output." Given a mode declaration for the top-level goal, an inference-transformation system infers the modes of the other predicates in the program with the objectives of avoiding indefinite modes, preferring determinate modes to indeterminate modes, and according to several other criteria. Such a mode assignment is obtained by exhaustive search. Debray [4] describes an algorithm based on flow analysis for the static inference of modes in static programs. A set of five modes is used. This information can be useful for the generation of efficient code. Debray and Warren [5] use a set of three modes for program predicates and use

the mode information about the predicates in a logic program to determine if the predicates, clauses and literals of the program are functional or not. This knowledge can then be used to optimize the program. Mellish [7] uses information provided by the user about the direction in which programs will be used to try and infer mode assignments for a program. The mode assignments thus obtained can be used for optimizing a Prolog compiler in several ways.

In this paper we describe a simple test that guarantees that unification without occur check is equivalent to unification with occur check in Prolog programs. Briefly, the method designates each argument position of every predicate as either an input position or an output position. After such a designation is found, we show that a sufficient condition for the soundness of unification without occur check is that all heads of clauses have all input positions collinear (see Section 2 for the definition of collinearity). We then describe two ways of implementing this test in Section 3. The methods are shown to be sound, and are compared in Section 3.3. Unification with occur check can be performed in places in the program where this test is not satisfied. In Section 4, we present some of the results obtained by running the first algorithm of this paper on some benchmark Prolog programs, and compare our method with other methods in the literature.

It is assumed that standard SLD resolution is used in this paper. The given results hold for any unification algorithm.

2. THEORETICAL BACKGROUND

We give in the following text some definitions and some theorems on which the test described in this paper is based.

Definition. We say a term or literal is *linear* if it has no repeated variables. Say a collection of terms or literals is *collinear* if they have, between them, no repeated variables.

Definition. If a predicate p has n arguments a_1 through a_n in that order, then we say that a_i is in position i in p.

Theorem 2.1. Suppose we unify terms s and t, and s is linear and s and t have no common variables. Then unification without occur check is equivalent to unification with occur check.

PROOF. The proof is by induction on the number of steps required to unify the terms. We know that any two terms that are unifiable can be unified in a finite number of steps. If one of the terms is a variable, say we are unifying X and t, then X does not occur in t by hypothesis, so only one step is required to unify the terms and the unifier is just $\{X \leftarrow t\}$. The inductive hypothesis is that when unifying two terms, if one term is linear and has no variables in common with the other, and if k or fewer steps are needed for the unification, then the occur check is not needed. Now suppose that we are unifying two terms that require $k + 1$ steps to be unified. If one of the terms is a variable, say we are unifying X and t, then as before, the unifier is just $\{X \leftarrow t\}$. If neither term is a variable, then the unification fails unless their principal functors are the same. Suppose we are unifying $f(s_1, s_2, \ldots, s_n)$ with $f(t_1, t_2, \ldots, t_n)$, where the first term is linear and the two terms

have no common variables. Then s_i must be unified with t_i for all i such that $1 \le i \le n$. Suppose s_j, t_j are unified first. Let α be the most general unifier of s_j and t_j. We then must unify the lists $(s_1\alpha, \ldots, s_{j-1}\alpha, s_{j+1}\alpha, \ldots, s_n\alpha)$ and $(t_1\alpha, \ldots, t_{j-1}\alpha, t_{j+1}\alpha, \ldots, t_n\alpha)$. Now, none of the variables in $s_1, \ldots, s_{j-1}$, $s_{j+1}, \ldots, s_n$ are changed by α, because none of these variables appear in s_j, and therefore $s_i = s_i\alpha$, for all i such that $1 \le i \le n$, $i \ne j$. The substitution α may result in $t_i\alpha$ containing some variable that appeared in s_j; however, because none of the s_is have any variable in common with s_j, for $1 \le i \le n$, $i \ne j$, the two lists $(s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n)$ and $(t_1\alpha, \ldots, t_{j-1}\alpha, t_{j+1}\alpha, \ldots, t_n\alpha)$ again have no variables in common, and the first of these lists is still collinear. We are, therefore, essentially unifying the terms $f(s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n)$ and $f(t_1\alpha, \ldots, t_{j-1}\alpha, t_{j+1}\alpha, \ldots, t_n\alpha)$, which requires less than $k+1$ steps, so by induction we can assume that the occur check is not needed for this and the proof is complete. □

We identify positions of a predicate as input and output positions as follows.

Definition. The kth argument of p is an *output position* only if, in each clause

$L: -L_1, \ldots, L_n.$,

if the predicate of L_i is p then all variables in the kth argument of L_i do not appear in L_j for $j < i$, and do not appear elsewhere in L_i, and if they appear in L, they only appear in output positions of L. Because these variables can appear only once in L_i, the output positions of P in L_i must be collinear. Also, in a query

$? - L_1, \ldots, L_n.$,

similar restrictions apply, except that there is no L. We call positions that are not output positions *input positions*. Note that variables in output positions of L_j can be in input positions of L_i for $i > j$. Also, output positions of clause heads need not be collinear. There is no restriction on output positions of facts L.

The preceding definition gives a necessary condition for a position to be an output position. Thus no positions that do not satisfy this condition can be labelled as output positions. However, if a position does satisfy this condition, it can be labelled either as an input position or as an output position.

Definition. An *input/output position combination* is a designation of all the argument positions of the predicates of a program as input or output positions that satisfies the foregoing constraints.

Note in particular that a predicate may have several permissible input/output position combinations.

Example 2.1. Consider the following Prolog program for detecting palindromes:

```
1. palindrome (L):- reverse(L, L).
2. reverse(L1, L2):- reverse(L1, [ ], L2).
3. reverse([ ], L, L).
4. reverse([H|L1], L2, L3):- reverse(L1, [H|L2], L3).
5. ? - palindrome([m, a, d, a, m]).
```

We shall illustrate the definition of input and output positions by computing all possible input/output position combinations for this example. There are three predicates in this program, namely, *palindrome*, *reverse*/2, and *reverse*/3. First we

examine the clause bodies. In the first clause body, the *reverse*/2 predicate has the same variable L occurring in both argument positions; therefore, both the positions of *reverse*/2 must be marked as input positions. No other restrictions on the marking of positions can be obtained by examining clause bodies.

The next step is to check whether the positions of *palindrome* and *reverse*/3 can be marked as output positions. For the predicate *reverse*/3, we see that in clause 2, the variables L1 and L2 occur in the first and third positions of *reverse*/3, respectively, in the clause body, and these variables occur in the clause head in input positions of *reverse*/2. This means that the first and third positions of *reverse*/3 must be marked as input positions. Also in clause 4, the variable H occurs in the second position of the predicate *reverse*/3 in the clause body, and it also occurs in the first position of the predicate *reverse*/3 in the clause head, which is an input position of *reverse*/3. Therefore, the second position of *reverse*/3 must also be marked as an input position. The position of predicate *palindrome* can be marked either as an output position or as an input position. Thus we have the following two possible input/output position combinations for this program:

palindrome(output)
reverse/2(input, input)
reverse/3(input, input, input)

or

palindrome(input)
reverse/2(input, input)
reverse/3(input, input, input).

Lemma. Suppose all positions in clauses of a program have been marked as input or output positions. Then during any unification between a clause head

$$p(u_1, u_2, \ldots, u_m, v_1, v_2, \ldots, v_n)$$

and a subgoal

$$p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n),$$

where the first m positions of p are assumed to be input positions and the last n positions of p are assumed to be output positions, $t_1, t_2, \ldots, t_n$ *are collinear and share no variables with* $s_1, s_2, \ldots, s_m$. *(In other words, the collinearity of output positions in clause bodies and the fact that none of the variables of* $t_1, t_2, \ldots, t_n$ *appear in* $s_1, s_2, \ldots, s_m$ *are invariant properties throughout the resolution process.)*

PROOF. The proof is by induction on the number of unifications performed so far between clause heads and subgoals during the resolution process. When the first unification is performed, the unification is between a subgoal that has never been instantiated (namely, the goal or query of the program) and a clause head. Therefore, all output positions of the subgoal are collinear and do not share variables with any of the input positions, by definition.

Now assume that for all unifications performed so far, the hypothesis stated in the lemma holds, and suppose we want to unify a clause head $p(u_1, u_2, \ldots, u_m,$

$v_1, v_2, \ldots, v_n$) and a subgoal $p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n)$. We need to prove that (i) the set $\{t_1, t_2, \ldots, t_n\}$ is collinear and (ii) the set $\{t_1, t_2, \ldots, t_n\}$ shares no variables with any of $s_1, s_2, \ldots, s_m$.

We prove (i) and (ii) in the following text.

(i) Suppose $\{t_1, t_2, \ldots, t_n\}$ is not collinear. This can only happen if $p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n)$ is some instance of a clause from a clause body, instantiated by previous unifications involving heads of clauses and subgoals. Then there must exist a variable, say Z, that appears at least twice in $\{t_1, t_2, \ldots, t_n\}$.

Now, because $p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n)$ is an instance of a clause from a clause body, there exists some substitution σ such that

$$p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)\sigma = p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n),$$

where $p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)$ is the original, uninstantiated clause from a clause body. By definition, $\{t'_1, t'_2, \ldots, t'_n\}$ is a collinear set. Now, σ is a substitution consisting of one or more substitutions for the original clause, performed during previous unifications. Because $\{t'_1, t'_2, \ldots, t'_n\}$ is collinear, therefore σ assigned Z to two distinct terms (or subterms) in this set; hence, obviously these two distinct terms (or subterms) were distinct variables, say X and Y. Therefore,

$$\{X \leftarrow Z, Y \leftarrow Z\} \subseteq \sigma.$$

Now, X and Y must have been instantiated (to Z) during the unification of the head H of the clause whose body contains $p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)$ with another subgoal. The reason for this is that variables X and Y cannot appear in any clause preceding $p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)$ in the clause body (by definition of output position); hence the only time these variables could be instantiated is during a previous unification of the clause head H and another subgoal, say S. Also, X and Y only appear in output positions of H, by definition (because they appear in output positions of $p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)$).

Now the substitutions $X \leftarrow Z$ and $Y \leftarrow Z$ were performed during the unification of S and H; therefore, Z must be a variable appearing twice in output positions of S, which contradicts the induction hypothesis (by the induction hypothesis, when the unification of H and S was performed, all output positions of S were collinear). Therefore, our assumption that $\{t_1, t_2, \ldots, t_n\}$ was not collinear must be wrong.

(ii) The proof follows similar lines to that of (i). Suppose $\{t_1, t_2, \ldots, t_n\}$ does share variables with $s_1, s_2, \ldots, s_m$. This can only happen if $p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n)$ is some instance of a clause from a clause body, instantiated by previous unifications involving heads of clauses and subgoals. Then there must exist a variable, say Z, that appears in some term t_i of $\{t_1, t_2, \ldots, t_n\}$ and also appears in some term s_j of $\{s_1, s_2, \ldots, s_m\}$.

As before, because $p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n)$ is an instance of a clause from a clause body, there exists some substitution σ such that

$$p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)\sigma = p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n),$$

where $p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)$ is the original, uninstantiated clause from a clause body. By definition, $\{t'_1, t'_2, \ldots, t'_n\}$ and $\{s'_1, s'_2, \ldots, s'_m\}$ do not share any variables. Now, σ is a substitution consisting of one or more substitutions for the original clause, performed during previous unifications. Because $\{t'_1, t'_2, \ldots, t'_n\}$ and

$\{s'_1, s'_2, \ldots, s'_m\}$ do not share any variables, therefore σ assigned Z to two distinct terms in these sets (one from each set); hence, obviously these two distinct terms were distinct variables, say X (belonging to t'_i) and Y (belonging to s'_j). Therefore,

$$\{X \leftarrow Z, Y \leftarrow Z\} \subseteq \sigma.$$

Now, X and Y must have been instantiated (to Z) during the unification of the head H of the clause whose body contains $p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)$, with another subgoal. The reason for this is that the variable X cannot appear in any clause preceding $p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)$ in the clause body (by definition of output position); hence, the only time X and Y could be instantiated to the same variable Z is during a previous unification of the clause head H and another subgoal, say S. Also, X only appears in output positions of H, by definition (because it appears in an output position of $p(s'_1, s'_2, \ldots, s'_m, t'_1, t'_2, \ldots, t'_n)$); Y could occur either in an input or an output position of H. Now the substitutions $X \leftarrow Z$ and $Y \leftarrow Z$ were performed during the unification of S and H. If Y occurs in an input position of H, then Z must be a variable appearing both in an input and in an output position of S, which contradicts the induction hypothesis. If Y occurs in an output position of H, then Z must be a variable appearing twice in output positions of S, which again contradicts the induction hypothesis (by the induction hypothesis, when the unification of H and S was performed, none of the output positions of S shared any variables with the input positions of S and all output positions of S were collinear). Therefore, our assumption that $\{t_1, t_2, \ldots, t_n\}$ shares some variables with $s_1, s_2, \ldots, s_m$ must be wrong. □

Theorem 2.2. Suppose in a Prolog program, the head of a clause C has all input positions collinear. Then for C, unification without occur check is equivalent to unification with occur check.

PROOF. Consider a unification between the head of clause C and a subgoal. Say we have $p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n)$ as the subgoal and $p(u_1, u_2, \ldots, u_m, v_1, v_2, \ldots, v_n)$ as the clause head, where without loss of generality the last n positions are assumed to be output positions. Then $u_1, u_2, \ldots, u_m$ are collinear (because all input positions of the clause head are collinear) and $t_1, t_2, \ldots, t_n$ are collinear (by the lemma). The unifier is the same as if the two literals were $p(s_1, s_2, \ldots, s_m, v_1, v_2, \ldots, v_n)$ and $p(u_1, u_2, \ldots, u_m, t_1, t_2, \ldots, t_n)$. Now, the list of terms $s_1, s_2, \ldots, s_m$ has no variables in common with the list $t_1, t_2, \ldots, t_n$ (by the lemma) or the lists $u_1, u_2, \ldots, u_m$ and $v_1, v_2, \ldots, v_n$ (because when unifying a subgoal and a clause head, all variables in the clause head are assigned new variable names); similarly, the list of terms $t_1, t_2, \ldots, t_n$ has no variables in common with any of the lists $s_1, s_2, \ldots, s_m$, $u_1, u_2, \ldots, u_m$, and $v_1, v_2, \ldots, v_n$. The lists $u_1, u_2, \ldots, u_m$ and $v_1, v_2, \ldots, v_n$ are, therefore, the only lists of terms that could possibly have any variables in common.

We will perform the unification of $p(s_1, s_2, \ldots, s_m, v_1, v_2, \ldots, v_n)$ and $p(u_1, u_2, \ldots, u_m, t_1, t_2, \ldots, t_n)$ in two steps. The first step consists of unifying $p(s_1, s_2, \ldots, s_m)$ with $p(u_1, u_2, \ldots, u_m)$ and unifying $p(t_1, t_2, \ldots, t_n)$ with $p(v_1, v_2, \ldots, v_n)$. Because $\{u_1, u_2, \ldots, u_m\}$ is a collinear set of terms, $p(u_1, u_2, \ldots, u_m)$ is linear; and because the list $s_1, s_2, \ldots, s_m$ has no variables in common with $u_1, u_2, \ldots, u_m$, by Theorem 2.1 unification without occur check is sound. Similarly we can unify $p(t_1, t_2, \ldots, t_n)$

with $p(v_1, v_2, \ldots, v_n)$ without performing the occur check because $\{t_1, t_2, \ldots, t_n\}$ is a collinear set of terms and the two lists of terms have no variable in common.

The second step consists of merging these two unifications. If the lists $u_1, u_2, \ldots, u_m$ and $v_1, v_2, \ldots, v_n$ have no common variables, then the unification is complete. Therefore, suppose they do have some common variables. Suppose X is a variable occurring in u_i and v_j for some $1 \le i \le m$, $1 \le j \le n$, such that the two unifications performed bound X to a term T_1 containing variables from $\{s_1, s_2, \ldots, s_m, u_1, u_2, \ldots, u_m\}$ and to a term T_2 containing variables from $\{t_1, t_2, \ldots, t_n\}$, respectively (if there is no such X then we are done). Note that T_2 cannot contain variables from $\{v_1, v_2, \ldots, v_n\}$ because $\{t_1, t_2, \ldots, t_n\}$ is a collinear set of terms. Then we must unify T_1 and T_2. However, we remarked earlier that $\{t_1, t_2, \ldots, t_n\}$ shares variables with none of $\{s_1, s_2, \ldots, s_m\}, \{u_1, u_2, \ldots, u_m\}$; hence, by Theorem 2.1 the occur check is not needed when unifying T_1 and T_2. After this is done for every such variable X occurring in both $u_1, u_2, \ldots, u_m$ and $v_1, v_2, \ldots, v_n$, the unification of $p(s_1, s_2, \ldots, s_m, v_1, v_2, \ldots, v_n)$ and $p(u_1, u_2, \ldots, u_m, t_1, t_2, \ldots, t_n)$ is complete and no occur check was required for this unification. This means that no occur check is needed in the unification of $p(s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n)$ and $p(u_1, u_2, \ldots, u_m, v_1, v_2, \ldots, v_n)$ either, and the proof is complete. □

Corollary to Theorem 2.2. Suppose in a Prolog program, all heads of clauses (including facts) have all input positions collinear. Then for this program, unification without occur check is equivalent to unification with occur check.

PROOF. All unifications will be between subgoals and clause heads. Because all clause heads are assumed to have all input positions collinear, by Theorem 2.2 the result follows. □

Example 2.2. Consider the well-known "append" program:

1. append([ ], X, X).
2. append([U|X], Y, [U|Z]): – append(X, Y, Z).

There are eight possible input/output position combinations for the predicate "append," of which there are five for which all input positions in clause heads are collinear. The eight input/output position combinations are:

(1) append(output, output, output)
(2) append(output, output, input)
(3) append(output, input, output)
(4) append(input, output, output)
(5) append(input, input, output)
(6) append(output, input, input)
(7) append(input, output, input)
(8) append(input, input, input).

Of these, the last three do not satisfy the condition of Theorem 2.2, namely, that all heads of clauses have all input positions collinear. For the position assignment given in (6), the last two argument positions of "append" are input positions, and the first clause of the program has an "X" in both these positions in the clause head. For the position assignment in (7), where the first and third argument positions are input positions, we see that in the second clause of the program a

"U" occurs in both the first and third argument positions of the clause head and thus these input positions are not collinear. For the assignment given in (8), the collinearity condition for input positions in clause heads is violated by both the clauses of the program. Hence we have five input/output position combinations for this program for which all input positions in clause heads are collinear.

Example 2.3. Consider the following program taken from [8]:

1. ? – q(U, V).
2. q(X, Y): – ancestor(X, Y), ancestor(Y, X).
3. ancestor(father(X), X).
4. ancestor(mother(X), X).
5. ancestor(X, X).

The two predicates in this program are "q" and "ancestor." From clause 2, because the variables in the first and second positions of ancestor(Y, X) appear in the literal preceding it (viz. in ancestor(X, Y)), the first and second positions of the predicate "ancestor" are input positions. It can be seen that both the argument positions of the predicate "q" satisfy the requirements for being output positions. Hence we have the following possible input/output position combination for the predicates of this program:

ancestor(input, input)
q(output, output).

Now we check the heads of clauses for collinearity of input positions. In clauses 3, 4, and 5, the two input positions of the clause head, which is the predicate ancestor, are not collinear. Thus the program requires an occur check to be performed with any unification involving these three clauses.

3. IMPLEMENTATION

In this section we will describe two algorithms for implementing the test given in the previous section. The first algorithm finds an optimal input/output position combination for each predicate of a given Prolog program; the second algorithm finds a set of valid input/output position combinations for each predicate of a given Prolog program. The second method will be shown to be better than the first one, in the sense that the first method may result in introducing occur checks into some Prolog programs that would not be introduced by the second method (and that are, therefore, not necessary for the soundness of unification in the program).

In what follows, we shall try to find an input/output position combination that chooses as many positions as possible to be output positions. This is because our test requires all input positions of clause heads and facts to be collinear; hence, the fewer input positions we have, the more likely we are to have collinear input positions. This in turn reduces the number of occur checks inserted in the program. Note that some of the occur checks inserted may still be superfluous; however, the given test does not detect this fact. Thus our approach will be to call a position an input position only if it is impossible to call it an output position without violating some of the constraints given about input and output positions. Note it is always possible to find an input/output position combination, because there exists

the obvious input/output position combination obtained by setting all argument positions to be input positions.

Let us mention at this point that it is always possible to transform a clause that does not satisfy the collinearity condition of Theorem 2.2 into one that does, by "pulling" some of the unifications into the body. For example, given the clause

$$p(X, X) :- \cdots,$$

where both argument positions for $p/2$ are input positions, it can be transformed to

$$p(X, Y) :- X = Y, \ldots.$$

The way this is handled is to assume that the predicate ' = '/2 is defined by a single clause

$$=(X, X).$$

3.1. First Method

The method that we present here determines a valid input/output position combination for a given Prolog program. The input/output position combination obtained is optimal in the sense that no input positions can be changed into output positions without violating the definition of input and output positions.

First we create a list L(p) for each predicate p of the program, which consists of N elements, where N is the arity of p. The N elements in the list initially are N distinct variables. As soon as the argument position i of the predicate p is determined to be an input position, the ith variable in this list is set to "input."

For example, for the ancestor program given in Example 2.3, we have the following lists for the predicates of the program:

$$\text{L(ancestor)} = [\_5, \_10]$$
$$\text{L(q)} = [\_15, \_20],$$

where _5, _10, _15, and _20 are all new variables. We now perform a first pass over the program in order to determine some of the arguments of the predicates that have to be input positions. Suppose we have a clause

$$L :- L_1, \ldots, L_n$$

and suppose the predicate of L_i is p, $1 \leq i \leq n$. Consider the kth argument of p. If any of the variables in this argument appear elsewhere in L_i, then the kth position of p has to be an input position. Similarly if any of the variables in the kth argument of p appear in any of $L_1, L_2, \ldots, L_{i-1}$, then the kth position of p is an input position. At this stage we do not check the heads of clauses or facts. We store the information thus obtained about input positions in the lists L(p).

For example, for the program given earlier, after this step the lists become:

$$\text{L(ancestor)} = [\text{input}, \text{input}], \text{L(q)} = [\_15, \_20].$$

The next step is to verify whether the unmarked positions (i.e, those that have not been marked as input positions) are eligible to be output positions or whether they

too need to be marked as input positions. We do this by an iterative process. We repeatedly perform the following iteration on the lists until a fixpoint is reached:

> Examine all the clauses of the program one by one. For every variable in the body of the clause, occurring in a position α of a predicate p, which has not yet been marked as an input position in L(p), we need to check whether that variable occurs in any position in the head of that clause, which has already been marked as an input position. If it does, then the position α in which this variable occurs in p in the body of the clause must be marked as an input position too.

The program is thus repeatedly examined as previously explained until all the lists L(p) remain unchanged by the application of the preceding step, i.e, until fixpoints are obtained for all the lists L(p). These lists now have some predicate positions marked as input positions and the others are still unmarked. We now mark all the unmarked positions as output positions in these lists. Note that the positions that were marked as input positions by the preceding procedure could not have been marked as output positions, by definition. Therefore, the resulting input/output position combination, given by the lists L(p), is the input/output position combination with the least possible number of positions marked as input positions.

Once this input/output position combination has been found, we need to check whether all the heads of clauses (including facts) have all input positions collinear. If they do, we are done and the given program requires no modification; if not, this means that no input/output position combination can be found that makes the input positions in heads of clauses collinear. This is because the input/output position combination obtained by the foregoing fixpoint iteration marks the smallest possible number of argument positions as input positions; hence, in any other admissible input/output position combination, the set of argument positions that will be marked as input positions will be a superset of the set of argument positions that were marked as input positions by our fixpoint iteration. This means that the heads of clauses still cannot have all input positions collinear. The input program, therefore, requires some modification to ensure that unification without occur check will be sound, which can be done by performing occur checks for all unifications involving heads of clauses that do not have all input positions collinear.

Time Complexity Analysis. The complexity of the preceding algorithm can be analyzed in terms of the number of distinct predicates (p) in the program, the total number of literals (P) in the program, the maximum arity A of predicates in the program, and the maximum number of variables N occurring in arguments of a literal in the program, counting all occurrences of a variable (e.g., if a literal $p(f(x,y), g(g(x)))$ is given, the number of variables occurring in arguments of p is 3, because there are three occurrences of variables occurring in arguments of p, namely, x, y, x).

There are p lists $L(p)$, each with at most A elements. The first pass over the program must examine, for each argument of each predicate occurrence, all the variables occurring in arguments of that predicate and all predicates to the left of it. The total time taken, therefore, is in $O(N^2 * P^2)$ (because there are at most $N * P$ variables, each of which have to be compared with each other).

The second step, consisting of the fixpoint iteration, will be repeated at most $A * p$ times (the sum of the number of positions in all the lists $L(p)$). Each such iteration compares all the (at most) N variables occurring as arguments of every literal in the body of a clause (there are at most P of these) with all the (at most N) variables occurring in the head of that clause. Thus, each iteration takes time in $O(N^2 * P)$. Therefore, the total time for the fixpoint iteration is in $O(N^2 * P * A * p)$.

Thus the asymptotic (worst-case) time complexity of this algorithm is in

$$O(N^2 * P * A * p + N^2 * P^2).$$

Example 3.1. The following example shows how the computation of an input/output position combination proceeds using this method. Consider the following Prolog program:

1. p(X):− q(X, Y).
2. s(X):− r(X, X).
3. r(X, Y):− p(X), t(Y).
4. t(a).
5. q(a, b).
6. ? − s(a).

Initially, L(p) = [_5], L(q) = [_10, _15], L(r) = [_20, _25], L(s) = [_30], and L(t) = [_35]. After the first pass over the program, we get

$$\text{L(p)} = [\_5], \quad \text{L(q)} = [\_10, \_15], \quad \text{L(r)} = [\text{input}, \text{input}],$$
$$\text{L(s)} = [\_30], \quad \text{L(t)} = [\_35].$$

The list L(r) is obtained by noting that in the body of the second clause of the program, the predicate "r" has the same variable occurring in both argument positions. This forces these two positions to be input positions. The argument positions for all the other predicates of the program remain unmarked.

We now begin iteratively processing the program. After the first iteration, we get

$$\text{L(p)} = [\text{input}], \quad \text{L(q)} = [\_10, \_15], \quad \text{L(r)} = [\text{input}, \text{input}],$$
$$\text{L(s)} = [\_30], \quad \text{L(t)} = [\text{input}].$$

Here the argument position for predicate "p" has been marked as an input position because in clause 3 of the program, the variable "X" occurs as an argument of p and also as an argument of r. Because both argument positions of r are input positions, the argument position of p is also marked as an input position. A similar explanation holds for the predicate "t" and its argument "Y."

After the second iteration of the procedure, we get

$$\text{L(p)} = [\text{input}], \quad \text{L(q)} = [\text{input}, \_15], \quad \text{L(r)} = [\text{input}, \text{input}],$$
$$\text{L(s)} = [\_30], \quad \text{L(t)} = [\text{input}].$$

Here the first argument position of the predicate "q" has been marked as an input position because in clause 1 of the program, the variable "X" occurs as the first argument of q and also as an argument of p. Because the argument position of p is an input position, the first argument position of q is also marked as an input position.

The third iteration leaves all the lists unchanged. Hence, the procedure terminates and we fill in all unmarked positions in these lists to be output positions, obtaining the following input/output position combination for the program:

$$L(p) = [\text{input}], \quad L(q) = [\text{input}, \text{output}], \quad L(r) = [\text{input}, \text{input}],$$
$$L(s) = [\text{output}], \quad L(t) = [\text{input}].$$

It is easy to prove the following result about this method:

Theorem 3.1. The method described in the preceding text terminates and produces a valid input/output position combination for a given Prolog program, with as few positions as possible marked as input positions.

PROOF. First of all, we note that this method does not mark any position as an input position unless and until it is found to be strictly necessary to do so in order to comply with the definition of input and output positions. Thus at the end of the fixpoint iteration, none of the positions that were marked as input positions could have been marked as output positions without violating the definition of an output position. The remaining unmarked positions are all eligible to be marked as output positions, which is then done for all unmarked positions in the lists associated with the predicates of the program.

To see that the iteration terminates, note that there are a finite number of predicates in the program, each of which is associated with a list. At each iteration, zero or more unmarked positions in each list are marked as input positions. If zero positions are marked as input positions for each predicate during a certain iteration, then the algorithm halts. If at least one position in at least one list marked as an input position each time an iteration is performed, then this process must eventually end because each list is of finite length. □

3.2. Second Method

In this section, we will take an approach to the problem that differs from that adopted in the previous section, although the definitions used are the same. Now, a predicate in a program can be called in several different ways. It may happen that the permissible input/output position combinations for one call of the predicate may be different from those for another call of the same predicate. Thus our task will be to find a set of optimum input/output position combinations for each predicate (by "optimum" we mean an input/output position combination with as few positions labelled as input positions as possible), instead of just one input/output position combination. Then the collinearity condition for input positions of clause heads will be checked for all the input/output position combinations obtained for each predicate.

In what follows, we will be illustrating the concepts involved with the help of the following simple example. Consider the following "remove" program:

1. remove(E, L1, L2): – append(U, [E|V], L1), append(U, V, L2).
2. append([ ], X, X).
3. append([U|X], Y, [U|Z]: – append(X, Y, Z).
4. ? – remove(X, [U, X, Y, Z], L2), append(X, X, [1, 2, 1, 2]), append(V, X, X).

First, for each predicate p in the body of a clause, we add a new subscript for each occurrence of that predicate. For example, if the predicate p occurs four times in clause bodies, then these four occurrences of p will be renamed p_1, p_2, p_3, and p_4, respectively. Thus every predicate in a body clause is now distinct. For the foregoing "remove" program, this transformation yields the following program:

1. remove(E, L1, L2): – append_1(U, [E|V], L1), append_2(U, V, L2).
2. append([ ], X, X).
3. append([U|X], Y, [U|Z]): – append_3(X, Y, Z).
4. ? – remove_1(X, [U, X, Y, Z], L2), append_4(X, X, [1, 2, 1, 2]), append_5(V, X, X).

As before, we create a list for each predicate p_i, which consists of N elements, where N is the arity of p. The N elements of the list are N distinct variables. As soon as the argument position i of the predicate p_i is determined to be an input position, the ith variable in this list is set to "input." For each predicate p_i, let the set $S(p_i)$ consist of this list.

For the "remove" program, these sets are:

$S(\text{append}_1) = \{[\_5, \_10, \_15]\}$,

$S(\text{append}_2) = \{[\_20, \_25, \_30]\}$,

$S(\text{append}_3) = \{[\_35, \_40, \_45]\}$,

$S(\text{append}_4) = \{[\_50, \_55, \_60]\}$,

$S(\text{append}_5) = \{[\_65, \_70, \_75]\}$,

$S(\text{remove}_1) = \{[\_80, \_85, \_90]\}$.

We now perform a first pass over the program in order to determine some of the arguments of the predicates that have to be input positions. Suppose we have a clause

$$L: -L_1, \ldots, L_n.$$

and suppose the predicate of L_i is p_j. Consider the kth argument of p_j. If any of the variables in this argument appear elsewhere in L_i, then the kth position of p_j has to be an input position. Similarly, if any of the variables in the kth argument of p_j appear in any of $L_1, L_2, \ldots, L_{i-1}$, then the kth position of p_j is an input position. At this stage we do not check the heads of clauses or facts. We store the information thus obtained about input positions in the list in set $S(p_j)$, repeating this for all the predicates in all the clause bodies of the program.

For the "remove" program, this step yields:

$S(\text{append}_1) = \{[\_5, \_10, \_15]\}$,

$S(\text{append}_2) = \{[\text{input}, \text{input}, \_30]\}$

(because variables U and V appear in predicate append_1),

$S(\text{append}_3) = \{[\_35, \_40, \_45]\}$,

$S(\text{append}_4) = \{[\text{input}, \text{input}, \_60]\}$

(because the first and second arguments are identical),

$S(\text{append}_5) = \{[\_65, \text{input}, \text{input}]\}$

(because the second and third arguments are identical),

$S(\text{remove}_1) = \{[\text{input}, \text{input}, \_90]\}$

(because X occurs in the first and second arguments).

The next step is to verify whether the unmarked positions (i.e., those that have not been marked as input positions) are eligible to be output positions or whether they too need to be marked as input positions. We do this by an iterative process. We repeatedly perform the following iteration until a fixpoint is reached for all the sets $S(p_i)$.

Examine all the clauses of the program one by one. Suppose we are examining a clause

$$L: -L_1, L_2, \ldots, L_n.$$

Then for each L_j, $1 \leq j \leq n$, do the following. Suppose the predicate of L_j is p_i and the predicate of L is q. Suppose there are k occurrences of the predicate q in clause bodies. Then a new subscript would have been added to each of these k occurrences of q in a clause body. Let the union of these k occurrences be $\bigcup_j S(q_j)$.

For every list M in the set $S(p_i)$ do

For every list P in $\bigcup_j S(q_j)$ do

$M' \leftarrow M$;

For every variable occurring in a position α of p_i that has not yet been marked as an input position in the list M', check whether that variable occurs in any position in the head of the clause containing p_i. If it does, then check whether that position has been marked as an input position in P. If it has, mark the position α in the list M' as an input position;

Add M' to the set $S(p_i)$.

The program is thus repeatedly examined as explained in the preceding text until all the sets $S(p_i)$ remain unchanged by the application of the foregoing step, i.e., until a fixpoint is obtained for all these sets. Now remove from list $S(p_i)$ any list N if there exists a list N' in $S(p_i)$ such that all the input positions of N are also input positions of N'. This can be done because if all input positions in a clause head are collinear using N' as an input/output position combination, then all input positions in that clause head will be collinear using N as an input/output position combination too. These sets now have some predicate positions marked as input positions and the others still unmarked. We now mark all the unmarked positions as output positions. Note that the positions that were marked as input positions by the preceding procedure could not have been marked as output positions, by definition. Therefore, the resulting input/output position combination, given by the sets $S(p_i)$, are the input/output position combinations with the least possible number of positions marked as input positions using this method.

For the program "remove," the iteration proceeds as follows:

First iteration:

$$S(\text{append}_1) = \{[\_5, \text{input}, \text{input}]\},$$

$$S(\text{append}_2) = \{[\text{input}, \text{input}, \_30]\},$$

$$S(\text{append}_3) = \{[\text{input}, \text{input}, \_45], [\_35, \text{input}, \text{input}]\},$$

$$S(\text{append}_4) = \{[\text{input}, \text{input}, \_60]\},$$

$$S(\text{append}_5) = \{[\_65, \text{input}, \text{input}]\},$$

$$S(\text{remove}_1) = \{[\text{input}, \text{input}, \_90]\}.$$

The second iteration yields no changes and, therefore, the procedure halts here. We mark all unmarked positions as output positions and get

$$S(\text{append}_1) = \{[\text{output}, \text{input}, \text{input}]\},$$

$$S(\text{append}_2) = \{[\text{input}, \text{input}, \text{output}]\},$$

$$S(\text{append}_3) = \{[\text{input}, \text{input}, \text{output}], [\text{output}, \text{input}, \text{input}]\},$$

$$S(\text{append}_4) = \{[\text{input}, \text{input}, \text{output}]\},$$

$$S(\text{append}_5) = \{[\text{output}, \text{input}, \text{input}]\},$$

$$S(\text{remove}_1) = \{[\text{input}, \text{input}, \text{output}]\}.$$

Now, from these sets $S(p_i)$ for different i, we build a set S(p) of input/output position combinations for each predicate p of the original program. S(p) is simply defined to be the union of all the sets $S(p_i)$, for every such i. The set S(p), therefore, gives a set of permissible input/output position combinations for the predicate p of the program.

For our remove program, we get

$$S(\text{append}) = \{[\text{input}, \text{input}, \text{output}], [\text{output}, \text{input}, \text{input}]\},$$

$$S(\text{remove}) = \{[\text{input}, \text{input}, \text{output}]\}.$$

We now need to check whether all the heads of clauses (including facts) have all input positions collinear. Because each clause head p has a set of possible input/output position combinations S(p), we need to check that p has all input positions collinear for every input/output position combination in S(p). If this condition is satisfied for every p, then we are done and the given program requires no occur check; if not, this means that no input/output position combination can be found that makes the input positions in heads of clauses collinear. This is because the input/output position combination obtained by the foregoing fixpoint iteration marks the smallest possible number of argument positions as input positions. Hence, in any other admissible input/output position combination, the set of argument positions that will be marked as input positions will be a superset of the set of argument positions that were marked as input positions by our fixpoint iteration. This means that the heads of clauses still cannot have all input positions collinear. As before, the input program requires some modification to ensure that unification without occur check will be sound.

Concluding our example, we need to check the clause heads of the program. We find that only the second clause has noncollinear input positions for the input/output position combination given by (output, input, input); hence, an occur check will be required here. None of the other clause heads requires an occur check according to this method.

A quick glance at the preceding algorithm immediately reveals that its time complexity is exponential. This is because the size of a set $S(p_i)$ could theoretically grow to 2^A, where A is the arity of the predicate p_i. Thus it is clear that the first algorithm described has an advantage over this one in terms of efficiency.

It is easy to prove that the foregoing method terminates and is sound.

Theorem 3.2. If for every clause head with some predicate p, the set of input positions of p is collinear for every input/output position combination in S(p), then the program requires no occur check.

PROOF. Suppose that for every clause head with some predicate p, the set of input positions is collinear for every input/output position combination in S(p). Now, as mentioned in Theorem 2.2, every unification takes place between a subgoal and a clause head. Consider any subgoal being unified with a clause head. Suppose the predicate in this subgoal and clause head is p, and suppose that in this subgoal, p was replaced by p_i during the process of finding input/output position combinations described earlier. Then every element of $S(p_i)$ is an input/output position combination for p_i satisfying our definition of input and output positions (this is clear from the way we calculated these input/output position combinations). Also note that because $S(p_i) \subseteq S(p)$, the predicate p in the clause head being unified here has all input positions collinear for all input/output position combinations in $S(p_i)$. Therefore, by Theorem 2.2, no occur check is required for this unification. □

Theorem 3.3. The algorithm described in this section will eventually terminate.

PROOF. To see that the iteration in this algorithm terminates, note that there is a finite number of predicates in the program, each of which is associated with a set of input/output position combinations. At each iteration, zero or more input/output position combinations are added to each such set. If zero input/output position combinations are added to each such set during a given iteration, then the algorithm halts. If at least one input/output position combination is added to at least one set each time an iteration is performed, then this process must eventually end because the number of possible input/output position combinations for any predicate is finite and thus the maximum possible cardinality of every set is finite (and equals $2^{\text{arity of predicate}}$). □

3.3. Comparison of the Two Methods

We note that the methods that we have developed here for detecting places in a program where occur checks should be added give sufficient conditions for the unifications performed in a program to be sound; however, the conditions are not necessary. In other words, an occur check found to be necessary using our methods

may not actually be required for the soundness of the program. For example, consider the following program:

1. palindrome(L): – reverse(L, L).
2. reverse(L1, L2): – reverse(L1, [ ], L2).
3. reverse([ ], L, L).
4. reverse([H|L1], L2, L3): – reverse(L1, [H|L2], L3).
5. ? – palindrome([m, a, d, a, m]).

Here all the three argument positions of *reverse*/3 are input positions (see Example 2.1), and in clause 3, the three input positions in the clause head are not collinear. However, this program does not require any occur check.

We will now compare the two methods given in Sections 3.1 and 3.2. We start by proving the following theorem:

Theorem 3.4. If an occur check is determined to be necessary for a given program using the second method, then the same occur check will be found necessary using the first method. (In other words, the second method is "at least as good as" the first one.)

PROOF. Suppose that an occur check is determined to be necessary for a given program using the method of Section 3.2. This means that there is some clause head H with predicate p for which all input positions are not collinear for at least one input/output position combination α in S(p). Now, S(p) is the union of n sets $S(p_i)$, n being the number of occurrences of literals with predicate symbol p in bodies of clauses in the program. Hence, the input/output position combination α belongs to at least one of these sets, say $S(p_i)$ for some $1 \leq i \leq n$. Let the clause that contains a literal with predicate symbol p_i in its body be

$$L: -L_1, L_2, \ldots, L_{j-1}, L_j, L_{j+1}, \ldots, L_m.,$$

where the predicate of the literal L_j is p_i. Suppose that the kth and lth positions in the clause head H mentioned earlier are input positions in α and are not collinear. For either of these two positions, there are two possible reasons for which the position could have been marked as an input position in α: either (i) some variable in that position in L_j occurred in some L_r for $1 \leq r \leq j-1$, or some variable in, that position occurred elsewhere in L_j, or (ii) some variable in that position in L_j occurred in a position in L that had been marked as an input position in some input/output position combination belonging to S(q), where q is the predicate of the clause head L.

For exactly the same two possible reasons, both the positions k and l would have been marked as input positions in L(p) too (using the first method). Thus using the first method, we would also have obtained noncollinear input positions k and l in the same clause head p for the input/output position combination of p. Thus we would also require the same occur check using the first method, and the theorem is proved. □

We can prove an even stronger result: the second method is strictly better than the first one in some cases. This can be demonstrated using the example that was used in Section 3.2. Recall that we had the following *remove* program:

1. remove(E, L1, L2): – append(U, [E|V], L1), append(U, V, L2).
2. append([ ], X, X).
3. append([U|X], Y, [U|Z]): – append(X, Y, Z).
4. ? – remove(X, [U, X, Y, Z], L2), append(X, X, [1, 2, 1, 2]), append(V, X, X).

Using the second method, we obtained

S(append) = {[input, input, output], [output, input, input]},

S(remove) = {[input, input, output]}

and concluded that an occur check was required only for the second clause of the program, because the second and third positions in its head are not collinear and are marked as input positions in the first element of S(append). Let us now apply the method of Section 3.1 to the same example. After the first pass over the program, we get

L(append) = [input, input, input],

L(remove) = [input, input, _30].

Iterating once, we get

L(append) = [input, input, input],

L(remove) = [input, input, _30]

and these lists are not changed by a second iteration. Thus we get the following optimal input/output position combination for the program:

L(append) = [input, input, input],

L(remove) = [input, input, output].

We now check for the collinearity of input positions in clause heads. We find that occur checks are required for clauses 2 and 3, because the second and third positions in the head of clause 2 are not collinear, and the first and third positions in the head of clause 3 are also not collinear. Thus we see that two occur checks are required using this method, whereas only one occur check was required using the method from Section 3.2. This shows that the second method performs better than the first one in some cases.

4. RESULTS AND DISCUSSION

4.1. Results

The first algorithm described in this paper, in Section 3.1, was implemented and run on two sets of benchmark programs. The programs in the first set are "toy" programs and were chosen for comparison purposes, e.g. with [2]. Those in the second set are much larger, "real-world" programs; these programs were used in [12] for benchmarking purposes. The 10 toy programs used are listed in the Appendix. Only 4 out of the 10 programs listed required some modification to include unification with occur check at some places in the program. Of these four, only the "ancestor" program actually required unification with occur check. The original "ancestor" program got into an infinite loop when trying to print a circular term. Table 1 shows the number of occur checks inserted into each of these programs. From the table, it can be seen that two unnecessary occur checks were inserted into the "bubblesort" and the "remove" programs, and one unnecessary occur check was inserted in the "palindrome" program.

TABLE 1. Results for toy programs

Program Name	Number of Occur Checks Inserted	Time Taken (s)	Program Size (Number of Clauses)
ancestor	3	0.067	5
append	0	0.05	3
bubblesort	2	0.133	5
insert	0	0.1	5
palindrome	1	0.167	5
quicksort	0	0.184	7
queens	0	3.283	19
remove	2	0.1	4
reverse	0	0.066	4
unify	0	0.35	13

The results from the second set of programs were equally encouraging (see Table 2). Although the programs in these sets were sizeable, not more than two occur checks were inserted in 7 out of the 10 programs, and the maximum number inserted was 17. This leads us to two conclusions. First, it is clear that the occur check is required relatively rarely in practice. Second, the method given in this paper does a fairly conservative job of inserting occur checks. The algorithm is reasonably efficient, as the running times in the third column indicate; these were obtained by running Quintus Prolog on a SUN SPARCstation. It is likely that the implementation used can be made more efficient by some fine tuning.

4.2. Comparison with Other Methods

In the previous section, we saw that it is possible for our method to insert unnecessary occur checks into some programs. However, the number of unnecessary occur checks inserted was very small in these cases. We now compare our method with those already existing in the literature. Plaisted's method [8] works by generating a set of instances of each clause that can be generated by an execution of the Prolog program and then examining this set to detect places where loops

TABLE 2. Results for some larger programs

Program Name	Number of Occur Checks Inserted	Time Taken (s)	Program Size (Number of (Clauses)
asm	0	0.534	264
boyer	13	10.167	135
browse	1	4.2	31
fourqueens	0	1.634	24
func	1	5.466	159
peephole	17	3.667	130
preprocess	7	6.517	121
projgeom	1	0.517	18
read	2	5.9	108
serialize	0	0.383	11

may be created. Occur checks can then be added to the program where appropriate. A "depth" parameter is used to control the precision of the method. If this depth parameter is large enough, then the method will not insert unnecessary occur checks. However, there is no way of knowing how large this parameter must be made in order to avoid the insertion of unnecessary occur checks. If the parameter is not sufficiently large, this method could insert unnecessary occur checks in the program. Also, the method becomes more laborious as this depth parameter is increased.

Beer [2] presents a method for inserting occur checks into Prolog programs that is based on a dynamic classification of the context in which logical variables occur. The method is discussed in terms of an implementation based on the Warren abstract Prolog instruction set. His method can also insert unnecessary occur checks into Prolog programs, and inserted 49 occur checks into the "quicksort" benchmark program. It did not insert unnecessary occur checks into other benchmark programs on which the method was tested. Our method, on the other hand, inserted no occur checks into the quicksort program and did not insert more than two unnecessary occur checks into any of the toy benchmark programs. Thus our method seems to compare more than favorably with Beer's method.

A number of abstract interpretation methods have been suggested in the literature for detecting places in Prolog programs where the occur check could be required. The approach adopted by these methods is different from ours because they depend on the semantics of the Prolog program, whereas our method is more concerned with the pattern of occurrence of variables in terms and the linearity and collinearity of terms. Sodergaard's method [11] is one of these and is conjectured to be exponential in the largest number of variables in a clause of a program. His method has not yet been implemented and it is, therefore, difficult to compare our results with his.

Probably one of the strongest arguments in favor of the method presented in this paper is its simplicity and efficiency. In contrast with Plaisted's method [8], which has not yet been implemented and indeed may prove to be too cumbersome to implement, our method has been implemented and shown to be practicable based on the fact that it does not insert more than a couple of unnecessary occur checks in the programs in the benchmark set in most cases. It can probably be incorporated with ease into Prolog compilers for inserting occur checks in programs to remedy the problem of unsound unification.

APPENDIX

This appendix lists the "toy" programs that were used for testing the method described in this paper. The programs mentioned in Table 2 are those cited in [12] for benchmarking purposes.

- *ancestor*

```
q(X, Y):- ancestor(X, Y), ancestor(Y, X).
ancestor(father(X), X).
ancestor(mother(X), X).
ancestor(X, X).
? - q(U, V).
```

• *append*

```
append([ ], X, X).
append([U|X], Y, [U|Z]): – append(X, Y, Z).
? – append(X, X, Y).
```

• *bubblesort*

```
busort(L, S): –
    append(U, [A, B|V], L),
    B < A,
    !,
    append(U, [B, A|V], W),
    busort(W, S).
busort(L, L).
append([ ], X, X).
append(U|X], Y, [U|Z]): – append(X, Y, Z).
? – busort([4, 12, 3, 1], Ans).
```

• *insert*

```
insert([ ], [ ]).
insert([X|L], M): – insert(L, N), insert(X, N, M).
insert(X, [A|L], [A|M]): – A < X, !, insert(X, L, M).
insert(X, L, [X|L]).
? – insert([3, 7, 4, 8, 1], Z).
```

• *palindrome*

```
palindrome(L): – reverse(L, L).
reverse(L1, L2): – reverse(L1, [ ], L2).
reverse([ ], L, L).
reverse([H|L1], L2, L3): – reverse(L1, [H|L2], L3).
? – palindrome([m, a, d, a, m]).
```

• *quicksort*

```
qsort([H|T], S): –
    split(H, T, A, B),
    qsort(A, A1),
    qsort(B, B1),
    append(A1, [H|B1], S).
split(H, [A|X], [A|Y], Z): – A < H, !, split(H, X, Y, Z).
split(H, [A|X], Y, [A|Z]): – H =< A, !, split(H, X, Y, Z).
split(_, [ ], [ ], [ ]).
append([ ], X, X).
append([U|X], Y, [U|Z]): – append(X, Y, Z).
? – qsort([3, 8, 1, 2], Ans).
```

• *queens*

```
all_queens: –
    bagof(X, get_solutions(X), L),
    length(L, N),
    write('Number of Solutions' = N), nl,
    write('Time' = DeltaTime), nl.
size(8).
int(1).
int(2).
```

```
int(3).
int(4).
int(5).
int(6)
int(7).
int(8).
get_solutions(Soln): – solve([ ], Soln).
newsquare([ ], square(1, X)): – int(X).
newsquare([square(I, J)|Rest], square(X, Y)): –
      X is I + 1,
      int(Y),
      not_threatened(I, J, X, Y),
      safe(Rest, X, Y).
safe([ ], X, Y).
safe([square(I, J)|L], X, Y): – not_threatened(I, J, X, Y), safe(L, X, Y).
not_threatened(I, J, X, Y): – I = \ = X,  J = \ = Y,  I – J = \ = X – Y,  I + J =
   \ = X + Y.
solve([square(Bs, Y)|L], [square(Bs, Y)|L]): – size(Bs).
solve(Initial, Final): – newsquare(Initial, Next), solve([Next|Initial], Final).
? – all_queens.
```

• *remove*

```
remove(E, L1, L2): – append(U, [E|V], L1), append(U, V, L2).
append([ ], X, X).
append([U|X], Y, [U|Z]): – append(X, Y, Z).
? – remove(X, [U, X, Y, Z], L2).
```

• *reverse*

```
reverse(L1, L2): – rev(L1, [ ], L2).
rev([ ], L, L).
rev([H|L1], L2, L3): – rev(L1, [H|L2], L3).
? – reverse([1, 2, 3, 4], Answer).
```

• *unify*

```
occ_check(Term, Var): – var(Term), !, Term \ == Var.
occ_check(Term, Var): – functor(Term, _, Num_args),
   do_occ_check(Num_args, Term, Var).
do_occ_check(0, _, _): – !.
do_occ_check(N, Term, Var): –
      arg(N, Term, Arg),
      occ_check(Arg, Var),
      M is N – 1, !,
      do_occ_check(M, Term, Var).
un(X, Y): – X == Y, !.
un(X, Y): – unif(X, Y), !.
unif(X, Y): – var(X), var(Y), !, X = Y.
unif(X, Y): – var(X), !, occ_check(Y, X), X = Y.
unif(X, Y): – var(Y), !, occ_check(X, Y), Y = X.
unif(X, Y): – atomic(X), !, X = Y.
unif(X, Y): – functor(X, F, N), functor(Y, F, N), unifying(N, X, Y).
unifying(0, X, Y): – !.
```

```
unifying(N,X,Y):-
    arg(N,X,Arg1),
    arg(N,Y,Arg2),
    unif(Arg1,Arg2),
    M is N - 1,
    !,
    unifying(M,X,Y).
```

We would like to thank the anonymous reviewers for their helpful suggestions and comments. We would also like to mention that while this paper was under review, Apt and Pellegrini [1] independently discovered a proof of our Theorem 2.2 and its corollary.

REFERENCES

1. Apt, K. R. and Pellegrini, A., Why the Occur-Check is Not a Problem, *PLILP '92, Leuven, Belgium. Lecture Notes in Computer Science*, Vol. 631, Springer, Berlin, 1992, pp. 69–86.
2. Beer, J., The Occur-Check Problem Revisited, *J. Logic Programming* 5:243–261 (1988).
3. Colmerauer, A., Prolog and Infinite Trees, in K. L. Clark and S. A. Tärnlund (eds.), *Logic Programming*, Academic, New York, 1982.
4. Debray, S. K., Static Inference of modes and Data Dependencies in Logic Programs, *ACM Trans. Programming Lang. Syst.* 11(3):418–450 (1989).
5. Debray, S. K. and Warren, D. S., Functional Computations in Logic Programs, *ACM Trans. Programming Lang. Syst.* 11(3):451–481 (1989).
6. Deransart, P. and Maluszynski, J., Relating Logic Programs and Attribute Grammars, *J. Logic Programming* 2:119–155 (1985).
7. Mellish, C. S., Some Global Optimizations for a Prolog Compiler, *J. Logic Programming* 1:43–66 (1985).
8. Plaisted, D. A., The Occur-Check Problem in Prolog, *New Generation Comput.* 2:309–322 (1984).
9. Reddy, U. S., Transformation of Logic Programs into Functional Programs, *Proceedings of the 1984 International Symposium on Logic Programming*, Atlantic City, New Jersey, IEEE, New York, 1984, pp. 187–196.
10. Robinson, J., A Machine-Oriented Logic Based on the Resolution Principle, *J. ACM* 12:23–41 (1965).
11. Sondergaard, H., An Application of Abstract Principles of Logic Programs: Occur-Check Reduction, *European Symposium on Programming, Saarbrücken, Federal Republic of Germany*, 1986, pp. 327–338.
12. Warren, R., Hermenegildo, M., and Debray, S. K., On the Practicality of Global Flow Analysis of Logic Programs, *International Conference on Logic Programming, Seattle*, 1988, pp. 684–699.