Unification-free Execution of Well-moded and Well-typed Prolog Programs[1]

M.R.K. Krishna Rao
and
R.K. Shyamasundar
Computer Science Group,
Tata Institute of Fundamental Research,
Colaba, BOMBAY 400 005, INDIA.
e-mail: {`krishna,shyam`}`@tcs.tifr.res.in`

Abstract

In this paper, a static analysis of Prolog programs has been proposed to improve the efficiency by replacing run-time unification with matching. Programs for which run-time unification can be replaced by matching are called *unification-free programs*. We propose a set of sufficient conditions for unification-freedom improving the existing results and discuss the application of our results in improving WAM code of Prolog programs. `Permutation` and PRESS (a system developed at Edinburgh for solving symbolic equations) and programs `flatten-dl, quick-sort-dl` using difference-lists are example programs which can be handled by our results but not by the earlier results.

1 Introduction

Unification is the basic step in the computation of logic programs and is often the bottle-neck for efficient execution. There has been a considerable amount of research [8, 14, 5, 3] in arriving at a set of sufficient conditions under which the costly unification can be replaced by rather less expensive iterated matching. The basic idea is to arrive at sufficient conditions that are simple and can be easily verified using static/global analysis.

A recent paper by Apt and Etalle [3] presents the best known results on unification-freedom by characterizing a few classes of Prolog programs for which unification can be replaced by iterated matching. They use *moding* information in handling programs running on queries with ground inputs (basically, well-moded programs) and both *typing* and *moding* information in handling programs running on queries with non-ground inputs (i.e., well-typed programs). In the context of well-moded programs and queries, the main result of [3] gives the following set of sufficient conditions for replacing unification by iterated matching:

(i) input variables of the head do not occur in output positions of the body atoms,

(ii) no variable occurs more than once in output positions of the body atoms,

(iii) no variable occurs in both input and output positions of any body atom, and

(iv) terms occurring in output positions of the body atoms are variables.

The programs (clauses and queries) satisfying the above four properties are refered to as *simply-moded* in [3]. In the context of well-typed programs and queries, a further condition is imposed on the input terms of heads. By listing a series of Prolog programs from the book of Sterling and Shapiro [16] which satisfy these conditions, Apt and Etalle [3] demonstrate that these conditions are reasonable to a great extent. Notable exceptions are `permutation` and the Prolog

[1]This is a revised and extended version of [11]

system PRESS (a system developed by Bundy and Welham [7] at Edinburgh for solving symbolic equations) which contains a non-variable term in an output position in the body. Other programs which do not satisfy these conditions include `flatten` and `quick-sort` programs using difference-lists. It must however be noted that all these programs are unification-free. In the following, we generalize the results of [3] to handle these programs and show that the above conditions (i), (ii) and (iii) can be relaxed.

We first discuss how to exploit the unification-freedom in improving efficiency and motivate the static analysis. Two natural approaches are (a) to transform programs into a known class of unification-free programs and (b) to develop sufficient conditions to characterize larger classes of unification-free programs and use static analysis. First, we show that every well-moded program can be transformed into an equivalent simply-moded program. However, the resulting program can be inefficient due to backtracking (see section 3). In line with the second approach, we propose sufficient conditions for unification-freeness of well-moded and well-typed programs and discuss application of the obtained results in improving WAM code of Prolog programs through static analysis.

Rest of the paper is organized as follows. The next section introduces the required terminology and preliminary results and section 3 deals with unification-freeness of well-moded programs. Section 4 deals with well-typed programs and section 5 discusses application of our results in improving WAM code of Prolog programs. Section 6 provides a comparison with related works and section 7 concludes with a summary of results.

2 Preliminaries

We follow the notations of Lloyd [12] and Apt [2] in general.

2.1 Well-moded Programs

In this subsection, we define the notion of well-moded Prolog programs/queries and give a few basic lemmas. Though we assume in the rest of the paper that moding information is available, the programmer need not always supply the moding information as there are many techniques available in the literature for inferring moding information from a given program.

In the following, we give a different (from the definitions used in the literature) definition of well-moded Prolog programs using the concepts of producer and consumer of variables. The concepts of producer and consumer facilitate the separation of the notion of well-modedness from the computation rule used in the computation (see [10] for the most general notion of well-moded logic programs). Since, we are considering Prolog programs in this paper, the selection rule is fixed and our definition coincides with the other definitions.

Definition 1 A *mode* m of an n-ary predicate p is a function from $\{1, \cdots, n\}$ to the set $\{in,\ out\}$. The set $\{j \mid m(j) = in\}$ is the set of input positions of p and $\{k \mid m(k) = out\}$ is the set of output positions of p.

Notation: The terms $invar(L)$ and $outvar(L)$ denote the sets of variables occurring in the input and output positions of L respectively and $Var(L) = invar(L) \cup outvar(L)$.

Definition 2 Let $H \leftarrow B_1, \cdots, B_k$ be a clause and X be a variable occurring in it. The atom B_i is a *consumer* of X if $X \in invar(B_i)$, and B_i is a *producer* of X if $X \in outvar(B_i)$. The head H is a *producer* of variable X if $X \in invar(H)$, and H is a *consumer* of X if $X \in outvar(H)$.

Definition 3 A Prolog clause $H \leftarrow B_1, \cdots, B_k$ is *well-moded* if (a) every variable in it has at least one producer and (b) every variable in $invar(B_i)$ has at least one producer in $H,\ B_1, \cdots, B_{i-1}$. A Prolog program P is *well-moded* if every clause in it is well-moded. A *well-moded query* is a well-moded clause without head.

The following lemmas are well-known.

Lemma 1 *If* $\leftarrow B_1, \cdots, B_n$ *is a well-moded Prolog query then* $invar(B_1) = \phi$.

Lemma 2 *If* $H \leftarrow B_1, \cdots, B_n$ *is a well-moded Prolog clause then each variable in* $(outvar(H) - invar(H))$ *occurs in at least one output position in the body.*

Lemma 3 *If* $H \leftarrow B_1, \cdots, B_n$ *is a well-moded Prolog clause and* θ *is a substitution then* $H\theta \leftarrow B_1\theta, \cdots, B_n\theta$ *is a well-moded Prolog clause too.*

Lemma 4 *If* P *is a well-moded Prolog program and* G *is a well-moded goal then every goal in any LD-derivation (SLD-derivation under Prolog's selection rule) of* $P \cup \{G\}$ *is well-moded.*

For the sake of completeness, we give the formal definition of simply-moded programs.

Definition 4 (simply-moded programs [4])

- We say that a Prolog goal $\leftarrow p_1(\overline{s}_1, \overline{t}_1), \cdots, p_n(\overline{s}_n, \overline{t}_n)$ is *simply-moded* if $\overline{t}_1, \cdots, \overline{t}_n$ is a linear family of *variables* and for each $i \in [1, n]$

$$Var(\overline{s}_i) \cap (\bigcup_{j=i}^{n} Var(\overline{t}_j)) = \phi.$$

- We say that a Prolog clause $p_0(\overline{s}_0, \overline{t}_0) \leftarrow p_1(\overline{s}_1, \overline{t}_1), \cdots, p_n(\overline{s}_n, \overline{t}_n)$ is *simply-moded* if the goal $\leftarrow p_1(\overline{s}_1, \overline{t}_1), \cdots, p_n(\overline{s}_n, \overline{t}_n)$ is simply-moded and

$$Var(\overline{s}_0) \cap (\bigcup_{j=1}^{n} Var(\overline{t}_j)) = \phi.$$

- A Prolog program is simply-moded if each clause in it is simply-moded.

2.2 Unification and Iterated Matching

We use the following conventions of Apt and Etalle [3]. Given two sequences of terms $\overline{s} = s_1, \cdots, s_n$ and $\overline{t} = t_1, \cdots, t_n$, we abbreviate the set of equations $\{s_1 = t_1, \cdots, s_n = t_n\}$ to $\{\ \overline{s} = \overline{t}\ \}$. With two atoms $p(\overline{s})$ and $p(\overline{t})$ to be unified, we associate $\{\ \overline{s} = \overline{t}\ \}$.

Definition 5 Consider a set of equations $E = \{\ \overline{s} = \overline{t}\ \}$.

1. A substitution θ such that $\overline{s}\theta \equiv \overline{t}\theta$ is called a *unifier* of E. A unifier θ of E is called a *match* for E if either $\overline{s}\theta \equiv \overline{s}$ or $\overline{t}\theta \equiv \overline{t}$. Further, θ is *idempotent* if $\theta\theta \equiv \theta$.
2. E is called *left-right disjoint* if $Var(\overline{s}) \cap Var(\overline{t}) = \phi$.
3. A left-right disjoint set of equations E is *solvable by matching* if E is unifiable implies that a match for E exists.

4. We say an atom (or term) is *pure* if it is of the form $p(\overline{x})$ (resp. $f(\overline{x})$), where $\overline{x}$ is a sequence of distinct variables.
5. Two atoms (or terms) are *disjoint* if they have no variable in common.

A simple lemma stated in [3] is reformulated below.

Lemma 5 *The set of left-right disjoint equations* $E = \{ \overline{s} = \overline{t} \}$ *is solvable by matching if* $\overline{s}$ *(or* $\overline{t}$*) is either (i) a sequence of ground terms or (ii) a sequence of distinct variables.*

The following simple lemma is useful in establishing solvability-by-matching of larger sets of equations from that of smaller sets.

Lemma 6 If sets of equations $E_1 = \{ \overline{s}_1 = \overline{t}_1 \}$ and $E_2 = \{ \overline{s}_2 = \overline{t}_2 \}$ are (i) solvable by matching with matching substitutions θ_1 and θ_2 such that $\overline{s}_1\theta_1 \equiv \overline{t}_1$ and $\overline{s}_2\theta_2 \equiv \overline{t}_2$, (ii) $Var(\overline{s}_1) \cap Var(\overline{s}_2) = \phi$ and (iii) $E_1 \cup E_2$ is left-right disjoint, then $E_1 \cup E_2$ *is solvable by matching and* $\theta = \theta_1\theta_2 = \theta_2\theta_1$ *is one of its matching substitutions.*

The notion of iterated matching defined in [3] is given below.

Definition 6 A left-right disjoint set of equations E is *solvable by iterated matching* if E *is unifiable implies that* for some $E_1, \cdots, E_n$ and substitutions $\theta_1, \cdots, \theta_n$,

- $E = E_1 \cup \cdots \cup E_n$ and
- for each $i \in [1, n]$, $E_i\theta_1 \cdots \theta_{i-1}$ is left-right disjoint and θ_i is a match for $E_i\theta_1 \cdots \theta_{i-1}$.

It is easy to see that $\theta = \theta_1 \cdots \theta_n$ is an idempotent mgu of E.

3 Unification-free Well-moded Prolog Programs

In this section, we study unification-freeness of well-moded Prolog programs. First, we show that every well-moded program can be transformed into a unification-free program and argue that such a transformation may involve a loss of efficiency, thus establishing the importance of tools for verifying unification-freeness of a given program. Then, we present a set of sufficient conditions for unification-freeness.

Definition 7 (**unification-free programs**)
Let D be a LD-derivation of a Prolog program P and a Prolog goal G.

1. If $p(\overline{s})$ is an atom selected in D and $p(\overline{t})$ is the head of the input clause used in resolving $p(\overline{s})$ then we say that the equation set $\{\overline{s} = \overline{t}\}$ is *considered* in D.
2. We say that $P \cup \{G\}$ is *unification-free* if each equation set considered in every LD-derivation of $P \cup \{G\}$ is solvable by iterated matching.

The following transformation derives an equivalent unification-free well-moded program from any given well-moded Prolog program.

Transformation: A given well-moded Prolog clause c

$$\mathtt{p}(\overline{\mathtt{s}}, \overline{\mathtt{t}}) \leftarrow \mathtt{q_1}(\overline{\mathtt{s}}_1, \overline{\mathtt{t}}_1), \cdots, \mathtt{q_n}(\overline{\mathtt{s}}_\mathtt{n}, \overline{\mathtt{t}}_\mathtt{n})$$

where $\overline{\mathtt{s}}_\mathtt{i}$ (resp. $\overline{\mathtt{t}}_\mathtt{i}$) is the sequence of input (resp. output) terms of $\mathtt{q_i}(\cdots)$, is transformed into a clause c'

$$\mathtt{p}(\overline{\mathtt{s}}, \overline{\mathtt{t}}) \leftarrow \mathtt{q_1}(\overline{\mathtt{s}}_1, \overline{\mathtt{X}}_1), \overline{\mathtt{t}}_1 \Longleftarrow \overline{\mathtt{X}}_1, \cdots, \mathtt{q_n}(\overline{\mathtt{s}}_\mathtt{n}, \overline{\mathtt{X}}_\mathtt{n}), \overline{\mathtt{t}}_\mathtt{n} \Longleftarrow \overline{\mathtt{X}}_\mathtt{n}$$

where each $\overline{\mathtt{X}}_\mathtt{i}$ is a sequence of distinct variables not occurring in c and $\Longleftarrow$ is a built-in predicate with moding `(out, in)`, called `match` predicate. In the above, $\overline{\mathtt{t}}_\mathtt{i} \Longleftarrow \overline{\mathtt{X}}_\mathtt{i}$ denotes a sequence of match atoms $\mathtt{t}^1_\mathtt{i} \Longleftarrow \mathtt{X}^1_\mathtt{i}, \cdots, \mathtt{t}^k_\mathtt{i} \Longleftarrow \mathtt{X}^k_\mathtt{i}$ when $\mathtt{t}^1_\mathtt{i}, \cdots, \mathtt{t}^k_\mathtt{i}$ constitutes the sequence $\overline{\mathtt{t}}_\mathtt{i}$.

Lemma 7 If P and Q are given well-moded Prolog program and query and P' and Q' are the program and the query derived from P and Q respectively by the above transformation, then *P' and Q' are well-moded and $P' \cup \{Q'\}$ is unification-free.*

Proof (sketch): From the moding of the `match` predicate, it easily follows that P' and Q' are well-moded. Since output positions of every non-`match` atom in the body of any clause in P' are filled by distinct variables which do not occur in the head and the output terms of Q' are distinct variables, it follows that every selected non-`match` atom in any LD-derivation of $P' \cup \{Q'\}$ has distinct variables in output positions. Therefore, unification can be replaced by double matching (first match input terms and then match output terms using Lemma 5). Further, each selected `match` atom has ground input and hence the input term can be matched onto the output term. Therefore, $P' \cup \{Q'\}$ is unification-free. $\square$

From the above theorem, it follows that every well-moded Prolog program can be transformed into a unification-free program. However, such a transformation can lead to a loss of efficiency as follows.

> Since we replace an atom $\mathtt{q_i}(\overline{\mathtt{s}}_\mathtt{i}, \overline{\mathtt{t}}_\mathtt{i})$ in the body of a clause by $\mathtt{q_i}(\overline{\mathtt{s}}_\mathtt{i}, \overline{\mathtt{X}}_\mathtt{i}), \overline{\mathtt{t}}_\mathtt{i} \Longleftarrow \overline{\mathtt{X}}_\mathtt{i}$, the output values of $\mathtt{q_i}(\cdots)$ are first computed and then tested whether they are in the specified from (by the `match` atoms) in $P' \cup \{Q'\}$ whereas these two tasks are done hand-in-hand in $P \cup \{Q\}$. Therefore, a backtracking due to incompatible outputs leads to a complete waste of their computation in $P' \cup \{Q'\}$ whereas such incompatibility is discovered much earlier (possibly at the time of unification with head itself) in $P \cup \{Q\}$.

In general, the above transformation can result in an inefficient program and hence it is not useful in getting efficient programs by exploiting unification-freeness. Thus, it's important to develop tools for verifying unification-freeness of a given program so that one can gain efficiency by replacing unification by iterated matching.

3.1 Neatly-moded Programs

The following definition characterizes the class of programs we are interested in.

Definition 8 (neatly-moded programs)

1. We say that a well-moded Prolog clause $H \leftarrow B_1, \cdots, B_n$ is *neatly-moded* if each B_i, $i \in [1, n]$ has the following property: *if X is a variable that occurs either (a) in multiple output positions of B_i or (b) in a non-variable output term of B_i then X also occurs (i) either in $B_1, \cdots, B_{i-1}$, (ii) or in input terms of H.*
2. A Prolog program is *neatly-moded* if each clause in it is neatly-moded.
3. We say that a Prolog goal $\leftarrow B_1, \cdots, B_n$ is *neatly-moded* if the clause $dummy \leftarrow B_1, \cdots, B_n$ is neatly-moded, where $dummy$ is a new predicate symbol of arity 0.

Note: By definition, a neatly-moded program/clause/goal is well-moded. It is easy to see that every simply-moded well-moded program is neatly-moded. However, a neatly-moded program need not be a simply-moded well-moded program. In particular, the following are allowed in neatly-moded programs but not in simply-moded well-moded programs

(i) a variable can occur in output positions of more than one atom in the body,
(ii) an input variable of the head can occur in output positions in the body,
(iii) a variable can occur in both input and output positions of an atom in the body, and

(iv) a non-variable term can occur in an output position of the body.

The following lemmas establish that all the goals in an LD-derivation of $P \cup \{G\}$ are neatly-moded if P and G are neatly-moded.

Lemma 8 Let $C = H \leftarrow B_1, \cdots, B_m$ be a well-moded Prolog clause and θ be an idempotent substitution such that

(i) $(domain(\theta) \cup range(\theta)) \cap (Var(C) - Var(H)) = \phi$,

(ii) $X\theta$ is ground for each variable $X \in invar(H)$,

(iii) for each variable $X \in (Var(H) - invar(H))$, $X\theta$ is either ground or a variable and

(iv) if X and Y are two different variables in $Var(H)$ then the terms $X\theta$ and $Y\theta$ are disjoint.

Then, *the clause $C\theta$ is neatly-moded if C is neatly-moded.*

Proof: Consider atom $B_i\theta$, $i \in [1, n]$ in $C\theta$. We first prove that each variable in a non-variable output term of $B_i\theta$ occurs in $B_1\theta, \cdots, B_{i-1}\theta$. Let t be an output term in B_i. If t is a variable not occurring in H then $t\theta \equiv t$. If t is a variable occurring in H then $t\theta$ is either ground or a variable. Now, consider the case of t being a non-variable term. By the definition 8, each variable (say, X) in t occurs either in $invar(H)$ or in $B_1, \cdots, B_{i-1}$. In the former case $X\theta$ is ground and in the latter case all the variables in $X\theta$ occur in $B_1\theta, \cdots, B_{i-1}\theta$.
Suppose a variable Y occurs in multiple output terms of $B_i\theta$. There are two cases. If $Y \notin Var(H)$, by properties (i) and (iv) of θ there is a variable Y' occurring in multiple output terms of B_i and $Y \in Var(Y'\theta)$. Since C is neatly-moded, Y' occurs in $B_1, \cdots, B_{i-1}$ and hence Y occur in $B_1\theta, \cdots, B_{i-1}\theta$. If $Y \in Var(H)$, the properties (i) and (iv) of θ imply that Y occurs in multiple output terms of B_i and $Y \equiv Y\theta$. Therefore, Y occur in $B_1\theta, \cdots, B_{i-1}\theta$. This completes the proof. □

Lemma 9 If $\leftarrow A_1, \cdots, A_n$ is a neatly-moded Prolog goal, *then the output terms of A_1 are disjoint and each of them is either ground or a variable.*

Proof : Follows from definition 8. □

Lemma 10 If $G = \leftarrow A_1, \cdots, A_n$ and $C = H \leftarrow B_1, \cdots, B_m$ are neatly-moded and do not share any variable, then *the LD-resolvent of G and C is also neatly-moded.*

Proof : Let θ be the mgu of A_1 and H. Then the LD-resolvent of G and C is $G' = \leftarrow B_1\theta, \cdots, B_m\theta, A_2\theta, \cdots, A_n\theta$. From Lemmas 1 and 9, it is easy to see that θ satisfies all the properties given in Lemma 8. Therefore, atoms $B_1\theta, \cdots, B_m\theta$ satisfy the requirements.

Consider atom $A_i\theta$, $i \in [2, n]$ in G'. We first prove that each variable in a non-variable output term of $A_i\theta$ occurs in $B_1\theta, \cdots, B_m\theta, A_2\theta, \cdots A_{i-1}\theta$. Let t be an output term in A_i. If t is a variable not occurring in A_1 then $t\theta \equiv t$. If t is a variable occurring in A_1 then $t\theta$ is an output term of $H\theta$ and hence by Lemmas 3 and 2, all the variables in $t\theta$ occur in $B_1\theta, \cdots, B_m\theta$. Now, consider the case of t being a non-variable term. By hypothesis, each variable in t occurs in $A_1, \cdots, A_{i-1}$ and hence each variable in $t\theta$ occurs in $B_1\theta, \cdots, B_m\theta, A_2\theta, \cdots, A_{i-1}\theta$.
Suppose a variable Y occurs in multiple output terms of $A_i\theta$. It is easy to observe that there are two cases: either (a) some occurrences of Y in $A_i\theta$ are due to application of θ on some variables in A_i or (b) all the occurrences of Y in $A_i\theta$ are already there in A_i. In case (a), it follows that Y occurs in $A_1\theta$ (note that θ is applicable on only those variables of A_i which also occur in A_1) and hence, by Lemmas 3 and 2, Y occur in $B_1\theta, \cdots, B_m\theta$. In case (b), Y occurs in $A_1, \cdots, A_{i-1}$ and $Y\theta \equiv Y$ by hypothesis. Therefore, Y occurs in $B_1\theta, \cdots, B_m\theta, A_2\theta, \cdots, A_{i-1}\theta$. This completes the proof. □

Lemma 11 If P is a neatly-moded Prolog program and G is a neatly-moded Prolog goal, then *all the goals in every LD-derivation of* $P \cup \{G\}$ *are neatly-moded.*

Proof: Follows from the above lemma by induction. □

Now, we are in a position to establish that the neatly-moded Prolog programs and queries are unification-free.

Theorem 1 *If P is a neatly-moded Prolog program and G is a neatly-moded Prolog goal, then $P \cup \{G\}$ is unification-free.*

Proof: By Lemma 11, every goal in the LD-tree of $P \cup \{G\}$ is neatly-moded and by Lemmas 1 and 9, the selected atom has ground inputs and its output terms are disjoint and each of them is either a ground term or a variable.

Now, consider the unification $A = H$ in the LD-tree, where A is a selected atom and H is the head of the input clause used. We can take E_1 as the set of equations $\{s = t \mid t$ is a ground term in A and s is the term in the corresponding position in $H\}$. By Lemma 5, E_1 is solvable by matching (say, with match θ_1). Now, take E_2 as the set of equations $\{s\theta_1 = t \mid t$ is a non-ground term in A and s is the term in the corresponding position in $H\}$. It is trivial to see that E_2 is left-right disjoint. Since right-hand sides of the equations in E_2 are distinct variables, E_2 is solvable by matching (say, with match θ_2) by Lemma 5. It is easy to see that $\theta_1\theta_2$ is an idempotent mgu of A and H. Therefore $A = H$ is solvable by iterated matching and hence, $P \cup \{G\}$ is unification-free. □

The following corollary gives a simple condition for unification-freeness and relaxes the first three requirements (see Introduction in this paper) of [3].

Corollary 1 If P is a well-moded Prolog program and G is a well-moded Prolog goal such that each atom B in the body of clauses/goal has disjoint output terms and each output term is either a ground term or a variable, then $P \cup \{G\}$ *is unification-free.*

Proof: It is easy to see that P and G are neatly-moded. □

Example 1 Consider the following `flatten` program from Sterling and Shapiro [16].

```
moding:  flatten (in, in)  and flatten-dl (in, in, out)

flatten(Xs, Ys) ← flatten-dl(Xs, Ys, [ ])
flatten-dl([X|Xs], Ys, Zs) ← flatten-dl(X, Ys, Ys1),  flatten-dl(Xs, Ys1, Zs)
flatten-dl(X, [X|Xs], Xs) ← constant(X), X ≠ [ ]
flatten-dl([ ], Ys, Ys) ←
```

This program is neatly-moded and hence, unification-free for all neatly-moded goals by Theorem 1.

It may be noted that this program is not simply-moded (notice the non-variable term in the output position of body in the first clause) and hence, the results of [3] are not applicable. □

3.2 Programs with non-ground non-variable output terms

In this subsection, we relax the requirement that all the variables in a non-variable output term occur in the preceding atoms or in the input terms of the head. Our approach here is that, given a well-moded program, we identify the 'errant output positions' — positions with non-variable output terms whose variables are not occurring in the preceding atoms and output positions sharing a variable which does not occur in the preceding atoms — and give a set of sufficient conditions on the heads of the clauses. We use the following predicate `errant` to identify the errant output positions.

The idea behind the definition of the predicate `errant` is to propagate the information about the errant output positions. If a selected atom has errant output positions, then it can induce errant output positions in the body of the (instance of) input clause which will be included in the resolvent goal. For example, if A is a selected atom with errant (say, j^{th}) output position and $C = H \leftarrow B_1, \cdots, B_k$ is the input clause and θ is the mgu then a variable in j^{th} output position of H can get bound to a non-variable non-ground term and the output positions in $B_1, \cdots, B_k$ with that variable can have a non-variable non-ground term in $B_1\theta, \cdots, B_k\theta$ (a part of the new goal). The third clause in the definition of `errant` captures this propagation. The first clause identifies the positions with non-variable non-ground output terms and the second clause identifies the positions sharing a variable which has no producer to the left of that particular atom.

The clauses in the program and atoms in the body are given numbers so that they can be easily identified. For example, `errant(cl,at,pos)` means that the term in $\texttt{pos}^{th}$ output position of $\texttt{at}^{th}$ atom in the body of $\texttt{cl}^{th}$ clause in the program is *errant* (also called *marked*).

`errant(cl,at,pos)` **if**
(i) the term (say t) in that position is a non-variable non-ground term and
(ii) Some variable $X \in Var(t)$ has no producer before the atom $\texttt{atom(cl, at)}$.

`errant(cl,at,pos)` **if**
(i) the term in that position is a variable (say, X),
(ii) X occurs in more than one output positions of the atom $\texttt{atom(cl, at)}$ and
(iii) X has no producer before the atom $\texttt{atom(cl, at)}$.

`errant(cl,at,pos)` **if**
(i) the term in that position is a variable (say, X),
(ii) X occurs in $\texttt{pos2}^{th}$ output term of head $\texttt{head(cl)}$,
(iii) X has no producer before the atom $\texttt{atom(cl, at)}$ and
(iv) there is a clause `cl2` in the program such that $\texttt{errant(cl2, at2, pos2)}$ and $\texttt{head(cl)}$ and $\texttt{atom(cl2, at2)}$ are unifiable.

Example 2 Consider the following quick-sort program adapted from [16].

`moding: qs (in, in); qs-dl (in, in, out);` and `part (in, in, out, out)`

```
1. qs(Xs, Ys) ← qs-dl(Xs, Ys, [ ])
2. qs-dl([X|Xs],Ys,Zs) ← part(Xs,X,L,B), qs-dl(L,Ys,[X|Ys1]), qs-dl(B,Ys1,Zs)
3. qs-dl([ ], Ys, Ys) ←
4. part([X | Xs], Y, [X | Ls], Bs) ← X ≤ Y, part(Xs, Y, Ls, Bs)
5. part([X | Xs], Y, Ls, [X | Bs]) ← X > Y, part(Xs, Y, Ls, Bs)
6. part([ ], Y, [ ], [ ]) ←
```

By the first clause in the definition of `errant`, the output term `[X | Ys1]` is marked. Now, we have a marked output term `[X | Ys1]` in atom `qs-dl(L, Ys, [X | Ys1])` which unifies with the head of the non-unit clause 2 and `Zs` is the variable in the corresponding position in the head. Hence the output term `Zs` in the atom `qs-dl(B, Ys1, Zs)` is marked by the third clause in the definition of `errant`. □

The following example illustrates the second clause in the definition of `errant`.

Example 3 Consider the following program P.

```
moding:  q (in, out);  p1 (in, out, out);  and p2 (in, out, out)

1. q(X, Y) ← p1(X, Y, Y)
2. p1(X, Y, f(Z)) ← p2(X, Y, Z), ···
         ⋮
```

By the second clause in the definition of `errant`, the output term `Y` in both the output positions of the atom `p1(X, Y, Y)` is marked. Since this atom unifies with the head of the (non-unit) clause 2 and `Y` and `Z` are variables in the output positions in the head, the two output terms of the atom `p2(X, Y, Z)` are marked. Notice that we get a query $\leftarrow$ `p2(a, f(Z), Z)`, $\cdots$ with a non-variable non-ground term in an output position in an LD-derivation of $P \cup \{G\}$, where G is a neatly-moded query $\leftarrow$ `q(a, Y)`. □

The following theorem is useful in obtaining a sufficient condition for unification-freeness of non-neatly-moded programs.

Theorem 2 Let P be a well-moded program (not necessarily neatly-moded), G_0 be a neatly-moded goal and $G_0, G_1, \cdots, G_n$ be a LD-derivation of $P \cup \{G_0\}$. Then, the following holds for each goal G_m, $m \in [1, n]$: *If an output position* `pos` *in an atom (say A) in G_m is errant then* `errant(cl, at, pos)` *for some clause* `cl` *in P such that A is an instance of the atom* `atom(cl, at)`.

Proof : Induction on m.

Basis : $m = 0$. The theorem holds vacuously in this case as G_0 is neatly-moded.

Induction hypothesis : Assume that theorem holds for all G_m, $m \leq l$.

Induction step : $m = l + 1$. Let G_l be $\leftarrow A_1, \cdots, A_k$, clause (say, $\mathtt{cl}^{th}$ clause in P) $H \leftarrow B_1, \cdots, B_{k'}$ be the input clause and θ be the mgu used in deriving G_{l+1} from G_l. Then G_{l+1} is $\leftarrow B_1\theta, \cdots, B_{k'}\theta, A_2\theta, \cdots, A_k\theta$. We first show that the theorem holds for all the errant output positions in $B_1\theta, \cdots, B_{k'}\theta$ and then show for the positions in $A_2\theta, \cdots, A_k\theta$.

Let `pos` in atom $B_i\theta$ be an errant output position. There are two cases: (i) the term (say, $t\theta$) in this position is a non-variable non-ground term having a variable which has no producer in $B_1\theta, \cdots, B_{i-1}\theta$ and (ii) the term in this position is a variable (say, X) which occurs in more than one output positions of $B_i\theta$ and has no producer in $B_1\theta, \cdots, B_{i-1}\theta$. Assume for contradiction that `errant(cl, i, pos)` is not true.

In case (i), there are two subcases: (a) t is a non-variable term and (b) t is a variable (say, Y). Since `errant(cl, i, pos)` is not true, all the variables in t have at least one producer in $H, B_1, \cdots, B_{i-1}$ in subcase (a) and it is obvious that all the variables in $t\theta$ have at least one producer in $B_1\theta, \cdots, B_{i-1}\theta$ (since θ grounds all the variables in the input positions of H) and `pos` is not an errant position in $B_i\theta$. In subcase (b), since t is a variable (say, Y) and $t\theta$ is a non-variable non-ground term, it follows that Y occurs in an output position (say, `pos1`) of H such that either (1) A_1 has a non-variable in position `pos1` or (2) A_1 has a variable (say, Z) in position `pos1` and Z also occurs in another output position (say, `pos2`) such that H has a non-variable term in position `pos2`. In either of these two cases, `pos1` is an errant position in G_l. By induction hypothesis, `errant(cl2, i2, pos1)` for some clause such that A_1 is an instance of `atom(cl2, i2)`. Then it follows from the definition of the predicate `errant` that `errant(cl, i, pos)`, a contradiction.

Case (ii). Variable X occurs in another output position (say, `pos2`) of $B_i\theta$. In this case, either (1) a variable Z occurs in positions `pos` and `pos2` in B_i such that $Z\theta \equiv X$ or (2) a variable

$Z1$ occurs in position pos and a variable $Z2$ occurs in position pos2 such that $Z1\theta \equiv X$ and $X \in Var(Z2\theta)$. In subcase (1), it is obvious that errant(cl, i, pos). In subcase (2), variables $Z1$ and $Z2$ occur in output positions of H such that either (a) $Z1$ and $Z2$ both occur in a position (say pos3) of H such that the term in output position pos3 of A_1 is non-linear (and hence non-variable non-ground) term or (b) $Z1$ and $Z2$ occur in two different output positions (say pos3 and pos4) of H such that the terms in positions of A_1 share a variable. In both these situations, position pos3 is errant in A_1 and it follows from the induction hypothesis and the definition of errant that errant(cl, i, pos), a contradiction.

Now, we prove that the theorem holds for errant positions in $A_2\theta, \cdots, A_k\theta$. In fact, we prove that all the errant positions in $A_2\theta, \cdots, A_k\theta$ are also errant positions in $A_2, cdots, A_k$ (i.e., in G_l). Then by the induction hypothesis it follows that the theorem holds for these positions. Let pos in atom $A_i\theta$ be an errant output position. There are two cases: (i) the term (say, $t\theta$) in this position is a non-variable non-ground term having a variable which has no producer in $B_1\theta, cdots, B_{k'}\theta, A_2\theta, cdots, A_{i-1}\theta$ and (ii) the term in this position is a variable (say, X) which occurs in more than one output positions of $B_i\theta$ and has no producer in $B_1\theta, cdots, B_{k'}\theta, A_2\theta, cdots, A_{i-1}\theta$. Let us first consider case (i). Due to well-modedness, each variable $X \in t$ with $X\theta \not\equiv X$ occurs in output positions of A_1. By Lemma 3, each variable in $(Var(t\theta) - Var(t))$ occurs in output positions of $B_1\theta, \cdots, B_{k'}\theta$ and hence the output position pos is errant in G_{l+1} if and only if it is errant in G_l. Now consider case (ii). If X is a variable in G_l, then the theorem obviously holds for position pos by the induction hypothesis. If X is does not occur in G_l then there is a variable Y in an output position of A_1 such that $X \in Var(Y\theta)$. Then by Lemma 3, X should occur in output positions of $B_1\theta, \cdots, B_{k'}\theta$ and hence pos cannot be an errant position in G_{l+1}. □

Now, we give a set of sufficient conditions for unification-freeness of non-neatly-moded programs. The following two lemmas give sufficient conditions for replacing unification by two or three matchings.

Lemma 12 Let P be a well-moded program and G be a neatly-moded goal. Then, $P \cup \{G\}$ *is unification-free if each marked atom B in P satisfies the following property: for each head H unifiable with B, the terms of H in the output positions corresponding to the marked positions of B (i) have distinct variables and (ii) are disjoint from input terms of H.*

Proof: By the above theorem, if a selected atom has errant output positions then there is an atom in some clause in P such that the corresponding positions are marked.

Consider unification of a selected atom A and a unifiable head H. Let EO1 be the set of errant output positions. Since P and G are well-moded, A has ground inputs and the terms in output positions not in EO1 are either ground or distinct variables. By the hypothesis of the lemma, the terms in EO1 positions of H are distinct variables and disjoint from input terms. Take E_1 as the set of equations $\{ s_i = t_i \mid t_i$ is a term in input or EO1 position of A and s_i is the term in H in the corresponding position $\}$. It is easy to see that E_1 is solvable by matching (by Lemmas 5 and 6). Let θ_1 be the match (i.e., $s_i\theta_1 \equiv t_i$). Then, take E_2 as the set of equations $\{ s_i\theta_1 = t_i \mid t_i$ is a ground term in non-EO1 output position of A and s_i is the term in H in the corresponding position $\}$. The right-hand sides of E_2 are ground terms and hence E_2 is solvable by matching again. Let θ_2 be the match of E_2 (i.e., $s_i\theta_1\theta_2 \equiv t_i$). Now, take E_3 as the set of equations $\{ s_i\theta_1\theta_2 = t_i \mid t_i$ is a variable in non-EO1 output position of A and s_i is the term in H in the corresponding position $\}$. The right-hand sides of E_3 are distinct variables and they occur only once in A, hence E_3 is left-right disjoint. Therefore, E_3 is solvable by matching (by Lemma 5). Let θ_3 be the match of E_3 (i.e., $s_i\theta_1\theta_2 \equiv t_i\theta_3$). It is easy to see that $\theta_1\theta_2\theta_3$ is an idempotent mgu of A and H. □

Lemma 13 Let P be a well-moded program and G be a neatly-moded goal. Then, $P \cup \{G\}$ *is unification-free if each marked atom B in P satisfies the following property: for each head H unifiable with B, all the variables occurring in H in the output positions corresponding to the marked positions in B are included in $invar(H)$.*

Proof: Consider unification of a selected atom A and a unifiable head H. Let EO2 be the set of errant output positions. For similar reasons (as in the above lemma), A has ground inputs and the terms in output positions not in EO2 are either ground or distinct variables. Take E_1 as the set of equations $\{ s_i = t_i \mid t_i$ is a ground term in input/output position of A and s_i is the term in H in the corresponding position $\}$. Since each t_i is ground, E_1 is clearly solvable by matching. Let θ_1 be the match (i.e., $s_i\theta_1 \equiv t_i$). Then, take E_2 as the set of equations $\{ s_i\theta_1 = t_i \mid t_i$ is a term EO2 position of A and s_i is the term in H in the corresponding position $\}$ and E_3 as the set of equations $\{ s_i\theta_1 = t_i \mid t_i$ is a non-ground term (variable) in non-EO2 output position of A and s_i is the term in H in the corresponding position $\}$. Since variables in EO2 positions of H are included in $invar(H)$, left-hand sides of E_2 are ground terms and hence E_2 is solvable by matching. E_3 is solvable by matching as the right-hand sides of E_3 are distinct variables. Since E_2 and E_3 share no variable, it follows by Lemma 6 that $E_2 \cup E_3$ is solvable by matching (say with match θ_2, i.e., $s_i\theta_1 \equiv t_i\theta_2$). It is easy to see that $\theta_1\theta_2$ is an idempotent mgu of A and H. □

The following theorem combines the above two lemmas and gives a sufficient condition for replacing unification by a maximum of four matchings.

Theorem 3 Let P be a well-moded program and G be a neatly-moded goal. Then, $P \cup \{G\}$ *is unification-free if each marked atom B in P satisfies the following property: for each head H unifiable with B, the marked positions in B can be divided into two sets EO1 and EO2 such that (a) the terms in EO1-positions of H are distinct variables and disjoint from the input terms of H and (b) all the variables occurring in EO2-positions of H are included in $invar(H)$.*

Proof: Consider unification of a selected atom A and a unifiable head H. For similar reasons (as in Lemma 12), A has ground inputs and the terms in output positions not in EO1∪EO2 are either ground or distinct variables. Take E_1 as the set of equations $\{s_i = t_i \mid t_i$ is a term in input/EO1 position of A and s_i is the term in H in the corresponding position$\}$. By the proof of Lemma 12, E_1 is solvable by matching. Let θ_1 be the match (i.e., $s_i\theta_1 \equiv t_i$). Then, take E_2 as the set of equations $\{s_i\theta_1 = t_i \mid t_i$ is a term EO2 position of A and s_i is the term in H in the corresponding position$\}$. By the proof of Lemma 13, E_2 is solvable by matching. Let θ_2 be the match (i.e., $s_i\theta_1 \equiv t_i\theta_2$). Now, take E_3 as the set of equations $\{s_i\theta_1\theta_2 = t_i \mid t_i$ is a ground term in non-(EO1∪EO2) output position of A and s_i is the term in H in the corresponding position$\}$. Since right-hand side terms in E_3 are ground, E_3 is solvable by matching. Let θ_3 be the match (i.e., $s_i\theta_1\theta_2\theta_3 \equiv t_i$). Then, take E_4 as the set of equations $\{s_i\theta_1\theta_2\theta_3 = t_i\theta_1\theta_2\theta_3 \mid t_i$ is a non-ground term (variable) in non-(EO1∪EO2) output position of A and s_i is the term in H in the corresponding position$\}$. The terms in in non-(EO1∪EO2) output position of A are distinct variables and they occur only once in A, hence E_4 is left-right disjoint and $t_i\theta_1\theta_2\theta_3 \equiv t_i$. Therefore, E_4 is solvable by matching (by Lemma 5). Let θ_4 be the match (i.e., $s_i\theta_1\theta_2\theta_3 \equiv t_i\theta_4$). It is easy to see that $\theta_1\theta_2\theta_3\theta_4$ is an idempotent mgu of A and H. Therefore, $P \cup \{G\}$ is unification-free. □

Remark : The unification can be replaced by just three matchings in the above theorem/proof if the variables in EO1-positions of H do not occur in output positions of H corresponding to the ground output (non-marked) terms of A. In that case, E_3 could have been included in E_1.

Example 4 Consider the quick-sort program given in Example 2. Evaluation of the predicate `errant` marks output terms `[X | Ys1]` and `Zs` in clause 2.

$$\mathtt{qs-dl([X|Xs],Ys,Zs)} \leftarrow \mathtt{part(Xs,X,L,B),qs-dl(L,Ys,[X|Ys1]),qs-dl(B,Ys1,Zs)}$$

The heads of the clauses 2 and 3 are unifiable with the marked atoms. While considering clause 2 and the marked atom `qs-dl(L, Ys, [X|Ys1])`, we can put the output position in EO1 and apply the above theorem. While considering clause 3 and the marked atom `qs-dl(L, Ys, [X|Ys1])`, we can put the output position in EO2 and apply the above theorem. For the marked atom `qs-dl(B, Ys1, Zs)`, the situation is completely similar. Therefore, this program is unification-free for all neatly-moded goals, by the above theorem. □

Example 5 Consider the following permutation program.

`moding: perm (in, out); ap1 (in, in, out);` and `ap2 (out, out, in)`

1. `perm([ ], [ ])` ←
2. `perm(Xs, [X|Ys])` ← `ap2(X1s, [X | X2s], Xs), ap1(X1s, X2s, Zs), perm(Zs, Ys)`
3. `ap1([ ], Ys, Ys)` ←
4. `ap1([X | Xs], Ys, [X | Zs])` ← `ap1(Xs, Ys, Zs)`
5. `ap2([ ], Ys, Ys)` ←
6. `ap2([X | Xs], Ys, [X | Zs])` ← `ap2(Xs, Ys, Zs)`

Evaluation of the predicate `errant` marks the output term `[X | X2s]` in atom `ap2(X1s, [X | X2s], Xs)` of clause 2 and the output term `Ys` in atom `ap2(Xs, Ys, Zs)` of clause 6. Clauses 5 and 6 unify with these two marked atoms. While considering clause 5 and the marked atom `ap2(X1s, [X | X2s], Xs)`, we can put the marked output position in EO2 and apply the above theorem. While considering clause 6 and the marked atom `ap2(X1s, [X | X2s], Xs)`, we can put the marked output position in EO1 and apply the above theorem. For the marked atom `ap2(Xs, Ys, Zs)`, the situation is completely similar. Therefore, this program is unification-free for all neatly-moded goals, by the above theorem. □

It may be noted that unification-freeness of the above two programs `quick-sort-dl` and `permutation` cannot be proved using either of the above Lemmas 12 and 13.

4 Unification-freeness of Well-typed Programs

All the results in the previous section deals with well-moded programs and queries. One of the important property of well-moded programs/queries is that the selected atom always has ground inputs. However, it is easy to be convinced that many properties (e.g., termination, occur-check freeness etc) of LD-derivations starting with well-moded queries hold for derivations starting with queries having some non-ground inputs. For example, programs like `append, reverse, split, permutation` can run with any lists, not necessarily lists of ground elements. Basically, many programs work well for certain types of (not necessarily ground) inputs. This naturally brings the concept of *types* in logic programming. In this section, we study unification-freeness of programs using types.

4.1 Well-typed Prolog Programs and Generic expressions

Some of the definitions and results of [6, 3] we need are given below.

Definition 9 A *type* is a decidable set of terms closed under instantiation (substitution). By a *typed term,* we mean a construct of the form $s : S$, where s is a term and S is its type. Throughout the paper, we fix a specific set of types, denoted by *Types.*

The following types are of special interest:

U — the set of all terms,
$List$ — the set of all lists,
$BinTree$ — the set of all binary trees,
$Ground$ — the set of all ground terms.

Definition 10 Let p be a n-ary predicate symbol. A *type* for p is a mapping t_p from $\{1, \cdots, n\}$ to the set of types (i.e., *Types*). We say that $t_p(i)$ is the type associated with the position i of p and an atom $p(s_1, \cdots, s_n)$ is *correctly typed* in position i if $s_i \in t_p(i)$.

The following two definitions explain the notion of *well-typed* programs.

Definition 11 (type judgement)

- A *type judgement* is a statement of the form $\overline{s} : \overline{S} \ \Rightarrow \ \overline{t} : \overline{T}$.
- We say that the above *type judgement* is *true* and denote

$$\models \ \overline{s} : \overline{S} \ \Rightarrow \ \overline{t} : \overline{T},$$

  if for all substitutions θ, $\overline{s}\theta : \overline{S}$ implies $\overline{t}\theta : \overline{T}$.

To simplify the notation, we write an atom as $p(\mathbf{u} : \mathbf{U},\ \mathbf{v} : \mathbf{V})$ to mean that $\mathbf{u} : \mathbf{U}$ is a sequence of typed terms filling in the input positions of p and $\mathbf{v} : \mathbf{V}$ filling in the output positions. We call a construct of the form $p(\mathbf{u} : \mathbf{U},\ \mathbf{v} : \mathbf{V})$ a *typed atom.*

Definition 12 (well-typed programs)

- A goal $\leftarrow p_1(\mathbf{u_1} : \mathbf{U_1},\ \mathbf{v_1} : \mathbf{V_1}), \cdots, p_n(\mathbf{u_n} : \mathbf{U_n},\ \mathbf{v_n} : \mathbf{V_n})$ is *well-typed* if for $j \in [1, n]$

$$\models \ \mathbf{v_1} : \mathbf{V_1}, \cdots, \mathbf{v_{j-1}} : \mathbf{V_{j-1}} \ \Rightarrow \ \mathbf{u_j} : \mathbf{U_j}.$$

- A clause $p_0(\mathbf{v_0} : \mathbf{V_0},\ \mathbf{u_{n+1}} : \mathbf{U_{n+1}}) \ \leftarrow \ p_1(\mathbf{u_1} : \mathbf{U_1},\ \mathbf{v_1} : \mathbf{V_1}), \cdots, p_n(\mathbf{u_n} : \mathbf{U_n},\ \mathbf{v_n} : \mathbf{V_n})$ is *well-typed* if for $j \in [1, n+1]$

$$\models \ \mathbf{v_0} : \mathbf{V_0}, \cdots, \mathbf{v_{j-1}} : \mathbf{V_{j-1}} \ \Rightarrow \ \mathbf{u_j} : \mathbf{U_j}.$$

- A program is *well-typed* if every clause in it is *well-typed.*

The following lemma from Bronsard, Lakshman and Reddy [6] is very useful.

Lemma 14 *An LD-resolvent of a well-typed goal with a well-typed clause is well-typed.*

The following notion of generic expression and the lemma below (from [3]) are useful in establishing unification-freeness of well-typed programs.

Definition 13 A term t is a *generic expression* for type T if for every $s \in T$ disjoint with t the following property holds: *if s unifies with t then s is an instance of t.*

For example, a variable is a generic expression for every type T, every term is a generic expression for the type $Ground$ and $[\ ]$, $[X]$, $[H|T]$, $[X, Y|Z]$, $\cdots$ are generic expressions for the type $List$. Note that a generic expression for type T need not be a member of T — e.g., term $f(X)$ is a generic expression for the type $List$.

Lemma 15 (from [3]) Let A and H be two disjoint typed atoms with same predicate symbol and P be a set of positions such that (i) A is correctly typed in these positions and (ii) these positions in H are filled in by disjoint terms which are generic expressions for its position's type. Then, *$\overline{s} = \overline{t}$ is solvable by matching, where $\overline{s}$ (and $\overline{t}$) is the sequence of terms occurring in P-positions in H (resp. A). Moreover, if $\overline{s}$ and $\overline{t}$ are unifiable, there exists a substitution θ such that $domain(\theta) \subseteq Var(\overline{s})$ and $\overline{s}\theta \equiv \overline{t}$.*

The following definition is a generalization of the corresponding definition in [3].

Definition 14 A typed atom A is *safe for a set of positions* P if (i) the terms filling in these positions of A are generic expressions for the corresponding types and (ii) the terms filling in non-ground (type) positions (among P) of A are disjoint. A typed atom is *input-safe* if it is safe for its input positions.

4.2 Unification-freeness of nicely-moded well-typed programs

Our aim here is to study unification-freeness of nicely-moded well-typed programs. The class of nicely-moded well-typed programs is a superset of the class of simply-moded well-typed programs. In particular, a non-variable term can occur in output positions of the body of nicely-moded clauses, which is not allowed in simply-moded clauses.

Definition 15 (nicely-moded programs [4])

- We say that a Prolog goal $\leftarrow p_1(\overline{s}_1, \overline{t}_1), \cdots, p_n(\overline{s}_n, \overline{t}_n)$ is *nicely-moded* if the family of output terms $\overline{t}_1, \cdots, \overline{t}_n$ is linear (i.e., no variable occurs more than once in these terms) and for each $i \in [1, n]$
$$Var(\overline{s}_i) \cap (\bigcup_{j=i}^{n} Var(\overline{t}_j)) = \phi.$$

- We say that a Prolog clause $p_0(\overline{s}_0, \overline{t}_0) \leftarrow p_1(\overline{s}_1, \overline{t}_1), \cdots, p_n(\overline{s}_n, \overline{t}_n)$ is *nicely-moded* if the goal $\leftarrow p_1(\overline{s}_1, \overline{t}_1), \cdots, p_n(\overline{s}_n, \overline{t}_n)$ is nicely-moded and
$$Var(\overline{s}_0) \cap (\bigcup_{j=1}^{n} Var(\overline{t}_j)) = \phi.$$

- A Prolog program is nicely-moded if each clause in it is nicely-moded.

The following lemma from [4] establishes that nicely-modedness persists in LD-derivations.

Lemma 16 *An LD-resolvent of a nicely-moded goal with a nicely-moded clause is nicely-moded.*

Our basic idea is to compute and analyze all the possible non-variable output terms in selected atoms in LD-derivations of nicely-moded programs and give sufficient conditions for unification-freeness. To facilitate the application of Lemma 15, we identify the output terms which are guaranteed to be either correctly-typed or generic expressions. The following predicate `mark` captures all the possible non-variable output terms in LD-derivations of nicely-moded programs starting with simply-moded goals, predicate `ct` captures the positions which are guaranteed to have correctly typed terms and predicate `ge` captures the positions which are guaranteed to have generic expressions. We assume that the starting goals are simply-moded to avoid taking the (potentially infinite) queries of interest into account while evaluating the above predicates.

`mark(cl,at,pos)` **if** the term in that output position is a non-variable term.

`mark(cl,at,pos)` **if**
(i) the term in that output position is a variable (say, X),
(ii) X occurs in $\texttt{pos2}^{th}$ output term of head `head(cl)` and
(iii) there is a clause `cl2` in the program such that `mark(cl2, at2, pos2)` and `head(cl)` and `atom(cl2, at2)` are unifiable.

`ct(cl,at,pos)` **if** `mark(cl,at,pos)` and the term in that position is correctly-typed.

`ge(cl,at,pos)` **if**
(i) `mark(cl,at,pos)`,
(ii) the term (say, t) in that position is a generic expression for its type and
(iii) for each variable $X \in Var(t)$ and for each output position `pos2` in `head(cl)` containing X, the following holds:
if A is a body atom in the program which is unifiable with `head(cl)` then $\texttt{pos2}^{th}$ position of A is not marked.

`ge(cl,at,pos)` **if**
(i) `mark(cl,at,pos)`,
(ii) the term in that position is a variable (say, X) which occurs in a unique position `pos2` in `head(cl)` such that the following holds:
(1) the term in $\texttt{pos2}^{th}$ position of `head(cl)` is X,
(2) if A is a body atom `atom(cl2, at2)` (with `(cl, at)` $\neq$ `(cl2, at2)`) in the program which is unifiable with `head(cl)` **then**
`mark(cl2, at2, pos2)` implies `ge(cl2, at2, pos2)` and
(3) the asserted type of `pos2` is the same as that of `pos`.

$\texttt{ct}^*$`(cl, at, pos)` **if** `mark(cl,at,pos)` and
(i) the term in that position is a variable (say, X),
(ii) for each atom `atom(cl2, at2)` unifiable with `head(cl)` and for each position `pos2` $\in$ `POS` such that `mark(cl2, at2, pos2)`, one of the following holds
either (a) `ct(cl2, at2, pos2)` and $\models s : S \Rightarrow X : T$
or (b) $\texttt{ct}^*$`(cl2, at2, pos2)` and $\models s : S \Rightarrow X : T$
where `POS` is the set of output positions in `head(cl)` having X in them,
T is the asserted type of position `pos` in `atom(cl, at)` and
s is the term in `head(cl)` in position `pos2` and S is the asserted type of `pos2`.

By definition, all the instances of a correctly typed term are correctly typed. This ensures that all `ct` positions are correctly typed in the instances of input clauses used in LD-derivations. Since, an instance of a generic expression need not be a generic expression, we have to make sure that the variables in the term (designated as `ge`) do not get instantiated to non-variable terms. The definition of the predicate `ge` ensures this property. The predicate $\texttt{ct}^*$ captures the marked positions which are guaranteed to have either correctly typed terms or just variables in the instances of input clauses used in LD-derivations.

Example 6 The permutation program is a nicely-moded well-typed program with type *List* for all the arguments. Evaluation of the predicate `mark` marks the output term `[X|X2s]` in atom `ap2(X1s, [X|X2s], Xs)` of clause 2 and the output term `Ys` in atom `ap2(Xs, Ys, Zs)` of clause 6. The output term `[X | X2s]` is a generic expression and by the first clause in the definition of `ge`, we get `ge(2, 1, 2)` as `perm` has no marked output term. By the second clause in the definition of `ge`, we get `ge(6, 1, 2)`. □

Example 7 The following code is from the Prolog program PRESS developed by Bundy and Welham [7] at the university of Edinburgh for solving symbolic equations. The moding and typing information is as follows:

```
moding and typing information
solveequation (in:Equation, in:Alg-variable, out:Diff-list),
singleoccurrence (in:Alg-variable, in:Term),
position (in:Term, in:Term, out:Natlist),
position' (in:Nat, in:Term, in:Term, out:Natlist),
maneuversides (in:Nat, in:Equation, out:Equation),
compound (in:Term),
arg (in:Nat, in:Term, Term),
functor (in:Term, out:Function, out:Nat) and
isolate (in:Natlist, in:Equation, out:Diff-list).

solveequation(Eq, Var, Ss) ←
            singleoccurrence(Var, Eq), !,
            position(Var, Eq, [Side|Position]),
            maneuversides(Side, Eq, Eq1),
            isolate(Position, Eq1, Ss)

position(Term, Term, [ ]) ←
position(Term, Exp, Path) ←
            compound(Exp),
            functor(Exp, F, N),
            position′(N, Term, Exp, Path), !

position′(N, Term, Exp, Path) ←
            N > 0,
            arg(N, Exp, Arg),
            position(Term, Arg, Path′)
            Path ⟸ [N|Path′]
position′(N, Term, Exp, Path) ←
            N1 is N − 1,
            N1 > 0,
            position′(N1, Term, Exp, Path)
```

The other clauses in PRESS are simply-moded and well-typed, so we only have to analyze the above clauses. The infix predicate $\Longleftarrow$ is a built-in predicate for matching the first (output) argument with the second (input) argument.

Since we have a non-variable term `[Side|Position]` in the output position of atom `position(Var, Eq, [Side|Position])` in the first clause, we get `mark(1,2,3)`. By the second clause in the definition of `mark`, this induces `mark(3,3,4)` which in turn induces `mark(4,4,1)` and `mark(5,3,4)`.

Since the term `[Side|Position]` is a generic expression and none of its variables occur in the head, we get `ge(1,2,3)` by the first clause in the definition of `ge`. By the second clause in the definition of `ge`, this induces `ge(3,3,4)` which in turn induces `ge(4,4,1)` and `ge(5,3,4)`. □

The following theorem is useful in giving a sufficient condition for unification-freeness of nicely-moded well-typed programs. This theorem basically establishes correctness of the above definitions of predicates `mark`, `ct`, `ge` and `ct*`.

Theorem 4 Let P be a nicely-moded well-typed program, G be a simply-moded well-typed goal and $G_0, G_1, \cdots, G_n$ be an LD-derivation of $P \cup \{G\}$. *If an atom A in G_i, $i \in [0,n]$ has a non-variable output term (say, term t in position* `pos`*), then there is a clause $H \leftarrow B_1, \cdots, B_k$*

in P such that (i) A is an instance of some B_j, (ii) $\mathtt{mark(cl,j,pos)}$ *and (iii)* $H \leftarrow B_1, \cdots, B_k$ *is the* $\mathtt{cl}^{th}$ *clause in* P.
Further, (a) if $\mathtt{ct(cl,j,pos)}$ *or* $\mathtt{ct^*(cl,i,pos)}$*, then t is correctly typed, (b) if* $\mathtt{ge(cl,j,pos)}$*, then t is a generic expression for the asserted type of position* $\mathtt{pos}$.

Proof :Induction on i.

Basis : $i = 0$. The theorem is vacuously true as the initial goal G is simply-moded, i.e., all the output terms are distinct variables.

Induction Step : $i = l + 1$. Let G_l be the goal $\leftarrow A_1, \cdots, A_m$, $H \leftarrow B_1, \cdots, B_{m'}$ be the input clause and θ be the mgu applied in deriving G_{l+1} from G_l. Then, $G_{l+1} = B_1\theta, \cdots, B_{m'}\theta, A_2\theta, \cdots, A_m\theta$. Since G_l is nicely-moded, the output terms of $A_2, \cdots, A_m$ do not share any variable with A_1. Therefore, the output terms of $A_2\theta, \cdots, A_m\theta$ are same as those of $A_2, \cdots, A_m$ and hence the theorem holds for these atoms by the induction hypothesis. Now, consider a non-variable output term (say, term $t\theta$ in position $\mathtt{pos}$), of an atom $B_j\theta$, $j \in [1, m']$. There are two cases : (1) t is a non-variable term or (2) t is a variable.

Case (1). In this case, it is obvious that $\mathtt{mark(cl,j,pos)}$. If $\mathtt{ct(cl,j,pos)}$ then t is correctly typed and hence $t\theta$ is correctly typed as well. Since t is a non-variable term, the case $\mathtt{ct^*(cl,j,pos)}$ does not arise. If $\mathtt{ge(cl,j,pos)}$ then t is a generic expression by the definition of $\mathtt{ge}$. Since any variant of a generic expression is also a generic expression, it is enough to show that $t\theta$ is just a variant of t. If no variable of t occurs in the head of the clause, $t\theta \equiv t$. By nicely-modedness of the program, no variable of t occurs in input positions of the head. Consider a variable X in t which occurs in output positions of H. By (first clause in) the definition of $\mathtt{ge}$, these positions are not marked in every atom in the program unifiable with H. By induction hypothesis, A_1 has distinct variables in its unmarked output positions. Therefore, $X\theta$ is not a non-variable term and hence $t\theta$ is a variant of t.

Case (2). t is a variable (say, Y). By nicely-modedness, Y can only occur in output positions of H. Since output terms of A_1 do not share variables and $Y\theta$ is a non-variable term, A_1 should have a non-variable term in an output position (say, $\mathtt{pos2}$) such that Y occurs in that position in H. By induction hypothesis, there is an atom B in the body of a clause in P such that A_1 is an instance of B and $\mathtt{mark^*(cl2,j2,pos2)}$, where $\mathtt{pos2}$ is the corresponding position in B and B is $\mathtt{j2}^{th}$ atom in the body of clause $\mathtt{cl2}$. Since H unifies with A_1, it also unifies with B and by the definition of $\mathtt{mark}$, it follows that $\mathtt{mark(cl,j,pos)}$. If $\mathtt{ct(cl,j,pos)}$ then t is correctly typed and hence $t\theta$ is correctly typed as well. If $\mathtt{ct^*(cl,j,pos)}$ it follows by the definition of $\mathtt{ct^*}$ that (i) $\mathtt{ct(cl2,j2,pos2)}$ or (ii) $\mathtt{ct^*(cl2,j2,pos2)}$. By induction hypothesis, A_1 has a correctly typed term in position $\mathtt{pos2}$ and hence $Y\theta$ is a correctly typed term of its position (by the definition of $\mathtt{ct^*}$). If $\mathtt{ge(cl,j,pos)}$ it follows by (second clause in) the definition of $\mathtt{ge}$ that (i) $\mathtt{ge(cl2,j2,pos2)}$ and (ii) Y occurs in a unique position ($\mathtt{pos2}$) in the head (the first clause in the definition of $\mathtt{ge}$ is not applicable as atom B which is unifiable with H is marked in $\mathtt{pos2}$). By induction hypothesis, A_1 has a generic expression in position $\mathtt{pos2}$ and hence $Y\theta \equiv t\theta$ is a generic expression. □

Now, we are in a position to state our result on unification-freeness of nicely-moded well-typed programs.

Theorem 5 Let P be a nicely-moded well-typed program and G be a simply-moded well-typed goal. Then, *$P \cup \{G\}$ is unification-free if each clause in P is input-safe and each marked atom B (say,* $\mathtt{i}^{th}$ *atom in the body of clause* $\mathtt{cl}$*) in P satisfies the following property: for each head H unifiable with B, the marked output positions in B can be divided into five sets* OP1, OP2, OP3, OP4 *and* OP5 *such that* *(a)* H *is* safe *for positions* $(\mathrm{OP1} \cup \mathrm{input})$ *and* $\mathtt{ct(cl,i,pos)}$ *for each* pos *in OP1,*

(b) the terms in OP2 *positions of* H *are distinct variables disjoint from the terms in positions* $(\mathrm{input} \cup \mathrm{OP1})$,

(c) the terms in OP3 *positions of* H *are disjoint pure terms disjoint from the terms in positions* $(\mathrm{input} \cup \mathrm{OP1} \cup \mathrm{OP2})$,

(d) the terms in OP4 *positions of* H *are disjoint generic expressions disjoint from the terms in positions* $(\mathrm{input} \cup \mathrm{OP1} \cup \mathrm{OP2} \cup \mathrm{OP3})$ *and* $\mathtt{ct^*(cl,i,pos)}$ *for each* pos *in OP4 and*

(e) for each pos *in* OP5, *either (i)* $\models \overline{s} : \overline{S} \Rightarrow ground(t)$ *or (ii)* $\mathtt{ge(cl,i,pos)}$ *and* $\models \overline{s} : \overline{S} \Rightarrow t : T$, *where* t *is the term in position* pos, T *is the asserted type of* pos, $\overline{s}$ *is the sequence of terms in* $(\mathrm{input} \cup \mathrm{OP1})$ *positions and* $\overline{S}$ *is the sequence of their asserted types.*

Proof: Consider unification of a selected atom A and a unifiable head H. By Lemma 14, the input terms of A are correctly-typed. If all the output terms of A are variables, we can first match the input terms (note that H is input-safe) and then the output terms (i.e., solvable by matching). If there is a non-variable output term in A, there is a marked atom B in a clause in P such that A is an instance of B (by the above theorem). Let the marked output positions of B be partitioned into OP1, OP2, OP3, OP4 and OP5 satisfying the above requirements.

Take E_1 as the set of equations $\{s_i = t_i \mid s_i$ is a term in input/OP1 position of non-ground type in H and t_i is the term in A in the corresponding position$\}$ and E_2 as $\{s_i = t_i \mid s_i$ is a term in OP2 position of H and t_i is the term in A in the corresponding position$\}$. Since A is correctly typed in $(\mathrm{input} \cup \mathrm{OP1})$ positions, E_1 is solvable by matching due to Lemma 15 and E_2 is solvable by matching as the left-hand side terms are variables. By Lemma 6, $E_1 \cup E_2$ is solvable by matching with a match σ_1, i.e., $s_i\sigma_1 \equiv t_i$. Let E_1' be $\{s_i\sigma_1 = t_i \mid s_i$ is a term in input/OP1 position of ground type in H and t_i is the term in A in the corresponding position$\}$. By Lemma 5, E_1' is solvable by matching with a match σ_1', i.e., $s_i\sigma_1\sigma_1' \equiv t_i$.

Let E_3 be $\{s_i = t_i \mid t_i$ is a variable-term in OP3 position of A and s_i is the term in H in the corresponding position$\}$ and E_4 be $\{s_i = t_i \mid t_i$ is a variable-term in OP4 position of A and s_i is the term in H in the corresponding position$\}$. By nicely-modedness, the right side terms of $E_3 \cup E_4$ are distinct variables and hence, $E_3 \cup E_4$ is solvable by matching (say, with matching substitution σ_3, i.e., $s_i \equiv t_i\sigma_3$). Now, let E_3' be $\{s_i = t_i \mid t_i$ is a non-variable term in OP3 position of A and s_i is the term in H in the corresponding position$\}$ and E_4' be $\{s_i = t_i \mid t_i$ is a non-variable term in OP4 position of A and s_i is the term in H in the corresponding position$\}$. E_3' is solvable by matching as the left-hand side terms are disjoint pure terms and the right-hand side terms are non-variable terms. By the above theorem, the right-hand side terms of E_4' are correctly-typed and hence E_4' is solvable by matching as the left-hand side terms are disjoint generic expressions. By Lemma 6, $E_3' \cup E_4'$ is solvable by matching (say, with matching substitution σ_3', i.e., $s_i\sigma_3' \equiv t_i$). Let $\theta_1 \equiv \sigma_1\sigma_1'\sigma_3\sigma_3'$.

Now, take E_5 as $\{s_i\theta_1 = t_i \mid t_i$ is a term in OP5 position of A and s_i is the term in H in the corresponding position$\}$ and E_6 as $\{s_i\theta_1 = t_i \mid t_i$ is a term (i.e., variable) in unmarked output position of A and s_i is the term in H in the corresponding position$\}$. Due to nicely-modedness, the right-hand side terms of E_5 and E_6 disjoint and $E_5 \cup E_6$ is left-right disjoint. Due to condition (e), we can partition E_5 into two sets E_5' and E_5'' such that (i) the left-hand side terms of E_5' are ground and (ii) the left-hand side terms of E_5'' are correctly-typed and the right-hand side terms are disjoint generic expressions. By Lemma 5, E_5' is solvable by matching. By Lemma 15, E_5'' is solvable by matching. E_6 is solvable by matching as its right-hand side terms are distinct variables. Now, $E_5 \cup E_6$ is solvable by matching by Lemma 6. Let θ_2 be its matching substitution (i.e., $s_i\theta_1 \equiv t_i\theta_2$). It is easy to see that $\theta_1\theta_2$ is an idempotent mgu of A and H. $\square$

Example 8 Consider the permutation program given in the previous example. It is an input-safe nicely-moded well-typed program with type *List* for all the arguments and the evaluation of predicates `mark`, `ct`, `ge` and `ct`* yields the following facts.

`mark(2, 1, 2)`, `mark(6, 1, 2)`

`ge(2, 1, 2)`, `ge(6, 1, 2)`

This marks the non-variable output term `[X | X2s]` in atom `ap2(X1s, [X | X2s], Xs)` of clause 2 and `Ys` in atom `ap2(Xs, Ys, Zs)` of clause 6. Heads of clauses 5 and 6 in P unify with these two marked atoms. While considering clause 5 and the marked atom `ap2(X1s, [X | X2s], Xs)`, we can put the marked output position in OP5 and apply the above theorem. While considering clause 6 and the marked atom `ap2(X1s, [X | X2s], Xs)`, we can put the marked output position in OP2 and apply the above theorem. For atom `ap2(Xs, Ys, Zs)`, the situation is completely similar. Therefore, this program is unification-free for all simply-moded well-typed goals, by the above theorem. □

Example 9 (Unification-freeness of PRESS)
Consider the program PRESS given in Example 7. It is an input-safe nicely-moded well-typed program. The evaluation of predicates `mark`, `ct`, `ge` and `ct`* yields the following facts.

`mark(1, 2, 3)`, `mark(3, 3, 4)`, `mark(4, 4, 1)`, `mark(5, 3, 4)` and

`ge(1, 2, 3)`, `ge(3, 3, 4)`, `ge(4, 4, 1)`, `ge(5, 3, 4)`.

While considering the marked atom `position(Var, Eq, [Side|Position])` in the first clause and the head of the third clause, we can put this position in OP2 and apply Theorem 5. Now, consider the marked atom `position'(N, Term, Exp, Path)`. This unifies with the heads of clause 4 and 5. Since the output term (variable `path`) do not share variables with input terms in both these heads, we can put this position in OP2 and apply Theorem 5. The case with the marked atom `position'(N1, Term, Exp, Path)` is the same. Finally consider the marked atom `Path` $\Longleftarrow$ `[N|Path']`. By well-typedness, the input term will be correctly-typed and can be matched onto the generic expression (in the output position). Therefore, this program is unification-free for all simply-moded well-typed goals, by Theorem 5. □

5 Practical Implications

Now, we discuss the practical implications of the theoretical results presented in the previous sections. In particular, we study how to improve the WAM code of Prolog programs by replacing unification by matching. The WAM instructions related to the unification operation are `get-structure, get-list, get-value, unify-value` and `unify-local-value` (see [17] for the introduction of these instructions and [1] for the details of their implementation). The instructions `get-value, unify-value` and `unify-local-value` explicitly call the *unify*-operation and the instructions `get-structure, get-list` set the moding of the *unify*-operation to `read` or `write`. In `read` mode, data on the heap is matched against while data on the heap is constructed in `write` mode. Informally, variables in the head of the input clause get instantiated in `read` mode while in `write` mode, variables in the selected atom also get instantiated.

From the proof of our (main) Theorem 5, it is clear that variables in OP1, OP2 and input positions of the *head* and variables in OP5 and unmarked output positions of the *selected atom* get instantiated in the unification. However, the direction of the instantiation is not clear for the terms in OP3 and OP4 positions (in some positions, variables in the head get instantiated, while variables in the selected atom get instantiated in other positions). We propose to use this knowledge about the direction of the instantiation in improving the WAM code.

First, consider programs in which `head` of every clause satisfy the following property: the partition of marked output positions (implied by Theorem 5) of all the `body-atoms` unifiable with the `head` is the same. For such programs, the direction of the instantiation for OP1, OP2, OP5, input and unmarked output positions of each head is clear at the compile time itself. The

WAM code can be improved by suitably changing the instructions `get-structure, get-list, get-value, unify-value` and `unify-local-value` to use a *match*-operation instead of the *unify*-operation in OP1, OP2, OP5, input and unmarked output positions.

For programs not satisfying the above property, i.e., programs in which the partition is not same for all the `body-atoms` unifiable with a `head`, we need to have some book-keeping. With each head, we associate the list of unifiable body-atoms and their partitions of marked positions. At run time, this book-keeping information can be used to replace *unify*-operation with *match*-operation and to decide the direction of the matching.

6 A comparison with related works

Our results are essentially generalizations of the recent results of Apt and Etalle [3]. Programs `quick-sort-dl, flatten-dl, permutation` are examples of well-moded programs which can be certified to be unification-free by our results but not by the earlier results. Programs `permutation` and PRESS (a system developed at Edinburgh for solving symbolic equations) are examples of well-typed programs (i.e., programs running on non-ground inputs) which can be certified to be unification-free by our results but not by the results of [3].

Contemporarily, Etalle [9] extended the results of [3] to handle `quick-sort-dl, flatten-dl, permutation`. Etalle [9] considers restrictions which ensure that output terms of the selected atoms are variables and pure terms and studies unification-freeness. He uses (i) moding annotations which assigns `input, output` and `unknown` to the predicate positions and (ii) types (namely, variables and pure-terms) which are not closed under instantiation. For programs (without `unknown` moding) satisfying the above restrictions, the computation of our predicates `errant, mark, ge, ct` and `ct*` is very simple (in fact, the predicates `ct` and `ct*` will be empty) and it is easy to establish unification-freeness using our results as well. However, our approach and that of Etalle [9] are incomparable as we allow arbitrary terms in output positions and [9] additionally uses the moding `unknown`. The following two examples illustrate the incomparability and advantages of the two approaches.

Allowing arbitrary terms in output positions is very useful in handling programs where a specific form of output is expected from the execution of an atom in the body and our results can handle such programs which are beyond the scope of the results of [9]. The need to expect output terms to be of a specific form arises very naturally in many application domains and it is useful to have results which can handle such situations.

Example 10 Consider the following nicely-moded well-typed program (schema) which expects the output term of q(...) to be a list with at least `k` elements.

$$\mathtt{p}(\cdots) \leftarrow \cdots, \mathtt{q}(\cdots, [\mathtt{X}_1, \mathtt{X}_2, \cdots, \mathtt{X}_k|\mathtt{X}]), \cdots$$

$$\vdots$$

$$\mathtt{q}(\mathtt{H}, \mathtt{T}, [\mathtt{H}|\mathtt{T}]) \leftarrow \mathtt{Body}$$

For simplicity assume that the output terms in bodies of clauses not shown above are variables. Let `(in:U, in:List, out:List)` be the mode and type of `q`.

The output term $[\mathtt{X}_1, \mathtt{X}_2, \cdots, \mathtt{X}_k|\mathtt{X}]$ of $\mathtt{q}(\cdots)$ in the body of the above clause is a generic expression for type `List` and the evaluation of our predicates `mark, ct, ge` and `ct*` marks this term as `ge`. By our Theorem 5, we conclude that this program is unification-free — condition (e) of the theorem holds if we put this position in OP5.

In Etalle's approach, the third position of q cannot be an output position as this position has a term which is neither a variable nor a pure term. An alternative is to give moding **unknown** (i.e., this position is a U-position). In this case, the head q(H, T, [H|T]) is not U-safe and hence his results cannot certify this program as unification-free. See [9] for the notions of U-position and U-safe. □

However, the moding **unknown** does help in concluding unification-freeness of some programs as illustrated by the following example.

Example 11 Consider the following `member` program.

```
member(X, [X|Xs]) ←
member(Y, [X|Xs]) ← member(Y, Xs)
```

Our results as well as the results of [9] certify this program to be unification-free for all the natural modes and types. However, our results cannot certify this program to be unification-free for (some what artificial) type `member (out:pure-term, unknown:U)` whereas the results of [9] can certify.

Since, we do not have moding `unknown`, we may take the second position of `member` as either input or output position. If it is taken as output position, the program is not well-moded (nor well-typed). So, only alternative is to take it as an input position with type U. In this case, the program is not input-safe (as `[X|Xs]` is not a generic expression for type U) and hence our results do not apply. □

It may be noted that the above program can be handled by our results if we slightly change the notion of input-safety by allowing pure-terms instead of variables in the positions of type `U` (note that variables are the only generic-expressions of `U` and the present notion allows only variables in these positions). However, with this change, the direction of matching for input positions will be not clear at compile time as the matching of a pure-term and an arbitrary term can be in either direction (if the arbitrary term is non-variable, matching instantiates the pure term, otherwise it instantiates the arbitrary term). Throughout the paper, we attempt to make the direction of matching as clear as possible at compile time to capitalize the results in improving WAM code of programs. For this reason, we prefer the existing notion of input-safety.

Another related and independent work is that of Marchiori [13]. The objective of [13] is quite different from ours and [3, 9] and is concerned with theoretical aspects of local properties (properties which can be verified clause by clause) that ensure unification-freeness of well-moded programs. The results are very interesting and there is a need to pursue this direction further as the present results only deal with well-moded programs. Some programs inherently need a static/global analysis (hence they are beyond the scope of results based on local properties) to conclude unification-freeness and our results are very useful is such situations.

7 Conclusion

In this paper, a static analysis is proposed to study unification-freeness of well-moded and well-typed Prolog programs. It is proved that any well-moded Prolog program can be transformed into an equivalent unification-free program and showed that the resulting program can be inefficient than the original program when backtracking is needed, thereby establishing the importance of techniques for verifying unification-freeness of a given program to facilitate an efficient execution. Some sufficient conditions for unification-freeness are presented and the application of the results in improving WAM code of Prolog programs is discussed. A comparison with the related works is provided.

References

[1] H. Ait-Kaci (1990), *The WAM: a (real) tutorial*, Tech. Rep., DIGITAL, Paris.

[2] K.R. Apt (1990), *Logic Programming*, in J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. **B**, pp. 493-574, North-Holland.

[3] K.R. Apt and S. Etalle (1993), *On the unification free Prolog programs*, Proc. MFCS'93, Lect. Notes in Comp. Sci., **711**, pp. 1-19, Springer-Verlag.

[4] K.R. Apt and A. Pellegrini (1992), *Why the occur-check is not a problem*, Proc. of PLILP'92, Lect. Notes in Comp. Sci., **631**, pp. 69-86, Springer-Verlag.

[5] I. Attali and P. Franchi-Zannettacci (1988), *Unification-free execution of TYPOL programs by semantic attribute evaluation*, Proc. Joint Intl. Conf. and Symp. on Logic Prog., JICSLP'88, pp. 160-177, MIT Press.

[6] F.Bronsard, T.K. Lakshman and U.S. Reddy (1992), *A framework of directionality for proving termination of logic programs*, Proc. Joint Intl. Conf. and Symp. on Logic Prog., JICSLP'92, pp. 321-335, MIT Press.

[7] A. Bundy and B. Welham (1981), *Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation*, Artificial Intelligence **16**, pp. 189-121.

[8] P. Deransart and J. Maluszinski (1985), *Relating Logic Programs and Attribute Grammars*, Journal of Logic Programming **2**, pp. 119-155.

[9] S. Etalle (1994), *More (on) unification-free Prolog programs*, manuscript, CWI Amsterdam, September.

[10] M.R.K. Krishna Rao, D. Kapur and R.K. Shyamasundar (1991), *A Transformational methodology for proving termination of logic programs*, Proc. of Computer Science Logic, CSL'91, Lect. Notes in Comp. Sci., **626**, pp. 213-226, Springer-Verlag.

[11] M.R.K. Krishna Rao and R.K. Shyamasundar (1994), *Unification-free execution of well-moded Prolog programs*, Technical Report, TIFR Bombay, April.

[12] J. W. Lloyd (1987), *Foundations of Logic Programming*, Springer-Verlag.

[13] M. Marchiori (1994), *Localization of unification freedom through matching directions*, Proc. Intl. Logic Prog. Symp., ILPS'94, MIT Press.

[14] J. Maluszinski and H.J. Komorowski (1985), *Unification-free execution of Prolog programs*, Proc. IEEE Symp. on Logic Programming, pp. 78-86.

[15] L. Sterling and M. Codish (1987), *PRESSing for parallelism: a Prolog program made concurrent*, in E. Shapiro (ed.), *Concurrent Prolog: collected papers*, Vol. **2**, pp. 304-349, MIT press.

[16] L. Sterling and E. Shapiro (1986), *Art of Prolog*, MIT Press.

[17] D.H.D. Warren (1983), *An abstract Prolog instruction set*, Tech. Rep., SRI International.