On the Occur-Check-Free PROLOG Programs

KRZYSZTOF R. APT
CWI and University of Amsterdam
and
ALESSANDRO PELLEGRINI
Università di Padova

In most PROLOG implementations, for efficiency occur-check is omitted from the unification algorithm. This paper provides natural syntactic conditions that allow the occur-check to be safely omitted. The established results apply to most well-known PROLOG programs, including those that use difference lists, and seem to explain why this omission does not lead in practice to any complications. When applying these results to general programs, we show their usefulness for proving absence of floundering. Finally, we propose a program transformation that transforms every program into a program for which only the calls to the built-in unification predicate need to be resolved by a unification algorithm with the occur-check.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.4 [**Programming Languages**]: Processors—*preprocessors*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming*

General Terms: Languages

Additional Key Words and Phrases: Moded programs, occur-check problem, PROLOG programs, unification algorithm

1. INTRODUCTION

The occur-check is a special test used in the unification algorithm. In most PROLOG implementations, it is omitted for efficiency. This omission affects the unification algorithm and introduces a possibility of divergence, or may yield incorrect results. This is obviously an undesired situation. This problem

The work of K. R. Apt was partly supported by ESPRIT Basic Research Action 6810 (Compulog 2). This research was partly carried out during A. Pellegrini's stay at the Centre for Mathematics and Computer Science, Amsterdam. His stay was supported by the 2060th District of the Rotary Foundation, Italy. A shorter version of this paper appeared as Apt and Pellegrini [1992].
Authors' addresses: K. R. Apt, CWI, P.O. Box 94079, 1090 GB Amsterdam, and Faculty of Mathematics and Computer Science, University of Amsterdam, 1018 TV Amsterdam, The Netherlands; A. Pellegrini, Dipartimento di Matematica Pura ed Applicata, Università di Padova, 35131 Padova, Italy.

was studied in the literature under the name of the *occur-check problem* (see, e.g., Plaisted [1984] and Deransart and Maluszynski [1985]).

The aim of this paper is to provide easy-to-check syntactic conditions that ensure that for PROLOG programs the occur-check can be safely omitted. We use here a result of Deransart et al. [1991] and build upon it within the context of moded programs. This allows us to extend the results of Deransart and Maluszynski [1985], to generalize the arguments of Chadha and Plaisted [1994], and to offer a uniform presentation. Additionally, the results of the former paper needed here are proved directly, without resorting to the techniques of the attribute grammars theory. We also consider general programs and show the usefulness of our approach for proving the absence of floundering. Finally, we show how the problem of inserting occur-checks in a program execution can be resolved by means of a program transformation that inserts calls of the built-in unification predicate into the program text. The obtained results apply to most well-known PROLOG programs.

In this paper we need a slightly more liberal definition of an SLD-derivation, according to which the selection of the atom in the current goal is combined with the selection of the input clause used to resolve this atom. Then an SLD-derivation fails if the selected atom does not unify with the head of the input clause selected to resolve it.

To see the difference with the customary definition, consider the program $\{p(0) \leftarrow , p(x) \leftarrow\}$. According to our definition, the goal $\leftarrow p(s(0))$ is not only the root of an SLD-refutation, but also a root of an immediately failing SLD-derivation (when the first clause is selected).

In what follows we study logic programs executed by means of the *LD-resolution*, which consists of the SLD-resolution combined with the leftmost selection rule. An SLD-derivation in which the leftmost selection rule is used is called an *LD-derivation*. We allow in programs various first-order built-ins, like $=$, $\leq$, $>$, etc., and assume that they are resolved in the way conforming to their interpretation.

Throughout the paper we use the standard notation of Lloyd [1987] and Apt [1990]. In particular, given a syntactic construct E (e.g., a term, an atom, or a set of equations) we denote by $Var(E)$ the set of the variables appearing in E. Given a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, we denote by $Dom(\theta)$ the set of variables $\{x_1, \ldots, x_n\}$, by $Range(\theta)$ the set of terms $\{t_1, \ldots, t_n\}$, and by $Ran(\theta)$ the set of variables appearing in $\{t_1, \ldots, t_n\}$. Finally, we define $Var(\theta) = Dom(\theta) \cup Ran(\theta)$.

Recall that a substitution θ is called a *grounding* if $Ran(\theta)$ is empty and is called a *renaming* if it is a permutation of the variables in $Dom(\theta)$. Given a substitution θ and a set of variables V, we denote by $\theta \mid V$ the substitution obtained from θ by restricting is domain to V.

2. OCCUR-CHECK-FREE PROGRAMS

We start by recalling a unification algorithm due to Martelli and Montanari [1982]. We use the notions of sets and of systems of equations interchangeably. Two atoms can unify only if they have the same relation symbol. With

two atoms $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$ to be unified, we associate the set of equations

$$\{s_1 = t_1, \ldots, s_n = t_n\}.$$

In the applications we often refer to this set as $p(s_1, \ldots, s_n) = p(t_1, \ldots, t_n)$. The algorithm operates on such finite sets of equations. A substitution θ such that $s_1\theta = t_1\theta, \ldots, s_n\theta = t_n\theta$ is called a *unifier* of the set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$. Thus, the set of equations $E = \{s_1 = t_1, \ldots, s_n = t_n\}$ has the same unifiers as the atoms $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$.

A unifier θ of a set of equations E is called a *most general unifier* (in short, *mgu*) of E if it is more general than all unifiers of E. An mgu θ of a set of equations E is called *relevant* if $Var(\theta) \subseteq Var(E)$.

Two sets of equations are called *equivalent* if they have the same set of unifiers, and a set of equations is called *solved* if it is of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$, where the x_i's are distinct variables and none of them occurs in a term t_j. The interest in solved sets of equations is revealed by the following lemma:

LEMMA 2.1. *If* $E = \{x_1 = t_1, \ldots, x_n = t_n\}$ *is solved, then* $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ *is a relevant mgu of* E.

We call θ the *unifier determined by* E. Thus, to find an mgu of two atoms it suffices to transform the associated set of equations into an equivalent one that is solved. The following algorithm does it, if this is possible, and otherwise halts with failure:

Martelli–Montanari Algorithm. Nondeterministically choose from the set of equations an equation of a form below, and perform the associated action:

(1)	$f(x_1, \ldots, s_n) = f(t_1, \ldots, t_n)$	*replace by the equations* $s_1 = t_1, \ldots, s_n = t_n$,
(2)	$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ where $f \not\equiv g$	*halt with failure*,
(3)	$x = x$	*delete the equation*,
(4)	$t = x$ where t is not a variable	*replace by the equation* $x = t$,
(5)	$x = t$ where $x \not\equiv t$, x does not occur in t, and x occurs elsewhere	*perform the substitution* $\{x/t\}$ *in every other equation*,
(6)	$x = t$ where $x \not\equiv t$ and x occurs in t	*halt with failure*.

The algorithm terminates when no action can be performed or when failure arises. To keep the formulation of the algorithm concise, we identified constants with 0-ary functions. Thus, action (2) includes the case of two different constants. The following theorem holds (see Martelli and Montanari [1982]):

UNIFICATION THEOREM 2.2. *The Martelli–Montanari algorithm always terminates. If the original set of equations* E *has a unifier, then the algorithm successfully terminates and produces a solved set of equations determining a relevant mgu of* E; *otherwise, it terminates with failure.*

The Martelli–Montanari algorithm does not generate all mgu's of a set of equations E, but the following lemma, proved in Lassez et al. [1988], will allow us to cope with this peculiarity.

LEMMA 2.3. *Let θ_1 and θ_2 be mgu's of a set of equations. Then for some renaming η, we have $\theta_2 = \theta_1\eta$.*

Finally, the following lemma allows us to search for mgu's in an iterative fashion:

LEMMA 2.4. *Let E_1 and E_2 be two sets of equations. Suppose that θ_1 is a relevant mgu of E_1 and θ_2 is a relevant mgu of $E_2\theta_1$. Then $\theta_1\theta_2$ is a relevant mgu of $E_1 \cup E_2$. Moreover, if $E_1 \cup E_2$ is unifiable then such a θ_1 exists, and for any such θ_1, an appropriate θ_2 exists, as well.*

PROOF. If e is an equation of E_1, then it is unified by θ_1 and so a fortiori by $\theta_1\theta_2$. If e is an equation of E_2, then $e\theta_1$ is an equation of $E_2\theta_1$. Thus, $e\theta_1$ is unified by θ_2, and consequently, e is unified by $\theta_1\theta_2$. This proves that $\theta_1\theta_2$ is a unifier of $E_1 \cup E_2$. Moreover, $Var(\theta_1\theta_2) \subseteq Var(\theta_1) \cup Var(\theta_2) \subseteq Var(E_1) \cup Var(E_2\theta_1) \subseteq Var(E_1) \cup Var(E_2) \cup Var(\theta_1) \subseteq Var(E_1 \cup E_2)$, so $\theta_1\theta_2$ is relevant.

Now let η be a unifier of $E_1 \cup E_2$. By the choice of θ_1, there exists a substitution λ_1 such that $\eta = \theta_1\lambda_1$. Thus, λ_1 is a unifier of $(E_1 \cup E_2)\theta_1$ and a fortiori of $E_2\theta_1$. By the choice of θ_2 for some λ_2, we have $\lambda_1 = \theta_2\lambda_2$. Thus, $\eta = \theta_1\lambda_1 = \theta_1\theta_2\lambda_2$. This proves that $\theta_1\theta_2$ is an mgu of $E_1 \cup E_2$.

Finally, note that if $E_1 \cup E_2$ is unifiable then a fortiori E_1 is unifiable, and Unification Theorem 2.2 tells us that a relevant mgu θ_1 for E_1 is produced by the Martelli–Montanari algorithm. The previously inferred existence of λ_1 implies that, for such a θ_1, $E_2\theta_1$ is unifiable, and again, the Martelli–Montanari algorithm can be used to produce for this set a relevant mgu θ_2. □

Return now to the Martelli–Montanari algorithm. The test "x does not occur in t" in action (5) is called the *occur-check*. In most PROLOG implementations, the occur-check is omitted. Recall that this omission can in some cases bring the cost of unification from linear time down to constant time. An example is the concatenation of the lists by means of the difference-list representation. (For a thorough analysis of the time complexity of the unification algorithm with and without the occur-check, see Albert et al. [1993]. By omitting the occur-check in (5) and deleting action (6) from the Martelli–Montanari algorithm, we are still left with two options, depending on whether the substitution $\{x/t\}$ is performed in t itself. If it is, the divergence can result, because if x occurs in t then x occurs in $t\{x/t\}$. If it is not (as in the case of the modified version of the algorithm just mentioned), then an incorrect result can be produced, as in the case of the single equation $x = f(x)$, which yields the substitution $\{x/f(x)\}$.

None of these alternatives is desirable. It is natural then to seek conditions that guarantee that, in absence of the occur-check, in all PROLOG evalua-

tions of a given goal w.r.t. a given program, unification is correctly performed. This leads us to the following notion due to Deransart et al. [1991].

Definition 2.5. A set of equations E is *not subject to occur-check* (*NSTO*, in short) if action (6) cannot be performed in any execution of the Martelli–Montanari algorithm started with E.

We now introduce the key definition of the paper:

Definition 2.6.

—Let ξ be an LD-derivation. Let A be an atom selected in ξ, and let H be the head of the input clause selected to resolve A in ξ. Suppose that A and H have the same relation symbol. Then we say that the system $A = H$ *is considered in* ξ.

—Suppose that all systems of equations considered in the LD-derivations of $P \cup \{G\}$ are NSTO. Then we say that $P \cup \{G\}$ is *occur-check free*.

This definition assumes a specific unification algorithm, but allows us to derive precise results. Moreover, the nondeterminism built into the Martelli–Montanari algorithm allows us to model executions of various other unification algorithms, including Robinson's algorithm (see, e.g., Albert et al. [1993]). In contrast, no specific unification algorithm in the definition of the LD-resolution is assumed.

By Theorem 2.2, if a considered system of equations is unifiable, then it is NSTO as well. Thus, the property of being occur-check free rests exclusively upon those considered systems that are not unifiable. As in the definition of the occur-check freedom, *all* LD-derivations of $P \cup \{G\}$ are considered, it follows that all systems of equations that can be considered in a possibly backtracking PROLOG evaluation of a goal G w.r.t. the program P are taken into account.

In Deransart et al. [1991], a related concept of an NSTO program is studied that essentially states that, independently of the selection rule and the resolution strategy chosen, all considered systems are NSTO. The definition of the occur-check freedom refers to the leftmost selection rule, and the results we obtain usually cannot be extended to those dealing with NSTO programs.

The aim of this paper is to offer simple syntactic conditions that imply that $P \cup \{G\}$ is occur-check free. As expected, the property of being occur-check free is undecidable (see Deransart and Maluszynski [1985], and, for a strengthened version, the Appendix). On the other hand, the problem of determing whether a set of equations is NSTO is decidable. In fact, Apt et al. [1994] recently proved that this problem is CoNP-hard.

For further analysis we need the following concepts:

Definition 2.7.

—We call a family of terms (resp., an atom) *linear* if every variable occurs at most once in it.

—We call a set of equations *left linear* (resp., *right linear*) if the family of terms formed by their left-hand (resp., right-hand) sides is linear.

Thus, a family of terms is linear iff no variable has two distinct occurrences in any term and no two terms have a variable in common.

Definition 2.8. Let E be a set of equations. We denote by $\rightarrow_E$ the following relation defined on the elements of E: $e_1 \rightarrow_E e_2$ iff the left-hand side of e_1 and the right-hand side of e_2 have a variable in common.

In particular, if a variable occurs both in the left-hand and right-hand sides of an equation e of E, then $e \rightarrow_E e$. The following result, due to Deransart et al. [1991], will be helpful in the sequel.

NSTO LEMMA 2.9. *Suppose that the equations in E can be oriented in such a way that the resulting system F is left linear and the relation* $\rightarrow_F$ *is cycle free. Then E is NSTO.*

Note that the converse of this lemma is not true—just take $E = \{f(x) = g(x)\}$. The original formulation of this lemma is slightly stronger, but for our purposes the above version is sufficient.

3. MODED PROGRAMS

For further analysis we introduce also modes, first considered in Mellish [1981] and more extensively studied in Reddy [1984] and in Dembinski and Maluszynski [1985].

Definition 3.1. Consider an n-ary relation symbol p. By a *mode* for p, we mean a function d_p from $\{1, \ldots, n\}$ to the set $\{+, -\}$. If $d_p(i) = +$, we call i an *input position* of p, and if $d_p(i) = -$, we call i an *output position* of p (both w.r.t. d_p).

We write d_p in the more suggestive form $p(d_p(1), \ldots, d_p(n))$. By *moding* we mean a collection of modes, each for a different relation symbol.

Intuitively, the modes indicate how the arguments of a relation should be used, though the distinction between the input and output positions is not clear when all positions in an atom of a goal are filled in by compound terms.

The definition of moding assumes one mode per relation in a program. Multiple modes may be obtained by simply renaming the relations. From now on we assume that *every considered relation* has a mode associated with it. This will allow us to discuss input positions and output positions of an atom.

We now introduce the following concepts:

Definition 3.2.

—An atom is called *input* (resp., *output*) *linear* if the family of terms occurring in its input (resp., output) positions is linear.

—An atom is called *input-output disjoint* if the family of terms occurring in its input positions has no variable in common with the family of terms occurring in its output positions.

The following lemma is crucial:

NSTO VIA MODES LEMMA 3.3. *Consider two atoms A and H with the same relation symbol. Suppose that*

—*they have no variable in common,*
—*one of them is input-output disjoint, and*
—*one of them is input linear and the other is output linear.*

Then $A = H$ is NSTO.

PROOF. Suppose first that A is input-output disjoint and input linear and that H is output linear. Let $i_1^A, \ldots, i_m^A$ (resp., $i_1^H, \ldots, i_m^H$) be the terms filling in the input positions of A (resp., H), and let $o_1^A, \ldots, o_n^A$ (resp., $o_1^H, \ldots, o_n^H$) be the terms filling in the output positions of A (resp., H).

The system under consideration is

$$E = \{i_1^A = i_1^H, \ldots, i_m^A = i_m^H, o_1^A = o_1^H, \ldots, o_n^A = o_n^H\}.$$

Reorient it as follows:

$$F = \{i_1^A = i_1^H, \ldots, i_m^A = i_m^H, o_1^H = o_1^A, \ldots, o_n^H = o_n^A\}.$$

By assumption, A and H have no variable in common. This implies that

—F is left linear (because, additionally, A is input linear and H is output linear), and
—the equations $i_j^A = i_j^H$ have no successor in the $\rightarrow_F$ relation and the equations $o_j^H = o_j^A$ have no predecessor (because, additionally, A is input-output disjoint).

Thus by NSTO Lemma 2.9, $A = H$ is NSTO. The proofs for the remaining three cases are analogous and, hence, omitted. □

We now prove two results, allowing us to conclude that $P \cup \{G\}$ is occur-check free. The first uses the the following notion introduced in Dembinski and Maluszynski [1985]:

Definition 3.4. We call an LD-derivation *data driven* if all atoms selected in it are ground in their input positions.

THEOREM 3.5. *Suppose that*

—*the head of every clause of P is output linear, and*
—*all LD-derivations of $P \cup \{G\}$ are data driven.*

Then $P \cup \{G\}$ is occur-check free.

PROOF. Consider an LD-derivation of $P \cup \{G\}$. Let A be an atom selected in it, and suppose that H is the head of an input clause such that A and H have the same relation symbol. By assumption, A is ground in its input positions, so it is input-output disjoint and input linear. By assumption, H is output linear, and A and H have no variable in common. So, by NSTO via Modes Lemma 3.3, $A = H$ is NSTO. □

The second result uses the following notion:

Definition 3.6. We call an LD-derivation *output driven* if all atoms selected in it are output linear and input-output disjoint.

THEOREM 3.7. *Suppose that*

—the head of every clause of P is input linear, and
—all LD-derivations of $P \cup \{G\}$ are output driven.

Then $P \cup \{G\}$ is occur-check free.

PROOF. Let A and H be as in the proof of Theorem 3.5. NSTO via Modes Lemma 3.3 applies and yields that $A = H$ is NSTO. □

This theorem is implicit in Chadha and Plaisted [1994] (see the proof of their Theorem 2.2). Clearly, through different "distributions" of the conditions of NSTO via Modes Lemma 3.3, other applications can be obtained. We found the above two least restrictive.

Note that the theorems established above generalize the following well-known result stated in Clark [1979, p. 15] and established in Deransart et al. [1991], as a direct consequence of the NSTO Lemma 2.9.

COROLLARY 3.8. *Suppose that the head of every clause of P is linear. Then $P \cup \{G\}$ is occur-check free for every goal G.*

PROOF. By Theorem 3.5, by moding every relation completely output, or by Theorem 3.7, by moding every relation completely input. □

This corollary can be applied to some well-known PROLOG programs, for example, to the unification program (see Sterling and Shapiro [1986, p. 150]) and, paradoxically, to the unification with the occur-check program (see Sterling and Shapiro [1986, p. 152]). However, to most programs this corollary does not apply. The subsequent sections provide some other options.

So far we have isolated two properties of LD-derivations, each of which implies occur-check freedom. In both cases we have had to impose some restrictions on the heads of the clauses. When we combine these two properties, we get occur-check freedom directly.

THEOREM 3.9. *Suppose that all LD-derivations of $P \cup \{G\}$ are both data and output driven. Then $P \cup \{G\}$ is occur-check free.*

PROOF. Let A and H be as in the proof of Theorem 3.5. By assumption, the system $A = H$ is left linear. Moreover, A and H have no variable in common, so the relation $\rightarrow_{A=H}$ is empty and a fortiori cycle free. So, by the NSTO Lemma 2.9, $A = H$ is NSTO. □

4. WELL-MODED PROGRAMS

The obvious problem with Theorems 3.5, 3.7, and 3.9 is that it is not easy to check their conditions. In fact, one can show that, in general, it is undecidable whether for a given program P and goal G the conditions of Theorem 3.5, 3.7, or 3.9 hold (see the Appendix).

The aim of this section is to propose some simple syntactic restrictions that imply the conditions of Theorem 3.5. We then show that these restrictions are satisfied by a number of well-known programs.

Here we use the notion of a well-moded program. The concept is due to Dembinski and Maluszynski [1985]; we use an elegant formulation due to Rosenblueth [1991] (which is equivalent to that of Drabent [1987], where well-moded programs are called simple). The definition of a well-moded program constrains the "flow of data" through the clauses of the programs. To simplify the notation, when writing an atom as $p(\mathbf{u}, \mathbf{v})$, we now assume that $\mathbf{u}$ is a sequence of terms filling in the input positions of p and that $\mathbf{v}$ is a sequence of terms filling in the output positions of p.

Definition 4.1.

—A goal $\leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ is called *well moded* if for $i \in [1, n]$

$$Var(\mathbf{s_i}) \subseteq \bigcup_{j=1}^{i-1} Var(\mathbf{t_j}).$$

—A clause

$$p_0(\mathbf{t_0}, \mathbf{s_{n+1}}) \leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$$

is called *well moded* if for $i \in [1, n + 1]$

$$Var(\mathbf{s_i}) \subseteq \bigcup_{j=0}^{i-1} Var(\mathbf{t_j}).$$

—A program is called *well moded* if every clause of it is.

Thus, a goal is well moded if

—every variable occurring in an input position of an atom ($i \in [1, n]$) occurs in an output position of an earlier ($j \in [1, i - 1]$) atom.

And a clause is well moded if

—($i \in [1, n]$) in every variable occurring in an input position of a body atom occurs either in an input position of the head ($j = 0$) or in an output position of an earlier ($j \in [1, i - 1]$) body atom, and

—($i = n + 1$) every variable occurring in an output position of the head occurs in an input position of the head ($j = 0$) or in an output position of a body atom ($j \in [1, n]$).

A test of whether a goal or clause is well moded can be efficiently performed by noting that a goal G is well moded iff every first from the left occurrence of a variable in G is within an output position. And a clause $p(\mathbf{s}, \mathbf{t}) \leftarrow \mathbf{B}$ is well moded iff every first from the left occurrence of a variable in the sequence s, $\mathbf{B}, \mathbf{t}$ is within the input position of $p(\mathbf{s}, \mathbf{t})$ or within an output position in $\mathbf{B}$. (We assume in this description that in every atom the input positions occur first.)

Note that a goal with only one atom is well moded iff this atom is ground in its input positions. The definition of a well-moded program is designed in such a way that the following theorem due to Dembinski and Maluszynski [1985] holds:

THEOREM 4.2. *Let P and G be well moded. Then all LD-derivations of $P \cup \{G\}$ are are data driven.*

In Dembinski and Maluszynski [1985], a different formulation of well modedness is given, and the above theorem is actually presented without a proof. So we allow ourselves to give a proof here.

Note that the first atom of a well-moded goal is ground in its input positions and a variant of a well-moded clause is well moded. Thus, it suffices to prove the following lemma, which shows the "persistence" of the notion of well modedness:

LEMMA 4.3. *An LD-resolvent of a well-moded goal and a well-moded clause that is variable-disjoint with it is well moded.*

PROOF. An LD-resolvent of a goal and a clause is obtained by means of the following three operations:

(1) instantiation of a goal;
(2) instantiation of a clause; and
(3) replacement of the first atom, say, H, of a goal by the body of a clause whose head is H.

So we only need to prove the following two claims:

CLAIM 1. *An instance of a well-moded goal (resp., clause) is well moded.*

PROOF. It suffices to note that, for any sequences of terms $\mathbf{s}, \mathbf{t}_1, \ldots, \mathbf{t}_n$ and a substitution σ,

$$Var(s) \subseteq \bigcup_{j=1}^{n} Var(\mathbf{t}_j)$$

implies

$$Var(\mathrm{s}\sigma) \subseteq \bigcup_{j=1}^{n} Var(\mathbf{t}_j\sigma). \qquad \square$$

CLAIM 2. *Suppose that $\leftarrow H, \mathbf{A}$ is a well-moded goal and $H \leftarrow \mathbf{B}$ is a well-moded clause. Then $\leftarrow \mathbf{B}, \mathbf{A}$ is a well-moded goal.*

PROOF. Let $H = p(\mathbf{s}, \mathbf{t})$ and $\mathbf{B} = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$. We have $Var(\mathrm{s}) = \emptyset$ since H is the first atom of a well-moded goal. Thus $\leftarrow \mathbf{B}$ is well moded. Moreover, $Var(\mathbf{t}) \subseteq \bigcup_{j=1}^{n} Var(\mathbf{t}_j)$, since $H \leftarrow \mathbf{B}$ is a well-moded clause and $Var(\mathbf{s}) = \emptyset$. These two observations imply the claim. $\square$

As a digression recall the following immediate and well-known conclusion of Theorem 4.2:

COROLLARY 4.4. *Let P and G be well moded. Then, for every computed answer substitution σ, $G\sigma$ is ground.*

PROOF. Let $\mathbf{x}$ stand for the sequence of all variables that appear in G. Let p be a new relation of arity equal to the length of $\mathbf{x}$ and with all positions moded as input. Then $\leftarrow \mathbf{A}, p(\mathbf{x})$ is a well-moded goal, where $G = \leftarrow \mathbf{A}$.

Now, σ is a computed answer substitution for $P \cup \{G\}$ iff $p(\mathbf{x})\sigma$ is a selected atom in an LD-derivation of $P \cup \{\leftarrow \mathbf{A}, p(\mathbf{x})\}$. The conclusion now follows from Theorem 4.2. □

We shall see in Section 7 that there are natural PROLOG programs for which data drivedness cannot be established using the concept of well modedness. Still, the above theorem brings us to the following conclusion, which can be easily applied to a number of well-known PROLOG programs:

COROLLARY 4.5. *Let P and G be well moded. Suppose that the head of every clause of P is output linear. Then $P \cup \{G\}$ is occur-check free.*

PROOF. By Theorems 3.5 and 4.2 □

Example 4.6. When presenting the programs, we adhere to the usual syntactic conventions of PROLOG, with the exception that PROLOG's :- is replaced by the logic programming $\leftarrow$.

(1) Consider the program **append**;

app([X|Xs], Ys, [X|Zs]) ← app(Xs, Ys, Zs).
app([ ], Ys, Ys).

with the mode **app(+, +, −)**. It is easy to check that **append** is well moded and that the head of every clause is output linear. By Corollary 4.5, we conclude that, for **s** and **t** ground, **append** $\cup$ {← **app(s, t, u)**} is occur-check free.

(2) Now examine the program **append** with the mode **app(−, −, +)**. Again, by Corollary 4.5, we conclude that, for **u** ground, **append** $\cup$ {← **app(s, t, u)**} is occur-check free.

(3) Consider the program **permutation**, which consists of the clauses

perm(Xs, [X|Ys]) ←
app(X1s, [X|X2s], Xs),
app(X1s, X2s, Zs),
perm(Zs, Ys).
perm([ ], [ ]).
augmented by the **append** program.

Here we use the following moding: **perm(+, −), app(−, −, +)** for the first call to **append**, and **app(+, +, −)** for the second call to **append**. It

is easy to check that **permutation** is then well moded and that the heads of all clauses are output linear. By Corollary 4.5, we get that, for **s** ground, **permutation** $\cup$ {$\leftarrow$ **perm(s, t)**} is occur-check free.

(4) Now examine the program **quicksort**, which consists of the clauses

qs([X | Xs], Ys) $\leftarrow$
  partition(X, Xs, Littles, Bigs),
  qs(Littles, Ls),
  qs(Bigs, Bs),
  app(Ls, [X | Bs], Ys).
qs([ ], [ ]).

partition(X, [Y | Xs], [Y | Ls], Bs) $\leftarrow$
  X > Y,
  partition(X, Xs, Ls, Bs).
partition(X, [Y | Xs], Ls, [Y | Bs]) $\leftarrow$
  X $\leq$ **Y,**
  partition(X, Xs, Ls, Bs).
partition(X, [ ], [ ], [ ]).
augmented by the **append** program.

We mode it as follows; **qs(+, −), partition(+, +, −, −), app(+, +, −).** Again, it is easy to check that **quicksort** is then well moded and that the heads of all clauses are output linear. By Corollary 4.5, we conclude that, for **s** ground, **quicksort** $\cup$ {$\leftarrow$ **qs(s, t)**} is occur-check free.

(5) Finally, consider the program **palindrome**:

palindrome(Xs) $\leftarrow$ **reverse(Xs, Xs).**
reverse(X1s, X2s) $\leftarrow$ **reverse(X1s, [ ], X2s).**
reverse ([X | X1s], X2s, Ys) $\leftarrow$ **reverse(X1s, [X | X2s], Ys).**
reverse ([ ], Xs, Xs).

We mode it as follows: **palindrome(+), reverse(+, −), reverse(+, +, −).** Then **palindrome** is well moded, and the heads of all clauses are output linear. By Corollary 4.5, we conclude that, for **s** ground, **palindrome** $\cup$ {$\leftarrow$ **palindrome(s)**} is occur-check free.

Note that Corollary 3.8 cannot be applied to any of these programs.

5. NICELY MODED PROGRAMS

The above conclusions are still restrictive, because in each case we had to assume that the input positions of the one-atom goals are ground. To alleviate this restriction, we now consider some syntactic restrictions that imply the conditions of Theorem 3.7.

The following notion was introduced in Chadha and Plaisted [1994]. (We found essentially the same concept independently, though later; the name and formulation are ours.)

Definition 5.1

—A goal $\leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ is called *nicely moded* if $\mathbf{t_1}, \ldots, \mathbf{t_n}$ is a linear family of terms and if for $i \in [1, n]$

$$Var(\mathbf{s_i}) \cap \left(\bigcup_{j=i}^{n} Var(\mathbf{t_j}) \right) = \varnothing. \tag{1}$$

—A clause

$$p_o(\mathbf{s_0}, \mathbf{t_0}) \leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$$

is called *nicely moded if* $\leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ is nicely moded and if

$$Var(\mathbf{s_0}) \cap \left(\bigcup_{j=1}^{n} Var(\mathbf{t_j}) \right) = \varnothing. \tag{2}$$

In particular, every unit clause is nicely moded.

—A program is called *nicely moded* if every clause of it is.

Thus, assuming that in every atom the input positions occur first, a goal is nicely moded if

—every variable occurring in an output position of an atom does not occur earlier in the goal.

And a clause is nicely moded if

—every variable occurring in an output position of a body atom occurs neither earlier in the body nor in an input position of the head.

So, intuitively, the concept of being nicely moded prevents a "speculative binding" of the variables that occur in output positions; these variables are required to be "fresh." Note that a goal with only one atom is nicely moded iff it is output linear and input-output disjoint. The following theorem clarifies our interest in nicely moded programs:

THEOREM 5.2. *Let P and G be nicely moded. Then all LD-derivations of $P \cup \{G\}$ are output driven.*

Note that the first atom of a nicely moded goal is output linear and input-output disjoint, and a variant of a nicely moded clause is nicely moded. Thus, to prove Theorem 5.2 it suffices to prove the following lemma, which shows the "persistence" of the notion of being nicely moded:

LEMMA 5.3. *An LD-resolvent of a nicely moded goal and a nicely moded clause that is variable-disjoint with it is nicely moded.*

PROOF. The proof is quite long and can be found in the Appendix. □

This lemma leads us to the following conclusion:

COROLLARY 5.4. *Let P and G be nicely moded. Suppose that the head of every clause of P is input linear. Then $P \cup \{G\}$ is occur-check free.*

PROOF. By Theorems 3.7 and 5.2. □

This corollary is independently established in Chadha and Plaisted [1994]. Pierre Deransart (private communication) pointed out to us that this corollary is a consequence of Theorem 4.1 in Deransart et al. [1991], whose conditions are satisfied for a nicely moded program P and a nicely moded goal G. This suggests a stronger result, namely, that then $P \cup \{G\}$ is NSTO. On the other hand, our proof establishes Lemma 5.3, which will allow us to deal in Section 7 with programs that use difference lists and in Section 10 with programs that do require the use of unification with the occur-check.

Note that to prove Corollary 5.4 it is actually sufficient to prove Lemma 5.3, under the assumption that the head of every clause of P is input linear. The proof is considerably simpler than that of Lemma 5.3.

To apply Corollary 4.5, it is natural to start by moding the relations used in the goal so that this goal becomes well moded. Then one should try to mode other relations used in the program, so that the remaining conditions of this corollary are satisfied. The important clue comes from the fact that the input positions of the first atom of a well-moded goal are filled in by ground terms. This is not the case for the nicely moded goals, so, for example, it is not clear how to mode the relation **app** when considering the goal $\leftarrow$ **app([X, 2], [Y, U], [3, Z, 0, Z])** (which succeeds with the c.a.s. **{X / 3, Z / 2, Y / 0, U / 2}**). We shall see later that in the presence of difference lists there is no clear intuition either about the modes of certain positions in the relations.

Consequently, as noted by Chadha and Plaisted [1994], to apply Corollary 5.4 it is probably more natural to investigate, first, all of the modings for which the program is nicely moded and for which the heads of all clauses are input linear. Then one should check for which modings the given goal is nicely moded. To this end Chadha and Plaisted [1994] proposed two efficient algorithms for generating modings with the minimal number of input positions, for which the program is nicely moded. These algorithms were implemented and applied to a number of well-known PROLOG programs.

In the case of the **append** program, the conditions of Corollary 5.4 are satisfied for only five of the eight modes. Out of the five, only the mode **app**$(-, -, +)$ can be used to deal with the goal $\leftarrow$ **app([X, 2], [Y, U], [3, Z, 0, Z])**.

Let us see now how this corollary can be applied to the previously studied programs.

Example 5.5.

(1) Again consider the program **append** with the moding **app**$(+, +, -)$. Clearly, **append** is nicely moded, and the head of every clause is input linear. By Corollary 5.4, we conclude that, when **u** is linear and when $Var(\mathbf{s}, \mathbf{t}) \cap Var(\mathbf{u}) = \emptyset$, **append** $\cup$ {$\leftarrow$ **app(s, t, u)**} is occur-check free.

(2) With the moding **app**$(-, -, +)$, the program **append** is nicely moded as well, and the head of every clause is input linear. Again, by Corollary 5.4, we conclude that, when **s, t** is a linear family of terms and when $Var(\mathbf{s}, \mathbf{t}) \cap Var(\mathbf{u}) = \emptyset$, **append** $\cup$ {$\leftarrow$ **app(s, t, u)**} is occur-check free.

(3) Now reconsider the program **permutation** with the modings as before. Again, it is easy to check that **permutation** is nicely moded and that the heads of all clauses are input linear. By Corollary 5.4, when $\mathbf{t}$ is linear and when $Var(\mathbf{s}) \cap Var(\mathbf{t}) = \emptyset$, **permutation** $\cup$ {$\leftarrow$ **perm(s, t)**} is occur-check free.

(4) Examine the program **quicksort** with the modings as before. Again, by Corollary 5.4, when $\mathbf{t}$ is linear and when $Var(\mathbf{s}) \cap Var(\mathbf{t}) = \emptyset$, **quicksort** $\cup$ {$\leftarrow$ **qs(s, t)**} is occur check free.

(5) So far it seems that Corollary 5.4 allows us to draw more useful conclusions than Corollary 4.5. However, reconsider the program **palindrome**. In Chadha and Plaisted [1994], it is shown that no moding exists in which **palindrome** is nicely moded with the heads of all clauses being input linear. Thus, Corollary 5.4 cannot be applied to this program.

6. STRICTLY MODED PROGRAMS

Next, consider syntactic restrictions that imply the condition of Theorem 3.9. To this end it is sufficient to combine the properties of being well moded and nicely moded. Indeed, we observe the following:

COROLLARY 6.1. *Let P and G be well moded and nicely moded. Then $P \cup \{G\}$ is occur-check free.*

PROOF. By Theorems 4.2, 5.2, and 3.9. □

In the remainder of this section, we show that the conditions of this corollary can be weakened. First, note that, when a goal $\leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is well moded and the family $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is linear, condition (1) of Definition 5.1 is satisfied and, thus, the goal is nicely moded. A similar observation can be made about a clause $p_0(\mathbf{s}_0, \mathbf{t}_0) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$.

Thus, the assumptions of the above corollary can be simplified. We now show that a further simplification is possible; namely, condition (2) of Definition 5.1 can be omitted as well.

Definition 6.2.

—A goal $\leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is called *strict* if $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a linear family of terms.

—A clause $H \leftarrow \mathbf{B}$ is called *strict* if $\leftarrow \mathbf{B}$ is strict.

—A program is called *strict* if every clause of it is.

—A goal (clause, program) is called *strictly moded* if it is both strict and well moded.

THEOREM 6.3. *Let P and G be strictly moded. Then all LD-derivations of $P \cup \{G\}$ are both data and output driven.*

Note that the first atom of a strictly moded goal is ground in its input positions, output linear, and input-output disjoint, and a variant of a strictly moded clause is strictly moded. Thus, to prove this theorem it suffices to show, as in the case of well-modedness and being nicely moded, that the notion of strict modedness is "persistent."

LEMMA 6.4. *An LD-resolvent of a strictly moded goal and a strictly moded clause that is variable-disjoint with it is strictly moded.*

PROOF. Consider a strictly moded goal $\leftarrow A, \mathbf{A}$ and a strictly moded clause $H \leftarrow \mathbf{B}$ that is variable-disjoint with it. We start by proving three claims that appropriately refine those of Lemma 4.3.

Suppose $A = p(\mathbf{s}_A, \mathbf{t}_A)$ and $H = p(\mathbf{s}_H, \mathbf{t}_H)$. Assume that $A = H$ is unifiable. Take as E_1 the system of equations $s_A = s_H$ and as E_2 the system of equations $t_A = t_H$. Let θ_1 be a relevant mgu of E_1 and let θ_2 be a relevant mgu of $E_2\theta_1$. The existence of these substitutions is assured by Lemma 2.4, which also gives that $\theta_1\theta_2$ is a relevant mgu of $A = H$.

Let p^- be a new relation symbol of arity equal to the cardinality of $\mathbf{t}_A$ (and of $\mathbf{t}_H$) with all positions moded as output. Note that θ_1 is grounding, so, by the definition of a nicely moded goal and clause, $\leftarrow (p^-(\mathbf{t}_A), \mathbf{A})\theta_1$ and $p^-(\mathbf{t}_H\theta_1) \leftarrow \mathbf{B}\theta_1$ are both nicely moded and have no variables in common. By Lemma 5.3, their resolvent $\leftarrow (\mathbf{B}\theta_1, \mathbf{A}\theta_1)\theta_2$ is nicely moded, as well.

This and Lemma 4.3 allow us to conclude that the LD-resolvent $\leftarrow (\mathbf{B}, \mathbf{A})\theta_1\theta_2$ of the goal $\leftarrow A, \mathbf{A}$ and the clause $H \leftarrow \mathbf{B}$ is both well moded and nicely moded, and is, thus, strictly moded.

$\theta = \theta_1\theta_2$ is just one specific mgu of $A = H$. By Lemma 2.3, every other mgu of $A = H$ is of the form $\theta\eta$ for a renaming η. But a renaming of a strictly moded goal is strictly moded, so we conclude that every LD-resolvent of $\leftarrow A$, $\mathbf{A}$ and $H \leftarrow \mathbf{B}$ is strictly moded. □

The following result improves upon Corollary 6.1:

COROLLARY 6.5. *Let P and G be strictly moded. Then $P \cup \{G\}$ is occur-check free.*

PROOF. By Theorems 6.3 and 3.9. □

Example 6.6. In contrast to the case of well-moded and nicely moded programs, it is difficult to come up with a natural example to which the notion of a strictly moded program could be applied. Still, consider the program **derivative** from Sterling and Shapiro [1986, p. 63], which computes a derivative of an expression w.r.t. a variable. To save space, here we only reproduce a couple of crucial clauses. The used function symbols are written in an infix form, and for simplicity, unary notation for natural numbers is used.

der(X, X, s(0)).
der(X↑s(N), X, s(N)∗X↑N).
der(F + G, X, DF + DG) ← der(F, X, DF), der(G, X, DG).
der(F∗G, X, F∗DG + DF∗G) ← der(F, X, DF), der(G, X, DG).

To compute the derivative of an expression **e**, say, **x↑s(0) + x∗y + y↑s(s(s(0)))**, w.r.t a variable, say, **x**, one uses the goal ← **der(e, x, Y)**. In the mode **der(+, +, −)**, this program is both well moded and nicely moded, and consequently, it is also strictly moded. By Corollary 6.5, we conclude that,

when **e** and **t** are ground and **u** is linear, **derivative** $\cup$ {$\leftarrow$ **der(e, t, u)**} is occur-check free.

Note that the head of the first clause is not input linear, and the head of the second clause is not output linear. Consequently, neither Corollary 4.5 nor Corollary 5.4 can be applied here.

7. DIFFERENCE LISTS

It is well known that programs with difference lists easily lead to complications in absence of the occur-check. For example, the program **empty**,

```
empty(L\L).
```

when executed with the goal $\leftarrow$ **empty([a|X]\X)** leads to the consideration of the system {**[a|X]** = **L**, **X** = **L**}, which is subject to the occur-check. It is worthwhile to note that programs that use difference lists can be handled by the methods proposed. For example, Corollary 5.4 immediately implies that, for **s** and **t** linear and variable-disjoint, **empty** $\cup$ {$\leftarrow$ **empty(s, t)**} is occur-check free.

However, we did find two programs in Sterling and Shapiro [1986] that use difference lists and to which we could not apply the results so far established. These are **flatten_dl** [Sterling and Shapiro 1986, program 15.2, p. 241];

```
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([ ], Xs\Xs).
```

and **quicksort_dl** [Sterling and Shapiro 1986, program 15.4, p. 244],

```
qs(Xs, Ys) ← qs_dl(Xs, Ys\[ ]).
qs_dl([X|Xs], Ys\Zs) ←
  partition(X, Xs, Littles, Bigs),
  qs_dl(Littles, Ys\[X|Ys1]),
  qs_dl(Bigs, Ys1\Zs).
qs_dl([ ], Xs\Xs).
```

augmented by the **partition** program.

These programs are customarily used in the modes **flatten**(+, −) and **qs**(+, −). It is easy to check that for both programs no completion of the moding exists for which the program is well moded, or nicely moded and with the heads of all clauses being input linear.

For example, for **flatten_dl** the attempt to get it well moded fails as follows: Assume the mode **flatten**(+, −). For the first clause, we have to use a mode of the form **flatten_dl**(?, −, ?). Now, due to the last clause, we actually have to use a mode of the form **flatten_dl**(?, −, +). But, then, in the

recursive clause for **flatten_dl** we cannot satisfy the requirement of well modedness concerning the variable **Y1s**.

However, it is possible to modify the results on well moded and nicely moded programs in such a way that the occur-check freedom of the above two programs used in the discussed modes still can be established. The idea is quite simple, though some work is needed to make it precise. Suppose that we know that some program atoms when selected in a derivation are ground in specific input positions. Then these input positions do not need to be considered when proving that the program is occur-check free.

To formalize this idea, we consider a program and goal in two different modings. First we prove that, when an atom is selected in a derivation, its input positions, which are "shared" in both modes, are ground. Then we consider a derived program obtained by removing from each atom these shared input positions and apply to it the previous results on the nicely moded programs.

To avoid confusion we write m-well moded (resp., m-nicely moded, etc.) when a given goal (clause, program) is well moded (resp., nicely moded, etc.) with respect to the moding m. Now assume two modings m_1, and m_2. Fix a relation symbol p. Some positions in p are both m_1-input and m_2-input. We now associate with p a new relation symbol p^- in which these shared input positions are removed and the remaining positions are moded as in m_2. The moding so obtained for the relation symbols of the form p^- is denoted by $m_2 - m_1$.

For example, if m_2 is **flatten_dl**$(+, +, -)$ and m_1 is **flatten_dl** $(+, -, -)$, then $m_2 - m_1$ is **flatten_dl**$^-(+, -)$. These new relation symbols allow us to associate with any atom A written in the mode m_2 as $p(\mathbf{u}, \mathbf{v})$, an atom A^- written in the mode $m_2 - m_1$ as $p^-(\mathbf{u}^-, \mathbf{v})$, where $\mathbf{u}^-$ is obtained by removing from $\mathbf{u}$ the terms that are in m_1-input positions. For example, for the above two modes m_1 and m_2 and $A =$ **flatten_dl([X | Xs], Ys, Zs)** we get $A^- =$ **flatten_dl**$^-$**(Ys, Zs)**.

Now given a sequence of atoms $\mathbf{A}$, we associate with it a sequence $\mathbf{A}^-$ obtained by replacing in $\mathbf{A}$ every atom A by A^-.

Definition 7.1.

—A goal $\leftarrow \mathbf{A}$ is called *m_2-nicely moded w.r.t. m_1* ($m_2 \mid m_1$-*nice*, in short) if the goal $\leftarrow \mathbf{A}^-$ is $(m_2 - m_1)$-nicely moded.

—A clause $H \leftarrow \mathbf{B}$ is called $m_2 \mid m_1$-*nice* if the clause $H^- \leftarrow \mathbf{B}^-$ is $m_2 \mid m_1$-nice.

—A program is called $m_2 \mid m_1$-*nice* if every clause of it is.

Thus, a goal (clause, program) is $m_2 \mid m_1$-nice if its "$^-$" version is $(m_2 - m_1)$-nicely moded. Note that the notion of $m_2 \mid m_1$-nice goal (clause, program) extends that of nice goal (clause, program). Indeed, if a goal (clause, program) is m_2-nice then it is $m_2 | m_1$-nice for every m_1. Also, a goal (clause, program) is $m_1 \mid m_1$-nice and m_1-well moded iff it is m_1-strictly moded.

The following theorem explains the usefulness of this concept:

THEOREM 7.2. *Suppose that*

—all LD-derivations of $P \cup \{G\}$ *are* m_1*-data driven, and*

—P *and* G *are* $m_2 \mid m_1$*-nice.*

Then all LD-derivations of $P \cup \{G\}$ *are* m_2*-output driven.*

PROOF. First, we prove that all goals appearing in an LD-derivation of $P \cup \{G\}$ are $m_2 \mid m_1$-nice. To this end, due to the assumption of m_1-data drivedness, it suffices to prove that, for A m_1-input ground, an LD-resolvent of an $m_2 \mid m_1$-nice goal $\leftarrow A, \mathbf{A}$ and a disjoint with it variant $H \leftarrow \mathbf{B}$ of an $m_2 \mid m_1$-nice clause is $m_2 \mid m_1$-nice as well.

Assume that A and H are unifiable. $A = H$ equals $E \cup (A^- = H^-)$, where the left-hand sides of the equations from E are ground. Let θ_1 be a relevant mgu of E and let θ_2 be a relevant mgu of $(A^- = H^-)\theta_1$. The existence of these substitutions is assured by Lemma 2.4, which also gives that $\theta_1\theta_2$ is a relevant mgu of $A = H$.

θ_1 is grounding, so by the definition of a nicely moded goal and clause, both $\leftarrow (A^-, \mathbf{A}^-)\theta_1$ and $(H^- \leftarrow \mathbf{B}^-)\theta_1$ are $(m_2 - m_1)$-nicely moded. By Lemma 5.3, their LD-resolvent $\leftarrow (\mathbf{B}^-, \mathbf{A}^-)\theta_1\theta_2$ is $(m_2 - m_1)$-nicely moded; that is, $\leftarrow (\mathbf{B}, \mathbf{A})\theta_1\theta_2$, the resolvent of $\leftarrow A, \mathbf{A}$ and $H \leftarrow \mathbf{B}$, is $m_2 \mid m_1$-nice. To draw the same conclusion for an arbitrary LD-resolvent of $\leftarrow A, \mathbf{A}$ and $H \leftarrow \mathbf{B}$, it suffices now to use Lemma 2.3.

Now consider a goal appearing in an LD-derivation of $P \cup \{G\}$. We just established that it is $m_2 \mid m_1$-nice, so its first atom A is such that A^- is $(m_2 - m_1)$-output linear and $(m_2 - m_1)$-input-output disjoint. By assumption, A is also m_1-input ground, so A is actually m_2-output linear and m_2-input-output disjoint. This proves the claim. □

This brings us to the following conclusion:

COROLLARY 7.3. *Suppose that*

—all LD-derivations of $P \cup \{G\}$ *are* m_1*-data driven,*

—P *and* G *are* $m_2 \mid m_1$*-nice, and*

—for a head H*, of a clause of* P*,* H^- *is* $(m_2 - m_1)$*-input linear.*

Then $P \cup \{G\}$ *is occur-check free.*

Note that the last assumption is weaker than the statement that the head of every clause of P is m_2-input linear.

PROOF. Let A be an atom selected in an LD-derivation of $P \cup \{G\}$, and suppose that H is a head of an input clause such that A and H have the same relation symbol. By Theorem 7.2, A is m_2-output linear, and m_2 is input-output disjoint. Let m_3 be the mode obtained from m_2 by reversing the m_1-input positions to output positions. By assumption, the m_1-input positions of A are ground, so A is m_3-output linear and m_3-input-output disjoint. Moreover, by assumption, H is m_3-input linear. Now NSTO via Modes Lemma 3.3 applies and yields that $A = H$ is NSTO. □

We have already noticed that neither **flatten_dl** nor **quicksort_dl** is well moded with the relation **flatten** (resp., **qs**) is moded, (+, −). Thus, to be able to apply this corollary we need another method for establishing data-drivedness than proving well modedness. Our idea is to weaken the latter notion by assuming that only part of the program is well moded and by requiring that the previous inclusions referring to the output variables refer now to the output variables of the atoms defined in this well-moded part. We begin with some definitions.

Definition 7.4. Let P be a program, and let p, q be relations.

—We say that p *refers to* q iff there is a clause in P that uses p in its head and q in its body.

—We say that p *depends on* q iff (p, q) is in the reflexive, transitive closure of the relation *refers to*.

—We say that a clause of P *defines the relation* p if p is used in its head.

Definition 7.5. Consider a program P and an atom A in a given moding.

—We denote by P_A the set of clauses of P that define the relation p of A and the relations on which p depends.

—We say that A is *well moded in* P if P_A is.

For example, in the moding **qs**(+, −), **qs_dl**(+, +, −), **partition**(+, +, −, −), for A = **partition(X, Xs, Littles, Bigs)** we have **quicksort_dl**$_A$ = **partition**, so A is well moded in **quicksort_dl.**

We now introduce the following modification of the notion of a well-moded program and goal:

Definition 7.6. Let P be a program.

—A goal $\leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ is called *weakly moded w.r.t.* P if for $i \in [1, n]$

$$Var(\mathbf{s_i}) \subseteq \bigcup_{j=1}^{k} Var(\mathbf{t_{i_j}}),$$

where $p_{i_1}(\mathbf{s_{i_1}}, \mathbf{t_{i_1}}), \ldots, p_{i_k}(\mathbf{s_{i_k}}, \mathbf{t_{i_k}})$ are all of the atoms among $p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_{i-1}(\mathbf{s_{i-1}}, \mathbf{t_{i-1}})$ that are well moded in P. (Here and below, k depends on i.)

—A clause

$$p_0(\mathbf{s_0}, \mathbf{t_0}) \leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$$

is called *weakly moded w.r.t.* P if for $i \in [1, n]$

$$Var(\mathbf{s_i}) \subseteq Var(\mathbf{s_0}) \cup \bigcup_{j=1}^{k} Var(\mathbf{t_{i_j}}),$$

where $p_{i_1}(\mathbf{s_{i_1}}, \mathbf{t_{i_1}}), \ldots, p_{i_k}(\mathbf{s_{i_k}}, \mathbf{t_{i_k}})$ are all of the atoms among $p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_{i-1}(\mathbf{s_{i-1}}, \mathbf{t_{i-1}})$ that are well moded in P. In particular, every unit clause is weakly moded w.r.t. P

—A program is called *weakly moded* if every clause of it is weakly moded w.r.t. it.

Thus, a goal is weakly moded w.r.t. P if

—every variable occurring in an input position of an atom occurs in an output position of an earlier, well-moded in P atom.

And a clause is weakly moded w.r.t. P if

—($i \in [1,n]$) every variable occurring in an input position of a body atom occurs either in an input position of the head or in an output position of an earlier body atom, which is well moded in P.

Observe that a goal with only one atom is weakly moded w.r.t. a program P iff its atom is ground in its input position. The notion of being weakly moded is obviously related to that of being well moded. In fact, if a program P is well moded, then it is weakly moded. Next, assuming that P is well moded, if a goal is well moded, then it is weakly moded w.r.t. P. Thus, the following theorem generalizes Theorem 4.2:

THEOREM 7.7. *Let P be weakly moded, and let G be weakly moded w.r.t. P. Then all LD-derivations of $P \cup \{G\}$ are data driven.*

Note that the first atom of a weakly moded goal is ground in its input positions and that a variant of a weakly moded clause is weakly moded (all w.r.t. a program P). Thus, as in the case of Theorem 4.2, it suffices to prove the following lemma, showing the persistence of the notion of being weakly moded:

LEMMA 7.8. *An LD-resolvent of a weakly moded goal and a weakly moded clause that is variable-disjoint with it is weakly moded, all w.r.t. a program P.*

PROOF. The proof is analogous to that of Lemma 4.3. We prove two claims:

CLAIM 1. *An instance of a weakly moded goal (resp., clause) is weakly moded, w.r.t. a program P.*

PROOF. As the proof of Claim 1 of Lemma 4.3. □

CLAIM 2. *Suppose $\leftarrow H, \mathbf{A}$ is a weakly moded goal and $H \leftarrow \mathbf{B}$ is a weakly moded clause. Then $\leftarrow \mathbf{B}, \mathbf{A}$ is a weakly moded goal, all w.r.t. a program P.*

PROOF. Let $H = p(\mathbf{s}, \mathbf{t})$ and $\mathbf{B} = p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$. *We have* $Var(\mathrm{s}) = \emptyset$, since H is the first atom of a weakly moded goal. Thus, $\leftarrow \mathbf{B}$ is weakly moded. Now, if H is well moded in P, then $H \leftarrow \mathbf{B}$ is a well-moded clause, and consequently, $Var(\mathbf{t}) \subseteq \bigcup_{j=1}^{n} Var(\mathbf{t_j})$, since $Var(\mathbf{s}) = \emptyset$. Moreover, all atoms in $\mathbf{B}$ are then well moded. So $\leftarrow \mathbf{B}, \mathbf{A}$ is weakly moded.

If H is not well moded in P, then, by assumption, $\leftarrow \mathbf{A}$ is weakly moded, and so $\leftarrow \mathbf{B}, \mathbf{A}$ is as well. □

In contrast, Corollary 4.4 does not generalize to the case of weak modedness. Indeed, consider $P = \{p(x) \leftarrow\}$ and $G = \leftarrow p(y)$, where the relation p is moded $p(-)$.

Now, combining Theorem 7.7 with Corollary 7.3 we obtain the generalization of Corollaries 5.4 and 6.5 we aimed at:

COROLLARY 7.9. *Suppose that*

—*P is m_1-weakly moded and G is m_1-weakly moded w.r.t. P;*
—*P and G are $m_2 \mid m_1$-nice, and*
—*for a head H of a clause of P, H^- is $(m_2 - m_1)$-input linear.*

Then $P \cup \{G\}$ is occur-check free.

The conditions of this corollary look quite elaborate, but it is easy to check them for specific programs.

Example 7.10.

(1) Consider the **flatten_dl** program with "\" replaced by ",". Choose m_2 = **flatten(+, −), flatten_dl(+, +, −), constant(+), ≠ (+, +)**. It is easy to see that in this moding **flatten_dl** is nicely moded, so it is $m_2 \mid m_1$-nice for any moding m_1. However, in the moding m_2 the head of the third clause is not input linear, so here we cannot apply Corollary 5.4.

On the other hand, by choosing m_1 with **flatten_dl(+, −, −)** and all other modes as in m_2, we get for a head H of a clause of **flatten_dl** that H^- is $(m_2 - m_1)$-input linear. Additionally, **flatten_dl** is m_1-weakly moded. We can now apply Corollary 7.9 and conclude that, when **xs** is ground and **ys** is linear, **flatten_dl** $\cup$ {← **flatten(xs, ys)**} is occur-check free.

(2) Now examine the program **quicksort_dl**, again with "\" replaced by ",". Choose m_2 = **qs(+, −), qs_dl(+, +, −), partition(+, +, −, −,)**. In this mode **quicksort_dl** is not nicely moded, since in the second clause the variable **X** occurs both in an input position of the head and in an output position of a body atom.

However, by choosing m_1 with **qs_dl(+, −, −,)** and all other modes as in m_2, we get that **quicksort_dl** is $m_2 \mid m_1$-nice. Additionally, **quicksort_dl** is m_1-weakly moded, since, as we already noted, **partition(X, Xs, Littles, Bigs)** is well moded in **quicksort_dl.** Also, for a head H of a clause of **quicksort_dl**, H^- is $(m_2 - m_1)$-input linear. By Corollary 7.9 we get that, when **xs** is ground and **ys** is linear, **quicksort_dl** $\cup$ {← **qs(xs, ys)**} is occur-check free.

(3) Finally, consider the following program **normalize** from Sterling and Shapiro [1986, p. 248], in which we replace the binary infix symbol "+ +" (symbolizing the sum still to be performed) by ",":

```
normalize(Exp, Norm) ← normalize_ds(Exp, Norm, 0).

normalize_ds(A + B, Norm, Space) ←
  normalize_ds(A, Norm, NormB)
  normalize_ds(B, NormB, Space).
normalize_ds(A, (A + Space), Space) ← constant (A).
```

normalize converts a sum **Exp** into a normalized form **Norm** that is bracketed to the right. For example, $(a + b) + (c + d)$ is normalized to $(a + (b + (c + d)))$.

Assume the mode **normalize** (+, −). We leave to the reader the task of checking that Corollaries 4.5, 5.4, or 6.5 cannot be applied here. Now consider m_2 = **normalize**(+, −), **normalize_ds**(+, +, −), **constant** (+), and let m_1 consist of **normalize_ds**(+, −, −) and all other modes, as in m_2. The same reasoning as in the case of **flatten_dl** applies and yields that, for **exp** ground and **norm** linear, **normalize** $\cup$ {← **normalize(exp, norm)**} is occur-check free.

8. GENERAL PROGRAMS

We now consider an extension of these results to the case of general programs, that is, programs in which negative literals in the clause bodies are allowed. We also show that the concept of well modedness can be used to prove the absence of floundering, that is, selection of a negative, nonground literal in a derivation.

First, we need to extend the basic definitions. By the *LDNF-resolution* we mean the SLDNF-resolution of Clark [1979] with the leftmost selection rule. When studying the occur-check problem, we need to use a definition of SLDNF-resolution that guarantees that for every general program P and a general goal G the SLDNF-tree, which comprises all SLDNF-derivations, exists. (The definition provided in Lloyd [1987] is too restrictive for this purpose; e.g., for the program $P = \{p \leftarrow p\}$ and the general goal $G = \leftarrow \neg p$, no SLDNF-derivation or tree exists.) Such a definition was recently given in Apt and Doets [1994].

Here we only need to know the general goals that can appear in an LDNF-derivation of $P \cup \{G\}$. This leads us to the following definition, where, for a general goal H and a literal L, $H - \{L\}$ stands for the result of removing L from H:

Definition 8.1. Consider an LDNF-derivation of $P \cup \{G\}$. Let $\mathcal{G}_{P,G}$ be the least set of general goals such that

(1) $G \in \mathcal{G}_{P,G}$;
(2) if $H \in \mathcal{G}_{P,G}$, the first literal of H is positive, and H' is an LDNF-resolvent of H and a general clause of P that is variable-disjoint with it, then $H' \in \mathcal{G}_{P,G}$;
(3) if $H \in \mathcal{G}_{P,G}$ and the first literal, L, of H is negative and ground, then $H - \{L\} \in \mathcal{G}_{P,G}$; and
(4) if H is ground, then $H \in \mathcal{G}_{P,G}$.

Using the definition of SLDNF-resolution provided in Apt and Doets [1994], it is straightforward to prove the following lemma, whose proof we omit:

LEMMA 8.2. *Consider an LDNF-derivation ξ of $P \cup \{G\}$. Every general goal that appears in ξ belongs to $\mathcal{G}_{P,G}$.*

When computing with general programs, one of the complications is so-called floundering. We study it here for the case of the LDNF-resolution.

Definition 8.3. Let P be a general program and G a general goal. We say that $P \cup \{G\}$ *flounders* if in the LDNF-tree of $P \cup \{G\}$ a general goal appears with the first literal negative and nonground.

Next, when considering the notion of the occur-check freedom for general programs and general goals, we simply reuse the original Definition 2.6. In this way, we ignore the selection of negative literals, but this does not matter, as the choice of a negative literal $\neg A$ leads either to floundering or to the consideration of the goal $\leftarrow A$ whose selected literal is positive. In both cases no unification is performed.

The concepts of data- and output-driven derivations extend to LDNF-derivations in a straightforward way by considering selected literals instead of selected atoms.

Now, we generalize the notion of well modedness to general programs and general goals by simply allowing in Definition 4.1 the negation symbol to occur in front of any atom $p_i(\mathbf{s_i}, \mathbf{t_i})$, where $i \in [1, n]$. Theorem 4.2 easily generalizes to general programs and general goals. More precisely, we have the following result:

THEOREM 8.4. *Consider a general program P and a general goal G. Let P and G be well moded. Then all LDNF-derivations of $P \cup \{G\}$ are data driven.*

PROOF. First, we prove that every general goal in $\mathscr{G}_{P,G}$ is well moded. Lemma 4.3 generalizes to LDNF-resolvants, so clause (2) of Definition 8.1 preserves well modedness. Obviously, so does clause (3) and clauses (1) and (4) admit only well-moded general goals in $\mathscr{G}_{P,G}$. The desired conclusion now follows from Lemma 8.2 and from the fact that the first literal of a well-moded general goal is ground in its input positions. □

Consequently, Corollary 4.5 holds for general programs and general goals, this time by virtue of Theorems 3.5 and 8.4.

The following simple result shows that the concept of well modedness is also very helpful for the study of floundering. It was independently discovered by Stroetman [1993]:

THEOREM 8.5. *Consider a general program P and a general goal G. Suppose that P and G are well moded and that all relations that appear in negative literals of P and G are moded completely input. Then $P \cup \{G\}$ does not flounder.*

PROOF. Using the inductive definition of $\mathscr{G}_{P,G}$, it is straightforward to show that every negative literal L occurring in a general goal $H \in \mathscr{G}_{P,G}$ is an instance of a negative literal occurring in P or G. So, by assumption, the relation appearing in such L is moded completely input. The claim now follows by Lemma 8.2 and by Theorem 8.4. □

Note also that Theorem 7.7 easily generalizes to general goals and general programs. Consequently, Theorem 8.5 can be strengthened to the case of

(appropriately defined) weakly moded general goals and general programs. Theorem 8.5 and the above generalization of Corollary 4.5 are easily applicable.

Example 8.6. All general programs below are understood to be augmented by the program **member**:

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

(1) Consider the general program **disjoint:**

disjoint(X, Y) ← ¬ overlap(X, Y).
overlap(X, Y) ← member(Z, X), member(Z, Y).

with the moding **disjoint(+, +), overlap(+, +), member(−, +).** Of course, **disjoint** checks whether two lists are disjoint. **disjoint** is clearly well moded, and the heads of all general clauses are output linear; so for **s** and **t** ground, **disjoint** ∪ {← **disjoint(s, t)**} is occur-check free, and by virtue of Theorem 8.5, it does not flounder.

(2) The following well-known general program **trans** computes the transitive closure of a binary relation:

trans(X, Y, E, V) ← member([X, Y], E).
trans(X, Z, E, V) ←
  member([X, Y], E),
  ¬ member(Y, V),
  trans(Y, Z, E, [Y | V]).

In a typical use of this program, in order to check that **[x, y]** is in the transitive closure of the binary relation **e**, one evaluates the goal ← **trans(x, y, e, [x]).**

With the moding **trans(−, −, +, +), member(+, +)** for the occurrence of **member** in the negative literal ¬ **member(Y, V)** and **member(−, +)** for the other occurrences of **member**, the program **trans** is well moded, and the heads of all general clauses are output linear. So we conclude that, for **e, v** ground, **trans** ∪ {← **trans(s, t, e, v)**} is occur-check free, and by Theorem 8.5, it does not flounder. The mode **member(+, +)** is needed here only to draw the latter conclusion.

(3) Finally consider the general program **sink**:

sink(X, E) ← ¬ interior(X, E).
interior(X, E) ← member([X, Y], E).

with the moding **sink(+, +), interior(+, +), member(−, +).** For a binary relation **e**, the goal ← **sink(a, e)** succeeds if **a** is a sink point in **e. sink** is well moded, and the heads of all general clauses are output linear, so for **a, e** ground, **sink** ∪ {← **sink(a, e)**} is occur-check free and does not flounder.

The usual approach to prove the absence of floundering by syntactic means concentrates on SLDNF-resolution and is based on various generalizations of

the concept of allowedness (see Decker [1991] for the strongest results in this direction). However, these techniques, in general, cannot be applied to LDNF-resolution, which employs a more specific notion of floundering.

Using Lemma 8.2 it is also possible to generalize the results on nicely and strictly moded programs (viz, Corollaries 5.4 and 6.5) to the case of general programs. However, the concept of a strictly moded general program is rarely needed, and that of a nicely moded general program does not prevent the use of nonground input positions in the goals. As a result, general programs to which the results on nicely moded general programs can be applied usually flounder. So, in the framework of LDNF-resolution, these generalizations are of limited interest and, consequently, are omitted.

9. DISCUSSION

To apply the established results to a (general) program and a (general) goal, one needs to find appropriate modings for the considered relations such that the conditions of one of the established corollaries are satisfied. In Table I several programs taken from Sterling and Shapiro [1986] are listed. (A similar analysis of the notion of a well-moded program was carried out in Drabent [1987]). Corollary 3.8 can be applied to none of them. For each program it is indicated which of the relevant conditions for a given moding are satisfied. All built-ins are moded completely input.

In programs that use difference lists, we replaced "\" by ",", thus splitting a position filled in by a difference list into two positions. Because of this change, in some relations additional arguments are introduced, and so certain clauses have to be modified in an obvious way. For example, in the parsing program in Sterling and Shapiro [1986], each clause of the form $\mathbf{p(X)} \leftarrow \mathbf{r(X)}$ has to be replaced by $\mathbf{p(X, Y)} \leftarrow \mathbf{r(X, Y)}$. Such changes are purely syntactic and allow us to draw conclusions about the occur-check freedom of the original program. The modings considered are usually intuitive, and at least one of the Corollaries 4.5, 5.4, or 6.5 applies.

The appropriate entry in Table I indicates that, after replacing "\" by "," in the mode **flatten**(+, +) and **flatten_dl**(+, +, −), **flatten_dl** is well moded and the heads of the clauses are output linear. Thus, by virtue of Corollary 4.5 for **s** and **t** ground, all LD-derivations of **flatten_dl** $\cup$ { $\leftarrow$ **flatten(s, t)**} are occur-check free. Similar conclusions can be drawn about **quicksort_dl** moded **qs**(+, +), **qs_dl**(+, +, −), **partition**(+, +, −, −). Thus, for a restricted class of goals, the occur-check freedom of these two programs can be established by means of the elementary techniques presented in Section 4.

10. WHEN OCCUR-CHECK IS NEEDED

Still, the results of this paper should be interpreted with caution. When Corollary 5.4 cannot be applied to a given program, the only alternatives are Corollaries 4.5, 6.5, or 7.9. In such cases, well- or weak-modedness is required, and thus, groundness of the inputs of the one-atom goal has to be assumed. Thus, no conclusion about the occur-check freedom for one-atom goals with nonground inputs can be drawn. For example, for the **member**

Table I

Program	Page	Moding	Well moded	Heads output linear	Nicely moded	Heads input linear	Strictly moded
member	45	(−, +)	Yes	Yes	Yes	Yes	Yes
member	45	(+, +)	Yes	Yes	Yes	No	Yes
prefix	45	(−, +)	Yes	Yes	Yes	Yes	Yes
prefix	45	(+, +)	Yes	Yes	Yes	No	Yes
suffix	45	(−, +)	Yes	Yes	Yes	Yes	Yes
suffix	45	(+, +)	Yes	Yes	Yes	No	Yes
naive reverse	48	**r**(+, −) **a**(+, +, −)	Yes	Yes	Yes	Yes	Yes
reverse-acc.	48	**r**(+, −) **r**(+, +, −)	Yes	Yes	Yes	Yes	Yes
delete	53	(+, +, −)	Yes	Yes	Yes	No	Yes
select	53	(+, +, −)	Yes	Yes	Yes	No	Yes
insertion sort	55	**s**(+, −) **i**(+, +, −)	Yes	Yes	Yes	Yes	Yes
tree-member	58	(−, +)	Yes	Yes	Yes	Yes	Yes
tree-member	58	(+, +)	Yes	Yes	Yes	No	Yes
isotree	58	(+, +)	Yes	Yes	Yes	No	Yes
substitute	60	(+, +, +, −)	Yes	Yes	Yes	No	Yes
pre-order	60	**p**(+, −) **a**(+, +, −)	Yes	Yes	Yes	Yes	Yes
in-order	60	**i**(+, −) **a**(+, +, −)	Yes	Yes	Yes	Yes	Yes
post-order	60	**p**(+, −) **a**(+, +, −)	Yes	Yes	Yes	Yes	Yes
polynomial	62	(+, +)	Yes	Yes	Yes	No	Yes
derivative	63	(+, +, −)	Yes	No	Yes	No	Yes
hanoi	64	**h**(+, +, +, −) **a**(+, +, −)	Yes	Yes	Yes	Yes	Yes
append_dl	241	(+, −, +, +, −, −)	Yes	Yes	Yes	Yes	Yes
append_dl	241	(+, −, +, −, −, −)	No	No	Yes	Yes	No
flatten_dl	241	**f**(+, +,) **f_dl**(+, +, −)	Yes	Yes	Yes	No	Yes
flatten	243	**f**(+, −) **f**(+, +, −)	Yes	Yes	Yes	Yes	Yes
reverse_dl	244	**r**(+, −) **r_dl**(+, −, +)	Yes	Yes	Yes	Yes	Yes
quicksort_dl	244	**q**(+, +) **q_dl**(+, +, −) **p**(+, +, −, −)	Yes	Yes	No	Yes	Yes
dutch	246	**dutch**(+, −) **di**(+, −, −, −)	Yes	Yes	Yes	Yes	Yes
dutch_dl	246	**dutch**(+, −) **di**(+, −, +, −, +, −, +)	Yes	Yes	Yes	Yes	Yes
parsing	258	**all**(+, −)	Yes	Yes	Yes	Yes	Yes

program of Example 8.6 no conclusion can be drawn for the goal ← **member(Y1s, Y2s)** when **Y1s** and **Y2s** are not ground. And, indeed, when **Y2s** = **[f(Y1s)]** one of the considered systems is {**Y1s** = **X**, **f(Y1s)** = **X**, **[ ]** = **Xs**}, which *is* subject to occur-check. Thus, after all, even for simple programs, the occur-check problem can very easily creep in.

In view of this discussion, it is easy to interpret the obtained results as a statement that the occur-check problem can arise only when considering some "ill-designed programs" or "ill-posed goals." The following delightful example offered to us by Dino Pedreschi (private communication) shows that it is not so.

Consider the typed lambda calculus and Curry's system of type assignment (see Curry and Feys [1958]). It involves statements of the form $x: t$, which should be read as "term x has type t." Finite sequences of such statements are denoted by R. The following three rules allow us to assign types to lambda terms:

$$\frac{x: t \in R}{R \vdash x: t}$$

$$\frac{R \vdash m: s \to t, R \vdash n: s}{R \vdash (mn): t}$$

$$\frac{R, x: s \vdash m: t}{R \vdash (\lambda x, m): s \to t}$$

These rules translate directly into the following PROLOG program, called **curry,** which can be used to compute a type assignment to a lambda term, if such an assignment exists (see, e.g., Reddy [1986]):

curry(R, var(X), T) ← **in([X, T], R).**
curry(R, apply(M, N), T) ← **curry(R, M, S → T), curry(R, N, S).**
curry(R, lambda(X, M), S → T) ← **curry([[X, S]|R], M, T).**

in(X, [Y|Xs]) ← **X ≠ Y, in(X, Xs).**
in(X, [X|Xs]).

In the first clause, the function symbol **var** is used to enforce the interpretation of **X** as a variable and, consequently, to prevent the instantiations of the clause to statements about the application and lambda abstraction. → is a binary function symbol written in an infix form.

Now consider the lambda term $\lambda x.(x\,x)$, to which no type can be assigned, and its PROLOG representation **m** = **lambda(x, apply(var(x), var(x)))**. Then it is easy to prove that the goal ← **curry([ ], m, T)** finitely fails. However, when the unification without the occur-check is used, then, if in step (5) of the Martelli-Montanari algorithm the substitution $\{x/t\}$ is performed in t, the goal ← **curry([ ], m, T)** diverges; otherwise, it succeeds! Thus, for the above program it is essential that a unification algorithm with the occur-check be used.

To deal with such programs, we propose the use of a program transformation. The following strengthening of Corollary 5.4 is essential:

THEOREM 10.1. *Let P and G be nicely moded. All systems of equations that are considered in the LD-derivations of $P \cup \{G\}$ and that are obtained using a clause whose head is input linear are NSTO.*

PROOF. By Lemma 5.3, all goals that appear in the LD-derivations of $P \cup \{G\}$ are nicely moded. But the first atom of a nicely moded goal is output linear and input-output disjoint. So, when the head of the input clause used is input linear, by NSTO via Modes Lemma 3.3, the corresponding system of equations is NSTO. □

To use this result, we transform a program and a goal into a nicely moded program and a nicely moded goal using the relation "$=_{oc}$", which is defined by the single clause $\mathbf{X} =_{oc} \mathbf{X}$ moded completely input. In the transformed program, only this relation is evaluated by the unification algorithm with the occur-check. The subscript "$_{oc}$" is added to distinguish it from the PROLOG built-in " = ", which performs unification without the occur-check.

The idea is to replace the variables that "contradict nice modedness" by "fresh" variables. Consider a clause $H \leftarrow \mathbf{B}$. Assume for simplicity that in every atom input positions occur first. We say that a given occurrence of a variable x in $\mathbf{B}$ *contradicts nicety* of $H \leftarrow \mathbf{B}$ if x occurs in an output position of an atom in $\mathbf{B}$ and, x occurs earlier in $\mathbf{B}$ or in an input position of H.

Now consider an occurrence of x in $\mathbf{B}$ that contradicts nicety. Let A be the atom in $\mathbf{B}$ in which this occurrence of x takes place, and let z be a fresh variable. Replace this occurrence of x in A by z, and denote the resulting atom as A'. Replace A in $\mathbf{B}$ by $A', z =_{oc} x$.

Scan $\mathbf{B}$ and perform this replacement repeatedly for all occurrences of variables that contradict the nicety of the original clause $H \leftarrow \mathbf{B}$. Call the resulting sequence of atoms $\mathbf{B}'$. It is easy to see that $H \leftarrow \mathbf{B}'$ is nicely moded. Note that, by unfolding (in the sense of Tamaki and Sato [1984]) the inserted calls of "$=_{oc}$" in $H \leftarrow \mathbf{B}'$, we obtain the original clause $H \leftarrow \mathbf{B}$.

The same transformation applied to an arbitrary goal transforms it into a nicely moded goal. Finally, a similar transformation ensures that the head H of $H \leftarrow \mathbf{B}$ is input linear. It suffices to replace repeatedly every occurrence of a variable x that contradicts input linearity of H by a fresh variable z and, to replace $\mathbf{B}$ by $z =_{oc} x, \mathbf{B}$. Clearly, the head of the resulting clause $H' \leftarrow \mathbf{B}'$ is input linear, and this transformation does not destroy the nicety of the clause. Again, the original clause $H \leftarrow \mathbf{B}$ can be obtained by unfolding the inserted calls of "$=_{oc}$".

The following result summarizes the effect of these transformations:

THEOREM 10.2. *For every program P and goal G, there exists a program P' and a goal G' such that*

—P' and G' are nicely moded;
—the head of every clause of P' different from $\mathbf{X} =_{oc} \mathbf{X}$ is input linear;
—P is the result of unfolding some calls of "$=_{oc}$" in P';
—G is the result of evaluating some calls of "$=_{oc}$" in G'; and

—*all systems of equations considered in the LD-derivations of* $P' \cup \{G'\}$, *but not associated with the calls of* " $=_{oc}$ ", *are NSTO.*

Note that the clause $\mathbf{X} =_{oc} \mathbf{X}$ is not input linear, so Corollary 5.4 cannot be applied to P' and G'.

PROOF. By construction and Theorem 10.1. □

As behavior of an unfolded program is closely related to the original program (see, e.g., Bossi and Cocco [1990]), it is justifiable to summarize this result by saying that every program and goal are equivalent to a nicely moded program and nicely moded goal, respectively, such that the heads of all clauses, except $\mathbf{X} =_{oc} \mathbf{X}$, are input linear. In the PROLOG execution of the latter program and goal, *only* the inserted calls of " $=_{oc}$ " need to be evaluated by means of a unification algorithm with the occur-check. Note that this transformation trades some "fragments" of the unification with the call of the relation " $=_{oc}$ ". These inserted calls of " $=_{oc}$ " can be viewed as the overhead needed to implement the original program correctly without the occur-check. Alternatively, the part of the transformation that ensures that the head of each clause is input linear could be dropped, and Theorem 10.1 could be applied.

To conclude, let us see how this transformation can be applied to the program **curry**. Consider the moding **curry**(+, +, −), **in**(+, +). The second clause of **curry** is not nicely moded, because the second occurrence of **S** contradicts its nicety. The corresponding transformed clause is nicely moded:

$$\mathbf{curry(R, apply(M, N), T)} \leftarrow \mathbf{curry(R, M, S \rightarrow T), curry(R, N, Z), Z} =_{oc} \mathbf{S}.$$

Another problem is that the head of the second clause of **in** is not input linear. The transformed version is

$$\mathbf{in(X, [Z \mid Xs])} \leftarrow \mathbf{Z} =_{oc} \mathbf{X}.$$

Call the transformed program **curry**′. It is nicely moded, and the head of every clause is input linear. By Corollary 5.4, we conclude that when **t** is linear and $Var(\mathbf{r}, \mathbf{m}) \cap Var(\mathbf{t}) = \emptyset$, **curry**′ $\cup$ {← **curry(r, m, t)**} is occur-check free. This allows us to draw the desired conclusion for the previously considered goal ← **curry**([ ], **m**, **T**) with **m** = **lambda(x, apply(var(x), var(x)))**.

The proposed transformation can be easily used "manually" and also can be efficiently implemented using two passes through the goal and the program, one to ensure nice modedness and the other to ensure the input linearity of the heads of the program clauses. This approach deals with the problem of inserting occur-checks at the source level and is orthogonal to that of Beer [1988], who proposed a revised implementation in which a new tag in the Warren Abstract Machine is used. This tag maintains information about the context in which a variable is used. This makes it possible to optimize the generated code by avoiding calls to the occur-check routine at the cost a small overhead at run time. It should be pointed out that in Beer's approach

unnecessary calls to the occur-check routine can be generated. For example, Beer [1988] reports that 49 occur-checks were invoked for the **quicksort** program.

11. CONCLUDING REMARKS

We have provided a systematic account of an approach for proving occur-check freedom of PROLOG programs based on syntactic analysis. In this approach, also advocated by Chadha and Plaisted [1994], it is shown that the existence of specific relationships between the variables of the goal and the variables of the program implies occur-check freedom. As a side effect, we have also explained how this approach can be used to prove the absence of floundering. Finally, we have shown how these results can be used to deal with the problem of insertion of occur-check tests in the program text by means of a program transformation.

The results on the occur-check freedom were established following a similar approach. First, systems of equations that are free from the occur-check were identified. Then, a property of a moded goal and a moded program was defined and proved to be "persistent" throughout the executions of the programs. This property ensured that in all executions the selected atoms lead to desired systems of equations. To deal with programs that use difference lists, a modification of these results, which involved two different modes, was needed.

Two other approaches to proving occur-check freedom were proposed in the literature. One is based on the abstract interpretations, and the other uses the attribute grammars. The first approach originated with Plaisted [1984] and was further developed by Søndergaard [1986]. Søndergaard used an abstract interpretation in which the information on the possibility of creating a *sharing* of a variable or forming multiple occurrences of the same variable (called *spawning*) is maintained. Then using abstract unification this sharing and spawn information is propagated among the program representation, in order to discover whether the abstract unification may lead to circularity. The absence of circularity guarantees occur-check freedom.

The abstract interpretations were also used to prove the absence of floundering. The strongest results were obtained by Marriott et al. [1990]. To this end they expressed a data-flow analysis of a general program by means of a finitely computable approximation of the denotational semantics. Such an approximation is determined by suitable functions that approximate the groundness information and the unification algorithm.

The approach based on attribute grammars was originated by Deransart and Maluszynski [1985] and was further developed in Deransart et al. [1991]. This approach exploits a close relationship between the abstract skeletons associated with the executions of a goal and a program and the derivation trees of the grammar associated with the goal and the program. The attributes are used to model relations between equations (like variable sharing). It is shown that the occur-check freedom is implied by a combination of

syntactic conditions and of noncircularity of the attribute dependency scheme. This approach was applied recently by Dumant [1992] to deal with the problem of inserting occur-checks in arbitrary resolution strategies.

The syntactic approach advocated in this paper is much more straightforward and has, in our opinion, two important advantages: First, it can be trivially implemented; and second, it can be easily used "manually." In fact, we have shown that it can be readily applied to several well-known PROLOG programs. Consequently, it seems to be sufficient to deal satisfactorily with most common PROLOG programs. It would be interesting to clarify the precise relationship between this approach and the other two.

APPENDIX

We prove here the promised undecidability results and Lemma 5.3. The following theorem summarizes the undecidability issues:

THEOREM A.1. *For some moded program P, the following properties are undecidable*:

—*G is such that $P \cup \{G\}$ is occur-check free,*
—*G is such that all LD-derivatives of $P \cup \{G\}$ are data driven, and*
—*G is such that all LD-derivations of $P \cup \{G\}$ are output driven.*

PROOF. Below, M_P denotes the least Herbrand model of a program P and L_P denotes the language determined by P. Let P_0 be a strictly moded program; let p be a new binary relation, moded $p(+, -)$; and let $P_1 = P_0 \cup \{p(y, f(y)) \leftarrow\}$.

The system $E = \{x = y,\ x = f(y)\}$ is not NSTO, and by Corollary 6.5, for every ground atom A, $P_1 \cup \{\leftarrow A\}$ is occur-check free. Thus, for a ground atom A in L_{P_0}, $P_1 \cup \{\leftarrow A, p(x, x)\}$ is not occur-check free iff E is considered in an LD-derivation of $P_1 \cup \{\leftarrow A, p(x, x)\}$ iff there exists an LD-refutation of $P_1 \cup \{\leftarrow A\}$ iff (by the completeness of LD-resolution) $A \in M_{P_1}$ iff $A \in M_{P_0}$.

So we have shown that, for every ground atom A in L_{P_0}, $A \notin M_{P_0}$ iff $P_1 \cup \{\leftarrow A, p(x, x)\}$ is occur-check free. An analogous argument using Theorem 4.2 (resp., Theorem 5.2) shows that, for every ground atom A in L_{P_0}, $A \notin M_{P_0}$ iff all LD-derivations of $P \cup \{\leftarrow A, p(x, x)\}$ are data driven (resp., iff all LD-derivations of $P \cup \{\leftarrow A, p(x, x)\}$ are output driven).

Thus, to prove the theorem it suffices to show that there exists a strictly moded program P_0 for which the set M_{P_0} is undecidable. Now, Corollary 4.7 in Apt [1990] gave this result for some program P_0, so it suffices to check that this corollary can be appropriately sharpened.

To this end it is enough to show that every recursive function can be computed by a strictly moded program. The proof of computability of recursive functions by logic programs given in Shepherdson [1991] and based on a straightforward encoding of register machines yields the needed result. In-

deed, the obvious moding $p(+, +, \ldots, -)$ for all relations p turns the generated logic programs into strictly moded ones. This completes the proof. □

We now turn to Lemma 5.3. We start by establishing a number of auxiliary lemmas. The notation used below was defined in Section 1.

LEMMA A.2. *Let θ be a substitution, and let* $\mathbf{s}$ *and* $\mathbf{t}$ *be sequences of terms such that*

—$Var(\mathbf{s}) \cap Var(\mathbf{t}) = \emptyset$,
—$Ran(\theta \mid Var(\mathbf{s})) \cap Ran(\theta \mid Var(\mathbf{t})) = \emptyset$,
—$Var(\mathbf{s}) \cap Ran(\theta \mid Var(\mathbf{t})) = \emptyset$, *and*
—$Var(\mathbf{t}) \cap Ran(\theta \mid Var(\mathbf{s})) = \emptyset$.

Then $Var(\mathbf{s}\theta) \cap Var(\mathbf{t}\theta) = \emptyset$.

PROOF. This is an immediate consequence of the fact that for any sequence of terms $\mathbf{u}$ and substitution σ we have $Var(\mathbf{u}\sigma) \subseteq Var(\mathbf{u}) \cup Ran(\sigma \mid Var(\mathbf{u}))$. □

The next two lemmas use the following notion:

Definition A.3. A substitution $\{x_1/t_1, \ldots, x_n/t_n\}$ is called *linear* if $t_1, \ldots, t_n$ is a linear family of terms.

LEMMA A.4. *Let θ be a substitution, and let* $\mathbf{t}$ *be a family of terms. Suppose that*

—θ *is linear*,
—$\mathbf{t}$ *is linear, and*
—$Ran(\theta) \cap Var(\mathbf{t}) = \emptyset$.

Then $\mathbf{t}\theta$ *is a linear family of terms as well.*

PROOF. Suppose a variable x has two distinct occurrences in $\mathbf{t}\theta$. Then one of the following statements holds in regard to these occurrences:

—they are both occurrences in $Range(\theta)$;
—they are both occurrences in $\mathbf{t}$; or
—one is an occurrence in $Range(\theta)$, and the other is an occurrence in $\mathbf{t}$.

But each assumption of the lemma excludes the corresponding statement above, so the claim follows. □

The following lemma is stated in Deransart and Maluszynski [1985]:

LEMMA A.5. *Consider two atoms A and H with the same relation symbol. Suppose that*

—*they have no variable in common, and*
—*A is linear.*

Assume that A and H are unifiable. Then there exists a relevant mgu θ of A and H such that

—$\theta \mid Var(H)$ is linear, and
—$Ran\ (\theta \mid Var(H)) \subseteq Var(A)$.

PROOF. Given a set of equations E, let

$$RVar(E) = \bigcup_{s=t \in E} Var(t),$$

$$E_H = \{s = t \in E \mid Var(s) \cap Var(H) \neq \emptyset\}.$$

Also, call a term t *singular* if each variable of it occurs in E only once. Call an equation $s = t$ *singular* if either s is a variable and singular or t is singular. Finally call E *singular* if every equation in it is singular.

Consider the set of equations $H = A$ (note the reverse ordering). We claim that the conjunction of the following three statements is initially true for E equal to $H = A$ and is preserved by the action (1), (2), (3), (5), and (6) of the Martelli–Montanari algorithm:

(1) E_H is right linear,
(2) $RVar(E) \subseteq Var(A)$, and
(3) E is singular.

The checking of this claim is simple. The only subtle point arises when action (5) applies. Let $x = t$ be the chosen equation; x occurs elsewhere, so it is not singular. Thus t is singular, and by Lemma A.4, after performing action (5) E_H remains right linear. Moreover, x then becomes singular, so the equation $x = t$ remains singular, though now on account of x. The other equations clearly remain singular. The remaining cases are straightforward.

This shows that, when applying to the set $H = A$, the Martelli–Montanari algorithm with action (4) omitted, eventually a set of equations E is produced, which satisfies statements (1–3) and to which only action (4) can be applied. Now let

$$E_1 = \{s = t \in E \mid s \text{ is not a variable}\},$$
$$E_2 = E - E_1.$$

None of the actions (1), (2), (3), (5), or (6) can be applied to E_1. Thus, each of its equations is of the form $s = x$, where x is a variable. Moreover, by virtue of statement (3),

$$E_1 = \{s_1 = x_1, \ldots, s_n = x_n\},$$

where $x_1, \ldots, x_n$ are different variables, each of which occurs in E only once. Thus,

$$F = \{x = s \mid s = x \in E_1\}$$

is in solved form and determines a relevant mgu θ_1 of E_1 such that $E_2\theta_1 = E_2$.

Next, none of the actions of the Martelli–Montanari algorithm can be applied to E_2. Thus, E_2 is in solved form and determines a relevant mgu θ_2 of E_2 and so of $E_2\theta_1$. By statement (1), $\theta_2 \mid Var(H)$ is linear, and by statement (2), $Ran(\theta_2 \mid Var(H)) \subseteq Var(A)$.

By Lemma 2.4, $\theta_1\theta_2$ is a relevant mgu of E. Moreover, by statement (2), $Dom(\theta_1) \subseteq Var(A)$, so by the disjointness of A and H, we get $Dom(\theta_1) \cap Var(H) = \emptyset$. Thus, $\theta_1\theta_2 \mid Var(H) = \theta_2 \mid Var(H)$. This shows that $\theta = \theta_1\theta_2$ is the desired mgu. □

In Deransart and Maluszynski [1985, proposition 3, p. 143], this claim was actually stated (without proof) for an arbitrary mgu θ. However, for $A = p(z, u)$, $H = p(x, y)$, and $\theta = \{x/y, y/u, z/y\}$, we get a counterexample.

Below, given an atom A, we denote by $VarIn(A)$ (resp., $VarOut(A)$) the set of variables occurring in the input (resp., output) positions of A. Similar notation is used for sequences of atoms.

Finally, we need the following technical lemma:

LEMMA A.6. *Consider two atoms A and H with the same relation symbol. Suppose that*

—they have no variable in common, and

—A is input-output disjoint and output linear.

Assume that A and H are unifiable. Then there exists a relevant mgu θ of A and H such that for $V = VarOut(H) - VarIn(H)$, $\eta_1 = \theta \mid V$, and $\eta_2 = \theta \mid VarIn(H)$

(i) η_1 is linear.

(ii) $Ran(\eta_1) \subseteq Var(A)$, and

(iii) $Ran(\eta_2) \cap (Ran(\eta_1) \cup V) = \emptyset$.

PROOF. Let $i_1^A, \ldots, i_m^A$ (resp., $i_1^H, \ldots, i_m^H$) be the therms filling in the input positions of A (resp., H) and let $o_n^A, \ldots, o_n^A$ (resp., $o_1^H, \ldots, o_n^H$) be the terms filling in the output positions of A (resp., H). Let θ_1 be the relevant mgu of $\{o_1^A = o_1^H, \ldots, o_n^A = o_n^A\}$ constructed in the proof of Lemma A.5. By the disjointness of A and H, we have $\theta_1 \mid Var(H) = \theta_1 \mid VarOut(H)$, so by Lemma A.5,

$$\theta_1 \mid Var(H) \quad \text{is linear} \tag{3}$$

and

$$Ran(\theta_1 \mid Var(H)) \subseteq VarOut(A). \tag{4}$$

Let θ_2 be a relevant mgu of $\{i_1^A = i_1^H, \ldots, i_m^A = i_m^H\}\theta_1$. By Lemma 2.4, θ_2 exists and $\theta = \theta_1\theta_2$ is a relevant mgu of $A = H$.

By the relevance of θ_1, we have $Dom(\theta_1) \subseteq VarOut(A) \cup VarOut(H)$, so by the input-output disjointness of A and the disjointness of A and H, we get $\{i_1^A = i_1^H, \ldots, i_m^A = i_m^H\}\theta_1 = \{i_1^A = i_1^H\theta_1, \ldots, i_m^A = i_m^H\theta_1\}$. By the relevance of θ_2,

we have $Var(\theta_2) \subseteq Var(\{i_1^A = i_1^H\theta_1, \ldots, i_m^A = i_m^H\theta_1\}) \subseteq VarIn(A) \cup VarIn(H) \cup Ran(\theta_1 \mid VarIn(H))$. Thus, by the disjointness of A and H and (4),

$$Var(\theta_2) \cap V = \emptyset. \tag{5}$$

For the same reasons and, additionally, by the input-output disjointness of A and (3),

$$Var(\theta_2) \cap Ran(\theta \mid V) = \emptyset. \tag{6}$$

Now, (5) and (6) imply that

$$\eta_1 = \theta_1 \mid V. \tag{7}$$

Thus, $\eta_1 \subseteq \theta_1 \mid Var(H)$; so by (3) we conclude (i), and by (4) we conclude (ii).

Now consider η_2. Note that $\eta_2 \subseteq (\theta_1 \mid VarIn(H))\theta_2$, so

$$Ran(\eta_2) \subseteq Ran(\theta_1 \mid VarIn(H)) \cup Var(\theta_2). \tag{8}$$

But, by (3), (6), (4), disjointness of A and H, and (5),

$$(Ran(\theta_1 \mid VarIn(H)) \cup Var(\theta_2)) \cap (Ran(\theta_1 \mid V) \cup V) = \emptyset;$$

so by (8) and (7) we conclude (iii). □

We can now return to Lemma 5.3.

PROOF OF LEMMA 5.3. First, we prove three claims that appropriately refine those of Lemma 4.3:

CLAIM 1. *Suppose that A and H satisfy the assumptions of Lemma A.6, and assume that θ is a relevant mgu of $A = H$ that satisfies conditions (i)–(iii) of Lemma A.6. Let $H \leftarrow \mathbf{B}$ be a nicely moded clause with no variables in common with A. Then $\leftarrow \mathbf{B}\theta$ is nicely moded.*

PROOF. Below, by *standardization apart* we mean the assumption that $H \leftarrow \mathbf{B}$ and A have no variables in common. Let V, η_1, and η_2 be as in the formulation of Lemma A.6.

Let $\theta_1 = \theta \mid VarOut(\mathbf{B})$ and $\theta_2 = \theta \mid (VarIn(\mathbf{B}) - VarOut(\mathbf{B}))$. We first establish some claims about θ_1 and θ_2. By standardization apart and the definition of a nicely moded clause,

$$VarOut(\mathbf{B}) \cap (Var(A) \cup Var(H)) \subseteq V, \tag{9}$$

so by the fact that θ is relevant,

$$\theta_1 \subseteq \eta_1. \tag{10}$$

Thus, by the linearity of η_1 (condition (i) of Lemma A.6),

$$\theta_1 \quad \text{is linear.} \tag{11}$$

Moreover by (10), (ii) of Lemma A.6, and standardization apart,

$$Ran(\theta_1) \cap Var(\mathbf{B}) = \emptyset. \tag{12}$$

Now, let $\theta_2' = \theta_2 \mid V$ and $\theta_2'' = \theta_2 \mid VarIn(H)$. We have

$$\theta_2 = \theta_2' \dot{\cup} \theta_2'', \tag{13}$$

$$\theta_2' \subseteq \eta_1, \tag{14}$$

and

$$\theta_2'' \subseteq \eta_2. \tag{15}$$

Now consider θ_2'. We have $Dom(\theta_1) \cap Dom(\theta_2) = \varnothing$, so $Dom(\theta_1) \cap Dom(\theta_2') = \varnothing$. Thus, by (10), (14), and the linearity of η_1,

$$Ran(\theta_2') \cap Ran(\theta_1) = \varnothing. \tag{16}$$

Moreover, by (14), (ii) of Lemma A.6, and standardization apart

$$Ran(\theta_2') \cap VarOut(\mathbf{B}) = \varnothing. \tag{17}$$

Now consider θ_2''. By (10), (15), and (iii) of Lemma A.6,

$$Ran(\theta_2'') \cap Ran(\theta_1) = \varnothing. \tag{18}$$

Also, by the fact that θ is relevant, $Ran(\theta_2'') \subseteq Var(A) \cup Var(H)$, so by (9), $Ran(\theta_2'') \cap VarOut(\mathbf{B}) \subseteq V$. Thus, by (15) and (iii) of Lemma A.6,

$$Ran(\theta_2'') \cap VarOut(\mathbf{B}) = \varnothing. \tag{19}$$

Combining (16) with (18) and (17) with (19), we get, by virtue of (13),

$$Ran(\theta_2) \cap (Ran(\theta_1) \cup VarOut(\mathbf{B})) = \varnothing. \tag{20}$$

Now consider $\mathbf{B}$ in more detail. Suppose $\mathbf{B} = p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$. By assumption, $\mathbf{t}_1, \ldots, \mathbf{t_n}$ is a linear family of terms, and for $i \in [1, n]$, $\mathbf{t_i}\theta \equiv \mathbf{t_1}\theta_1$. So, by (11), (12), and Lemma A.4, $\mathbf{t}_1\theta, \ldots, \mathbf{t_n}\theta$ is a linear family of terms as well.

Now fix $i \in [1, n]$ and $j \in [i, n]$. We have

$$Ran(\theta \mid Var(\mathbf{s_i})) \subseteq Ran(\theta_1 \mid Var(\mathbf{s_i})) \cup Ran(\theta_2 \mid Var(\mathbf{s_i})) \tag{21}$$

and

$$Ran\big(\theta \mid Var(\mathbf{t_j})\big) = Ran\big(\theta_1 \mid Var(\mathbf{t_j})\big). \tag{22}$$

$\leftarrow \mathbf{B}$ is nicely moded, so

$$Var(\mathbf{s_i}) \cap Var(\mathbf{t_j}) = \varnothing. \tag{23}$$

Thus, by the linearity of θ_1, $Ran(\theta_1 \mid Var(\mathbf{s_i})) \cap Ran(\theta_1 \mid Var(\mathbf{t_j})) = \varnothing$ and consequently, by (21), (22), and (20),

$$Ran(\theta \mid Var(\mathbf{s_i})) \cap Ran\big(\theta \mid Var(\mathbf{t_j})\big) = \varnothing. \tag{24}$$

Next, by (22) and (12),

$$Var(\mathbf{s_i}) \cap Ran\big(\theta \mid Var(\mathbf{t_j})\big) = \varnothing. \tag{25}$$

Finally, by (21), (12), and (20),

$$Var(\mathbf{t}_j) \cap Ran(\theta \mid Var(\mathbf{s}_i)) = \emptyset. \tag{26}$$

Now, by (23), (24), (25), (26), and Lemma A.2, we conclude that $Var(\mathbf{s}_i\theta) \cap Var(\mathbf{t}_j\theta) = \emptyset$.

This proves that $\leftarrow \mathbf{B}\theta$ is nicely moded. □

CLAIM 2. *Let θ be a substitution, and let $\leftarrow \mathbf{A}$ be a nicely moded goal such that $Var(\theta) \cap VarOut(\mathbf{A}) = \emptyset$. Then $\leftarrow \mathbf{A}\theta$ is nicely moded as well.*

PROOF. For any term s and a substitution σ, we have $Var(s\sigma) \subseteq Var(s) \cup Var(\sigma)$. Moreover, for any term t occurring at an output position of $\mathbf{A}$ by the assumption about θ we have $t\theta = t$. The claim now follows by the definition of a nicely moded goal. □

CLAIM 3. *Suppose $\leftarrow \mathbf{A}$ and $\leftarrow \mathbf{B}$ are nicely moded goals such that $VarOut(\mathbf{A}) \cap Var(\mathbf{B}) = \emptyset$. Then $\leftarrow \mathbf{B}, \mathbf{A}$ is a nicely moded goal as well.*

PROOF. Immediate by the definition of a nicely moded goal. □

Now consider a nicely moded goal $\leftarrow A, \mathbf{A}$ and a nicely moded clause $H \leftarrow \mathbf{B}$ that is variable disjoint with it, such that A and H unify. Observe that A and H satisfy the assumptions of Lemma A.6. Assume that θ is a relevant mgu of $A = H$ that satisfies conditions (i)–(iii) of Lemma A.6. By Claim 1, $\leftarrow \mathbf{B}\theta$ is nicely moded.

θ is relevant, and $Var(A) \cap VarOut(\mathbf{A}) = \emptyset$; so by standardization apart,

$$Var(\theta) \cap VarOut(\mathbf{A}) = \emptyset. \tag{27}$$

By Claim 2, $\leftarrow \mathbf{A}\theta$ is nicely moded.

But (27) implies that $VarOut(\mathbf{A}\theta) = VarOut(\mathbf{A})$. Moreover, $Var(\mathbf{B}\theta) \subseteq Var(\mathbf{B}) \cup Var(\theta)$, and by standardization apart, $VarOut(\mathbf{A}) \cap Var(\mathbf{B}) = \emptyset$; so, again by (27),

$$VarOut(\mathbf{A}\theta) \cap Var(\mathbf{B}\theta) = \emptyset. \tag{28}$$

Now (28) establishes the last assumption of Claim 3 with $\leftarrow \mathbf{A}$ replaced by $\leftarrow \mathbf{A}\theta$ and $\leftarrow \mathbf{B}$ replaced by $\leftarrow \mathbf{B}\theta$. We conclude by Claim 3 that the LD-resolvent $\leftarrow (\mathbf{B}, \mathbf{A})\theta$ of the goal $\leftarrow A, \mathbf{A}$ and the clause $H \leftarrow \mathbf{B}$ is nicely moded. To draw the same conclusion for an arbitrary LD-resolvent of $\leftarrow A, \mathbf{A}$ and $H \leftarrow \mathbf{B}$, it suffices to use Lemma 2.3. □

ACKNOWLEDGMENTS

We thank Pierre Deransart and the referees for constructive remarks on the subject of this paper, and Dino Pedreschi for providing the **curry** program example.

REFERENCES

ALBERT, L., CASAS, R., AND FAGES, F. 1993. Average case analysis of unification algorithms. *Theor. Comput. Sci. 113*, 1, 24, 3–34.

APT, K. R. 1990. Logic programming. In *Handbook of Theoretical Computer Science*, vol. B, J. van Leeuwen, Ed. Elsevier, North-Holland, New York, 493–574.

APT, K. R. AND DOETS, K. 1994. A new definition of SLDNF-resolution. *J. Logic Program. 18*, 177–190.

APT, K. R., AND PELLEGRINI, A. 1992. Why the occur-check is not a problem. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP 92)*, M. Bruynooghe and M. Wirsing, Eds. Lecture Notes in Computer Science, vol 631, Springer-Verlag, Berlin, 69–86.

APT, K. R., VAN EMDE BOAS, P., AND WELLING, A. 1994. The STO problem is NP-hard. Res. Rep. CT-94-08, Dept of Mathematics and Computer Science, Univ. of Amsterdam, The Netherlands.

BEER, J. 1988. The occur-check problem revisited. *J. Logic Program. 5*, 243–261.

BOSSI, A., AND COCCO, N. 1990. Basic transformation operations for logic programs which preserve computed answer substitutions. Tech. Rep. 16, Dipartimento di Matematica Pura ed Applicata, Universitá di Padova, Italy.

CHADHA, R., AND PLAISTED, D. A. 1994. Correctness of unification without occur check in Prolog. *J. Logic Program. 18*, 99–122.

CLARK, K. L. 1979. Predicate logic as a computational formalism. Res. Rep. DOC 79/59, Dept. of Computing, Imperial College, London.

CURRY, H. B., AND FEYS, R. 1958. *Combinatory Logic. Vol. I. Studies in Logic and the Foundation of Mathematics*. North-Holland, Amsterdam.

DECKER, H. 1991. On generalized cover axioms. In *Proceedings of the 8th International Conference on Logic Programming*, K. Furukawa, Ed. MIT Press, Paris, France, 693–707.

DEMBINSKI, P., AND MALUSZYNSKI, J. 1985. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the International Symposium on Logic Programming* (Boston, Mass.). 29–38.

DERANSART, P., AND MALUSZYNSKI, J. 1985. Relating logic programs and attribute grammars. *J. Logic Program., 2*, 119–156.

DERANSART, P., FERRAND, G., AND TÉGUIA, M. 1991. NSTO programs (not subject to occur-check). In *Proceedings of the International Logic Symposium*, V. Saraswat and K. Ueda, Eds. MIT Press, Cambridge, Mass., 533–547.

DRABENT, W. 1987. Do logic programs resemble programs in conventional languages? In *International Symposium on Logic Programming* (San Francisco, Calif., Aug.) IEEE, New York, 389–396.

DUMANT, B. 1992. Checking soundness of resolution schemes. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, K. R. Apt, Ed. MIT Press, Cambridge, Mass., 37–51.

LASSEZ, J.-L., MAHER, M. J., AND MARRIOTT, K. 1988. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, Los Altos, Calif., 587–625.

LLOYD, J. W. 1987. *Foundations of Logic Programming*. 2nd ed. Springer-Verlag, Berlin.

MARIOTT, K., SØNDERGAARD, H., AND DART, P. 1990. A characterization of non-floundering logic programs. In *Proceedings of the North American Conference on Logic Programming '90*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, Mass. 661–680.

MARTELLI, A., AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 258–282.

MELLISH, C. S. 1981. The automatic generation of mode declarations for Prolog programs. DAI Res. Pap. 163, Dept of Artificial Intelligence, Univ. of Edinburgh, Aug.

PLAISTED, D. A. 1984. The occur-check problem in Prolog. In *Proceedings of the International Conference on Logic Programming*. IEEE, New York, 272–280.

REDDY, U. S. 1984. Transformation of logic programs into functional programs. In *International Symposium on Logic Programming* (Silver Spring, Md., Feb.). IEEE, New York, 187–198.

REDDY, U. S. 1986. On the relationship between logic and functional languages. In *Functional*

and Logic Programming, D. DeGroot and G. Lindstrom, Eds. Prentice-Hall, Englewood Cliffs, N.J., 3–36.

ROSENBLUETH, D. A. 1991. Using program transformation to obtain methods for eliminating backtracking in fixed-mode logic programs. Tech. Rep. 7, Instituto de Investigaciones en Matematicas Aplicadas y en Sistemas. Universidad Nacional Autonoma de Mexico.

SHEPHERDSON, J. C. 1991. Unsolvable problems for SLDNF resolution. *J. Logic Program. 10*, 1, 19–22.

SØNDERGAARD, H. 1986. An application of abstract interpretation of logic programs: Occur check reduction. In *Proceedings of ESOP'86* (Sarrbruecken, Germany). 327–338.

STERLING, L., AND SHAPIRO, E. 1986. *The Art of Prolog*, MIT Press, Cambridge, Mass.

STROETMAN, K. 1993. A completeness result for SLDNF resolution. *J. Logic Program. 15*, 337–357.

TAMAKI, H., AND SATO, T. 1984. Unfold/fold transformation of logic programs. In *Proceedings of the 2nd International Conference on Logic Programming* (Uppsala, Sweden). 127–137.

Received October 1992; accepted July 1993