On the unification free Prolog programs

K.R. Apt, S. Etalle

Computer Science/Department of Software Technology

Report CS-R9331 May 1993

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications. SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 4079, 1009 AB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

On the Unification Free Prolog Programs

Krzysztof R. Apt
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
and
Faculty of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands
and
Sandro Etalle
Dipartimento di Matematica Pura ed Applicata
Università di Padova
Via Belzoni 7, 35131 Padova, Italy

Abstract

We provide simple conditions which allow us to conclude that in case of several well-known Prolog programs the unification algorithm can be replaced by iterated matching. The main tools used here are types and generic expressions for types. As already noticed by other researchers, such a replacement offers a possibility of improving the efficiency of program's execution.

1991 Mathematics Subject Classification: 68Q40, 68T15.
CR Categories: F.3.2., F.4.1, H.3.3, I.2.3.
Keywords and Phrases: iterated matching, Prolog programs.

Notes. The work of the first author was partly supported by ESPRIT Basic Research Action 6810 (Compulog 2). This research was done partly during the second author's stay at Centre for Mathematics and Computer Science, Amsterdam. This paper will appear as invited lecture in: Proc. of Conference on Mathematical Foundations of Computer Science (MFCS '93), Lecture Notes in Computer Science, Springer-Verlag, S. Sokolowski (editor).

1 Introduction

Unification is heralded as one of the crucial features offered by Prolog, so it is natural to ask whether it is actually used in specific programs. The aim of this paper to identify natural conditions under which unification can be replaced by iterated matching and to show that they are applicable to several well-known Prolog programs. These conditions can be statically checked without analyzing the search trees for the queries. For programs which use ground inputs they can be efficiently tested.

The problem of replacing unification by iterated matching was already studied in the literature by a number of researchers – see e.g. Deransart and Maluszynski [DM85b], Maluszynski and Komorowski [MK85] and Attali and Franchi-Zannettacci [AFZ88]. As in the previous works on this subject, we use modes, which indicate how the arguments of a relation should be used. Our results improve upon the previous ones due to the additional use of types. This allows us to deal with non-ground inputs.

We use here a simple notion of a type, which is a set of terms closed under substitution. The main tool in our approach is the concept of a *generic expression*. Intuitively, a term s is a generic expression for a type T if it is more general than all elements of T which unify with s. This simple notion turns out to be crucial here, because surprisingly often the input positions of the heads of program clauses are filled in by generic expressions for appropriate types.

Report CS-R9331
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

We combine in our analysis the use of generic expressions with the notion of a *well-typed program*, recently introduced by Bronsard, Lakshman and Reddy [BLR92], which allows us to ensure that the input positions of the selected atoms remain correctly typed. As the table included at the end of this paper shows, our results can be applied to astonishingly many Prolog programs.

2 Preliminaries

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-in's, like $=$, $\neq$, $>$, etc, and assume that they are resolved in the way conforming to their interpretation.

We work here with *queries*, that is sequences of atoms, instead of *goals*, that is constructs of the form $\leftarrow Q$, where Q is a query. Apart from this we use the standard notation of Lloyd [Llo87] and Apt [Apt90]. In particular, given a syntactic construct E (so for example, 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 $\theta = \{x_1/t_1, ..., x_n/t_n\}$ 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 *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 $\theta|V$ the substitution obtained from θ by restricting its domain to V.

2.1 Unifiers

Given two sequences of terms $\mathbf{s} = s_1, ..., s_n$ and $\mathbf{t} = t_1, ..., t_n$ of the same length we abbreviate the set of equations $\{s_1 = t_1, ..., s_n = t_n\}$ to $\{\mathbf{s} = \mathbf{t}\}$ and the sequence $s_1\theta, ..., s_n\theta$ to $\mathbf{s}\theta$. Two atoms can unify only if they have the same relation symbol. With two atoms $p(\mathbf{s})$ and $p(\mathbf{t})$ to be unified we associate the set of equations $\{\mathbf{s} = \mathbf{t}\}$. In the applications we often refer to this set as $p(\mathbf{s}) = p(\mathbf{t})$. A substitution θ such that $\mathbf{s}\theta = \mathbf{t}\theta$ is called a *unifier* of the set of equations $\{\mathbf{s} = \mathbf{t}\}$. Thus the set of equations $\{\mathbf{s} = \mathbf{t}\}$ has the same unifiers as the atoms $p(\mathbf{s})$ and $p(\mathbf{t})$.

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 $Var(\theta) \subseteq Var(E)$.

The following lemma was proved in Lassez, Marriot and Maher [LMM88].

Lemma 2.1 *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 well-known lemma allows us to search for mgu's in an iterative fashion.

Lemma 2.2 *Let E_1, 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 θ_1 exists and for any such θ_1 an appropriate θ_2 exists, as well.* □

2.2 Modes and Types

Below we extensively use modes.

Definition 2.3 Consider an n-ary relation symbol p. By a *mode* for p we mean a function m_p from $\{1, \ldots, n\}$ to the set $\{+, -\}$. If $m_p(i) = '+'$, we call i an *input position* of p and if $m_p(i) = '-'$, we call i an *output position* of p (both w.r.t. m_p). □

Modes indicate how the arguments of a relation should be used. The definition of moding assumes one mode per relation in a program. Multiple modes may be obtained by simply renaming the relations. When every considered relation has a mode associated with it, we can talk about input positions and output positions of an atom. In that case for an atom A we denote by $In(A)$ and $Out(A)$ the family of terms filling in, respectively, the input and the output positions of A. 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.

In the sequel, we also use types. The following very general definition is sufficient for our purposes.

Definition 2.4 A *type* is a decidable set of terms closed under substitution. □

We call a type T *ground* if all its elements are ground, and *non-ground* if some of its elements is non-ground. By a *typed term* we mean a construct of the form $s : S$ where s is a term and S is a type. Given a sequence $\mathbf{s} : \mathbf{S} = s_1 : S_1, \ldots, s_n : S_n$ of typed terms we write $\mathbf{s} \in \mathbf{S}$ if for $i \in [1, n]$ we have $s_i \in S_i$.

Certain types will be of special interest:

U — the set of all terms,

List — the set of lists,

BinTree — the set of binary trees,

Nat — the set of natural numbers,

Ground — the set of ground terms.

Of course, the use of the type *List* assumes the existence of the empty list `[]` and the list constructor `[.|.]` in the language, and the use of the type *Nat* assumes the existence of the numeral `0` and the successor function `s(.)`, etc. Throughout the paper we fix a specific set of types, denoted by *Types*, which includes the above ones.

We also associate types with relation symbols.

Definition 2.5 Consider an n-ary relation symbol p. By a *type* for p we mean a function t_p from $[1, n]$ to the set *Types*. If $t_p(i) = T$, we call T *the type associated with the position i of p*. Assuming a type t_p for the relation p, we say that an atom $p(s_1, \ldots, s_n)$ is *correctly typed in position i* if $s_i \in t_p(i)$. □

When every considered relation has a mode and a type associated with it, we can talk about types of input positions and of output positions of an atom. An n-ary relation p with a mode m_p and type t_p will be denoted by

$$p(m_p(1) : t_p(1), \ldots, m_p(n) : t_p(n)).$$

For example, $\mathtt{app}(+ : List,\ + : List,\ - : U)$ denotes a ternary relation `app` with the first two positions moded as input and typed as *List*, and the third position moded as output and typed as U.

From the context it will be always clear whether modes and/or types are assumed for the considered relations. In this paper we shall always use types in presence of modes.

3 Solvability by (Iterated) Matching

3.1 Solvability by Matching

We begin by recalling the following concepts.

Definition 3.1 Consider a set of equations $E = \{\mathbf{s} = \mathbf{t}\}$.

- A substitution θ such that either $Dom(\theta) \subseteq Var(\mathbf{s})$ and $\mathbf{s}\theta = \mathbf{t}$ or $Dom(\theta) \subseteq Var(\mathbf{t})$ and $\mathbf{s} = \mathbf{t}\theta$, is called a *match* for E.
- E is called *left-right disjoint* if $Var(\mathbf{s}) \cap Var(\mathbf{t}) = \emptyset$. □

Clearly, if E is left-right disjoint, then a match for E is also a relevant mgu of E. The sets of equations we consider in this paper will always satisfy this disjointness proviso due to the standardization apart.

Definition 3.2 Let E be a left-right disjoint set of equations. We say that E is *solvable by matching* if E is unifiable implies that a match for E exists. □

A simple test allowing us to determine whether a given set of equations is solvable by matching is summarized in the following lemma.

Definition 3.3

- We call an atom (resp. a term) a *pure atom* (resp. *pure variable term*) if it is of the form $p(\mathbf{x})$ with $\mathbf{x}$ a sequence of different variables.
- Two atoms (resp. terms) are called *disjoint* if they have no variables in common. □

Lemma 3.4 (Matching 1) *Consider two disjoint atoms A and H with the same relation symbol. Suppose that*

- *one of them is ground or pure.*

Then $A = H$ is solvable by matching.

Proof. Clear. □

3.2 Generic Expressions

A more interesting condition for solvability by matching can be obtained using types. For example, assume the standard list notation and consider a term $t = [x|y]$ with x and y variables. Note that whenever a list l unifies with t, then l is an instance of t, i.e $l = t$ is solvable by matching.

Thus solvability by matching can be sometimes deduced from the shape of the considered terms. This motivates the following definition.

Definition 3.5 Let T be a type. A term t is a *generic expression* for T if for every $s \in T$ disjoint with t, if s unifies with t then s is an instance of t. □

In other words, t is a generic expression for type T iff all left-right disjoint equations $s = t$, where $s \in T$, are solvable by matching. Note that a generic expression for type T needs not to be a member of T.

Example 3.6

- 0, $s(x)$, $s(s(x))$, ... are generic expressions for the type *Nat*,
- $[\,]$, $[x]$, $[x|y]$, $[x|x]$, $[x, y|z]$, ... are generic expressions for the type *List*. □

Next, we provide some important examples of generic expressions which will be used in the sequel.

Lemma 3.7 *Let T be a type. Then*

- *variables are generic expressions for T,*
- *the only generic expressions for type U are variables,*
- *if T does not contain variables, then every pure variable term is a generic expression for T,*
- *if T is ground, then every term is a generic expression for T.*

Proof. Clear. □

When the types are defined by structural induction (as for example in Bronsard, Lakshman and Reddy [BLR92] or in Yardeni, T. Frühwirth and E. Shapiro [YFS92]), then it is easy to characterize the generic expressions for each type by structural induction.

We can now provide another simple test for establishing solvability by matching.

Lemma 3.8 (Matching 2) *Consider two disjoint typed atoms A and H with the same relation symbol. Suppose that*

- *A is correctly typed,*
- *the positions of H are filled in by mutually disjoint terms and each of them is a generic expression for its position's type.*

Then $A = H$ is solvable by matching. Moreover, if A and H are unifiable, then a substitution θ with $Dom(\theta) \subseteq Var(H)$ exists such that $A = H\theta$.

Proof. Clear. □

3.3 Solvability by Iterated Matching

Consider a selected atom A and the head H of an input clause used to resolve A. In presence of modes the input and output positions can be used to model a parameter passing mechanism as follows. First the input values are passed from the selected atom A to the head H. Then the output values are passed from H to A.

To formalize and extend this idea we introduce the following notion where passing a value is modeled by matching.

Definition 3.9 Let E be a left-right disjoint set of equations.

- We say that E is *solvable by iterated matching* if E is unifiable implies that for some $E_1, \ldots, E_n$ and substitutions $\theta_1, \ldots, \theta_n$
  - $E = \dot{\bigcup}_{j=1}^{n} E_j$,
  and for $i \in [1, n]$
  - $E_i\theta_1 \ldots \theta_{i-1}$ is left-right disjoint,
  - θ_i is a match for $E_i\theta_1 \ldots \theta_{i-1}$. □

We shall also call it *double matching* when $n = 2$. In fact, in this paper we shall only study this form of iterated matching.

Note that when $\theta_1, \ldots, \theta_n$ satisfy the above three conditions, then by Lemma 2.2 $\theta_1\theta_2 \ldots \theta_n$ is a relevant mgu of E.

A slightly less general definition of solvability by (iterated) matching was considered by Maluszynski and Komorowski [MK85], where for $E = \{s_1 = t_1, \ldots, s_n = t_n\}$ the fixed partition $E = \dot{\bigcup}_{j=1}^{n} E_j$ with $E_j = \{s_j = t_j\}$ is used.

According to this terminology the above modeling of a parameter passing mechanism amounts to solvability by double matching.

To study solvability by double matching, modes and types are useful. Again, let us consider the case of passing the input values from a selected atom A to the head H of the clause used to resolve A. In presence of types, we can expect those input values to be correctly typed. Then the Matching 2 Lemma 3.8 can be applicable to deal with the input positions. If we are able to combine it with the Matching 1 Lemma 3.4 applied to the output positions, we can then conclude that $A = H$ is solvable by double matching. This observation is at the base of the following definitions.

Definition 3.10 An atom is called *input safe* if

- each of its input positions is filled in with a generic expression for this position's type,
- either the types of all input positions are ground or the terms filling in the input positions are mutually disjoint. □

In particular, an atom is *input safe* if the types of all input positions are ground.

Definition 3.11 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. □

Definition 3.12 An atom A is called *i/o regular* if

(i) it is correctly typed in its input positions,

(ii) it is input-output disjoint,

(iii) each of its output positions is filled in by a distinct variable. □

We now prove a result allowing us to conclude that $A = H$ is solvable by double matching.

Lemma 3.13 ((Double Matching)) *Consider two disjoint atoms A and H with the same relation symbol. Suppose that*

- *A is i/o regular,*
- *H is input safe,*

Then $A = H$ is solvable by double matching.

Proof. Assume that $A = H$ is unifiable. Take as E_1 the subset of $A = H$ corresponding to the input positions, as E_2 the subset of $A = H$ corresponding to the output positions.

By the Matching 1 Lemma 3.4 or the Matching 2 Lemma 3.8 E_1 is solvable by matching, and it determines a match θ_1 such that $Dom(\theta_1) \subseteq Var(H)$ and $Ran(\theta_1) \subseteq VarIn(A)$. But A is input-output disjoint, so $Ran(\theta_1) \cap VarOut(A) = \emptyset$. Thus $E_2\theta_1$ is left-right disjoint. Applying to $E_2\theta_1$ the Matching 1 Lemma 3.4 we get a match θ_2 for $E_2\theta_1$ such that $Dom(\theta_2) \subseteq VarOut(A)$. □

3.4 Unification Free Programs

Recall that the aim of this paper is to clarify for what Prolog programs unification can be replaced by iterated matching. The following definition is the key one.

Definition 3.14

- Let ξ be an LD-derivation. Let A be an atom selected in ξ and H 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 $P \cup \{Q\}$ are solvable by iterated matching. Then we say that $P \cup \{Q\}$ is *unification free.* □

The Double Matching Lemma 3.13 allows us to conclude when $P \cup \{Q\}$ is unification free. We need this notion.

Definition 3.15 We call an LD-derivation *i/o driven* if all atoms selected in it are i/o regular. □

Theorem 3.16 *Suppose that*

- *the head of every clause of P is input safe,*
- *all LD-derivations of $P \cup \{Q\}$ are i/o driven.*

Then $P \cup \{Q\}$ is unification free. □

In order to apply this theorem we need to find conditions which imply that all considered *LD*-derivations are i/o driven. To deal with the first condition for an atom to be i/o regular we use the concept of well-typed queries and programs.

4 Well-Typed Programs

The notion of well-typed queries and programs relies on the concept of a type judgement.

Definition 4.1

- By a *type judgement* we mean a statement of the form

$$\mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}. \tag{1}$$

- We say that a type judgement (1) *is true*, and write

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

if for all substitutions θ, $\mathbf{s}\theta \in \mathbf{S}$ implies $\mathbf{t}\theta \in \mathbf{T}$. □

For example, the type judgement $s(s(x)) : Nat,\ l : ListNat \Rightarrow [x \mid l] : ListNat$ is true.

To simplify the notation, when writing an atom as $p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$ we now assume that $\mathbf{u} : \mathbf{S}$ is a sequence of typed terms filling in the input positions of p and $\mathbf{v} : \mathbf{T}$ is a sequence of typed terms filling in the output positions of p. We call a construct of the form $p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$ a *typed atom*.

The following notion is due to Bronsard, Lakshman and Reddy [BLR92].

Definition 4.2

- A query $p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$ is called *well-typed* if for $j \in [1, n]$
$$\models \mathbf{o_1} : \mathbf{O_1}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \Rightarrow \mathbf{i_j} : \mathbf{I_j}.$$
- A clause $p_0(\mathbf{o_0} : \mathbf{O_0}, \mathbf{i_{n+1}} : \mathbf{I_{n+1}}) \leftarrow p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$ is called *well-typed* if for $j \in [1, n+1]$
$$\models \mathbf{o_0} : \mathbf{O_0}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \Rightarrow \mathbf{i_j} : \mathbf{I_j}.$$
- A program is called *well-typed* if every clause of it is. □

Thus, a query is well-typed if

- the types of the terms filling in the *input* positions of an atom can be deduced from the types of the terms filling in the *output* positions of the previous atoms.

And a clause is well-typed if

- ($j \in [1, n]$) the types of the terms filling the *input* positions of a body atom can be deduced from the types of the terms filling in the *input* positions of the head and the *output* positions of the previous body atoms,
- ($j = n + 1$) the types of the terms filling in the *output* positions of the head can be deduced from the types of the terms filling in the *input* positions of the head and the types of the terms filling in the *output* positions of the body atoms.

Note that a query with only one atom is well-typed iff this atom is correctly typed in its input positions. The following lemma due to Bronsard, Lakshman and Reddy [BLR92] shows persistence of the notion of being well-typed.

Lemma 4.3 *An LD-resolvent of a well-typed query and a disjoint with it well-typed clause is well-typed.* □

Corollary 4.4 *Let P and Q be well-typed, and let ξ be an LD-derivation of $P \cup \{Q\}$. All atoms selected in ξ are correctly typed in their input positions.*

Proof. A variant of a well-typed clause is well-typed and the first atom of a well-typed query is correctly typed in its input positions. □

This shows that by restricting our attention to well-typed programs and queries we ensure that all atoms selected in the LD-derivations satisfy the first condition of i/o regularity.

5 Simply Moded Programs

To ensure that the other two conditions of i/o regularity are satisfied we introduce further syntactic restrictions. Later we shall discuss how confining these restrictions are. We need a definition first.

Definition 5.1 A family of terms is called *linear* if every variable occurs at most once in it. □

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

Definition 5.2

- A query $p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ is called *simply moded* if $\mathbf{t_1}, \ldots, \mathbf{t_n}$ is a linear family of variables and for $i \in [1, n]$

$$Var(\mathbf{s_i}) \cap (\bigcup_{j=i}^{n} Var(\mathbf{t_j})) = \emptyset.$$

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

$$Var(\mathbf{s_0}) \cap (\bigcup_{j=1}^{n} Var(\mathbf{t_j})) = \emptyset.$$

  In particular, every unit clause is simply moded.

- A program is called *simply moded* if every clause of it is. □

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

- all output positions are filled in by variables,
- every variable occurring in an output position of an atom does not occur earlier in the query.

And a clause is simply moded if

- all output positions of body atoms are filled in by variables,
- 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 simply moded prevents a "speculative binding" of the variables which fill in the output positions — these variables are required to be "fresh". A similar notion of nicely moded programs and queries was introduced in Chadha and Plaisted [CP91] and further studied in Apt and Pellegrini [AP92]. The difference is that the output positions do not need there to be filled in by variables.

Note that a query with only one atom is simply moded iff it is input-output disjoint and each of its output positions is filled in by a distinct variable, i.e. so iff the conditions (ii) and (iii) of i/o regularity are satisfied. The following lemma shows the persistence of the notion of being simply moded.

Lemma 5.3 *An LD-resolvent of a simply moded query and a disjoint with it simply moded clause is simply moded.*

Proof. First, we establish two claims.

Claim 1 *Let θ be a substitution and $\mathbf{A}$ a simply moded query such that $Var(\theta) \cap VarOut(\mathbf{A}) = \emptyset$. Then $\mathbf{A}\theta$ is simply moded, as well.*

Proof. θ does not affect the variables appearing in the output positions of $\mathbf{A}$ and does not introduce these variables when applied to the terms appearing in the input positions of $\mathbf{A}$. □

Claim 2 *Suppose $\mathbf{A}$ and $\mathbf{B}$ are simply moded queries such that $VarOut(\mathbf{A}) \cap Var(\mathbf{B}) = \emptyset$. Then $\mathbf{B}, \mathbf{A}$ is a simply moded query, as well.*

Proof. Immediate by the definition of a simply moded query. □

Consider now a simply moded query $A, \mathbf{A}$ and a disjoint with it simply moded clause $H \leftarrow \mathbf{B}$, such that A and H unify. Take as E_1 the subset of $A = H$ corresponding to the input positions and as E_2 the subset of $A = H$ corresponding to the output positions. Let θ_1 be a relevant mgu of E_1. Then $Var(\theta_1) \subseteq VarIn(H) \cup VarIn(A)$, so $Var(\theta_1) \cap VarOut(A) = \emptyset$, since A is input-output disjoint. Thus $E_2\theta_1$ is left-right disjoint and by virtue of the Matching 1 Lemma 3.4 is solvable by matching. Let θ_2 be a match for $E_2\theta_1$. Then $Dom(\theta_2) \subseteq VarOut(A)$ and $Ran(\theta_2) \subseteq VarOut(H)$.

Let $\theta = \theta_1\theta_2$. Then $Var(\theta) \subseteq Var(A) \cup Var(H)$, so by the disjointness assumption and the definition of simply modedness $Var(\theta) \cap VarOut(\mathbf{A}) = \emptyset$. Thus by Claim 1 $\mathbf{A}\theta$ is simply moded.

Next, $\theta = \theta_1 \dot{\cup} \theta_2$, since A is input-output disjoint. So by the disjointness assumption $\mathbf{B}\theta = \mathbf{B}\theta_1$. But $Var(\theta_1) \cap VarOut(\mathbf{B}) \subseteq (VarIn(A) \cup VarIn(H)) \cap VarOut(\mathbf{B}) = \emptyset$, so by Claim 1 $\mathbf{B}\theta$ is simply moded.

Finally, $VarOut(\mathbf{A}\theta) = VarOut(\mathbf{A})$ and $Var(\mathbf{B}\theta) \subseteq Var(\mathbf{B}) \cup Var(A) \cup Var(H)$, so by the disjointness assumption and the definition of simply modedness we have that $VarOut(\mathbf{A}\theta) \cap Var(\mathbf{B}\theta) = \emptyset$. By Claim 2 $(\mathbf{B}, \mathbf{A})\theta$ is simply moded. Now by Lemma 2.2 θ is an mgu of A and H, so $(\mathbf{B}, \mathbf{A})\theta$ is a resolvent of $A, \mathbf{A}$ and $H \leftarrow \mathbf{B}$.

$\theta = \theta_1\theta_2$ is just one specific mgu of $A = H$. By Lemma 2.1 every other mgu of $A = H$ is of the form $\theta\eta$ for a renaming η. But a renaming of a simply moded query is simply moded, so we conclude that every LD-resolvent of $A, \mathbf{A}$ and $H \leftarrow \mathbf{B}$ is simply moded. □

It is useful to note that the above Lemma can be easily established as a consequence of Lemma 5.9 of Apt and Pellegrini [AP92] stating persistence of the notion of being nicely moded. To keep the paper self-contained we preferred to give here a direct proof.

The following immediate consequence show that the notion of being simply moded is the one we need.

Corollary 5.4 *Let P and Q be simply moded, and let ξ be an LD-derivation of $P \cup \{Q\}$. All atoms selected in ξ are input-output disjoint and such that each of their output positions are filled in by a distinct variable.*

Proof. A variant of a simply moded clause is simply moded and the first atom of a simply moded query is input-output disjoint and each of its output positions is filled in by a distinct variable. □

Theorem 5.5 *Suppose that*

- *P and Q are well-typed and simply moded,*

Then all LD-derivations of $P \cup \{Q\}$ are i/o driven.

Proof. By Corollaries 4.4 and 5.4. □

This brings us to the desired conclusion.

Theorem 5.6 ((Main)) *Suppose that*

- *P and Q are well-typed and simply moded,*
- *the head of every clause of P is input safe.*

Then $P \cup \{Q\}$ is unification free.

Proof. By Theorems 3.16 and 5.5. □

6 Examples

Let us see now how the established result can be applied to specific programs. When presenting the programs we adhere here to the usual syntactic conventions of Prolog with the exception that Prolog's ":-" is replaced by the logic programming " $\leftarrow$ ".

(i) Consider the proverbial program append:

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

with the typing app(+:*List*,+:*List*,-:*List*). First note that append is well-typed in the assumed typing. Indeed, the following type judgements are true:

$$[X|Xs] : List \Rightarrow Xs : List,$$
$$Ys : List \Rightarrow Ys : List,$$
$$Zs : List \Rightarrow [X|Zs] : List.$$

append is also obviously simply moded and the heads of all clauses are input safe. By the Main Theorem 5.6 we conclude that for lists s and t, and a variable u , append $\cup$ { app(s, t, u)} is unification free.

(ii) Examine now the program append with the typing app(-:*List*,-:*List*,+:*List*). First note that by virtue of the same type judgements as above append is well-typed. Moreover, append is also simply moded and the heads of all clauses are input safe. The Main Theorem 5.6 yields that for a list u and variables s,t, append $\cup$ { app(s, t, u)} is unification free.

(iii) Consider now the program permutation sort which is often used as a benchmark program.

```
ps(Xs, Ys) ←permutation(Xs, Ys), ordered(Ys).

permutation(Xs, [Y | Ys]) ←
   select(Y, Xs, Zs),
   permutation(Zs, Ys).
permutation([], []).

select(X, [X | Xs], Xs).
select(X, [Z | Xs], [Z | Zs]) ← select(X, Xs, Zs).

ordered([]).
ordered([X]).
ordered([X, Y | Xs]) ←X ≤ Y, ordered([Y| Xs]).
```

With the following typing: ps(+:*List*,-:*List*), permutation(+:*List*, -:*List*), select(-:*U*,+:*List*,-:*List*), $\leq$(+:*U*,+:*U*), ordered(+:*List*), the program is well-typed. Indeed, in addition to the above type judgements the following type judgement is true:

$$[X, Y|Xs] : List \Rightarrow [Y|Xs] : List.$$

permutation sort is also simply moded and the heads of all clauses are input safe. By the Main Theorem 5.6 we get that for a list s and a variable t, permutation sort $\cup$ { ps(s, t)} is unification free.

In all the examples seen before, the generic expressions which were filling in the input positions of the clauses were always either variables or pure variable terms. This is not the case with permutation sort. Indeed, the terms [X] and [X, Y | Xs], filling in the input positions of, respectively, the first and the third clause defining the relation ordered, are generic expressions for *List*, but are not pure variable terms. In a sense we could say that [X] and [X, Y | Xs] are nontrivial generic expressions.

(iv) Finally, consider the program in-order which converts a (n ordered) binary tree into a (n ordered) list and consists in the following clauses:

```
in-order(tree(X, L, R), Xs) ←
   in-order(L, Ls),
   in-order(R, Rs),
   app(Ls, [X | Rs], Xs).
in-order(void, []).
```

augmented by the append program.

The type *BinTree* can be defined recursively as follows:
- `void` is a binary tree,
- if `l` and `r` are binary trees and `label` is a term, then `tree(label, l, r)` is a binary tree.

With the typing `in-order(+:`*BinTree*`,-:`*List*`)`, `app(+:`*List*`,+:`*List*`,-:`*List*`)`, the program `in-order` is well-typed. Indeed, in addition to the above type judgements the following type judgements are clearly true:

$$tree(X, L, R) : BinTree \Rightarrow L : BinTree,$$
$$tree(X, L, R) : BinTree \Rightarrow R : BinTree.$$

`in-order` is also simply moded and the heads of all clauses are input safe. By the Main Theorem 5.6 we conclude that for a binary tree `t` and a variable `s`, `in-order` $\cup$ { `in-order(t, s)`} is unification free.

7 Avoiding Unification Using Modes

When trying to apply the Main Theorem 5.6 one has to verify whether for a given typing a program and a query are well-typed. This can be inefficient and for some artificially constructed types even undecidable. However, when dealing with programs which use ground inputs the conditions of the Main Theorem 5.6 can be efficiently tested for a given moding, program and query.

This is due to the fact that it is possible then to formulate this theorem without explicit reference to types. The notion which is sufficient then is that of a well-moded program. The concept is essentially due to Dembinski and Maluszynski [DM85a]; we use here an elegant formulation due to Rosenblueth [Ros91]. 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 7.1

- A query $p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ 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 query 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

- ($i \in [1, n]$) 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,
- ($i = n + 1$) 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 ($j \in [1, n]$).

It is useful to note that the concept of a well-moded program (resp. query) is a particular case of that of a well-typed program. Indeed, if the only type used is *Ground*, then the notions of a well-typed program (resp. query) and a well-moded program (resp. query) coincide. All programs considered in Section 6 become well-moded when the type information is dropped.

This brings us to the following special case of the Main Theorem 5.6.

Corollary 7.2 *Suppose that*

- *P and Q are simply moded and well-moded.*

Then $P \cup \{Q\}$ is unification free.

Proof. When the only types used are ground, all atoms are input safe. □

Note that for a given moding it is easy to test whether conditions of this corollary are applicable. Indeed, assume that in every atom the input positions occur first. Then a query Q is simply moded and well-moded iff

- every first from the left occurrence of a variable in Q is within an output position,
- the output positions of Q are filled in by distinct variables.

And a clause $p(\mathbf{s}, \mathbf{t}) \leftarrow \mathbf{B}$ is simply moded and well-moded iff

- every first from the left occurrence of a variable in the sequence $\mathbf{s}, \mathbf{B}, \mathbf{t}$ is in $\mathbf{s}$ or within an output position in $\mathbf{B}$,
- the output positions of $\mathbf{B}$ are filled in by distinct variables which do not appear in $\mathbf{s}$.

This corollary allows us to deal with more restricted queries than the Main Theorem 5.6, but in a number of cases this is sufficient.

Example 7.3

(i) Examine the following program `palindrome`:

```
palindrome(Xs) ← reverse(Xs, Xs).

reverse(X1s, X2s) ← reverse(X1s, [], X2s).
reverse([X | X1s], X2s, Ys) ← reverse(X1s, [X | X2s], Ys).
reverse([], Xs, Xs).
```

With the typing `palindrome(+:`*List*`)`, `reverse(+:`*List*`, -:`*List*`)`, `reverse(+:`*List*`, +:`*List*`, -:`*List*`)`, `reverse` is simply moded, but `palindrome` is not, as the body of the first clause does not satisfy Definition 5.2 (the variable in the second position of `reverse` appears "earlier" twice). Switching to the typing `palindrome(+:`*List*`)`, `reverse(+:`*List*`,+:`*List*`)`, `reverse(+:`*List*`,+:`*List*`,+:`*List*`)`, does not help as now the head of the last clause is not input safe.

On the other hand, when adopting the mode `palindrome(+)`, `reverse(+,+)`, `reverse(+,+,+)`, `palindrome` is simply moded and well-moded. Hence, by Corollary 7.2, when `t` is ground, `palindrome` ∪ { `palindrome(t)`} is unification free.

It is worth noticing that for non-ground inputs `palindrome(+:`*List*`)` may actually require unification in order to run properly. Indeed, consider the following query: `palindrome([f(X,a),f(b,X)])`. When evaluated it eventually leads to the equation $\mathtt{f(X, a)} = \mathtt{f(b, X)}$, which to be solved requires unification.

(ii) Applying Corollary 7.2 to the programs handled in Section 6 we can only draw conclusions for the case when all terms filling in the input positions are ground.

8 Discussion

To apply the established results to a program and a query, one needs to find appropriate modings and typings for the considered relations such that the conditions of the Main Theorem 5.6 or of Corollary 7.2 are satisfied. In the table below several programs taken from the book of Sterling and Shapiro [SS86] are listed. For each program it is indicated for which *nonground* typing (i.e. one with some non-ground type) the Main Theorem 5.6 is applicable, and for which modings Corollary 7.2 is. All built-in's are moded completely input with all positions typed U.

In programs which 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 on page 258 each clause of the form `p(X)` $\leftarrow$ `r(X)` has to be replaced by `p(X,Y)` $\leftarrow$ `r(X,Y)`. Such changes are purely syntactic and they allow us to draw conclusions about the original program.

In the program `dutch` we refer to a new type *CList* which consists of lists of *colored objects*, where a colored object is a term of the form *red*(s), *blue*(s) or *white*(s).

program	page	(nonground) typings	modings
member	45	(−:U,+:*List*)	(−,+)
member	45	none	(+,+)
prefix	45	(−:*List*,+:*List*)	(−,+)
prefix	45	(+:*List*,−:*List*)	(+,−)
prefix	45	none	(+,+)
suffix	45	(−:*List*,+:*List*)	(−,+)
suffix	45	(+:*List*,−:*List*)	(+,−)
suffix	45	none	(+,+)
naive reverse	48	(+:*List*,−:*List*)	(+,−)
reverse-accum.	48	r(+:*List*,−:*List*)	r(+,−)
		r(+:*List*,+:*List*,−:*List*)	r(+,+,−)
reverse-accum.	48	none	r(+,+)
			r(+,+,+)
delete	53	none	(+,+,−)
delete	53	none	(+,+,+)
select	53	none	(+,+,−)
select	53	(−:U,+:*List*,−:*List*)	(−,+,−)
select	53	(+:U,−:*List*,+:*List*)	(+,−,+)
select	53	none	(+,+,+)
insertion sort	55	s(+:*List*,−:*List*)	s(+,−)
		i(+:*List*,+:*List*,−:*List*)	i(+,+,−)
quicksort	56	q(+:*List*,−:*List*)	q(+,−)
		p(+:*List*,+:U,−:*List*,−:*List*)	p(+,+,−,−)
		app(+:*List*,+:*List*,−:*List*)	app(+,+,−)
tree-member	58	(−:U,+:*BinTree*)	(−,+)
tree-member	58	none	(+,+)
isotree	58	(−:*BinTree*,+:*BinTree*)	(−,+)

isotree	58	(+:*BinTree*,−:*BinTree*)	(+,−)
isotree	58	none	(+,+)
substitute	60	none	(+,+,+,−)
pre-order	60	(+:*BinTree*,−:*List*)	(+,−)
post-order	60	(+:*BinTree*,−:*List*)	(+,−)
polynomial	62	none	(+,+)
derivative	63	none	(+,+,−)
derivative	63	none	(+,+,+)
hanoi	64	(+:*Nat*,+:*U*,+:*U*,+:*U*,−:*List*)	(+,+,+,+,−)
flatten	243	none	f(+,−) f(+,+,−)
reverse_dl	244	r(+:*List*,−:*List*) r_dl(+:*List*,−:*List*,+:*List*)	r(+,−) r_dl(+,−,+)
reverse_dl	244	none	r(+,+) r_dl(+,+,+)
dutch	246	dutch(+:*CList*,−:*CList*) di(+:*CList*,−:*CList*,−:*CList*,−:*CList*)	dutch(+,−) di(+,−,−,−)
parsing	258	none	all (+,−)

9 Conclusions

In view of the above results it is natural to ask when unification is intrinsically needed in Prolog programs. A canonic example is the Prolog program `curry` which computes a type assignment to a lambda term, if such an assignment exists (see e.g. Reddy [Red86]). We are not aware of other natural examples, though it should be added that for complicated queries which anticipate in their ouput positions the form of computed answers, almost any program will necessitate the use of unification.

In our analysis we restricted our attention to the case of programs in which the output positions of the clause bodies are filled in by variables. This obviously limits applicability of our results, since in a number of natural programs these output positions are compound terms. An example is the following program `permutation`:

```
perm(Xs, Ys) ← Ys is a permutation of the list Xs.

perm(Xs, [X | Ys]) ←
   app(X1s, [X | X2s], Xs),
   app(X1s, X2s, Zs),
   perm(Zs, Ys).
perm([], []).
```

augmented by the `APPEND` program.

in which the first call of `app` uses a compound term `[X | X2s]` in an output position. We checked by hand that when `s` is a list and `x` a variable, `permutation` ∪ { `perm(s, x)` } is unification free. Currently we are working on extension of the obtained results to the case of non-variable outputs.

This work is naturally related to the study of conditions which guarantee that Prolog programs can be executed using unification without the occur-check. It should be noted however, that unification freedom property rests exclusively upon those considered systems of equations which *are* unifiable, whereas the property of being occur-check free rests exclusively upon those considered systems which are *not* unifiable. Indeed, the occur-check is only needed to correctly identify the non-unifiable systems of equations. Still, when comparing the outcome of this paper with our previous work on the occur-check problem (Apt and Pellegrini [AP92]) we note an astonishing similarity between both classes of identified Prolog programs.

Acknowledgements

We thank Jan Heering and Alessandro Pellegrini for interesting discussions on the subject of this paper.

References

[AFZ88] I. Attali and P. Franchi-Zannettacci. Unification-free execution of TYPOL programs by semantic attribute evaluation. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 160–177. The MIT Press, 1988.

[AP92] K. R. Apt and A. Pellegrini. Why the occur-check is not a problem. In M. Bruynooghe and M. Wirsing, editors, *Proceeding of the Fourth International Symposium on Programming Language Implementation and Logic Programming (PLILP 92)*, Lecture Notes in Computer Science 631, pages 69–86, Berlin, 1992. Springer-Verlag.

[Apt90] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.

[BLR92] F. Bronsard, T.K. Lakshman, and U.S. Reddy. A framework of directionality for proving termination of logic programs. In K.R. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 321–335. MIT Press, 1992.

[CP91] R. Chadha and D.A. Plaisted. Correctness of unification without occur check in Prolog. Technical report, Department of Computer Science, University of North Carolina, Chapel Hill, N.C., 1991.

[DM85a] P. Dembinski and J. Maluszynski. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the International Symposium on Logic Programming*, pages 29–38, Boston, 1985.

[DM85b] P. Deransart and J. Maluszynski. Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming*, 2:119–156, 1985.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[LMM88] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[MK85] J. Maluszynski and H. J. Komorowski. Unification-free execution of logic programs. In *Proceedings of the 1985 IEEE Symposium on Logic Programming*, pages 78–86, Boston, 1985. IEEE Computer Society Press.

[Red86] U.S. Reddy. On the relationship between logic and functional languages. In D. DeGroot and G. Lindstrom, editors, *Functional and Logic Programming*, pages 3–36. Prentice-Hall, 1986.

[Ros91] D.A. Rosenblueth. Using program transformation to obtain methods for eliminating backtracking in fixed-mode logic programs. Technical Report 7, Universidad Nacional Autonoma de Mexico, Instituto de Investigaciones en Matematicas Aplicadas y en Sistemas, 1991.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[YFS92] E. Yardeni, T. Frühwirth, and E. Shapiro. Polymorphically typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 63–90. MIT Press, Cambridge, Massachussets, 1992.