More (on) unification-free Prolog programs

S. Etalle

Computer Science/Department of Software Technology

Report CS-R9454 September 1994

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 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

More (on) Unification-Free Prolog Programs

Sandro Etalle

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Dipartimento di Matematica Pura ed Applicata
Università di Padova
Via Belzoni 7, 35131 Padova, Italy

email: sandro@cwi.nl, sandro@hilbert.math.unipd.it

Abstract

We provide new 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. As already noticed by other researchers, such a replacement offers a possibility of improving the efficiency of program's execution. The results we prove improve on those in our previous paper ([AE93]) both because they allow to prove unification-freedom for a larger class of programs and queries and because the conditions are, in many cases, checkable in a much more efficient way.

1991 Mathematics Subject Classification: 68Q40, 68T15, 68N15.
CR Categories: D.1.6, F.3.2., F.4.1, H.3.3, I.2.3.
Keywords and Phrases: iterated matching, Prolog programs.

1. Introduction

Unification is the core of the resolution method employed by PROLOG, and its efficiency has great influence on the overall performance of the interpreter. The best sequential unification algorithm employs linear time (see for example Martelli-Montanari [MM82]), and, most likely, this result cannot be improved by the adoption of a parallel algorithm: Dwork et al. [DKM84] have shown that, unless PTIME $\subseteq$ NC (which is quite improbable) unification does not admit an algorithm that run polilogarithmic time using a polynomially bounded number of processors.

On the other hand, fast parallel algorithms are available for *term matching*: a special case of unification where one of the terms is always an instance of the other one [DKM84, DKS86]. This motivates the research for sufficient conditions for the replacement of unification with term matching (see, for instance [DM85b, MK85, AFZ88] and, more recently, [AE93, Mar94]).

In Deransart and Maluszynski [DM85b], Maluszynski and Komorowski [MK85] and Attali and Franchi-Zannettacci [AFZ88], the problem was tackled by using *modes*. Intuitively, a *mode* is a function that labels as *input* or *output* the positions of each relation in order to indicate how the arguments of a relation should be used. A limit of this approach is that the input positions of the queries are expected to be filled in by ground (i.e. variable-free) terms. Apt and Etalle [AE93] improved upon the previous results by additionally using *types*, which allow to deal with non-ground inputs.

Here, we generalize the results of [AE93]. The main tools of our approach can be summarized as follows:

Report CS-R9454
ISSN 0169-118X
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

First, in addition to *input* and *output* positions, we introduce here U-positions. Here "U" can be read as *unknown*, as the U-positions of a query can be filled in by any term. It turns out that for many of the programs mentioned in [AE93] we could simply turn some positions into U positions, both enlarging significantly the class of allowed queries and, when this process was applied to the nonground input positions, simplifying dramatically the method for proving that the program is unification-free.

Second, we now allow also *pure terms* to fill in *output* positions of the queries, again this enlarges the class of allowed queries.

Finally, by following Apt [Apt94], we adopt here a more flexible definition of *well-typed program*.

As in our previous paper, the conditions we provide can be statically checked without analyzing the search trees for the queries.

This paper is organized as follows. In the next section we introduce the concepts of solvability by sequential matching and of unification-free Prolog program. Section 3 contains the basic definitions of modes and types, which are the main tools we need in the sequel. Both concept are used in order to specify how the arguments of an atom should be used, and, ultimately, to restrict the set of allowed queries. In section 4 we begin to tackle the problem of how to prove that a program is unification-free: we introduce the definition of a Nicely Typed program and we show that, in some cases, this concept alone is sufficient for our purposes. This section can be also seen as an intermediate step: in the subsequent one we report the definition of Well-typed program. Programs which are both Well and Nicely Typed are the ones that will enable us to prove, in Section 5, our most general theorem (5.18). In Section 6 we give a more restrictive version of our Main Theorem. The relevance of this result lies in the fact that its applicability conditions can be tested in a much more efficient way. Section 7 contains some practical examples, and in Section 8 we conclude by comparing this paper with our previous one [AE93] and with another recent related paper [Mar94].

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, and 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 Solvability by (sequential) Matching

Following the notation of Apt and Etalle, [AE93], we begin by recalling the following concepts.

DEFINITION 2.3 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 2.4 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. □

Consider a selected atom $p(t_1, \ldots, t_n)$ and the head $p(s_1, \ldots, s_n)$ of an input clause used to resolve it. The unification mechanism tries then to find a mgu of the set of equations $t_1 = s_1, \ldots, t_n = s_n$. Sometimes such a set is not solvable by matching as a whole, but it can be solved by a *sequential* matching, that is, by considering the equations one at a time.

To formalize this idea we introduce the following notion.

DEFINITION 2.5 Let $E = E_1, \ldots, E_n$ be a left-right disjoint sequence of (sets of) equations.

- We say that E is *solvable by sequential matching* if E is unifiable implies that for some substitutions $\theta_1, \ldots, \theta_n$, 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 say that E is *solvable by sequential matching* wrt π if π is a permutation of $1, \ldots, n$, and
  - $E_{\pi(1)}, \ldots, E_{\pi(n)}$ is solvable by sequential matching. □

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

This Definition corresponds to the one considered by Maluszynski and Komorowski [MK85], and is slightly less general than the one of *iterated* matching given in [AE93], which makes no explicit reference to the order in which the equations are to be solved. Intuitively, E is solvable by *iterated* matching iff there exists a π such that E is solvable by sequential matching wrt π.

2.3 Unification Free Programs

Recall that the aim of this paper is to clarify for what Prolog programs unification can be replaced by sequential matching. The following Definition is then the key one. Here we denote by $rel(A)$ the relation symbol of the atom A.

DEFINITION 2.6

- 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 each system of equations $A = H$ considered in the LD-derivations of $P \cup \{Q\}$ is solvable by sequential matching wrt a permutation $\pi_{rel(A)}$, where $\pi_{rel(A)}$ is uniquely determined by the relation symbol of A. Then we say that $P \cup \{Q\}$ is *unification free*. □

A slightly more flexible definition of unification-free program was given in Apt-Etalle [AE93], where the equation $A = H$ may be solvable by *iterated* matching, i.e. the sequence π needs not to be determinable from the relations symbol of A.

3. TYPES AND MODES

The main tools that we are going to use in this paper are types and modes. The following very general definition of type is sufficient for our purposes.

DEFINITION 3.1

- A *type* is a set of atoms with the same relation symbol;
- A *type* is a type for a relation symbol p. □

Notice that, as opposed to [AE93], here we are also considering types which are not closed under substitution.

For the purpose of this paper, types for relations are always built by suitably combining set of terms.

DEFINITION 3.2

- A *term_type* is a set of terms. □

Here, we sometimes overload the term *type* to denote either a type or a term_type; the actual meaning will be clear from the context.

Certain term_types will be of special interest:

U — the set of all terms,

Var — the set of variables,

List — the set of lists,

BinTree — the set of binary trees,

Ground — the set of ground terms.

Of course, the use of the term_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.

The following notation will be used throughout the paper. Let p be an n-ary relation symbol, and let $T_1, \ldots, T_n$ be term_types. we denote by

$$p : T_1 \times \ldots \times T_n$$

the type for p given by the following set of atoms.

$$\{p(t_1, \ldots, t_n) \mid \text{ for } i \in [1, n], t_i \in T_i\}$$

Given a program P, a *typing* for P is a function that associate to each relation symbol p in P a type of the form $p : T_1 \times \ldots \times T_n$, consequently we also say that T_i is the term_type associated to the i-th *position* of p.

We need one final Definition.

DEFINITION 3.3 Let $p : T_1 \times \ldots \times T_n$ be the type for p.

- We say that an atom $p(t_1, \ldots, t_n)$ is *correctly typed* in his i-th position if $t_i \in T_i$;
- We say that an atom $p(t_1, \ldots, t_n)$ *correctly typed* if it is correctly type in all its positions. □

In the sequel we assume that each program has a (n often unspecified) typing associated to. The typing specifies how the argument of a relation should be used: as a general rule, we expect that the atoms selected in a LD-derivation are correctly typed (to make sure of this we'll introduce appropriate tools). Consider for instance the well-known program append:

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

append can be used for concatenating two lists, and this can be reflected by the adoption of the following "natural" typing:

app : $List \times List \times Var$

This typing expresses the fact that each time an atom of the form `:- append(s, t, u)` is selected in by the (leftmost) selection rule, we expect `s` and `t` to be lists, and `u` to be a variable. Multiple typings can be obtained by simply renaming the relations.

Before introducing modes, we need a last definition.

DEFINITION 3.4

- We call an atom (resp. a term) a *pure atom* (resp. *pure 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. □

To study solvability by matching, we keep in special consideration the following term_types.

- Var - the set of all variables;
- Pt - the set of variables and pure terms;
- U - the set of all terms.

Notice that $Var \subseteq Pt \subseteq U$. According to the typing used, we'll make some distinctions among the positions of an atom. Consider the case of a selected atom A and the head H of an input clause used to resolve A. In presence of types, we expect A to be correctly typed. It is then natural to consider the positions of A which are typed *Var* or *Pt*, which are filled in by variables or pure terms as *output* positions, as they contain no information. On the other hand for those positions which are typed U, since we really have no clue over the kind of parameter-passing that will take place in them, we use the special name of U-positions. The remaining positions will then by convention be considered as *input*. These considerations are at the base of the following Definition.

DEFINITION 3.5 Let $p : T_1 \times \ldots \times T_n$ be the type of the relation symbol p. We call the i-th position of an atom $p(t_1, \ldots, t_n)$

- A U-position if $T_i = U$
- An *output* position if $T_i = Var$ or $T_i = Pt$;
- An *input* position otherwise. □

This classification is actually a *moding*. Modes for logic programs were first considered by Mellish [Mel81] and then more extensively studied in Reddy [Red84] and in Dembinski and Maluszynski [DM85a]. Here we are departing from the previous works by using also the mode U, which can be seen as a way to avoid to commit ourselves to a specific mode when such a commitment is not necessary.

4. AVOIDING UNIFICATION USING THE MODES "U" AND "OUTPUT"

In order to introduce the tools we need in a gradual manner, we begin by excluding the presence of input positions.

Surprisingly, in many cases, this restriction does not represent a problem: in order to pass the information from the selected atom to the head of the input clause we can still use the U-positions. Consider for instance again the program append, as we mentioned before, when it is used for concatenating two lists, the "natural" typing is

append: $List \times List \times Var$.

Now, if we want to avoid the presence of input positions, we can simply use the following typing.

append: $U \times U \times Var$

Notice that the first two positions are U-positions, while the third one is and output one. The only practical difference between this and the "natural" typing is that in the query `app(s, t, u)` we now allow `s` and `t` to be any term, rather than just list. This is obviously no restriction. In general, using the U-positions for the parameter-passing task has the advantage of flexibility: since every term belongs to U we are making here no *a priori* assumption on the structure of the data. Moreover, as we'll show in the rest of this Section, proving unification-freedom is in this context particularly simple.

Throughout this Section we assume that the atoms have only U- and output positions: by Definition 3.5 this is equivalent to considering typings built only with the following term_types: U, Var and Pt.

4.1 Sequential Matching via Pure Terms

We start with a simple test allowing us to determine whether a given set of equations is solvable by matching.

LEMMA 4.1 (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. □

Now let us go back to the example of the (correctly typed) selected atom A and the head H of a clause used to resolve it. In order to apply the Matching 1 Lemma 4.1 to the part of $A = H$ corresponding to the U-positions, since we have no information about the shape of the terms filling in the U-positions of A, we have to impose some restrictions on H. Here we call a family of terms *linear* if every variable occurs at most once in it.

DEFINITION 4.2 (U-SAFE$^-$) An atom H is called *U-safe*$^-$ if the family of terms filling in its U-positions is linear and consists of only variables and pure terms. □

The minus sign in *U-safe*$^-$ is motivated by the fact that in Section 5 we'll introduce a more general definition of U-safeness, which will also take into account the presence of input positions. We need now one further notion.

DEFINITION 4.3 An atom A is called *output independent* if each term occurring in an output position is disjoint from the rest of A. □

Now we prove a result allowing us to conclude that $A = H$ is solvable by sequential matching.

LEMMA 4.4 (SEQUENTIAL MATCHING 1) Consider two disjoint atoms A and H with the same relation symbol p. Suppose that p has no input positions. If

- A is correctly typed and output independent,
- H is U-safe$^-$,

then there exists a permutation π such that $A = H$ is solvable by sequential matching wrt π.

In particular, $A = H$ is solvable by sequential matching wrt any permutation π of $1, \ldots, n$ such that, according to the order given by $\pi(1), \ldots, \pi(n)$, we have that the U-positions of p come first and the output positions come last.

PROOF. Suppose that $A = H$ is unifiable, we can then assume that A is $p(s_1, \ldots, s_n)$ and that H is equal to $p(t_1, \ldots, t_n)$, where $s_1, \ldots, s_n, t_1, \ldots, t_n$ have been reordered in such a way that U-positions come first (on the left) and the output positions are the rightmost ones.

We now need to prove that $s_1 = t_1, \ldots, s_n = t_n$ is solvable by sequential matching, that is we need to find $\theta_1, \ldots, \theta_n$ such that each θ_i is a match of $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$. For each i, we distinguish upon the kind of position where the equation $s_i = t_i$ is found.

If $s_i = t_i$ is found in a U-position then, since H is U-safe$^-$, we have that t_i is a variable or a pure term and $Var(t_i) \cap Var(\theta_1 \ldots \theta_{i-1}) = \emptyset$, so $t_i\theta_1 \ldots \theta_{i-1}$ is still a variable or a pure term and by the Matching 1 Lemma 4.1 $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching.

Finally, if $s_i = t_i$ is found in an output position then, from the assumptions we made on A, it follows that s_i is a variable or a pure term and that $Var(s_i) \cap Var(\theta_1, \ldots, \theta_{i-1}) = \emptyset$. So $s_i\theta_1, \ldots, \theta_{i-1}$ is still a variable or a pure term, and by the Matching 1 Lemma 4.1 $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching. □

When A and H satisfy the conditions of this Lemma, we can then solve $A = H$ by sequentially matching one position at a time. Still, we can improve on this result by showing that there exist some subsets of $A = H$ which correspond to more than one position and which can be solved by a single matching. This issue will be discussed in the Appendix.

We need one further notion.

DEFINITION 4.5 We call an LD-derivation *i/o driven* if all atoms selected in it are correctly typed and output independent. □

i/o driven derivations were introduced in [AE93], but the definition we give here is more general than the previous one. This is due to the fact that now we consider also U-positions, and that we allow Pt as a term_type for the output positions (in [AE93] the only term_type allowed for the output positions is Var).

The Sequential Matching Lemma 4.4 allows us to combine the notions of U-safe atom and of i/o driven derivation for concluding that $P \cup \{Q\}$ is unification free.

THEOREM 4.6 Suppose that each predicate symbol occurring in P has no input positions. If

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

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

4.2 Taking care of the output positions: Nicely Typed programs

In order to apply Theorem 4.6 we need to find conditions which imply that all considered *LD*-derivations are i/o driven. Since here we exclude the existence of input positions, all we have to do is to ensure that the selected atom A is correctly typed in its output position and output independent. For this we'll introduce the new concept of Nicely Typed program.

We start with the following notion which was introduced in Chadha and Plaisted [CP91]. Here we use the notation of Apt and Pellegrini [AP92]: when writing an atom as $p(\mathbf{rest}, \mathbf{out})$, we now assume that **out** is the sequence of terms filling in the output positions of p, while that **rest** is the sequence of terms filling its remaining positions.

DEFINITION 4.7 (NICELY MODED)

- A query $p_1(\mathbf{r_1}, \mathbf{o_1}), \ldots, p_n(\mathbf{r_n}, \mathbf{o_n})$ is called *nicely moded* if $\mathbf{o_1}, \ldots \mathbf{o_n}$ is a linear family of terms and for $j \in [1, n]$

$$Var(\mathbf{r_j}) \cap (\bigcup_{k=j}^{n} Var(\mathbf{o_k})) = \emptyset. \tag{4.1}$$

- A clause

$$p_0(\mathbf{r_0}, \mathbf{o_0}) \leftarrow p_1(\mathbf{r_1}, \mathbf{o_1}), \ldots, p_n(\mathbf{r_n}, \mathbf{o_n})$$

is called *nicely moded* if $p_1(\mathbf{r_1}, \mathbf{o_1}), \ldots, p_n(\mathbf{r_n}, \mathbf{o_n})$ is nicely moded and

$$Var(\mathbf{r_0}) \cap (\bigcup_{k=1}^{n} Var(\mathbf{o_k})) = \emptyset. \tag{4.2}$$

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 output positions are the rightmost ones, a query is nicely moded if

- every variable occurring in an output position of an atom does not occur earlier in the query.

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 a non-output position of the head.

So, intuitively, the concept of being nicely moded prevents a "speculative binding" of the variables which occur in output positions — these variables are required to be "fresh".

From the definition it follows that, if the query is nicely moded, then the selected atom is output independent. In order to fulfill the requirements of i/o drivenness we also ask the output positions to be correctly typed. For this reason we introduce a further Definition. Here and in the sequel, given an atom A, we denote by $VarOut(A)$ the set of variables occurring in the output positions of A. Similar notation is used for sequences of atoms.

DEFINITION 4.8 (NICELY TYPED)

- A nicely moded query $\mathbf{B}$ is called *nicely typed* if it is correctly typed in its output positions.
- a nicely moded clause $H \leftarrow \mathbf{B}$ is called *nicely typed* if $\mathbf{B}$ is nicely typed, and each term t filling in a position of H of type Pt satisfies the following

  If t is a variable and $t \cap VarOut(\mathbf{B}) \neq \emptyset$ then t fills in a position of $\mathbf{B}$ of type Pt. (4.3)

- A program is called *nicely typed* if every clause of it is. □

Nicely typed programs can be seen as a generalization of simply moded programs of [AE93]. The additional condition (4.3) that we impose on the clauses is needed to ensure the persistence of the notion of being nicely typed, which is proven in the following key Lemma.

LEMMA 4.9 An LD-resolvent of a nicely typed query and a disjoint with it nicely typed clause is nicely typed. □

PROOF. Consider a nicely typed query $A, \mathbf{A}$ and a disjoint with it nicely typed clause $H \leftarrow \mathbf{B}$, such that A and H unify. Take as E_0 the subset of $A = H$ corresponding to the non-output positions, and as $E_1, \ldots, E_n$ the subsets of $A = H$ each corresponding to an output position.

The proof is divided in steps.

CLAIM 1 There exist $\theta_0, \ldots, \theta_n$ such that, for $i \in [0, n]$,

(a) θ_i is a relevant mgu of $E_i\theta_0 \ldots \theta_{i-1}$,

(b) $\mathbf{B}\theta_0, \ldots, \theta_i$ is correctly typed in its output positions.

Proof. We proceed by induction.

Base case: $i = 0$.
Let θ_0 be any relevant mgu of E_0. Since $H \leftarrow \mathbf{B}$ is nicely moded, the variables in $VarOut(\mathbf{B})$ do not occur in the non-output positions of H, therefore the output positions of $\mathbf{B}$ are not affected by θ_0. Since by hypothesis $\mathbf{B}$ is correctly typed in its output positions, $\mathbf{B}\theta_0$ is correctly typed in its output positions as well.

Induction step: $i > 0$.
Let $E_i \equiv s = t$, where s and t are the terms filling the i-th output position respectively of A and H. First notice that since A is nicely moded, the variables of s do not occur anywhere else in A. Moreover, from the disjointness hypothesis (and the relevance of each θ_i) it follows then that $Var(s) \cap Var(\theta_0 \ldots \theta_{i-1}) = \emptyset$. Therefore we have that

$$s\theta_0 \ldots \theta_{i-1} = s$$

Keep in mind that by the inductive hypothesis $\mathbf{B}\theta_0 \ldots \theta_{i-1}$ is correctly typed in its output positions, and that $s = s\theta_0 \ldots \theta_{i-1}$. Since A is nicely typed, s may only be a variable or a pure term. Let us consider those two cases separately, and let us suppose that s is

a variable. Then we can take θ_i to be exactly $[s/t\theta_0 \ldots \theta_{i-1}]$. Therefore $Dom(\theta_i) = s$, and $\mathbf{B}\theta_0 \ldots \theta_{i-1}$ is not affected by θ_i, and the result follows from the inductive hypothesis.

a pure term. Since A is nicely typed, the type of the the i-th output position of A (and H) must be Pt. Let θ_i be any relevant mgu of $s\theta_1 \ldots \theta_{i-1} = t\theta_1 \ldots \theta_{i-1}$ We have to distinguish three cases:

First we consider the case in which $t\theta_0 \ldots \theta_{i-1}$ is a variable and it occurs in $VarOut(\mathbf{B}\theta_0 \ldots \theta_{i-1})$. Obviously, in this case t itself is a variable as well. Now notice that if r is any term filling in an output position of $\mathbf{B}$ then we have that

$$\text{if} \quad Var(r\theta_0 \ldots \theta_{i-1}) \cap t\theta_0 \ldots \theta_{i-1} \neq \emptyset \quad \text{then} \quad Var(r) \cap t \neq \emptyset \tag{4.4}$$

In other words, if r is disjoint from t then also $r\theta_0 \ldots \theta_{i-1}$ is disjoint from $t\theta_0 \ldots \theta_{i-1}$. This is due to the fact that, since $H \leftarrow \mathbf{B}$ is nicely moded, the variables of r may not occur in the input positions of H but only in the output ones, and, since A is output independent, the substitutions $\theta_0 \ldots \theta_{i-1}$ cannot bind them to other variables of $H \leftarrow \mathbf{B}$.

Since $t\theta_0 \ldots \theta_{i-1}$ occurs in $VarOut(\mathbf{B}\theta_0 \ldots \theta_{i-1})$, from (4.4) it follows that t occurs in $VarOut(\mathbf{B})$. Furthermore, from (4.4) and the fact that $H \leftarrow \mathbf{B}$ is nicely typed it follows that $t\theta_0 \ldots \theta_{i-1}$ fills in an output position of $\mathbf{B}\theta_0 \ldots \theta_{i-1}$, and (being $H \leftarrow \mathbf{B}$ nicely moded) it does not occur anywhere also in $\mathbf{B}\theta_0 \ldots \theta_{i-1}$.

Now, $s\theta_0 \ldots \theta_{i-1}$ is a pure term and $t\theta_0 \ldots \theta_{i-1}$ is a variable, therefore we have that $t\theta_0 \ldots \theta_{i-1}\theta_i$ is a pure term, and, since $t\theta_0 \ldots \theta_{i-1}$ fills in an output position of $\mathbf{B}\theta_0 \ldots \theta_{i-1}$ of type Pt, from the inductive hypothesis it follows that $\mathbf{B}\theta_0 \ldots \theta_{i-1}\theta_i$ is correctly typed in its output positions.

Secondly, if $t\theta_0 \ldots \theta_{i-1}$ is a variable and it does not occur in $VarOut(\mathbf{B})\theta_0 \ldots \theta_{i-1}$, then the output positions of $\mathbf{B}\theta_0 \ldots \theta_{i-1}$ are not affected by θ_i, and the result follows by the inductive hypothesis.

Finally, if $t\theta_0 \dots \theta_{i-1}$ is not a variable, then, since $s\theta_0 \dots \theta_{i-1}(= s)$ is a pure term, and since $(s = t)\theta_0 \dots \theta_{i-1}$ is unifiable, we have that $t\theta_0 \dots \theta_{i-1}$ is an instance of $s\theta_0 \dots \theta_{i-1}$. We can then take θ_i such that $Dom(\theta_i) = s\theta_0 \dots \theta_{i-1}$. It follows that $t\theta_0 \dots \theta_{i-1}$ is not affected by θ_i Consequently, $\mathbf{B}\theta_0 \dots \theta_{i-1}$ is not affected by θ_i as well and the result follows from the inductive hypothesis.

This ends the proof of Claim 1. □

Now let $\theta = \theta_0 \dots \theta_i$. By Lemma 2.2 θ is a relevant mgu of $A = H$. So far we have established that

$\mathbf{B}\theta$ is correctly typed in its output positions. (4.5)

In order to prove that also $(\mathbf{B}, \mathbf{A})\theta$ is nicely typed we have to go through a few more steps.

CLAIM 2 $\mathbf{A}\theta$ is correctly typed in its output position.

Proof. $\mathbf{A}$ is nicely moded, therefore $VarOut(\mathbf{A}) \cap Var(A) = \emptyset$. Since θ is relevant, from the disjointness hypothesis it follows then that $Var(\theta) \cap VarOut(\mathbf{A}) = \emptyset$. Since $\mathbf{A}$ is correctly typed in its output position, also $\mathbf{A}\theta$ is. □

Finally we have that

CLAIM 3 $(\mathbf{B}, \mathbf{A})\theta$ is nicely moded.

Proof. This is due to the fact that the resolvent of a nicely moded query and a (disjoint with it) nicely moded clause is nicely moded (Apt and Pellegrini in [AP92, Lemma 5.3]). □

From (4.5) and the last two Claims it follows that $(\mathbf{B}, \mathbf{A})\theta$ is nicely typed. Now $\theta = \theta_1 \dots \theta_n$ 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 nicely typed query is nicely typed, so we conclude that every LD-resolvent of $A, \mathbf{A}$ and $H \leftarrow \mathbf{B}$ is nicely typed. □

The following is an immediate consequence of Lemma 4.9 which will be soon needed.

COROLLARY 4.10 Let P and Q be nicely typed, and let ξ be an LD-derivation of $P \cup \{Q\}$. All atoms selected in ξ are correctly typed in their output positions and are output independent. □

4.3 Avoiding Unification with Nicely Typed Programs

Recall that in order to prove that $P \cup \{Q\}$ is unification-free using Theorem 4.6 we are looking for conditions which imply that all the LD-derivations starting in Q are i/o driven and that, since we are excluding the presence of input positions, this reduces to requiring that the selected atom are correctly typed in their output positions and output independent. By Corollary 4.10 the concept of being nicely typed is the one we need.

LEMMA 4.11 Suppose that each predicate symbol p occurring in P has no input positions. If

- P and Q are nicely typed.

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

PROOF. This follows directly form Corollary 4.10. □

We can now state the main result of this Section.

THEOREM 4.12 Suppose that each predicate symbol p occurring in P has no input positions. If

- P and Q are nicely typed,
- the head of every clause of P is U-safe$^-$

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

PROOF. From Lemma 4.11 and Theorem 4.6 □

This result, though rather simple, can be applied to a large number of programs.

EXAMPLE 4.13
(i) Consider again the program append, together with the following typing:

app : $U \times U \times Pt$

First note that append is nicely typed and that the head of both clauses are U-safe$^-$. Now let `t, s` be terms, and u be a variable (or a pure term), disjoint from `t, s`; `append(t,s,u)` is then a nicely typed query, and, from Theorem 4.12, it follows that append $\cup$ { `app(s, t, u)`} is unification free.

(ii) append can be used not only for concatenating two lists, but also for splitting a list in two. This is reflected by the adoption of the following typing:

app : $Pt \times Pt \times U$

Again, append is nicely typed, and the head of both clauses are U-safe$^-$. Theorem 4.12 yields that, for disjoint terms `u, v ,t`, where u and v are variables or pure terms, append $\cup$ { `app(u, v, t)`} is unification free.

(iii) Let us now consider the following permutation program:

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

perm(Xs, [X | Ys]) ←
   app1(X1s, [X | X2s], Xs),
   app2(X1s, X2s, Zs),
   perm(Zs, Ys).
perm([], []).
```

augmented by the app1 and app2 programs.

Where both app1 and app2 are renamings of the append program; we use here two distinct renamings in order to adopt two different types, namely

app1 : $Pt \times Pt \times U$
app2 : $U \times U \times Pt$

By the previous example we have that both app1 and app2 are nicely typed. Let us consider the following typing:

perm : $U \times Pt$

It is easy to check that perm is nicely typed, and that both clause's heads are U-safe$^-$. Hence, when u a variable or a pure term disjoint from t, permutation $\cup$ { perm(t, u)} is unification free. □

More examples of programs and typings that satisfy the hypothesis of Theorem 4.12 are provided by the list in Section 7.1.

5. Avoiding Unification using also the mode "input"

In the previous Section we have been using only the modes U and output. Therefore the parameter passing from the selected atom to the head of the input clause was always done via the U-positions. As we remarked before, this has the advantage of flexibility, as there is no assumption on the data structure used. However, in some cases, if we can be more precise about the kind of data structure is being used, we'll be able to broaden the range of of programs and queries that we can prove to be unification-free. Consider for instance the well-known member program.

```
member(Element, List) ←
   Element is an element of the list List.

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

It is easy to check (see Example 6.7 for a formalization of this statement) when the typing is member : $Pt \times U$, member satisfies the conditions of Theorem 4.12, therefore if s is in Pt and t is disjoint from s, then member $\cup$ { member(s, t) } is unification-free. On the other hand, it is also easy to (manually) check that if we know that t is ground, then we can drop the assumption that s is in Pt: member $\cup$ { member(s, t) } is still unification-free. In order to capture this situation, we need an extension of Theorem 4.12 that is applicable when the typing adopted is member : $U \times Ground$. In this situation, according to the convention of Definition 3.5, the second position is moded as *input*.

In this Section we provide the tools necessary to handle the presence of input positions. First notice that by Definition 3.5, the input positions of an atom are exactly the ones that are not typed *Var*, Pt or U. Consequently, considering also input positions tantamounts to considering also term_types which are not in { *Var*, *Pt*, *U* }.

The new types we interested in are *monotonic*, that is, they are closed under substitution. This property will simplify a lot the discussion.

Definition 5.1 We call a term_type T *monotonic* iff, for each substitution θ

- $t \in T$ implies $t\theta \in T$ □

From now on we make the following Assumption.

Assumption 5.2

- with the exception of term_types *Var*, *Pt*, all the term_types we refer to are monotonic. □

Notice that types *Ground*, *U* are by definition monotonic. Recall that we assume also that the type associated to a relation symbol p is always of the form $p : T_1 \times \ldots \times T_n$. The basic implication of Assumption 5.2 is then that the T_is corresponding to the input positions are always monotonic term_types.

5.1 Sequential Matching via Generic Expressions

Generic expressions were introduced by Apt-Etalle in [AE93], and can be used to obtain a new interesting condition for solvability by matching. For example, assume the standard list notation and consider a term $t = [x, y|z]$ with x, y and z variables. Note that (despite the fact that t is not a pure term), 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. In this subsection we will follow closely Apt and Etalle [AE93], and we begin with the following Definition.

DEFINITION 5.3 Let T be a term_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 the term_type T iff all left-right disjoint equations $s = t$, where $s \in T$, are solvable by matching.

EXAMPLE 5.4

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

Note that a generic expression for T needs not to be a member of T.

Next, we provide some important examples of generic expressions which will be used in the sequel. Here and in the following we call a (term_) type T *ground* if all its elements are ground, and *non-ground* if some of its elements is non-ground; consequently the *non-ground* positions of an atom H are those positions of H whose associated term_type is not a *ground* type.

LEMMA 5.5 Let T be a term_type. Then

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

PROOF. Clear. □

When the term_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 5.6 (MATCHING 2, [AE93]) Consider two disjoint 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 positions 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. □

Consider again the case of a selected atom A and the head H of a clause used to resolve A. In presence of arbitrary term_types, in order to apply the Matching 2 Lemma 5.6 to the subset of $A = H$ corresponding to the input positions, we have to impose some restrictions on H.

DEFINITION 5.7 An atom H is called *input safe* if each term t filling in a non-ground input position of H satisfies the following two conditions:

(i) t is a generic expression for this positions type,

(ii) t is disjoint from all the other terms occurring in the non-ground input positions of H. □

We also need to upgrade the Definition of U-safe$^-$ atom in order to take into account the presence of input positions.

DEFINITION 5.8 (U-SAFE) An atom H is called *U-safe* if for each term t filling in one of its U-positions one of the following two conditions holds:

(i) t is a variable or a pure term and it is disjoint from the terms occurring in the input and the other U-positions of H;

(ii) each variable occurring in t appears also in an *input* position of H of *ground* type. □

Note that when there are no input positions this Definition coincides with the one of U-safe$^-$ atom.

The above two conditions reflect two different way in which we can apply the Matching 1 Lemma 4.1 to the U-positions of $A = H$: the first conditions ensures that the term in the position we are considering is a variable or a pure term, and that it is not affected by the matching of the input and the other U-positions. On the other hand the second makes sure that after having matched the input positions of $A = H$, the term will be ground, so that the Matching 1 Lemma will still be applicable.

The above Definitions allow us to generalize Lemma 4.4 to the case in which we have also input positions.

LEMMA 5.9 (SEQUENTIAL MATCHING 2) Consider two disjoint atoms A and H with the same relation symbol. If

- A is correctly typed and output independent,
- H is input safe and U-safe,

Then there exists a permutation π such that $A = H$ is solvable by sequential matching wrt π.

In particular, $A = H$ is solvable by sequential matching wrt any permutation of $1, \ldots, n$ such that, according to the order given by $\pi(1), \ldots, \pi(n)$, we have that the non-ground input positions of p come first, the ground input positions come next, the U-positions come after them and the output positions come last.

PROOF. Suppose that $A = H$ is unifiable, we can then assume that A and H are equal respectively to $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$, where $s_1, \ldots, s_n$, $t_1, \ldots, t_n$ have been reordered in such a way that non-ground input positions come first (on the left), the ground (input) positions come next, the U-positions come third and the output positions are the rightmost ones.

We now need to prove that $s_1 = t_1, \ldots, s_n = t_n$ is solvable by sequential matching, that is we need to find $\theta_1, \ldots, \theta_n$ such that each θ_i is a match of $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$.

Let T_i be the term_type associated to the i-th position of p. Each equation $s_i = t_i$ corresponds to one position of $A = H$, we now distinguish four cases upon the kind of position the equation $s_i = t_i$ corresponds to.

First we consider the case when $s_i = t_i$ corresponds to a *non-ground* input position. Since H is input safe, t_i is a generic expression for T_i and $Var(t_i) \cap Var(\theta_1 \ldots \theta_{i-1}) = \emptyset$, so $t_i\theta_1 \ldots \theta_{i-1}$ is still a generic expression for T_i and, since $\theta_1 \ldots \theta_{i-1}$ are relevant, $t_i\theta_1 \ldots \theta_{i-1}$ is disjoint from $s_i\theta_1 \ldots \theta_{i-1}$. Moreover, A is correctly typed, thus s_i belongs to T_i, and, since by Assumption 5.2, T_i is monotonic, $s_i\theta_1 \ldots \theta_{i-1}$ belongs to T_i as well. From the Matching 2 Lemma 5.9 it follows then that $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching.

Second, we consider the case when $s_i = t_i$ corresponds to a *ground* input position. Since A is correctly typed, s_i is a ground term. From the Matching 1 Lemma 4.1 it follows then that $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching. Moreover, if $t_j, \ldots, t_k$ are the terms found in the ground input position of H, we also have that $(t_j, \ldots, t_k)\theta_1 \ldots \theta_k$ are ground terms.

Third, if $s_i = t_i$ is found in a U-position then, depending on which of the two conditions of U-safeness is satisfied we have that: (i) t_i is a variable or a pure term and $Var(t_i) \cap Var(\theta_1 \ldots \theta_{i-1}) = \emptyset$, so $t_i\theta_1 \ldots \theta_{i-1}$ is still a variable or a pure term and by the Matching 1 Lemma 4.1 $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching; (ii) $Var(t_i) \subseteq Var(t_j, \ldots, t_k)$ and, by the order hypothesis, the equations $1, \ldots, k$ have already been processed, from what noticed before it follows that $t_i\theta_1 \ldots \theta_{i-1}$ is a ground term, and again, by the Matching 1 Lemma 4.1, $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching.

Finally, if $s_i = t_i$ is found in an output position then s_i is a variable or a pure term and, since A is output independent, $Var(s_i) \cap Var(\theta_1, \ldots, \theta_{i-1}) = \emptyset$. So $s_i\theta_1, \ldots, \theta_{i-1}$ is still a variable or a pure term, and by the Matching 1 Lemma 4.1 $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching. □

This allows us to generalize Theorem 4.6. Recall that an LD-derivation is called *i/o driven* if all atoms selected in it are correctly typed and output independent.

THEOREM 5.10 Suppose that

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

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

5.2 Taking care of the input positions: Well-Typed Programs

In order to apply Theorem 5.10, we need again to find some conditions sufficient to ensure that the *LD*-derivations will be i/o-driven. As in the previous Section, the output positions will be taken care of by the fact that the programs we consider are nicely typed. Consequently, our concern is now to guarantee that the selected atoms will be correctly typed in their input positions. In presence of arbitrary term_types, the task is not trivial.

Substantially, the approach that we follow here is originally due to Bossi and Cocco [BC89], where

it was used for proving partial correctness. We use the concept of Well-Typed program, which was introduced by Bronsard, Lakshman and Reddy [BLR92], and we adopt the notation of Apt [Apt94].

We begin with the following Definition, where we assume that the input positions of atom are grouped on the left.

DEFINITION 5.11 Let $rel(A) : T_1 \times \ldots \times T_n$ be the type associated to the relation symbol of the atom A. Assume that the input positions of A are its leftmost m positions, then

- the *pre-type* for $rel(A)$ is the type

$$pre_{rel(A)} : T_1 \times \ldots \times T_m \times U \times \ldots \times U$$

and it is obtained by projecting $rel(A) : T_1 \times \ldots \times T_n$ onto its input positions. □

The pre-type of $rel(A)$ is then uniquely determined by the type of $rel(A)$; therefore from the assumption that each relation symbol has always a type associated to it it follows that each relation symbol has automatically also a pre-type associated to. The advantage of referring to the pre-type instead of the type is that by Assumption 5.2 the pre-type is always monotonic.

To give the definition of Well-Typed program we need two more notions.

DEFINITION 5.12 Let $A_1, \ldots, A_{n+1}$ be atoms and $\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}$ be monotonic types

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

$$\models A_1 \in \mathcal{T}_1 \wedge \ldots \wedge A_n \in \mathcal{T}_n \Rightarrow A_{n+1} \in \mathcal{T}_{n+1}$$

which denotes that, for all substitutions θ, $Dom(\theta) = Var(A_1, \ldots, A_n)$:

$$\text{if } A_1\theta \in \mathcal{T}_1 \wedge \ldots \wedge A_n\theta \in \mathcal{T}_n \text{ then } A_{n+1}\theta \in \mathcal{T}_{n+1}$$

□

Recall that in order to apply Theorem 5.10, we have to prove that each selected atom belongs to its pre-type; to do this we use type judgements and associate to each relation symbol also a *post-type*.

DEFINITION 5.13 A *post-type* for a relation symbol p, is a monotonic type for p. □

From now on we assume that each relations symbol has, together with the type, also a post-type associated to it.

As opposed to the type, we want the post-type to contain information about the state of the arguments of a query *after* the query itself has been successfully resolved. For example, consider again the program append. A typical typing for it is app: $List \times List \times Pt$[1]. This formalizes the idea that when and atom of the form `app(s, t, u)` is *selected*, we expect `s` and `t` to be variables and `u` to be a variable, or, at most, a pure term. On the other hand, we require the post-type to hold some knowledge over the situation of `s`, `t` and `u` after that the query `app(s, t, u)` has been successfully resolved. In this situation a natural post-type would be $post_{\texttt{app}} : List \times List \times List$, indicating that, after `app(s, t, u)` has succeeded, we also expect `u` to be a list. Notice also that when the type adopted is the above one, the the pre-type is $pre_{\texttt{app}} : List \times List \times U$.

In the following we write $pre(A)$ (resp. $post(A)$) as shorthand for $A \in pre_{rel(A)}$ (resp. $A \in pre_{rel(A)}$), where $pre_{rel(A)}$ and $post_{rel(A)}$ are the pre- and post-type of the relation symbol of A.

[1]This is a slight extension of the "natural" typing app: $List \times List \times Var$ that we mentioned in Sections 3 and 4

DEFINITION 5.14

- A query $A_1, \ldots, A_n$ is called *well-typed* if, for $j \in [1, n]$,

$$\models post(A_1) \wedge \ldots \wedge post(A_{j-1}) \Rightarrow pre(A_j).$$

- A clause $H \leftarrow B_1, \ldots, B_n$ is called *well-typed* if, for $j \in [1, n+1]$,

$$\models pre(H) \wedge post(B_1) \wedge \ldots \wedge post(B_{j-1}) \Rightarrow pre(B_j),$$

  where $pre(B_{n+1}) := post(H)$.

- A program is called *well-typed* if every clause of it is. □

Thus, a query is well-typed if

- the pre-type of an atom can be deduced from the post-types of previous atoms.

And a clause is well-typed if

- ($j \in [1, n]$) the pre-type a body atom can be deduced from the pre-type of the head and the post-types of the previous body atoms,
- ($j = n + 1$) the post-types of the head can be deduced from the pre-type of the head and the post-types of the body atoms.

In particular a query A is well-typed iff $\models pre(A)$, while a unit clause $A \leftarrow$ is well-typed iff $\models pre(A) \Rightarrow post(A)$.

The following result states the persistence of the notion of being well-typed (see Bossi-Cocco [BC89] or an account of it Apt-Marchiori [AM94]).

LEMMA 5.15 (PERSISTENCE) An LD-resolvent of a well-typed query and a well-typed clause that is variable disjoint with it, is well-typed. □

This brings us to the following conclusion.

COROLLARY 5.16 Let P and Q be well-typed, and let ξ be an LD-derivation of $P \cup \{Q\}$. Then every atom selected in ξ is correctly typed in its input positions.

PROOF. A variant of a well-typed clause is well-typed and for a well-typed query $A_1, \ldots, A_n$ we have $\models pre(A_1)$. □

5.3 Avoiding Unification with Well+Nicely Typed Programs

Recall that in order to prove that $P \cup \{Q\}$ is unification-free using Theorem 4.6 we are looking again for conditions which imply that all the LD-derivations starting in Q are i/o driven: we want that the selected atom is correctly typed and output independent.

The combination of the concepts of being well-typed and being nicely typed allows us to deal with all the cases in which the types used satisfy Assumption 5.2: well-typedness takes care of the input position, while nicely typedness takes care of the output ones.

LEMMA 5.17 Suppose that

- P and Q are nicely typed and well-typed.

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

PROOF. It follows from Corollaries 5.16 and 4.10. □

This brings us to the main result of this paper.

THEOREM 5.18 (MAIN) Suppose that

- P and Q are nicely typed and well-typed,
- the head of every clause of P is input safe and U-safe

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

PROOF. From Lemma 5.17 and Theorem 5.10. □

In particular, from the Sequential Matching 2 Lemma 5.9 it follows that each of the equations $A = H$ considered in the LD-derivations can be solved by sequentially matching (one by one) each of the atoms positions, provided that we observe the following order: first the nonground input positions, then the ground input positions, after that the U-positions and finally the output ones. In the Appendix we'll show how we can improve on this result by grouping some positions under the same match.

It is not difficult to check that this Theorem 5.18 generalizes our previous result, Theorem 4.12. Indeed if the program P and the query Q satisfy the conditions of Theorem 4.12, then, since the atoms have no input positions, we have that the heads of the clauses of P are trivially input-safe and, by assigning to each predicate symbol p the trivial post-type $p : U \times \ldots \times U$, we have that P and Q are well-typed. Therefore P and Q satisfy the hypothesis of Theorem 5.18 as well.

EXAMPLE 5.19 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]).
```

Let us associate to it the following typing,

	type	post-type
`ps`	: $List \times Pt$	$List \times List$
`permutation`	: $List \times Pt$	$List \times List$
`select`	: $Pt \times List \times Pt$	$U \times List \times List$
`ordered`	: $List$	$List$

Now, `permutation sort` is well-typed and nicely typed. Moreover, the heads of all clauses are input safe and U-safe[2]. By the Main Theorem 5.18 we get that for a list `s` and a disjoint with it variable or pure term `t`, `permutation sort` $\cup$ { `ps(s, t)`} is unification free.

Observe that 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 terms. In a sense we could say that `[X]` and `[X, Y | Xs]` are nontrivial generic expressions. □

6. A SIMPLER SPECIAL CASE: GROUND INPUT POSITIONS

Sometimes, a lot of the machinery needed by Theorem 5.18 is actually superfluous. In particular, this happens when the input positions are all of ground type. In this case, instead of requiring the program to be well-typed, we can use the more restrictive concept of well-moded program. This has two relevant advantages:

First, that we do not need to associate a post-type to each relation symbol.

Second, while checking that a program is well-typed is an algorithmically intractable problem, testing well-modedness can be done in polynomial (quadratic) time. A discussion on the algorithmic tractability of the concepts used in this paper is reported in Section 6.3.

In this Section we'll assume that the only term_type used for the input positions in *Ground*. Informally, this means that the information we pass to the program consists always of ground terms. By Definition 3.5 this is equivalent to assuming that we use types which are built using only the following term_types: *Ground, Pt, Var, U.*

6.1 Well-Moded programs

The concept of Well-Moded program is essentially due to Dembinski and Maluszynski [DM85a]; here we make use of the elegant formulation of Rosenblueth [Ros91] and of the same notation of [AE93]. In particular, 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 and the U-positions of p (notice that this shorthand is different from the one used for Definition 4.7).

DEFINITION 6.1

- 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

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

[2]The latter statement is trivial, as there are no U-positions: the fact that U appears in a post-type is of no relevance here.

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 a non-input 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 a non-input position of an earlier ($j \in [1, i-1]$) body atom,
- ($i = n+1$) every variable occurring in an non-input 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 important to notice 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 term_type used for the input positions is *Ground*, and the post-type associated to each relation symbol p is $p : Ground \times \ldots \times Ground$, then the notions of a well-typed program (resp. query) and a well-moded program (resp. query) coincide.

The following Lemma states the persistence of the notion of being well-moded. A proof of it can be found in Apt and Marchiori [AE93].

LEMMA 6.2 An LD-resolvent of a well-moded query and a disjoint with it well-moded clause is well-moded. □

The next result is originally due to Dembinski and Maluszynski and follows directly from the definition of well-moded program.

COROLLARY 6.3 Let P and Q be well-moded, and let ξ be an LD-derivation of $P \cup \{Q\}$. All atoms selected in ξ contain ground terms in their input positions. □

6.2 Avoiding Unification with Well-Moded Nicely Typed Programs

As we anticipated at the beginning of this Section, here we assume that the only term_type used for the input position is *Ground*, this is equivalent to making the following

ASSUMPTION 6.4 In this subsection we each predicate symbol has a type associated to it of the form $p : T_1 \times \ldots \times T_n$, where for $i \in [1, n]$, $T_i \in \{Ground, Var, Pt, U\}$. □

Once again we are going to use Theorem 4.6 for proving that $P \cup \{Q\}$ is unification-free. Therefore we are looking again for conditions which imply that all the LD-derivations starting in Q are i/o driven: the selected atoms in a LD-derivation need to be correctly typed and output independent. As in the previous two Sections, the concept of being nicely typed will take care of the output positions.

Since we are assuming that the input positions are always of ground type, from Corollary 6.3 it follows that well-modedness is what we need for taking care of the input positions.

LEMMA 6.5 If Assumption 6.4 is satisfied and

- P and Q are nicely typed and well-moded.

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

PROOF. Let A be a selected atom in an LD-derivation of $P \cup \{Q\}$. By Corollary 6.3 the input positions of A are correctly typed, and by Corollary 4.10, A is correctly typed in its output positions is output independent. □

This, together with Theorem 4.6, brings us to the following conclusion.

THEOREM 6.6 If Assumption 6.4 is satisfied and

- P and Q are nicely typed and well-moded,
- the head of every clause of P is U-safe

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

PROOF. It follows directly from Lemma 6.5 and Theorem 4.6. □

It is easy to check that this is a special case of Theorem 5.18: if P and Q satisfy its hypothesis, then P and Q are well-moded and, as we mentioned before, well-moded programs (and queries) are a special case of well-typed programs in which the only term_type used for the input positions is *Ground*. Therefore P and Q satisfy also the condition of being well-typed, moreover, we also have that the heads of P are (trivially) input safe. Consequently P and Q satisfy the hypothesis of Theorem 5.18 as well.

EXAMPLE 6.7
(i) First, let us go back to what we stated at the beginning of Section 5, and let us consider again the program `member`. With the typing `member`: $U \times$ *Ground*, `member` is well-moded and (trivially, as there are no output positions) nicely typed; moreover, all clause's heads are U-safe. By Theorem 6.6 if `t` is a ground term, then, for any `s`, `member` $\cup$ { `member(s, t)`} is unification free.

Let us compare this with what we could have obtained by using the result (namely, Theorem 4.12) given in the Section 4. Without using input positions we can prove that, when the following type is used:

`member` : $Pt \times U$

then `member` is nicely typed and all clause's heads are U-safe. By Theorem 4.12 this implies that if `s` is a variable or a pure term disjoint from `t`, then `member` $\cup$ { `member(s, t)`} is unification free. In this case, the advantage of Theorem 6.6 over Theorem 4.12 is that we can allow `s` to be any term. The price we have to pay for this is that Theorem 6.6 requires `t` to be ground. Symmetrically, Theorem 4.12 imposes no conditions on `t` (which can be then a nonground list, or any other term) but requires `s` to be a variable or a pure term.

Notice also that, when the above types are used, Theorem 6.6 is not applicable, as the program is not well-moded. This shows that Theorem 6.6 is not more general that Theorem 4.12.

(ii) Consider now the `MapColor` program:

```
color_map(Map, Colors) ←
   Map is correctly typed using Colors.

color_map([Region | Regions], Colors) ←
   color_region(Region, Colors),
   color_map(Regions, Colors),
color_map([], _ ).

color_region(Region, Colors) ←
   Region and its neighbors are correctly colored using Colors.

color_region(region(Name, Color, Neighbors) , Colors) ←
   select(Color, Colors, ColorsLeft),
   subset(Neighbors, ColorsLeft).

select(X, Xs, Zs) ←
   Zs is the result of deleting one occurrence of X from the list Zs.

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

subset(Xs, Ys) ←
   each element of the list Xs is also an element of the list Ys.

subset([X | Xs], Ys) ← member(X, Ys), subset(Xs, Ys).
subset([] , _ ).
```

augmented by the member program.

Let us associate to it the following typing:

$$\begin{aligned} \texttt{color\_map} &: U \times \mathit{Ground} \\ \texttt{color\_region} &: U \times \mathit{Ground} \\ \texttt{select} &: U \times \mathit{Ground} \times \mathit{Pt} \\ \texttt{subset} &: U \times \mathit{Ground} \\ \texttt{member} &: U \times \mathit{Ground} \end{aligned}$$

It is straightforward to check that with the above typing, MapColor is well-moded and nicely typed. Since the head of all clauses are U-safe, by Theorem 6.6 we have that, if t is a ground term, then, for any s, color_map $\cup$ { color_map(s, t)} is unification free. □

It is worth noticing that the U-positions have been used in (at least) two opposite ways: in Section 4 we they were actually used as "input" positions, in the sense that they were used to transfer information from the selected atom to the head of the clause used to resolve it, while in Section 6 they were more used as "output". This becomes noticeable in the moment that we compare Example 4.13 with Example 6.7. However, it should be mentioned that this distinction is not always so clear: consider for instance the program select (which is a subprogram of the above MapColor): A query select(s, t, u) can be used in two main ways: to delete the element s from the list t and report the result in u, or as a generalized member program, to report in s an element of t, and in u the remains of the list. In the first case the first position is used as "input", in the second as "output", but for both cases we can simply use the typing select : $U \times \mathit{Ground} \times \mathit{Pt}$. In this case the mode U takes care of the ambivalence of the first position. Notice also that when we adopt this typing the hypothesis of Theorem 6.6 are satisfied, therefore if t is ground, u is in Pt and s is disjoint from s then selectUselect(s, t, u) is unification-free.

6.3 Comparing Theorems 4.12, 5.18 and 6.6: efficiency issues
Theorem 5.18 is a generalization of Theorems 4.12 and 6.6, but the latter two are much more suitable for being used in an automatic way.

In fact, it is worth noticing that the applicability conditions of Theorems 4.12 and 6.6 can be statically and efficiently tested: in order to check that a program is nicely typed, well-moded and the head of its clauses are input safe, one can easily find some naive algorithms whose complexity is quadratic in the size of the clauses and linear in the number of clauses in a program. Indeed, all three concepts require procedures like the following one.

```
for each clause cl in P do
    for each variable v occurring in cl do
        begin
           check that all the other occurrences of v in cl satisfy the
           required conditions (this require re-scanning cl)
        end
```

On the other hand, to test the hypothesis of Theorem 5.18 one needs to check if some *type judgements* hold, and this is a much more complex problem, in fact, for artificially built types, it can even be undecidable. Aiken and Lakshman in [AL93] have investigated the problem of checking type judgements for monotonic types: they prove that it is EXPTIME-hard and they state that no upper bound is known, moreover, they show that also in the case that we use only *discriminative types*[3] then the problem has a a lower complexity bound of PSPACE, and a upper bound of NEXPTIME. In other words, even in this more restrictive case, the problem remains highly untractable.

Thus, checking the conditions of Theorems 4.12 and 6.6 is much simpler than checking the ones of Theorem 5.18, moreover, by checking the list in Section 7.1, one can easily realize that the practical cases in which Theorem 5.18 is really useful are a minority: in most cases Theorems 4.12 and 6.6 are sufficient for our purposes.

7. What have we done and what have we not done

7.1 What have we done: the List
To apply the established results to a program and a query, one needs to find appropriate typings for the considered relations such that the conditions of one of the Theorems 4.12, 5.18 or 6.6, 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 typings these theorems are applicable.

In programs which use difference-lists we replace "\" 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)` ← `r(X)` has to be replaced by `p(X,Y)` ← `r(X,Y)`. Such changes are purely syntactic and they allow us to draw conclusions about the original program.

We also report between parenthesis typings which are "subsumed" by other typings in the list, that is, typings for which there exists another typing which is more general. We report them here because they provide further examples of typings wrt which these programs are (unification-free and) well-typed (or well-moded).

program	page	Thm.	Typing

[3] a *discriminative type* is a type built using to some specific rules which include a fixpoint set construction; according to Aiken and Lakshman "The important restriction of discriminative set expressions are that no intersection operation is allowed and all union are formed from expressions with distinct outermost constructor". In any case, discriminative types are descriptive enough to be able to handle all the examples presented here.

member	45	4.12	$Pt \times U$	
		6.6	$U \times Ground$	
		(5.18)	$(Pt \times List)$	
prefix	45	4.12	$Pt \times U$	
		6.6	$Ground \times Ground$	
		(6.6)	$(Pt \times Ground)$	
		(5.18)	$(Pt \times List)$	
suffix	45	4.12	$Pt \times U$	
		6.6	$Ground \times Ground$	
		(6.6)	$(Pt \times Ground)$	
		(5.18)	$(Pt \times List)$	
naive reverse	48	4.12	$U \times Pt$	
		6.6	$Ground \times U$	
		(5.18)	$(List \times Pt)$	
reverse-accum.	48	4.12	$U \times Pt$,	$U \times U \times Pt$
		6.6	$Ground \times U$,	$Ground \times Ground \times U$
		(5.18)	$(List \times Pt$,	$List \times List \times Pt)$
delete	53	5.18	$Ground \times U \times Pt$	
		5.18	$Ground \times U \times Ground$	
		(6.6)	$(Ground \times Ground \times Pt)$	
select	53	4.12	$Pt \times U \times Pt$	
		4.12	$U \times Pt \times U$	
		6.6	$U \times Ground \times Pt$	
		6.6	$Ground \times Ground \times Ground$	
		(6.6)	$(Ground \times Ground \times Pt)$	
		(5.18)	$(Pt \times List \times Pt)$	
insertion sort	55	4.12	$s : U \times Pt$,	$i : U \times U \times Pt$
		(6.6)	$(s : Ground \times Pt$,	$i : Ground \times Ground \times Pt)$
		(5.18)	$(s : List \times Pt$,	$i : U \times List \times Pt)$
quicksort	56	4.12	$q : U \times Pt$,	$p : U \times U \times Var \times Var$
		(6.6)	$(q : Ground \times Pt$,	$p : Ground \times Ground \times Var \times Var)$
		(5.18)	$(q : List \times Pt$,	$p : U \times List \times Pt)$
tree-member	58	4.12	$Pt \times U$	
		6.6	$U \times Ground$	
		6.6	$Ground \times Ground$	
		(5.18)	$(Pt \times BinTree)$	
isotree	58	4.12	$U \times Pt$	
		4.12	$Pt \times U$	
		6.6	$Ground \times Ground$	
		(6.6)	$(Ground \times Pt)$	

		(6.6)	$(Pt \times Ground)$
		(5.18)	$(BinTree \times Pt)$
		(5.18)	$(Pt \times BinTree)$
substitute	60	5.18	$U \times U \times Ground \times Pt$
		5.18	$U \times U \times Pt \times Ground$
		5.18	$U \times U \times Ground \times Ground$
		(6.6)	$(Ground \times Ground \times Ground \times Pt)$
		(6.6)	$(Ground \times Ground \times Pt \times Ground)$
pre-order	60	4.12	$U \times Pt$
		6.6	$Ground \times U$
		(5.18)	$(BinTree \times Pt)$
in-order	60	4.12	$U \times Pt$
		6.6	$Ground \times U$
		(5.18)	$(BinTree \times Pt)$
post-order	60	4.12	$U \times Pt$
		6.6	$Ground \times U$
		(5.18)	$(BinTree \times Pt)$
polynomial	62	6.6	$Ground \times U$
derivative	63	6.6	$Ground \times U \times Pt$
		6.6	$Ground \times U \times Ground$
hanoi	64	4.12	$U \times U \times U \times U \times Pt$
		6.6	$U \times Ground \times Ground \times Ground \times U$
reverse_dl	244	4.12	$r : U \times Pt$, $r\_dl : U \times Pt \times U$
		6.6	$r : Ground \times U$, $r\_dl : Ground \times U \times Ground$
		(5.18)	$(r : List \times Pt$, $r\_dl : List \times Pt \times List)$
dutch	246	4.12	$dutch : U \times Pt$, $di : U \times Pt \times Pt \times Pt$
		6.6	$dutch : Ground \times U$, $di : Ground \times Pt \times Pt \times Pt$
dutch_dl	246	4.12	$dutch : U \times Pt$, $di : U \times Pt \times Pt \times Pt \times U$
parsing	258	6.6	all $Ground \times U$

7.2 What have we not done

Still, there are some natural programs that when executed do not require unification, while they cannot be proven unification-free using our method. We are aware of the following two examples: `quicksort_dl` and `flatten_dl` [SS86, pag. 244, 241].

First, let us consider `quicksort_dl`.

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

```
    qs_dl(Littles, Ys, [X|Y1s]),
    qs_dl(Bigs, Y1s, Zs).
qs_dl([], Xs, Xs).

partition(X, [Y | Xs], [Y | Ls], Bs) ← X > Y, partition(X, Xs, Ls, Bs).
partition(X, [Y | Xs], Ls, [Y | Bs]) ← X ≤ Y, partition(X, Xs, Ls, Bs).
partition(X, [], [], []).
```

By looking at the trace of the program, it is easy to see that, if `t` is a list and `s` is a variable disjoint with `t`, then `quicksort_dl`$\cup\{$ `qs(t, s)` $\}$ is unification free. Indeed, if we use the following types:

$$\begin{aligned} \texttt{qs} &: \mathit{List} \times \mathit{Var} \\ \texttt{qs\_dl} &: \mathit{List} \times \mathit{Var} \times U \\ \texttt{partition} &: U \times \mathit{List} \times \mathit{Var} \times \mathit{Var} \end{aligned}$$

then we have that the heads of all the clauses are input safe and U-safe, moreover, we can check "by hand" that, if $\{$ `qs(t, s)` $\}$ is correctly typed and output independent, all LD-derivations of `quicksort_dl` $\cup$ $\{$ `qs(t, s)` $\}$ are i/o driven, therefore, by Theorem 5.10, `quicksort_dl` $\cup$ $\{$ `qs(t, s)` $\}$ is unification-free. The problem here is that the program is not nicely typed: `Y1s` appears first in the U-position of `qs_dl(Littles, Ys, [X|Y1s])` and then in the output position of `qs_dl(Bigs, Y1s, Zs)`, therefore, with the tools in our possession, we cannot *prove* that the derivations are i/o driven, in particular we can't show that each time that an atom of the form `qs_dl(t, s, r)` is selected, `s` will be a variable[4].

Now, let us consider the program `flatten_dl`.

```
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).
```

Incidentally, the reasons why we cannot `flatten_dl` to be unification-free are the same ones found for the program `quicksort_dl`. If we associate to it the following types:

$$\begin{aligned} \texttt{flatten} &: \mathit{Ground} \times \mathit{Var} \\ \texttt{flatten\_dl} &: \mathit{Ground} \times \mathit{Var} \times U \end{aligned}$$

We have that the heads of all the clauses are input safe and U-safe, and, in the case that `t` is a list and `s` is a variable disjoint with `t`, all LD-derivations of `flatten_dl` $\cup$ $\{$ `flatten(t, s)` $\}$ are i/o driven, therefore, by Theorem 5.10, `flatten_dl` $\cup$ $\{$ `flatten(t, s)` $\}$ is unification-free. Again, the problem here is that the program is not nicely typed: `Y1s` appears first in the U-position of `flatten_dl(X, Ys, Ys1)` and then in the output position of `flatten_dl(Xs, Ys1, Zs)`; consequently, with our tools we cannot guarantee the i/o drivenness of the derivations.

In the literature we do find tools that would enable us to prove these two programs to be unification-free, namely *asserted programs*. *Assertions* can be viewed as extension of types, and provide a more expressive formalism for proving run-time properties like groundness of terms and independence of variables (see Apt-Marchiori [AM94]). Two are the reasons why we decided not to use assertions in this paper: in the first place, the machinery involved is far more complicated and computationally expensive than with types, and when we use types in full generality we already face the algorithmically

[4]It may be interesting to notice that, if we want to prove "by hand" that this program is unification-free, then the key step is indeed represented by showing that each time that an atom of the form `qs_dl(t, s, r)` is selected, `s` will be a variable.

intractable problem of checking type judgements. Secondly, the only two programs that we know of that can be proven to be unification-free using assertions and not with types are precisely `flatten_dl` and `quicksort_dl`. Summarizing, we strongly believe that the gain in generality is far not worth the loss in clarity and efficiency.

Of course, the results of this paper allow us to can prove `quicksort_dl` and `flatten_dl` are unification-free wrt the following types:

$$\begin{aligned} \texttt{qs} &: \mathit{Ground} \times \mathit{Ground} \\ \texttt{qs\_dl} &: \mathit{Ground} \times \mathit{Ground} \times U \\ \texttt{partition} &: \mathit{Ground} \times \mathit{Ground} \times \mathit{Var} \times \mathit{Var} \\ \texttt{flatten} &: \mathit{Ground} \times \mathit{Ground} \\ \texttt{flatten\_dl} &: \mathit{Ground} \times \mathit{Ground} \times U \end{aligned}$$

However this are not the natural typings for these programs: for instance they require that in the queries `qs(t, s)` and `flatten(t, s)` both `t` and `s` are ground terms. In practice we have to know the result of the computation in advance.

7.3 What cannot be done: when is unification needed

Considering the surprisingly large number of programs that could be proven to be unification-free, in [AE93] we raised the question of whether unification was actually intrinsically needed in Prolog programs: "A canonic example (of a program requiring unification) 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 output positions the form of computed answers, almost any program will necessitate the use of unification."

In one year we have been running into a couple of interesting examples. The first one is the program `append_dl` [SS86, Pag. 241].

```
append_dl(As, Bs, Cs) ←
    the difference-list Cs is the result concatenating the difference-lists As and Bs.
append_dl(Xs \ Ys, Ys \ Zs, Xs \ Zs).
```

`append_dl` can concatenate the difference lists `As` and `Bs` in constant time, a relevant improvement over the ordinary `append`, which takes linear time. However, it is easy to see that in most cases `append_dl` does requires the use unification.

A second example is provided by the Prolog formalization of a problem from Coelho and Cotta [CC88, pag. 193]: arrange three 1's, three 2's, ..., three 9's in sequence so that for all $i \in [1,9]$ there are exactly i numbers between successive occurrences of i.

```
sublist(Xs, Ys) ← Xs is a sublist of the list Ys.
sublist(Xs, Ys) ← app(_, Zs, Ys), app(Xs, _, Zs).

sequence(Xs) ← Xs is a list of 27 elements.
sequence([_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]).

question(Ss) ← Ss is a list of 27 elements forming the desired sequence.
question(Ss) ←
    sequence(Ss),
    sublist([1,_,1,_,1], Ss),
    sublist([2,_,_,2,_,_,2], Ss),
    sublist([3,_,_,_,3,_,_,_,3], Ss),
    sublist([4,_,_,_,_,4,_,_,_,_,4], Ss),
    sublist([5,_,_,_,_,_,5,_,_,_,_,_,5], Ss),
```

```
sublist([6,_,_,_,_,_,_,6,_,_,_,_,_,_,6], Ss),
sublist([7,_,_,_,_,_,_,_,7,_,_,_,_,_,_,_,7], Ss),
sublist([8,_,_,_,_,_,_,_,_,8,_,_,_,_,_,_,_,_,8], Ss),
sublist([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9], Ss).
```

augmented by the append program.

In this case Prolog provides a straightforward and elegant way of formalizing the problem, however by looking at the trace of the execution it is easy to check that, in order to run properly, the program fully uses unification.

8. Conclusions

8.1 Relations with [AE93]

This paper can be seen as an extension of Apt and Etalle [AE93]. Technically, the main differences between this and the previous paper can be summarized as follows:

- In [AE93] only input and output positions are considered while here we introduce and use U-positions as well.
- In [AE93] the only terms that are allowed to fill in the output positions of the queries are variables. Here, by using the type Pt, we often allow the presence of pure terms, and this broadens the class of programs and queries that we can prove to be unification-free.
- Like in here, in [AE93], the programs considered needed always to be well-typed[5], however, the definition of well-typed programs used in [AE93] is more restrictive than the present ones.

The practical consequence of these facts are manifold.

- The results can be applied to a larger class of programs.

  Examples of programs that could not be handled with the tools of [AE93] and that can be handled now are `permutation` and `color_map`.

- The results can be applied to a larger class of queries.

  In almost all cases, programs which could be handled in [AE93] can be now handled better, i.e. the class of allowed queries is now broader. To give a simple example, let us consider the program `member`. Using the tools of [AE93], we can prove to be unification-free wrt the following typings:

  (1) `member`: $Ground \times Ground$,

  (2) `member`: $Var \times Ground$,

  (3) `member`: $Var \times List$

  On the other hand, using the tools given in this paper we can prove `member` to be unification-free wrt the following typings:

  (a) `member`: $U \times Ground$

  (b) `member`: $Pt \times U$

  It is easy to see that the typing (a) is more general than both (1) and (2), while (b) is more general than both (2) (again) and (3): the class of queries for which we can prove unification freedom is now quite larger, and we can do this using a reduced number of different typings (two instead of three), thus reducing the machinery involved in the proof.

[5] recall that in the discussion after Theorem 5.18 we showed that, by appropriately choosing the *type* and the *post-type* for a relation symbol, all the programs that satisfy the conditions of Theorem 4.12 or the ones of Theorem 6.6 are well-typed.

- The hypothesis of the theorems are often checkable in a much more efficient way.

  In order to provide an example, let us consider again the **member** program, together with the typings given above. First recall that the typing (b) is more general that both typings (2) and (3). Now, an important advantage of (b) over (3) is the following: in order to use (3) we have to use Theorem 30 of [AE93][6] which requires to check some non-trivial type judgement, and this is, as discussed before, an algorithmically intractable problem. On the other hand, in order to prove unification freedom using typing (b), can use Theorem 4.12, our simplest result, whose hypothesis can be simply and efficiently tested.

  This situation is not incidental: by looking at the list of programs reported in [AE93, Section 8][7] and comparing it with the one in Section 7.1 of this paper, we see that in most of the cases in which we had some nonground input positions, we could simply turn these positions into U-positions, and prove unification freedom using Theorem 4.12 instead of Theorem 30 of [AE93], both enlarging the class of allowed queries and simplifying dramatically the process of proving that the program is unification-free.

8.2 Other related work

Other recent related works on unification-free programs are the ones of M. Marchiori [Mar94] and of Krishna Rao and Shyamasundar [RS94].

[Mar94] concentrates on Well-Moded programs and studies *maximal localizations* of the property of being Unification-Free. In order to compare his paper with our we have to introduce a bit of notation. Let us be brief and informal.

We say that a property $\mathcal{P}$ is *local* if for any two programs P and Q, we have that the P and Q satisfy $\mathcal{P}$ iff $P \cup Q$ satisfies $\mathcal{P}$ as well. In other words, $\mathcal{P}$ is local if it can be checked clause by clause. For instance the property *"P is Well-Moded and Nicely typed wrt the typing $\mathcal{T}$"* is local, while the property *"there exists a typing $\mathcal{T}$ such that P is Well-Moded and Nicely typed wrt it"* is not local, as we need to traverse the program more than once to check it (eventually we have to try different $\mathcal{T}$s). We also say that a property $\mathcal{Q}$ is more general than $\mathcal{P}$ if each program that satisfies $\mathcal{P}$ satisfies $\mathcal{Q}$ as well.

Now, the question addressed in [Mar94] is the following:

- assume that to each relation symbol is already associated a typing of the form

$$p: T_1 \times \ldots \times T_n, \text{ where, for each } i, T_i \in \{Ground, U\}. \tag{8.6}$$

  we want to find (if it exists) a *local* property $\mathcal{P}$ such that

  - each program that satisfies $\mathcal{P}$ is Well-Moded (wrt the given typing (8.6));
  - each program that satisfies $\mathcal{P}$ is Unification-Free;
  - $\mathcal{P}$ is maximal, that is, there is no other local property $\mathcal{Q}$ which is more general than $\mathcal{P}$ and that satisfies the above two conditions.

In [Mar94] it is proven that such properties exist, in particular two of them are defined in detail[8]. Of course there exist other maximal properties that satisfy the above conditions.

[6] Roughly speaking, [AE93, Theorem 30] is a restricted version of Theorem 5.18, and it is the most general result of [AE93].

[7] the reader who actually does so has to be warned that the notation is a bit different: for instance the type **select** (–:*U*, +:*List*, –:*List*) of [AE93] corresponds to our type **select** : *Var*, *List*, *Var* together with the post-type **select** : *U*, *List*, *List*.

[8] These two properties are named "(the property of being) *Flatly-Well-Moded*" and *"coFlatly-Well-Moded"*

Summarizing, the goal of [Mar94] is quite different from our own: [Mar94] focuses more on the theoretical aspects of local properties in the context of well-moded program, while here we want to provide (possibly simple) tools for proving unification freedom for a (possibly) large class of programs and queries. Indeed the class of programs and queries for which we can prove unification freedom is substantially larger than in [Mar94]; this is mainly due to two reason: firstly, because restricting to the class of Well-moded program already narrows sensibly the set of allowed queries (recall that of the programs of the List, the ones that are Well-Moded are the ones which are proven to be Unification-Free via Theorem 6.6); secondly, because *local* properties are, at least in this context, intrinsically rather weak.

Finally, in [RS94] it is proposed to use the same tools of [AE93] together with a top-down algorithm which, given the query we are interested in, searches (an abstraction of) its derivation tree in order to find out all the output positions in the selected atoms which may be filled in by a non-variable term. This allows to have terms other than variables in the output positions of the query. Their results are devised for well-moded programs, but can also partially be applied to well-typed ones. The basic difference between [RS94] and this paper lies in the fact that the method they propose is a top-down one, and involves the search of the derivation tree of a query. Moreover, the results in [RS94] are almost exclusively applicable to Well-Moded programs, and, as discussed before, this narrows sensibly the class of allowed queries.

9. Acknowledgements

Many of the ideas presented here originated during joint research and discussions with K. R. Apt, who also provided invaluable remarks all along the development of this paper. I also want to thank M. R. K. Krishna Rao, who found a mistake in a proof, and Maurizio Gabbrielli, Nicoletta Cocco and Massimo Marchiori for their helpful comments and corrections on the final draft.

10. Appendix: reducing the number of matches

Let $A = p(\mathbf{s})$ and $H = p(\mathbf{t})$ be two atoms. We know that if the hypothesis of the Sequential Matching 2 Lemma 5.9 are satisfied, then the equations in $\mathbf{s} = \mathbf{t}$ are solvable, *one at a time*, by matching.

Here we want to show that some subsets of $\mathbf{s} = \mathbf{t}$ containing more than one equation can be solved by a single matching. This reduces the total number of matchings needed to solve $\mathbf{s} = \mathbf{t}$, and results in an efficiency gain: since there are parallel algorithms for term matching that run in polilogarithmic time [DKM84, DKS86], matching more positions at once increases the execution speed.

Lemma 10.1 Consider two disjoint atoms $A = p(\mathbf{s})$ and $H = p(\mathbf{t})$ with the same relation symbol. Assume that A correctly typed and output independent, and that H is input safe and U-safe. Let us now divide the set of equations $\mathbf{s} = \mathbf{t}$ into the following subsets: let

- $\mathbf{s}_1 = \mathbf{t}_1$ be the subset of $\mathbf{s} = \mathbf{t}$ corresponding to the *nonground* input positions.
- $\mathbf{s}_2 = \mathbf{t}_2$ be the subset of $\mathbf{s} = \mathbf{t}$ corresponding to the *ground* input positions.
- $\mathbf{s}_3 = \mathbf{t}_3$ be the subset of $\mathbf{s} = \mathbf{t}$ corresponding to the U-positions with respect to which H satisfies condition (ii) of U-safeness (Definition 5.8).
- $\mathbf{s}_4 = \mathbf{t}_4$ be the subset of $\mathbf{s} = \mathbf{t}$ corresponding to those of the remaining U-positions of H which are filled in by a variable.
- $s_5 = t_5, \ldots, s_k = t_k$ be the subsets of $\mathbf{s} = \mathbf{t}$ such that for $i \in [5, k]$, each $s_i = t_i$ corresponds to one of the remaining U-positions.
- $s_{k+1} = t_{k+1}, \ldots, s_l = t_l$ be the subsets of $\mathbf{s} = \mathbf{t}$ such that for $i \in [k+1, l]$, each $s_i = t_i$ corresponds to a position of type Pt.

- $\mathbf{s}_{l+1} = \mathbf{t}_{l+1}$ be the subset of $\mathbf{s} = \mathbf{t}$ corresponding to the positions typed Var.

Then

$$\mathbf{s}_1 = \mathbf{t}_1,\ \mathbf{s}_2 = \mathbf{t}_2,\ \mathbf{s}_3 = \mathbf{t}_3,\ \mathbf{s}_4 = \mathbf{t}_4,\ s_5 = t_5, \ldots, s_k = t_k,\ s_{k+1} = t_{k+1}, \ldots, s_l = t_l,\ \mathbf{s}_{l+1} = \mathbf{t}_{l+1}$$

is solvable by sequential matching.

Here notice that $\mathbf{s}_1 = \mathbf{t}_1$, $\mathbf{s}_2 = \mathbf{t}_2$, $\mathbf{s}_3 = \mathbf{t}_3$, $\mathbf{s}_4 = \mathbf{t}_4$ and $\mathbf{s}_{l+1} = \mathbf{t}_{l+1}$ are *sets* of equations, and these are precisely the subsets of $\mathbf{s} = \mathbf{t}$ whose content can be processed by a single matching.

PROOF. We proceed as in the proof of Lemma 5.9: we'll find some substitutions $\theta_1, \ldots, \theta_l$ such that, for $i \in [1, l+1]$, θ_i is a match of $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ (here, for the sake of precision, for $i \in \{1, 2, 3, 4, l+1\}$, we should have used bold letters, and written $(\mathbf{s_i} = \mathbf{t_i})$). We have to consider seven distinct cases.

In $\mathbf{s}_1 = \mathbf{t}_1$, since H is input safe, each term in $\mathbf{t}_1$ is a generic expressions for the type of the positions it corresponds to; moreover, the terms in $\mathbf{t}_1$ are pairwise disjoint. Since A is correctly typed, from the Matching 2 Lemma 5.6 it follows that $\mathbf{s}_1 = \mathbf{t}_1$ is solvable by matching. Let θ_1 be a match of $\mathbf{s}_1 = \mathbf{t}_1$.

In $(\mathbf{s}_2 = \mathbf{t}_2)\theta_1$, since A is correctly typed, the terms in $\mathbf{s}_2$ are all ground. By the Matching 1 Lemma 4.1 $(\mathbf{s}_2 = \mathbf{t}_2)\theta_1$ is then solvable by matching. Let θ_2 be a match of it, and notice that $\mathbf{t}_2\theta_1\theta_2$ is a set of ground terms.

In $(\mathbf{s}_3 = \mathbf{t}_3)\theta_1\theta_2$, because of the way $\mathbf{s}_3 = \mathbf{t}_3$ was defined, we have that $Var(\mathbf{t}_3) \subseteq Var(\mathbf{t}_2)$, therefore $\mathbf{t}_3\theta_1\theta_2$ is a set of ground terms. Again, by the Matching 1 Lemma 4.1 $(\mathbf{s}_3 = \mathbf{t}_3)\theta_1\theta_2$ is then solvable by matching. Let θ_3 be a match of it.

In $(\mathbf{s}_4 = \mathbf{t}_4)\theta_1\theta_2\theta_3$, by the way $\mathbf{s}_4 = \mathbf{t}_4$ was defined, $\mathbf{t}_4$ consists of distinct variables, moreover $Var(\mathbf{t}_4) \cap Var(\mathbf{t}_1, \ldots, \mathbf{t}_3) = \emptyset$. By the relevance of $\theta_1, \theta_2, \theta_3$ (a match is always a *relevant* mgu) we then have that $\mathbf{t}_4\theta_1\theta_2\theta_3$ is a set of distinct variables. Again, by the Matching 1 Lemma 4.1 $(\mathbf{s}_4 = \mathbf{t}_4)\theta_1\theta_2\theta_3$ is then solvable by matching. Let θ_4 be a match of it.

The equations $(s_5 = t_5, \ldots, s_k = t_k, s_{k+1} = t_{k+1}, \ldots, s_l = t_l)\theta_1 \ldots \theta_4$ are then solvable (one at a time) by sequential matching. This follows at once from the proof of the Sequential Matching 2 Lemma 5.9. In particular we have that: for $i \in [5, k]$, since H is U-safe, $t_i\theta_1 \ldots \theta_{i-1}$ is a variable or a pure term, while for $i \in [k+1, l]$, since A is correctly typed and output independent, $s_i\theta_1 \ldots \theta_{i-1}$ is a variable or a pure term; here we (inductively) assume that for $i \in [5, l]$, θ_i is a match of $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$.

Finally, in $(\mathbf{s}_{l+1} = \mathbf{t}_{l+1})\theta_1 \ldots \theta_l$, since A is correctly typed and output independent, from the relevance of $\theta_1, \ldots, \theta_l$ it follows that the terms in $\mathbf{s}_{l+1}\theta_1 \ldots \theta_l$ are all distinct variables. Therefore, by the Matching 1 Lemma 4.1, $(\mathbf{s}_{l+1} = \mathbf{t}_{l+1})\theta_1 \ldots \theta_l$ is solvable by matching. This proves the Lemma. □

In practice, Lemma 10.1 states that we can solve by a single matching each of the following groups of positions:

- the *nonground* input positions.
- the *ground* input positions.
- the U-positions with respect to which H satisfies condition (ii) of U-safeness (Definition 5.8).
- those of the remaining U-positions of H which are filled in by a variable.
- the positions typed Var.

While the remaining positions should be processed one by one. These are

- the remaining U-positions.
- the position of type Pt.

The following Example shows that these last positions actually need to be processed one at a time.

EXAMPLE 10.2

(i) Consider $A = p(x, f(x,x))$ and $H = p(g(y), f(z,w))$, together with the typing $p : U \times U$. We have that A is correctly typed and that H is U-safe. Since here there are no input nor output positions, it follows that the hypothesis of the Sequential matching 2 Lemma 5.9 are satisfied, therefore $A = H$ is solvable by sequential matching. However $A = H$ is not solvable by matching, as there is no θ such that $A\theta = H$ or $A = H\theta$. This shows that the U positions of H which are filled in by pure terms and for which H satisfies condition (i) of U-safeness (Definition 5.8) need to be processed one at a time.

(ii) A perfectly symmetric reasoning applies for the positions typed Pt: consider $A = p(y, f(z,w))$ and $H = p(x,x)$, together with the typing $p : Pt \times Pt$. A is correctly typed and output independent, and since there are no input and U-positions, this is sufficient to satisfy the hypothesis of the Sequential 2 Lemma 5.9. Therefore $A = H$ is solvable by sequential matching, but not by a simple matching. As before, this is confirmed by the fact that there is no θ such that $A\theta = H$ or $A = H\theta$. □

Lemma 10.1 is an improved version of the Sequential Matching 2 Lemma 5.9, which in turn was the crucial step of Theorem 5.18. Therefore, its basic implication is that, when A and H are respectively the selected atom and the head of the input clause used to resolve it, then some positions of $A = H$ can be grouped in the same match (while others may not).

For this reason, in some situations, we might find convenient to adopt a typing which is more restrictive than another one, but which allows us to prove that we can solve the equations in the LD-derivations with a smaller number of matchings.

Consider for instance once again the program append, suppose that we want to use it for splitting a ground list in two. We might then want to adopt the following typing:

$$\mathcal{T}_1 = \texttt{app} : Pt \times U \times Ground$$

Here the (only) input position in the third one. From Theorem 6.6 it follows that, if t is a ground list, r is in Pt, then, for any term s disjoint from s, append ∪ { app(r, s, t)} is unification free.

However, if the kind of queries we are interested in are the ones in which the first two positions of append are filled in by variables (and this is a common situation), then we might find convenient to use the following typing:

$$\mathcal{T}_2 = \texttt{app} : Var \times Var \times Ground$$

Of course $\mathcal{T}_2$ is more restrictive than $\mathcal{T}_1$: every query that is correctly typed wrt $\mathcal{T}_2$ is also correctly typed wrt $\mathcal{T}_1$ (and not vice-versa). However, when we adopt $\mathcal{T}_1$, the best that we can prove is that all the equations considered in the LD-derivations of append ∪ { app(r, s, t)} are solvable by *triple* matching: first we match the rightmost position, then we match the middle one, and finally we match the leftmost one. On the other hand, if we adopt $\mathcal{T}_2$, from Lemma 10.1 it follows that all the equations considered in the LD-derivations of append ∪ { app(r, s, t)} are solvable by *double* (rather than triple) matching: first we match the rightmost position, then with a single match we can take care of the first two ones. Of course this holds provided that the queries satisfy the conditions of Theorem 5.18 wrt the adopted typing, and that is when they are correctly typed and output independent.

Finally, as a further example consider again the program select, which is reported in Example 6.7. As we mentioned in the discussion after Example 6.7, a query select(s, t, u) can be used in two main ways: to delete the element s from the list t and report the result in u, or as a generalized member program, to report in s an element of t, and in u the remains of the list. For both cases we can use the typing

$$\mathcal{T}_1 = \texttt{select} : U \times Ground \times Pt$$

When we use this typing, (assuming that the query satisfies the hypothesis of Theorem 5.18), from Lemma 10.1 it follows that all the equations considered in the LD derivations of `select` $\cup$ { `select(s, t, u)` } are solvable by triple matching.

However, when `select` is used in the first of the ways outlined above, then the first two arguments of the query are possibly ground terms. This allows us to use the typing

$$\mathcal{T}_2 = \texttt{select} : \mathit{Ground} \times \mathit{Ground} \times \mathit{Pt}$$

in this case, by Lemma 10.1, the equations considered in LD-derivations of `select` $\cup$ { `select(s, t, u)` } are solvable by *double* matching: first we match simultaneously the first two positions, then we match the third one.

A similar reasoning applies when we want to use select only as a generalized member program: we can reduce the number of matching needed in the LD-derivations by restricting the range of allowed queries, in particular by adopting the following typing:

$$\mathcal{T}_3 = \texttt{select} : \mathit{Var} \times \mathit{Ground} \times \mathit{Var}$$

In this case, from Lemma 10.1 it follows that the equations considered in the LD-derivations are again solvable by double matching, but this time we (obviously) match first the second position (the input one) and then, simultaneously, the first and third one (again, here we naturally assume that the queries satisfy the conditions of Theorem 5.18).

REFERENCES

[AE93] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, Lecture Notes in Computer Science, pages 1–19, Berlin, 1993. Springer-Verlag.

[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.

[AL93] A. Aiken and T.K. Lakshman. Type checking directionally typed logic programs. Technical report, Department of Computer Science, University of Illinois at Urbana Champaign, november 1993.

[AM94] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 1994. In print. Also Technical report CS-R9358, CWI, Amsterdam, The Netherlands. Available via anonymous ftp at ftp.cwi.nl, or via xmosaic at http://www.cwi.nl/cwi/publications/index.html.

[AP92] K. R. Apt and A. Pellegrini. Why the occur-check is not a problem. In M. Bruynooghe and M. Wirsing, editors, *Proceedings 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. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[Apt94] K. R. Apt. Program verification and prolog. In E. Børgher, editor, *Specification and validation methods for programming languages and systems*. Oxford University Press, 1994. to appear.

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. In J. Diaz and F. Orejas, editors, *TAPSOFT '89, Barcelona, Spain, March 1989, (Lecture Notes in Computer Science, vol. 352)*, pages 96–110. Springer-Verlag, 1989.

[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, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 321–335. MIT Press, 1992.

[CC88] H. Coelho and J.C. Cotta. *Prolog by Example.* Springer-Verlag, Berlin, 1988.

[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.

[DKM84] C. Dwork, P.C. Kanellakis, and J.C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[DKS86] C. Dwork, P. Kannellakis, and L. Stockmeyer. Parallel algorithms for term matching. In J. H. Siekmann, editor, *Proc. Eighth International Conference on Automated Deduction*, Lecture Notes in Computer Science 230, pages 416–430. Springer-Verlag, 1986.

[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, 1987. Second edition.

[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.

[Mar94] M. Marchiori. Localizations of unification freedom through matching directions. In M. Bruynooghe, editor, *Proc. Eleventh International Logic Programming Symposium.* MIT Press, 1994.

[Mel81] C. S. Mellish. The Automatic Generation of Mode Declarations for Prolog Programs. DAI Research Paper 163, Department of Artificial Intelligence, Univ. of Edinburgh, August 1981.

[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.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[Red84] U. S. Reddy. Transformation of logic programs into functional programs. In *International Symposium on Logic Programming*, pages 187–198, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[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.

[RS94] M. R. K. Krishna Rao and R. K. Shyamasundar. Unification-free execution of well-moded prolog programs. Technical report, TIFR, Bombay, India, March 1994.

[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.