Verification of Logic Programs with Delay Declarations

Krzysztof R. Apt[1,2] and Ingrid Luitjes[2]

[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[2] Department of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Abstract. Logic programs augmented with delay declarations form a higly expressive programming language in which dynamic networks of processes that communicate asynchronously by means of multiparty channels can be easily created. In this paper we study correctness these programs. In particular, we propose proof methods allowing us to deal with occur check freedom, absence of deadlock, absence of errors in presence of arithmetic relations, and termination. These methods turn out to be simple modifications of the corresponding methods dealing with Prolog programs. This allows us to derive correct delay declarations by analyzing Prolog programs. Finally, we point out difficulties concerning proofs of termination.

Note. The research of the first author was partly supported by the ESPRIT Basic Research Action 6810 (Compulog 2).

1 Introduction

In Kowalski [Kow79] the slogan "Algorithm = Logic + Control" was coined. This paper suggested logic programming as a formalism for a systematic development of algorithms. The idea was to endow a logic program with a control mechanism to obtain an executable program. Prolog is a realization of this idea where the control consists of the leftmost selection rule combined with the depth-first search in the resulting search tree.

But this idea of a control is in many cases overly restrictive. As an extreme example consider a theorem prover written in Prolog. A proof rule

$$\frac{A_1, \ldots, A_n}{B}$$

naturally translates into a Prolog clause

$$\texttt{prove(B)} \leftarrow \texttt{prove(A}_1\texttt{)}, \ldots, \texttt{prove(A}_\texttt{n}\texttt{)}$$

according to which the premises $A_1, \ldots, A_n$ have to be proved in the abovementioned order. In contrast, in the underlying logic the order in which the premises are to be proved is usually arbitrary.

In general, however, some order between the actions of a logic program is necessary. As an example consider the `QUICKSORT` program:

```
qs(Xs, Ys) ← Ys is an ordered permutation of the list Xs.
qs([], []).
qs([X | Xs], Ys) ←
   part(X, Xs, Littles, Bigs),
   qs(Littles, Ls),
   qs(Bigs, Bs),
   app(Ls, [X | Bs], Ys).

part(X, Xs, Ls, Bs) ← Ls is a list of elements of Xs which are < X,
                      Bs is a list of elements of Xs which are ≥ X.
part(_, [], [], []).
part(X, [Y | Xs], [Y | Ls], Bs) ← X > Y, part(X, Xs, Ls, Bs).
part(X, [Y | Xs], Ls, [Y | Bs]) ← X ≤ Y, part(X, Xs, Ls, Bs).
```

augmented by the following APPEND program:

```
app(Xs, Ys, Zs) ← Zs is the result of concatenating the lists Xs and Ys.
app([], Ys, Ys).
app([X | Xs], Ys, [X | Zs]) ← app(Xs, Ys, Zs).
```

It is easy to see that starting with a query qs(s, Ys), where s is a list of integers, QUICKSORT diverges when the rightmost atom is repeatedly selected. Also, a run time error results if the arithmetic atoms $\texttt{X} > \texttt{Y}$ and $\texttt{X} \leq \texttt{Y}$ are selected "too early". To prevent this type of undesired behaviour some coordination between the actions of the program is necessary.

For this purpose in Naish [Nai82] delay declarations were proposed. The idea is to replace the Prolog selection rule by a more flexible selection mechanism according to which atoms are delayed until they become "sufficiently" instantiated. This is achieved by adding to the program so-called delay declarations. Then at each stage of the execution of a logic program only atoms satisfying the delay declarations can be selected. In presence of trivial delay declarations any atom can be selected and a nondeterministic logic program without any control declaration is then obtained.

More recently, the delay declarations were studied in Lüttringhaus-Kappel [LK93] and also incorporated in various versions of Prolog, notably Sicstus Prolog, and in Gödel, the programming language proposed by Hill and Lloyd [HL94]. Actually, in all these references a more restricted selection rule is employed, according to which the leftmost non-delayed atom is selected. This selection rule allows us to model within the logic programming the coroutine mechanism and lazy evaluation. In contrast, the selection rules here studied. and originally considered in Naish [Nai88], allow us to model parallel executions.

Returning to the above QUICKSORT program we note that the coordination between the program actions has to do with the need to produce the appropriate "inputs" before executing the "calls" which use them. The requirement to compute these inputs can be expressed by means of the following delay declarations, where the last two declarations are meant to ensure that the arithmetic relations are called with the right inputs:

```
DELAY qs(X, _) UNTIL nonvar(X).
DELAY part(_, Y, _, _) UNTIL nonvar(Y).
DELAY app(X, _, _) UNTIL nonvar(X).
DELAY X > Y UNTIL ground(X) ∧ ground(Y).
DELAY X ≤ Y UNTIL ground(X) ∧ ground(Y).
```

The behaviour of the resulting program is highly non-trivial, since during its executions dynamic networks of asynchronously communicating processes are created.

Delay declarations form a powerful control mechanism. In general, we can identify three natural uses of them:

- to enforce termination,
- to prevent absence of errors in presence of arithmetic operations,
- to impose a synchronization between various actions of the program; this makes it possible to model parallel executions.

In this paper we illustrate each of these uses of delay declarations and provide formal means of justifying them. More specifically, we study here various correctness aspects of logic programs in presence of delay declarations, visibly occur check freedom, absence of deadlock, absence of errors in presence of arithmetic operations, and termination. In each case we propose a simple method which can be readily applied to several well-known programs.

These results imply that for the query `qs(s, Ys)`, where `s` is a list of integers, `QUICKSORT` augmented by the above delay declarations is occur-check free and deadlock free. Moreover, no errors due to the presence of arithmetic operations arise and under some additional assumptions all derivations terminate.

Interestingly, the suggested proof methods turn out to be simple modifications of the corresponding methods dealing with Prolog programs. So the transition from Prolog to programs with delay declarations is quite natural, even though the latter ones permit more execution sequences and more complex "intermediate situations". This observation is further substantiated by showing how "correct" delay declarations can be derived by analyzing Prolog programs so that the given Prolog program can be executed in a more flexible way.

2 Preliminaries

In what follows we use the standard notation of Lloyd [Llo87] and Apt [Apt90], though 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. In particular, given a syntactic construct E (so for example, a term, an atom or a set of term equations) we denote by $Var(E)$ the set of the variables appearing in E. Recall that an mgu θ of a set of term equations E is called *relevant* if $Var(\theta) \subseteq Var(E)$.

The following lemma (see e.g. Apt and Pellegrini [AP94]) will be needed in Section 6.

Lemma 1 (Iteration). *Let E_1, E_2 be two sets of term 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 a relevant mgu θ_1 of E_1 exists, and for any mgu θ_1 of E_1 a relevant mgu θ_2 of $E_2\theta_1$ exists, as well.* □

We now define the syntax of delay declarations. We loosely follow here Hill and Lloyd [HL94]. First, we define inductively a set of *conditions*:

- **true** is a condition,
- given a variable `X`, `ground(X)` and `nonvar(X)` are conditions,
- if c_1 and c_2 are conditions, then $c_1 \wedge c_2$ and $c_1 \vee c_2$ are conditions.

Next, we define inductively when an instance of a condition *holds*:

- **true** holds,
- `ground(s)` holds if `s` is a ground term,
- `nonvar(s)` holds if `s` is a non-variable term,
- $c_1 \wedge c_2$ holds if c_1 and c_2 hold,
- $c_1 \vee c_2$ holds if c_1 or c_2 holds.

Call an atom a *p-atom* if its relation symbol is p. A delay declaration associated for a relation symbol p has the form `DELAY A UNTIL COND`, where `A` is a p-atom and `COND` is a condition. From now on we consider logic programs augmented by the delay declarations, one for each of its relation symbols. In the presentation below we drop the delay declarations of the form `DELAY A UNTIL` **true**.

The following simple definition explains the use of delay declarations.

Definition 2.

- We say that an atom `B` *satisfies a delay declaration* `DELAY A UNTIL COND` if for some substitution θ both $\mathtt{B} = \mathtt{A}\theta$ and $\mathtt{COND}\theta$ hold.
  In particular, if $\mathtt{X_1}, \ldots, \mathtt{X_n}$ are different variables, then $\mathtt{p}(\mathtt{s_1}, \ldots, \mathtt{s_n})$ satisfies a delay declaration $\mathtt{DELAY\ p}(\mathtt{X_1}, \ldots, \mathtt{X_n})\ \mathtt{UNTIL\ COND}$ if $\mathtt{COND}\{\mathtt{X_1}/\mathtt{s_1}, \ldots, \mathtt{X_n}/\mathtt{s_n}\}$ holds.
- An SLD-derivation *respects the delay declarations* if all atoms selected in it satisfy their delay declarations. □

Intuitively, in presence of delay declarations only atoms which satisfy their delay declarations can be selected. So in presence of delay declarations we allow only those selection rules which generate SLD-derivations respecting the delay declarations. Note that in presence of delay declarations a query can be generated in which no atom can be selected because none of them satisfies its delay declaration. We view such a fragment of an SLD-derivation as a finite SLD-derivation.

In what follows we shall study correctness of logic programs augmented with the delay declarations. To show the usefulness of the obtained results and to see their limitations, we shall analyze in this paper three example programs: QUICKSORT from the introduction, and the following two.

The program IN_ORDER constructs the list of all nodes of a binary tree by means of an in-order traversal:

```
in_order(Tree, List) ← List is a list obtained by the in-order
                       traversal of the tree Tree.

in_order(void, []).
in_order(tree(X, Left, Right), Xs) ←
  in_order(Left, Ls),
  in_order(Right, Rs),
  app(Ls, [X | Rs], Xs).
```

augmented by the APPEND program.

together with the following delay declarations:

```
DELAY in_order(X, _) UNTIL nonvar(X).
DELAY app(X, _, _) UNTIL nonvar(X).
```

Finally, the program SEQUENCE (see Coelho and Cotta [CC88, page 193]) solves the following problem: 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) ← app(_, Zs, Ys), app(Xs, _, Zs).

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.

together with the following the delay declaration:

```
DELAY app(_, _, Z) UNTIL nonvar(Z).
```

3 Occur-check Freedom

In most Prolog implementations for the efficiency reasons so-called occur-check is omitted from the unification algorithm. It is well-known that this omission can lead to incorrect results. The resulting difficulties are usually called the *occur-check problem*. They have motivated a study of conditions under which the occur-check can be safely omitted. In this section we study this problem for logic programs augmented with delay declarations. To this end we recall the relevant definitions. We follow here Apt and Pellegrini [AP94] though we adapt its framework to arbitrary SLD-derivations. The following notion is due to Deransart, Ferrand and Téguia [DFT91].

Definition 3. A system of term equations E is called *not subject to occur-check* (NSTO in short) if no execution of the Martelli-Montanari unification algorithm (see Martelli and Montanari [MM82]) started with E ends with a system that includes an equation of the form $x = t$, where x is a variable and t a term different from x, but in which x occurs. □

We can now introduce the crucial notion of this section.

Definition 4.

- Consider an SLD-derivation ξ. 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.
- An SLD-derivation is called *occur-check free* if all the systems of equations considered in it are NSTO. □

Recall that in presence of delay declarations selection of an atom implies that it satisfies its delay declaration.

In what follows we identify some syntactic conditions that allow us to draw conclusions about the occur-check freedom. To this end we use modes. Informally, modes indicate how the arguments of a relation should be used.

Definition 5. Consider an n-ary relation symbol p. By a *mode* for p we mean a function m_p from $\{1, \ldots, n\}$ to the set $\{+, -\}$. If $m_p(i) =$ '$+$', we call i an *input position* of p and if $m_p(i) =$ '$-$', we call i an *output position* of p (both w.r.t. m_p). By a *moding* we mean a collection of modes, each for a different relation symbol. □

We write m_p in a more suggestive form $p(m_p(1), \ldots, m_p(n))$. For example, `member(-,+)` denotes a binary relation `member` with the first position moded as output and the second position moded as input.

The definition of moding assumes one mode per relation in a program. Multiple modes may be obtained by simply renaming the relations. In this paper we adopt the following

Assumption *Every considered relation* has a fixed mode associated with it.

This assumption will allow us to talk about input positions and output positions of an atom.

Definition 6.

- A family of terms is called *linear* if every variable occurs at most once in it.
- An atom is called *input* (resp. *output*) *linear* if the family of terms occurring in its input (resp. output) positions is linear.
- An atom is called *input-output disjoint* if the family of terms occurring in its input positions has no variable in common with the family of terms occurring in its output positions. □

In the sequel we shall use the following lemma of Apt and Pellegrini [AP94].

Lemma 7. *Consider two atoms A and H with the same relation symbol. Suppose that*

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

Then $A = H$ is NSTO. □

The first result is an immediate consequence of Lemma 7. The idea is that when the head of every clause of P is output linear, it suffices to delay all the atoms until their input positions become ground. To this end we introduce the following notion that relates the delay declarations to modes.

Definition 8. We say that the delay declarations *imply the moding* if every atom which satisfies its delay declaration is ground in its input positions. □

Theorem 9 (Occur-check Freedom 1). *Suppose that*

- *the head of every clause of P is output linear,*
- *the delay declarations imply the moding.*

Then all SLD-derivations of $P \cup \{Q\}$ which respect the delay declarations are occur-check free.

Proof. By Lemma 7. □

This result allows us to deal only with the delay "until ground" declarations. So for example, we cannot draw yet at this stage any conclusions concerning the `QUICKSORT` program from the introduction. To deal with other forms of delay declarations we use the following notion introduced in Chadha and Plaisted [CP94] and further studied in Apt and Pellegrini [AP94].

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

Definition 10.

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

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

– A clause

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

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

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

In particular, every unit clause is nicely-moded.

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

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". Note that a query with only one atom is nicely moded iff it is output linear and input-output disjoint.

The following lemma is crucial. It shows persistence of the notion of nice modedness in presence of a natural assumption.

Lemma 11 (Nice modedness). *Every SLD-resolvent of a nicely moded query and a nicely moded clause with an input-linear head, that is variable-disjoint with it, is nicely moded.*

Proof. The proof is quite long and can be found in Luitjes [Lui94]. It is similar to the proof of an analogous lemma, 5.3, in Apt and Pellegrini [AP94, pages 719-724]. □

Corollary 12 (Nice modedness). *Let P and Q be nicely moded. Suppose that the head of every clause of P is input linear. Then all queries in all SLD-derivations of $P \cup \{Q\}$ are nicely moded.* □

This corollary brings us to the following conclusion.

Theorem 13 (Occur-check Freedom 2). *Let P and Q be nicely moded. Suppose that*

– *the head of every clause of P is input linear.*

Then all SLD-derivations of $P \cup \{Q\}$ are occur-check free.

Proof. By the Nice modedness Corollary 12 all queries in all SLD-derivations of $P \cup \{Q\}$ are nicely moded. But every atom of a nicely moded query is input-output disjoint and output linear. So the claim follows by Lemma 7. □

This result shows that for nicely moded programs and queries occur-check freedom can be ensured inpendendently of the selection rule, so without taking into account the delay declarations. In Chadha and Plaisted [CP94] and Apt and Pellegrini [AP94] it was shown that the `QUICKSORT` program with the moding `qs(+,-)`, `partition(+,+,-,-)`, `app(+,+,-)`, `+>+`, `+≤+` satisfies the condition of the above theorem, so this result applies to any query of the form `qs(s, Ys)`, where `s` is a list of integers.

To apply the above result to the `IN_ORDER` program consider the following moding: `in_order(+,-)`, `app(+,+,-)`. It is straightforward to check that then `IN_ORDER` is nicely moded and the head of every clause of `IN_ORDER` is input linear.. So by the Occur-check Freedom 2 Theorem 13 we conclude that for any term `t` and a variable `Ys` that does not occur in `t`, all SLD-derivations of `IN_ORDER` $\cup$`{in_order(t, Ys)}` are occur-check free.

Finally, to deal with the `SEQUENCE` program take the following moding: `sublist(-,+)`, `sequence(+)`, `question(+)`, `app(-,-,+)`. Thanks to the use of anonymous variables it is easy to check that `SEQUENCE` is then indeed nicely moded and that the heads of all clauses are input linear. So by the Occur-check Freedom 2 Theorem 13 all SLD-derivations of `SEQUENCE` $\cup$`{question(Ss)}` are occur-check free.

4 Absence of Deadlock

In presence of delay declarations a query can be generated in which no atom can be selected, because none of them satisfies its delay declaration. Then the computation cannot proceed. Such a situation is obviously undesirable. We call it a deadlock and study here means to avoid it. Let us begin with a formal definition.

Definition 14. An SLD-derivation *flounders* if it contains a query no atom of which satisfies its delay declaration. We say that $P \cup \{Q\}$ *deadlocks* if an SLD-derivation of $P \cup \{Q\}$ which respects the delay declarations flounders. □

We now propose syntactic conditions which allow us to prove absence of deadlock. The main tool is the notion of a well-moded program and query. Let us recall the definition (see e.g. Dembinski and Maluszynski [DM85] and Drabent [Dra87]).

4.1 Well-moded Queries and Programs

Definition 15.

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

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

– A clause

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

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

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

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

The following lemma shows the persistence of the notion of well-modedness. It strengthens the version given in Apt and Pellegrini [AP94] to arbitrary SLD-resolvents.

Lemma 16 (Well-modedness). *Every SLD-resolvent of a well-moded query and a well-moded clause, that is variable-disjoint with it, is well-moded.*

Proof. An SLD-resolvent of a query and a clause is obtained by means of the following three operations:

– instantiation of a query,
– instantiation of a clause,
– replacement of an atom, say H, of a query by the body of a clause whose head is H.

So we only need to prove the following two claims.

Claim 1 *An instance of a well-moded query (resp. clause) is well-moded.*

Proof. It suffices to note that for any sequences of terms $\mathbf{s}, \mathbf{t}_1, \ldots, \mathbf{t}_n$ and a substitution θ, $Var(\mathbf{s}) \subseteq \bigcup_{j=1}^{n} Var(\mathbf{t}_j)$ implies $Var(\mathbf{s}\theta) \subseteq \bigcup_{j=1}^{n} Var(\mathbf{t}_j\theta)$. □

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

Proof. Let
$\mathbf{A} := p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_k(\mathbf{s}_k, \mathbf{t}_k),$
$H := p(\mathbf{s}, \mathbf{t}),$
$\mathbf{B} := p_{k+1}(\mathbf{s}_{k+1}, \mathbf{t}_{k+1}), \ldots, p_{k+l}(\mathbf{s}_{k+l}, \mathbf{t}_{k+l}),$
$\mathbf{C} := p_{k+l+1}(\mathbf{s}_{k+l+1}, \mathbf{t}_{k+l+1}), \ldots, p_n(\mathbf{s}_{k+l+m}, \mathbf{t}_{k+l+m}).$

Fix now $i \in [1, k+l+m]$. We need to prove that $Var(\mathbf{s}_i) \subseteq \bigcup_{j=1}^{i-1} Var(\mathbf{t}_j)$.
Case 1 $i \in [1, k]$.
Note that $\mathbf{A}$ is well-moded, since $\mathbf{A}, H, \mathbf{C}$ is well-moded. Hence the claim follows.
Case 2 $i \in [k+1, k+l]$.
$H \leftarrow \mathbf{B}$ is well-moded, so $Var(\mathbf{s_i}) \subseteq Var(\mathbf{s}) \cup \bigcup_{j=k+1}^{i-1} Var(\mathbf{t}_j)$. Moreover, $\mathbf{A}, H, \mathbf{C}$ is well-moded, so $Var(\mathbf{s}) \subseteq \bigcup_{j=1}^{k} Var(\mathbf{t}_j)$. This implies the claim.

Case 3 $i \in [k+l+1, k+l+m]$.
$\mathbf{A}, H, \mathbf{C}$ is well-moded, so $Var(\mathbf{s}_i) \subseteq \bigcup_{j=1}^{k} Var(\mathbf{t}_j) \cup Var(\mathbf{t}) \cup \bigcup_{j=k+l+1}^{i-1} Var(\mathbf{t}_j)$ and $Var(\mathbf{s}) \subseteq \bigcup_{j=1}^{k} Var(\mathbf{t}_j)$. Moreover, $H \leftarrow \mathbf{B}$ is well-moded, so

$$Var(\mathbf{t}) \subseteq Var(\mathbf{s}) \cup \bigcup_{j=k+1}^{k+l} Var(\mathbf{t}_j).$$

This implies the claim. □
□

Corollary 17 (Well-modedness). *Let P and Q be well-moded. Then all queries in all SLD-derivations of $P \cup \{Q\}$ are well-moded.* □

The following definition provides a link between the delay declarations and moding.

Definition 18. We say that the delay declarations *are implied by the moding* if every atom which is ground in its input positions satisfies its delay declaration. □

We can now state and prove the desired result.

Theorem 19 (Absence of Deadlock 1). *Let P and Q be well-moded. Suppose that the delay declarations are implied by the moding. Then $P \cup \{Q\}$ does not deadlock.*

Proof. By the Well-modedness Corollary 17 all queries in all SLD-derivations of $P \cup \{Q\}$ are well-moded. But the first atom of a well-moded query is ground in its input positions. Hence, by the assumption, in every SLD-derivation of $P \cup \{Q\}$ the first atom of every query satisfies its delay declaration. Consequently, no SLD-derivation of $P \cup \{Q\}$ flounders. □

Intuitively, the above result states that under the abovementioned conditions, at every stage of a computation the first atom can always be selected.

The above result can be applied to the `QUICKSORT` program. Indeed, with the moding considered in Section 3 `QUICKSORT` is easily seen to be well-moded, the query `qs(s, Ys)`, where `s` is a list of integers, is well-moded and the delay declarations considered in the introduction are clearly implied by this moding. In fact, the same reasoning applies when the original delay declarations are strengthened by replacing everywhere "`nonvar`" by "`ground`".

The Absence of Deadlock 1 Theorem 19 can also be used for the `IN_ORDER` program. Indeed, in the moding considered in the previous section `IN_ORDER` is clearly well-moded. So the above theorem is applicable to any query of the form `in_order(t, Ys)`, where `t` is a ground term. However, no conclusion can be drawn if `t` is not ground. Below we shall see how to draw such stronger conclusions.

Finally, the above theorem cannot be applied to the program `SEQUENCE`. Indeed, it is easy to see that no moding exists for which both `SEQUENCE` and the query `question(Ss)` are well-moded. So the above theorem cannot be applied to the `SEQUENCE` program *no matter* what delay declarations are used.

4.2 Well-typed Queries and Programs

To overcome these difficulties we generalize the above approach by using the notion of a type. The presentation below of well-typed queries and programs is taken from Apt and Etalle [AE93]. We begin by adopting the following general definition.

Definition 20. A *type* is an non-empty set of terms closed under substitution. □

We now fix a specific set of types, denoted by *Types*, which includes:

U — the set of all terms,
List — the set of lists,
Gae — the set of of all ground arithmetic expressions (gae's, in short),
ListGae — the set of lists of gae's.
Tree — the set of binary trees, defined inductively as follows: `void` is a tree and if `s, t` are trees, then for any term `u`, `tree(u,s,t)` is a tree.

Of course, the use of the type *List* assumes the existence of the empty list `[]` and the list constructor `[.|.]` in the language, etc.

We call a construct of the form $s : S$, where s is a term and S is a type, a *typed term*. Given a sequence $\mathbf{s} : \mathbf{S} = s_1 : S_1, \ldots, s_n : S_n$ of typed terms, we write $\mathbf{s} \in \mathbf{S}$ if for $i \in [1, n]$ we have $s_i \in S_i$. Further, we abbreviate the sequence $s_1\theta, \ldots, s_n\theta$ to $\mathbf{s}\theta$. Finally, we write

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

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

Next, we define types for relations.

Definition 21. By a *type* for an n-ary relation symbol$p$ we mean a function t_p from $[1, n]$ to the set *Types*. If $t_p(i) = T$, we call T *the type associated with the position i of p.* □

In the remainder of this paper we consider a combination of modes and types and adopt the following

Assumption *Every considered relation* has a fixed mode and a fixed type associated with it.

This assumption will allow us to talk about types of input positions and of output positions of an atom. An n-ary relation p with a mode m_p and type t_p will be denoted by

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

For example, $\texttt{member}(- : U,\ + : List)$ denotes a binary relation `member` with the first position moded as output and typed as U, and the second position moded as input and typed as $List$.

To simplify the notation, when writing an atom as $p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$ we now assume that $\mathbf{u} : \mathbf{S}$ is a sequence of typed terms filling in the input positions of p and $\mathbf{v} : \mathbf{T}$ is a sequence of typed terms filling in the output positions of p. We call a construct of the form $p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$ a *typed atom* and a sequence of typed atoms a *typed query*. We say that a typed atom $p(s_1 : S_1, \ldots, s_n : S_n)$ is *correctly typed in position i* if $s_i \in S_i$ and use an analogous terminology for typed queries.

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

Definition 22.

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

Note that a query with only one atom is well-typed iff this atom is correctly typed in its input positions. The following lemma shows persistence of the notion of well-typedness. It strenghtens a result mentioned in Bronsard, Lakshman and Reddy [BLR92] to arbitrary SLD-resolvents.

Lemma 23 (Well-typedness). *Every SLD-resolvent of a well-typed query and a well-typed clause, that is variable-disjoint with it, is well-typed.*

Proof. The proof is analogous to that of the Well-modedness Lemma 16 and is omitted. □

Corollary 24 (Well-typedness). *Let P and Q be well-typed. Then all queries in all SLD-derivations of $P \cup \{Q\}$ are well-typed.* □

Finally, we link the delay declarations with types.

Definition 25. We say that the delay declarations *are implied by the typing* if every atom which is correctly typed in its input positions satisfies its delay declaration. □

We can now state and prove the desired result.

Theorem 26 (Absence of Deadlock 2). *Let P and Q be well-typed. Suppose that the delay declarations are implied by the typing. Then $P \cup \{Q\}$ does not deadlock.*

Proof. By the Well-typedness Corollary 24 all queries in all SLD-derivations of $P \cup \{Q\}$ are well-typed. But the first atom of a well-typed query is correctly typed in its input positions. Hence, by the assumption, in every SLD-derivation of $P \cup \{Q\}$ the first atom of every query satisfies its delay declaration. Consequently, no SLD-derivation of $P \cup \{Q\}$ flounders. $\square$

Let us see how to apply this theorem to the `IN_ORDER` program. Consider the following typing:
`in_order`$(+ : Tree, - : List)$,
`app`$(+ : List, + : List, + : List)$.

We leave to the reader the task of checking that `IN_ORDER` is then well-typed and that the delay declarations are implied by the typing. We conclude that for any term `t` which is a tree, `IN_ORDER` $\cup\{$`in_order(t, Ys)`$\}$ does not deadlock, which strengthens the conclusion drawn by means of the Absence of Deadlock 1 Theorem 19.

Finally, note that this theorem can also be applied to the `SEQUENCE` program. Indeed, consider the following typing:
`question`$(- : List)$,
`sequence`$(- : List)$,
`sublist`$(+ : List, + : List)$,
`app`$(- : List, - : List, + : List)$.

Again, it is easy to see that `SEQUENCE` and the query `question(Ss)` are then well-typed and that the delay declaration is implied by the typing. It is worthwhile to note that if we change in this declaration "`nonvar`" to "`ground`", then `SEQUENCE` $\cup\{$`question(Ss)`$\}$ does deadlock.

5 Absence of Errors

One of the natural uses of the delay declarations is to prevent run time errors in presence of arithmetic relations. This is for example the idea behind the delay declarations

```
DELAY X > Y UNTIL ground(X) ∧ ground(Y).
DELAY X ≤ Y UNTIL ground(X) ∧ ground(Y).
```

used in the introduction which ensure that both relations are called only with ground arguments. Now, to prove absence of errors a stronger property is needed, namely that the arguments of these relations are ground arithmetic expressions. However, the syntax of the delay declarations does not allow us to express this stronger information.

The aim of this section is to provide means to deduce this stronger property and thus to prove absence of errors in presence of arithmetic relations. To this end we shall use the notions of a well-typed query and a well-typed program introduced in the previous section.

The following simple observation provides us with some means to prove that if an atom is ground in its input positions, then it is correctly typed in its input positions.

Lemma 27. *Suppose that* $\mathbf{A}, B, \mathbf{C}$ *is a well-typed query such that*

- B *is ground in its input positions,*
- *for some substitution* θ*,* $\mathbf{A}\theta$ *is correctly typed in its output positions.*

Then B *is correctly typed in its input positions.*

Proof. Let $\mathbf{A} = p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$ and $B = p_{n+1}(\mathbf{i_{n+1}} : \mathbf{I_{n+1}}, \mathbf{o_{n+1}} : \mathbf{O_{n+1}})$. By the definition of well-typedness

$$\models \mathbf{o_1} : \mathbf{O_1}, \ldots, \mathbf{o_n} : \mathbf{O_n} \ \Rightarrow \ \mathbf{i_{n+1}} : \mathbf{I_{n+1}}.$$

But by the assumption $\mathbf{o_i}\theta \in \mathbf{O_i}$ for $i \in [1, n]$, so $\mathbf{i_{n+1}}\theta \in \mathbf{I_{n+1}}$ which implies the claim since by assumption $\mathbf{i_{n+1}}\theta = \mathbf{i_{n+1}}$. □

We need to apply this lemma at every stage of the SLD-derivations. To prove the second assumption we shall use the following notion introduced in Apt and Etalle [AE93].

Definition 28.

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

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

- A clause

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

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

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

  In particular, every unit clause is simply moded.
- A program is called *simply moded* if every clause of it is. □

So simple modedness is a special case of nice modedness. The sole difference lies in the new assumption that each output position of a query or of a body of a clause is filled by a variable. The following lemma clarifies our interest in the notion of simple modedness.

Lemma 29. *Let* $\mathbf{A}$ *be a typed query which is simply moded. Then for some substitution* θ*,* $\mathbf{A}\theta$ *is correctly typed in its output positions.*

Proof. By the definition the output positions of $\mathbf{A}$ are filled by different variables. Choose for each variable occurring in an output position of $\mathbf{A}$ a term from the type associated with this position. So obtained bindings form the desidered substitution. □

The next step is to prove persistence of the notion of simple modedness. In Apt and Etalle [AE93] this property is established for the SLD-derivations w.r.t. the leftmost selection rule. But the generalization to arbitrary SLD-derivations does not work, as the following example shows. Consider the moding $p(-), r(+)$ and let $P = \{p(a) \leftarrow \}$ and $Q = p(x), r(x)$. Then $p(a)$ is an SLD-resolvent of Q and $p(a)$ is not simply moded.

However, a simple additional condition does ensure persistence. Namely, we have the following lemma.

Lemma 30 (Simple modedness). *Every SLD-resolvent of a simply moded query and a simply moded clause, that is variable-disjoint with it, is simply moded, when the input part of selected atom is an instance of the input part of the head of the clause.*

Proof. Omitted. □

Corollary 31 (Simple modedness). *Let* P *and* Q *be simply moded. Consider an SLD-derivation* ξ *of* $P \cup \{Q\}$ *such that the input part of each selected atom is an instance of the input part of the head of the used clause. Then all queries in* ξ *are simply moded.* □

To use this result we now link the delay declarations with matching.

Definition 32. We say that the delay declarations *imply matching* if for every atom $p(\mathbf{u}, \mathbf{v})$ which satisfies its delay declaration and for every head $p(\mathbf{u}', \mathbf{v}')$ of a clause if $p(\mathbf{u}, \mathbf{v})$ and $p(\mathbf{u}', \mathbf{v}')$ unify, then $\mathbf{u}$ is an instance of $\mathbf{u}'$. □

In particular, if the delay declarations imply the moding, then they imply matching, but not conversely. This brings us to the following conclusion.

Theorem 33 (Correct Typing). *Suppose that*

- *P and Q are well-typed and simply moded,*
- *the delay declarations imply matching.*

Then in all SLD-derivations of $P \cup \{Q\}$ *which respect the delay declarations every selected atom which is ground in its input positions is correctly typed in its input positions.*

Proof. It is a direct consequence of the Lemmata 27, 29, the Well-typedness Corollary 24 and the Simple modedness Corollary 31. □

This result can be used to prove absence of errors in presence of arithmetic relations by using for each arithmetic relation p a delay declaration

```
DELAY p(X, Y) UNTIL ground(X) ∧ ground(Y)
```

and a typing $p(+ : Gae, + : Gae)$. In the case of `QUICKSORT` take the typing
$\texttt{qs}(+ : ListGae, - : ListGae)$,
$\texttt{part}(+ : Gae, + : ListGae, - : ListGae, - : ListGae)$,
$\texttt{app}(+ : ListGae, + : ListGae, - : ListGae)$
$> (+ : Gae, + : Gae)$,
$\leq (+ : Gae, + : Gae)$,
Then `QUICKSORT` is well-typed (see Apt [Apt93]). Also, it is clearly simply moded. Moreover, the delay declarations from the introduction imply matching. Indeed, if a non-variable term unifies with `[X | Xs]`, then it is an instance of `[X | Xs]`.

We conclude that for `s` a list of integers, all SLD-derivations of `QUICKSORT` $\cup\{$`qs(s, X)`$\}$ which respect the delay declarations from the introduction do not end in an error.

6 Termination

Finally, we study termination of logic programs with delay declarations. The key idea in our approach is the restriction to a specific class of SLD-derivations.

6.1 Termination via Determinacy

Definition 34. We say that an SLD-derivation is *determinate* if every selected atom unifies with a variant of at most one clause head, that is, every selected atom can be resolved using at most one clause. □

The following simple observation, which is of independent interest, forms the basis of our approach.

Lemma 35. *Suppose that*

– *an SLD-derivation of $P \cup \{Q\}$ is successful.*

Then all determinate SLD-derivations of $P \cup \{Q\}$ are successful, hence finite.

Proof. Consider a determinate SLD-derivation ξ of $P \cup \{Q\}$. ξ is a branch in an SLD-tree $\mathcal{T}$ for $P \cup \{Q\}$. By the strong completeness of the SLD-resolution $\mathcal{T}$ is successful and since ξ is determinate, $\mathcal{T}$ has just one branch, namely ξ. So ξ is successful, and a fortiori finite. □

To use this result, we link the delay declarations with the notion of determinacy.

Definition 36. We say that the delay declarations *imply determinacy* if every atom which satisfies its delay declaration unifies with a variant of at most one clause head. □

This brings us to the following result.

Theorem 37 (Termination 1). *Suppose that*

- *an SLD-derivation of $P \cup \{Q\}$ is successful,*
- *the delay declarations imply determinacy.*

Then all SLD-derivations of $P \cup \{Q\}$ which respect the delay declarations are successful, hence finite.

Proof. It is an immediate consequence of Lemma 35, because if the delay declarations imply determinacy, then every SLD-derivation which respects these delay declarations is determinate. □

Let us see now how to apply this result to the `IN_ORDER` program. Using the approach of Apt [Apt93] it is straightforward to prove that for a tree `t`, `IN_ORDER` $\cup\{$`in_order(t, Ys)`$\}$ satisfies the first condition of the above theorem. Also, the assumed delay declarations clearly imply determinacy. We conclude that all SLD-derivations of `IN_ORDER` $\cup\{$`in_order(t, Ys)`$\}$ which respect the delay declarations are finite.

However, the above theorem cannot be directly applied to the `QUICKSORT` program because the delay declarations from the introduction do not imply determinacy in the case of the `part` relation. On the other hand, it is possible to adjust these delay declarations and slightly modify the execution of the program so that for the query `qs(s, Ys)`, where `s` is a list of integers, termination can be established.

Namely, consider the following alternative *set* of delay declarations for the `part` relation, given in Naish [Nai88]:

```
DELAY part(_, [], _, _) UNTIL true.
DELAY part(X, [Y | _], _, _) UNTIL ground(X) ∧ ground(Y).
```

We now say that an atom *satisfies a set of delay declarations* if it satisfies at least one of them. The idea behind these new delay declarations is to enforce that the arguments of the arithmetic atoms `X` $>$ `Y` and `X` $\leq$ `Y` are ground once they are introduced through the selection of a `part`-atom. Note that now the delay declarations for the arithmetic relations become superfluous in the sense that they are always satisfied.

Also, observe that for this new set of delay declarations absence of deadlock is now ensured by the Absence of Deadlock 2 Theorem 26 and not anymore by the Absence of Deadlock 1 Theorem 19. Indeed, these new delay declarations are clearly implied by the typing given in Section 5 but are not any longer implied by the moding given in Section 3, as not all ground terms are of the form `[y|ys]`.

Further, note that this new set of delay declarations still implies matching. So the proof of the absence of errors in all SLD-derivations of `QUICKSORT` $\cup$\{`qs(s, X)`\} respecting the delay declarations and given in the previous section remains valid.

However, the determinacy is still not ensured. To deal with this problem we now modify the execution of the programs by viewing the arithmetic atoms as guards in the sense of e.g. Shapiro [Sha89].

6.2 Termination for Guarded Programs

By treating atoms as guards we mean the following generalization of the SLD-resolution, which we call the *SLDG-resolution*. By a *guarded clause* we refer to a construct of the form $H \leftarrow \mathbf{G} \mid \mathbf{B}$, where H is an atom and $\mathbf{G}$ and $\mathbf{B}$ are sequences of atoms. A *guarded program* is a set of guarded clauses. If $\mathbf{G}$ is empty, then we drop the vertical bar "|". The atoms in $\mathbf{G}$ are called *guards*. Note that we do not insist that a guarded program is a finite set of clauses. The reason is that we wish to have the possibility of defining the relations used in the guards by a possibly infinite set of ground facts. It is clear how to extend the notions of well-modedness etc. to guarded programs.

We now view `QUICKSORT` as a guarded program by rewriting the last two clauses defining the `part` relation as follows:

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

We call the resulting program `QUICKSORT-G`. We assume that both in `QUICKSORT` and in `QUICKSORT-G` the arithmetic relations are defined by the infinite set of ground facts.

Consider now a query $\mathbf{A}, B, \mathbf{C}$ and a variable disjoint with it guarded clause $H \leftarrow \mathbf{G} \mid \mathbf{B}$. We say that B *guardedly unifies* with H if for some mgu θ of B and H the query $\mathbf{G}\theta$ is ground and succeeds. We call then $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ an *SLDG-resolvent* of $\mathbf{A}, B, \mathbf{C}$ and $H \leftarrow \mathbf{G} \mid \mathbf{B}$. So, intuitively, the test that the guards are ground and succeed forms now a part of the unification process. If $\mathbf{G}$ is empty, then the SLDG-resolvents coincide with the SLD-resolvents. So SLDG-resolution is indeed a generalization of the SLD-resolution. It is now clear how to define the SLDG-derivations and the SDLG-trees.

Given a guarded program, we now say that the delay declarations *imply determinacy* if every atom which satisfies its set of delay declarations guardedly unifies with a variant of at most one guarded clause head.

Note that the new set of delay declarations implies determinacy for the guarded program `QUICKSORT-G` because in the case of the `part` relation for any two ground terms `s,t` at most one of the queries `s > t` and `s ≤ t` succeeds. The remaining delay declarations imply determinacy because if a non-variable term unifies with `[X | Xs]`, then it is does not unify with `[ ]`.

Definition 38. We say that a guarded program is *regular* if

– every relation used in the guards is defined by a set of ground facts,
– for every other relation symbol p either
  - in all clauses defining p all guards are empty, or
  - for some sequence of terms $\mathbf{s}$ and a set of variables $V \subseteq Var(\mathbf{s})$, every guarded clause defining p is of the form

$$p(\mathbf{s}, \mathbf{t}) \leftarrow \mathbf{G} \mid \mathbf{B}$$

    for some $\mathbf{t}$, $\mathbf{G}$, $\mathbf{B}$, such that $Var(\mathbf{G}) = V$. □

Note that `QUICKSORT-G` is regular. In fact, we observed that most of the programs that use arithmetic comparison relations are regular when viewed as guarded programs.

The following theorem explains our approach to the proofs of termination for guarded programs. It is a modification of the Termination 1 Theorem 37.

Theorem 39 (Termination 2). *Suppose that*

– *P is a regular guarded program,*
– *P and Q are simply moded,*
– *the delay declarations imply determinacy and matching,*
– *an SLDG-derivation of $P \cup \{Q\}$ is successful.*

Then all SLDG-derivations of $P \cup \{Q\}$ which respect the delay declarations are successful, hence finite.

Proof. Omitted. □

Let us return now to `QUICKSORT-G`. In the previous sections we already checked before that `QUICKSORT-G` with the query `qs(s, Ys)`, where `s` is a list of integers, satisfies the first three condtions of the above theorem. Now, using the approach of Apt [Apt93] it is easy to show that `QUICKSORT-G` $\cup$\{`qs(s, Ys)`\} satisfies the last condition. We conclude by the Termination 2 Theorem 39 that all SLDG-derivations of `QUICKSORT` $\cup$\{`qs(s, Ys)`\} which respect the modified delay declarations are finite.

It is worthwhile to point out that the SLDG-resolution is very meaningful from the operational point of view. In the case of `QUICKSORT-G` it prevents a choice of a "wrong" alternative in the definition of the `part` relation and consequently, obviates a backtracking.

Finally, note that we cannot apply either the Termination 1 Theorem 37 or the Termination 2 Theorem 39 to the `SEQUENCE` program, because the delay declaration

```
DELAY app(_, _, Z) UNTIL nonvar(Z)
```

does not imply determinacy. Currently we are working on techniques allowing us to deal with termination in absence of determinacy.

The above two theorems are applicable only to the queries which have successful SLD-derivations. At this moment we know how to deal with termination for arbitrary queries at the cost of restricting attention to fair SLD-derivations. Recall that an SLD-derivation is *fair* if it is finite or if every atom occurring in it is eventually selected (after some possible instantiations).

Below, by an *LD-derivation* we mean an SLD-derivation via the leftmost selection rule. So LD-derivations are SLD-derivations generated by the Prolog selection rule. In the literature a lot of attention has been devoted to the study of termination with respect to the Prolog selection rule (see for example the survey article of De Schreye and Decorte [SD94]). This work can be use to deal with termination in presence of delay declarations. Namely, we have the following result.

Theorem 40 (Termination 3). *Suppose that*

– *all LD-derivations of* $P \cup \{Q\}$ *are finite.*

Then all fair SLD-derivations of $P \cup \{Q\}$ *are finite.*

Proof. The proof uses a generalization of the Switching Lemma (see Lloyd[Llo87, pages 50-51] to infinite SLD-derivations. Using it one can prove that if an infinite fair SLD-derivation exists, then an infinite LD-derivation exists. We omit the details. □

6.3 Termination of `APPEND`

It is important to realize that in general termination in presence of delay declarations is very subtle. As an example of the difficulties consider the `APPEND` program augmented with the delay declaration

```
DELAY app(X, _, _) UNTIL nonvar(X).
```

It seems that `APPEND` then terminates for all queries, that is, for all queries Q, all SLD-derivations of `APPEND` $\cup\{Q\}$ which respect this delay declaration are finite. The informal argument goes as follows. When the second clause of `APPEND` can be used to resolve a query of the form `app(s, t, u)`, then `s` is of the form `[x|xs]`, so in the next resolvent the first argument of `app` is `xs`, thus shorter. So eventually, the first argument is either a constant, in which case the second clause cannot be used, or else a variable and in this case the SLD-derivation terminates due to the delay declaration.

However, this reasoning is incorrect. Namely, consider the query `app([X | U], Ys, U)`. Using the second clause it resolves to itself and consequently an infinite SLD-derivation for `app([X | U], Ys, U)` can be generated with the first argument of `app` always being non-variable. (This fact was noticed in Naish [Nai92] and Lüttringhaus-Kappel [LK93]).

Thus the query `app([X | U], Ys, U)` does not terminate in presence of the delay declaration `DELAY app(X, _, _) UNTIL nonvar(X)`. Similarly, the query `app(U, Ys, [X | U])` does not terminate in presence of the delay declaration `DELAY app(_, _, Z) UNTIL nonvar(Z)`.

It seems that the problem has to do with the fact that the first and the third arguments of app share a variable. However, limiting one's attention to the queries of the form app(s, t, u) where s,t,u are pairwise variable disjoint does not help, as the query app([(U,X1,X1)|U], Ys, [([X2|V1],W,V1)|W]) resolves to the above query app([X | U], Ys, U).

What does hold is the following limited property.

Theorem 41. *Suppose that* s,t,u *are terms such that* s *and* u *are variable disjoint.*

(i) If u *is linear, then all SLD-derivations of* APPEND $\cup$ {app(s, t, u)} *that respect the delay declaration* DELAY app(X, _, _) UNTIL nonvar(X) *terminate.*
(ii) If t *is linear, then all SLD-derivations of* APPEND $\cup$ {app(s, t, u)} *that respect the delay declaration* DELAY app(_, _, Z) UNTIL nonvar(Z) *terminate.*
(iii) If s *and* t *are linear, then all SLD-derivations of* APPEND $\cup$ {app(s, t, u)} *that respect the delay declaration*
DELAY app(X, _, Z) UNTIL nonvar(X) $\vee$ nonvar(Z)
terminate.

Proof. (i) Let l(v) denote the number of symbols of the term v. Suppose that app(s, t, u) satisfies the delay declaration and resolves to app(s', t', u') using the second clause of APPEND. We prove that then s' and u' are variable disjoint, u' is linear and the pair (l(u'), l(s')) is smaller than (l(u), l(s)) in the lexicographic ordering.

s is not a variable, so it is of the form [x|xs], and u is not a constant. Two cases arise where we assume that app(s, t, u) and app([X | Xs], Ys, [X | Zs]) are variable disjoint.

Case 1 u is a variable.
Then {X/x, Xs/xs, Ys/t, u/[x|Zs]} is an mgu of app(s, t, u) and app([X | Xs], Ys, [X | Zs]), so app(s, t, u) resolves to app(xs, t, Zs). Now xs and Zs are variable disjoint, Zs is linear and the pair (l(Zs), l(xs)) is smaller than (l(u), l([x|xs])) in the lexicographic ordering.

Case 2 u is a compound term.
Then u is of the form [z|zs]. By the Iteration Lemma 1 the substitution {X/x, Xs/xs, Ys/t, Zs/zs}θ, where θ is a relevant mgu of x and z, is an mgu of app(s, t, u) and app([X | Xs], Ys, [X | Zs]). By assumption x is variable disjoint with zs. Moreover, [z|zs] is linear, so z is variable disjoint with zs, as well. Thus by the relevance of θ, zsθ = zs, so app(s, t, u) resolves to app(xsθ, tθ, zs). Now, zs is linear, so xsθ and zs are variable disjoint, because *Var*(xsθ) $\subseteq$ *Var*(s) $\cup$ *Var*(z). Moreover, the pair (l(zs), l(xsθ)) is smaller than (l([z|zs]), l(s)) in the lexicographic ordering.

This proves the claim.

(ii) By symmetry with (i).

(iii) Suppose that an infinite SLD-derivation ξ of `APPEND` $\cup\{$`app(s, t, u)`$\}$ exists that respects the delay declaration

`DELAY app(X, _, Z) UNTIL nonvar(X)` $\vee$ `nonvar(Z)`.

Then either `s` or `u` is not a variable. Assume without loss of generality that `s` is not a variable. By (i) a descendant of `app(s, t, u)` in ξ exists which is of the form `app(Xs, t', u')` for a variable `Xs` and some terms `t'` and `u'`. Since ξ is infinite, `u'` is not a variable.

In the moding `app(+,+,-)` `APPEND` and the query `app(s, t, u)` are nicely moded and the head of every clause of `APPEND` is input linear. Thus by the Nice modedness Corollary 12 `u'` is linear and `Xs` and `u'` are variable disjoint.

By (ii) a descendant of `app(Xs, t', u')` in ξ exists which is of the form `app(s'', t'', Zs)` for some terms `s''` and `t''` and a variable `Zs`. Moreover, in every descendant of `app(Xs, t', u')` in ξ the first argument of `app` remains a variable and consequently `app(s'', t'', Zs)` does not satisfy the delay declaration `DELAY app(X, _, Z) UNTIL nonvar(X)` $\vee$ `nonvar(Z)`. A contradiction.
$\square$

This isolated result shows how elaborate arguments are sometimes needed to prove simple termination results in presence of delay declarations. In this connection let us mention a related work of Naish [Nai86] and Lüttringhaus-Kappel [LK93] who automatically generate delay declarations which ensure that a given program terminates with respect to a selection rule according to which the leftmost non-delayed atom is selected.

7 Conclusions

In this paper we dealt with the correctness of logic programs augmented by the delay declarations. To this end we strengthened and adapted methods that were originally developed for the study of Prolog programs.

Interestingly, we can reverse the situation and derive the appropriate delay declarations by analyzing the Prolog programs. Consider for example the `QUICKSORT` program with the query `qs(s, Ys)`, where `s` is a list of integers. Once we have shown in Section 4 that it is well-moded with the moding `qs(+,-)`, `partition(+,+,-,-)`, `app(+,+,-)`, `+>+`, $+\leq+$ we can augment it by arbitrary delay declarations which are implied by this moding and conclude by virtue of the Absence of Deadlock 1 Theorem 19 that for the query in question no deadlock arises.

Once we have shown in Section 4 that it is well-typed and simply moded with the typing
`qs`$(+ : ListGae, - : ListGae)$,
`part`$(+ : Gae, + : ListGae, - : ListGae, - : ListGae)$,
`app`$(+ : ListGae, + : ListGae, - : ListGae)$
$> (+ : Gae, + : Gae)$,
$\leq (+ : Gae, + : Gae)$,
we can augment it by arbitrary delay declarations which imply matching and

conclude by virtue of the Correct Typing Theorem 33 that for the query in question no run time errors due to the use of arithmetic relations arise.

Finally, once we have shown in Section 6 that some SLDG-derivation of `QUICKSORT-G` $\cup$\{`qs(s, Ys)`\} is successful, we can augment `QUICKSORT-G` by arbitrary delay declarations which imply determinacy and matching, and conclude by virtue of the Termination 2 Theorem 39 that all SLDG-derivations of `qs(s, Ys)` which respect these delay declarations are finite. This shows that it is possible to derive correct parallel logic programs by analyzing Prolog programs.

We conclude by noticing that the "stronger" the delay declarations are the bigger the chance that a deadlock arises, but the smaller the chance that divergence can result. So deadlock freedom and termination seem to form two boundaries within which lie the "correct" delay declarations.

Acknowledgements

The first author would like to thank Elena Marchiori and Frank Teusink for helpful discussions on the subject of the delay declarations.

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.

[AP94] K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.

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

[Apt93] K. R. Apt. Declarative programming in Prolog. In D. Miller, editor, *Proc. International Symposium on Logic Programming*, pages 11–35. MIT Press, 1993.

[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.

[CP94] R. Chadha and D. A. Plaisted. Correctness of unification without occur check in Prolog. *Journal of Logic Programming*, 18(2):99–122, 1994.

[DFT91] P. Deransart, G. Ferrand, and M. Téguia. NSTO programs (not subject to occur-check). In V. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Symposium*, pages 533–547. The MIT Press, 1991.

[DM85] 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.

[Dra87] W. Drabent. Do Logic Programs Resemble Programs in Conventional Languages? In *International Symposium on Logic Programming*, pages 389–396. San Francisco, IEEE Computer Society, August 1987.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.

[Kow79] R. Kowalski. Algorithm = Logic + Control. *Communications of ACM*, 22:424–431, 1979.

[LK93] S. Lüttringhaus-Kappel. Control generation for logic programs. In D. S. Warren, editor, *Proceedings of the 10th Int. Conf. on Logic Programming, Budapest*, pages 478–495. MIT, July 1993.

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

[Lui94] I. Luitjes. Logic programming and dynamic selection rules. Scriptie (Master's Thesis), University of Amsterdam, 1994.

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

[Nai82] L. Naish. An Introduction to MU-PROLOG. Technical Report TR 82/2, Dept. of Computer Science, Univ. of Melbourne, 1982.

[Nai86] L. Naish. *Negation and Control in Prolog*. Number 238 in Lecture Notes in Computer Science. Springer-Verlag, 1986.

[Nai88] L. Naish. Parallelizing NU-Prolog. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 1546–1564. The MIT Press, 1988.

[Nai92] L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, Department of Computer Science, University of Melbourne, 1992.

[SD94] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

This article was processed using the LaTeX macro package with LLNCS style