# Proving Termination of Logic Programs with Delay Declarations

**Elena Marchiori** and **Frank Teusink**
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
e-mail: {elena,frankt}@cwi.nl

## Abstract

In this paper we propose a method for proving termination of logic programs with delay declarations. The method is based on the notion of *recurrent* logic program, which is used to prove programs terminating with respect to an arbitrary selection rule. Most importantly, we use the notion of bound query (as proposed by M. Bezem) in the definition of *cover*, a new notion which forms the kernel of our approach. We introduce the class of *delay recurrent* programs and prove that programs in this class terminate for all *local delay* selection rules, provided that the delay conditions imply boundedness. The corresponding method can be also used to transform a logic program into a terminating logic program with delay declarations.

## 1 Introduction

Delay declarations are used for the dynamic control of the selection of atoms in a derivation. The idea is that, besides the usual logic clauses, the program contains declarations of the form

DELAY *predicate* UNTIL *condition*

Then, a selection rule is used which only selects an atom from a query, if that atom is not delayed, i.e. the condition in the delay declaration for that atom is satisfied. Delay declarations are employed in many programming systems based on logic programming, like NU-Prolog [TJ86] and Gödel [HL94]. They are important for a number of reasons: for instance, they can be used to ensure termination of the program, or to support coroutining. As a consequence, efficient algorithms can be produced from a simple logical specification augmented with suitable delay declarations. This approach reflects the idea of considering a program as consisting of two parts: logic and control.

In this paper, we study termination of logic programs with delay declarations. To illustrate how delay declarations may affect the termination behaviour of a program, consider the well-known *append*/3 program:

$$app([x|xs], ys, [x|zs]) \leftarrow app(xs, ys, zs).$$
$$app([], ys, ys).$$

and the query $app(xs, [4,5], zs), xs = [1,2,3]$. This query does not terminate when the leftmost selection rule is used. However, suppose we add the following delay declaration for $append/3$:

$$\text{DELAY } app(xs, ys, zs) \text{ UNTIL } list(xs)$$

With this delay declaration, the leftmost atom in the query is delayed. Therefore, if we use a delay selection rule, only the second atom can be selected, resulting in the resolvent $app([1,2,3], [4,5], zs)$. Here the atom in the query is not delayed. Moreover, this query is terminating.

The termination behaviour of a logic program with delay declarations is rather subtle. There are various aspects, sometimes unexpected, that one has to take into account. A thorough discussion of these aspects is given by Naish in [Nai92]. For instance, one would expect the delay declaration

$$\text{DELAY } app(xs, ys, zs) \text{ UNTIL } nonvar(xs) \ \vee \ nonvar(zs)$$

to ensure the termination of $append/3$. However, as illustrated by Naish, the query $app([a|T], [\,], T)$ satisfies the delay declaration, but has an infinite derivation. The fact that termination behaviour of logic programs in the context of dynamic selection rules is very subtle, is reflected also in the various methods that have been introduced, which are either based on heuristics (e.g. [LK92, MNL90]), or are rather specialized (e.g. [AL95]).

In this paper we try to tackle the problem from a different perspective. That is, we do not consider general coroutining, with all its problems, but consider the class of delay selection rules which are "local". Local selection rules are introduced in [Vie89], and correspond to selecting always in a query one of the most recently introduced atoms in the derivation from the initial query. Local selection rules behave well w.r.t. semantic information, in the following sense. If an atom of a query in a derivation is selected, then the derivation is committed to resolve that atom, and only after that atom has been completely resolved, an other atom of the query can be selected. It is this semantic property of local selection rules, which allows us to define a simple, yet powerful, method for proving termination of logic programs with delay declarations.

Our method is based on the notion of bounded query, introduced by Bezem in [Bez89, Cav89] to study termination of logic programs. We use this notion to define the central concept of our method, namely the *covers* of a body atom of a clause (query). Then, using a combination of syntactical (covers) and semantical (model) information, we define the notion of delay recurrent program. This notion is a generalization to SLD-resolution with delay selection rules, of the one of recurrent program, introduced by Bezem to study termination of logic programs w.r.t. an arbitrary selection rule. We prove that a delay recurrent program terminates for every local selection rule which selects only bounded atoms. Thus, this notion provides a method for proving termination of a logic program with delay declarations, when the

delay declarations imply boundedness, i.e. if an atom satisfies its delay declaration, then that atom is bounded. Alternatively, this method can be used to find suitable delay declarations that ensure termination of goals for a given program, by choosing delay declarations which imply boundedness.

We believe that the contribution of this paper is important for at least two reasons: it provides a simple tool to reason about termination of logic programs with delay declarations, which can be also used to transform a logic program in a terminating logic program with delay declarations; moreover, it provides a new insight on the role of the selection rules when reasoning about the run-time behaviour of logic programs with delay declarations. In particular, it shows that the class of local selection rules is not only good because it supports efficient searching techniques, but also because it supports simple tools for proving termination.

The paper is organized as follows. After some preliminaries in Section 2, we present our method and the termination results in Section 3. Then, in Section 4 we give an example of proving the program *quicksort*/2 terminating in reverse order. In Section 5, we discuss some aspects of our method. For lack of space, the proofs have been omitted. They can be found in the full version.

## 2 Preliminaries

We shall use the following notation and terminology.

A *logic program*, called for brevity *program* and denoted by $P$, is a finite set of (universally quantified) clauses $H \leftarrow Q$, where $Q$ is a query, i.e. a sequence of atoms, and $H$ is an atom. In the following, the letters $A, B$ indicate atoms and $c$ a clause. For a query $Q$, define a $Q$-ground instance of a clause $c$ to be any instance of $c$ which grounds all the atoms of $Q$. Finally, c.a.s. is used as shorthand for computed answer substitution.

A sequence of atoms will also be denoted by $\tilde{A}$. As we are not interested in the order of atoms, we will sometimes treat sequences of atoms as multisets. Moreover, we will sometimes implicitly translate a sequence of atoms into a set of atoms, in order to be able to refer to elements, subsets, unions, etc. In those cases, multiplicity of atoms will be ignored, i.e. $p, p$ will be translated into $\{p\}$. We only do this where multiplicity of atoms is not an issue.

We shall use multisets and the multiset ordering (see [Der87]). Recall that a *multiset* is a unordered collection in which the number of occurrences of each element is significant. We shall consider here the multiset ordering on multisets of natural numbers. Formally, a multiset of natural numbers is a function from the natural numbers to itself, giving the multiplicity of each natural number. Then, given the standard order $<$ on natural numbers, the ordering $<_{mul}$ on multisets is defined as the transitive closure of the replacement of a natural number with any finite number (possibly zero) of natural numbers that are smaller under $<$. Since $<$ is well-founded, the

induced ordering $<_{mul}$ is also well-founded. For simplicity we shall omit in the sequel the subscript *mult* from $<_{mul}$.

A delay declaration is denoted as follows: for a predicate $p$ of arity $n$ a delay declaration has the form

$$\text{DELAY } p(x_1, \ldots, x_n) \text{ UNTIL } Cond(x_1, \ldots, x_n)$$

where $x_1, \ldots, x_n$ denote the arguments of $p$, and $Cond(x_1, \ldots, x_n)$ is a formula in some assertion language. We shall not fix the syntax of that assertion language, as it is not relevant for the sequel of the paper. The meaning of such a delay declaration is that in a query an atom $p(t_1, \ldots, t_n)$ can only be selected if the condition $Cond(t_1, \ldots, t_n)$ is satisfied. We shall assume that if an atom is selectable then all its instances are selectable too. This condition is satisfied by almost all the logic programming systems which use delay declarations. Its importance in the study of termination is crucial, and all the approaches we are aware of, for the study of properties of logic programs with delay declarations, use this assumption.

The delay declarations in a program define a class of selection rules, called *delay selection rules*. A *delay selection rule* selects an atom from a query, among those atoms which satisfy their delay declarations. If the query is non-empty and no such atom exists, no atom is selected and the query is *deadlocked*. When using delay declarations, we are only interested in SLD-derivations that are constructed using a delay selection rule. We call these derivations *delay SLD-derivations*.

## 3 Delay-Recurrent Programs

The aim of this paper is to define a class of programs that behave nicely with respect to termination. First, we introduce the notion of delay recurrent program. Then, we prove that, for a suitable delay declaration and a broad class of delay selection rules, every query in a delay recurrent program has only finite derivations. To this end, we use the notions of level mapping and of bounded query, introduced in [Bez89, Cav89].

**Definition 3.1 (level mapping)** Let $P$ be a program. A *level mapping for* $P$ is a function $|\ |: \mathcal{B}_P \rightarrow \mathbb{N}$ from the Herbrand base for $P$ to the set of natural numbers. □

Thus, $|\ |$ is only defined for ground atoms. However, one can associate to a non-ground atom, the image of its set of ground instances with respect to $|\ |$:

$$||A|| \stackrel{def}{=} \{|A'| \mid A' \text{ is a ground instance of } A\}$$

Using this, we define the notion of *bounded* atoms and queries.

**Definition 3.2 (bounded query)** An atom $A$ is *bounded* (with respect to $|\ |$), if $||A||$ is finite. A query $Q$ is bounded if all the atoms in it are bounded. □

With a bounded query $Q = A_1, \ldots, A_n$ is associated the multiset $|[Q]|$ as follows:

$$|[Q]| \stackrel{def}{=} [\![max||A_1||, \ldots, max||A_n||]\!]$$

where $max||A||$ denotes the maximum of $||A||$. In the sequel, we shall often refer to $|[Q]|$ as the *level mapping* of $Q$.

The idea of using a level mapping to prove termination is that one proves that, in a derivation, selected atoms are always bounded and that the level mappings of the queries decrease. We can use delay declarations to ensure that only bounded atoms are selected, i.e. that the delay declarations imply boundedness.

**Definition 3.3 (safe delay declaration)** A delay declaration is *safe with respect to* $|\ |$ if for every atom $A$, if $A$ satisfies its delay declaration then $A$ is bounded with respect to $|\ |$. □

So, by using safe delay declarations, we ensure that selected atoms are bounded. Now, we provide a method that ensures that the level mapping also decreases. For this, we use the information that selected atoms are bounded, together with the additional information provided by a model of the program. In order for an atom to be bounded, certain other atoms that originate from the same body, must have been (partially) resolved. We call these sets of atoms *covers*. To define the covers of a body atom, we need the notion of *direct covers*. Intuitively, a direct cover of an atom $A$ in a query is a subset $\tilde{B}$ of that query, such that for some instantiation $\theta$ of the variables in $\tilde{B}$, $A\theta$ is bounded.

**Definition 3.4 (direct cover)** Let $|\ |$ be a level mapping. Let $Q$ be a query, let $B$ be an atom in $Q$ and let $\tilde{C}$ be a subset of $Q$ such that $B \notin \tilde{C}$. We say that $\tilde{C}$ is a *direct cover for $B$ with respect to $Q$ (and $|\ |$)*, if there exists a substitution $\theta$ such that $B\theta$ is bounded with respect to $|\ |$ and $Dom(\theta) \subseteq Var(\tilde{C})$.

Let $H$ be an atom. We say that $\tilde{C}$ is a *direct cover for $B$ with respect to $H \leftarrow Q$ (and $|\ |$)*, if there exists a substitution $\theta$ such that $B\theta$ is bounded with respect to $|\ |$ and $Dom(\theta) \subseteq Var(H, \tilde{C})$.

Finally, a direct cover $\tilde{C}$ of $B$ is *minimal* if no proper subset of $\tilde{C}$ is a direct cover for $B$. □

One should note that a body atom $B$ can have zero, one, or more (minimal) direct covers. For instance when, for $B$ to become bounded, it is necessary to instantiate a variable of $B$ which does not occur anywhere else in the clause, $B$ will have no direct covers. On the other hand, if $B$ is bounded whenever $H$ is bounded, then there exists only one minimal direct cover, namely the empty set. It is worthwhile to notice that the direct covers of an atom depend on the level mapping one chooses. For instance, consider the clause $p(x) \leftarrow p(y)$, and the two level mappings $|\ |_1$ and $|\ |_2$ such that: if $s$ is a list then $|p(s)|_1$ is equal to its length, otherwise it is equal to 0;

and $|p(s)|_2$ equal to 0 for every $s$. Then $p(y)$ has no direct cover w.r.t. $|\ |_1$, while it has $\emptyset$ as direct cover w.r.t. $|\ |_2$. Finally, we would like to emphasize that direct covers can be 'cyclic', in the sense that two atoms can have each other in their direct covers. Take for instance the query $p(x), q(x)$ and a level mapping $|\ |$ in which boundedness of $p(x)$ and $q(x)$ depend on $x$. Then, $p(x)$ will have direct cover $\{q(x)\}$ and $q(x)$ will have direct cover $\{p(x)\}$.

In the definition of cover, we take a kind of 'closure' of the direct cover relation.

**Definition 3.5 (cover)** Let $Q$ be a query and let $|\ |$ be a level mapping. Let $B$ be an atom in $Q$ and let $\tilde{C}$ be a subset of $Q$. Then, $\tilde{C}$ is a *cover of $B$ (with respect to $Q$ and $|\ |$)*, if $\langle B, \tilde{C}\rangle$ is an element of the least set $\mathcal{C}$ ($\mathcal{C} \subseteq \mathcal{P}(Q \times \mathcal{P}(Q))$ such that

1. $\langle B, \emptyset\rangle \in \mathcal{C}$ whenever $B$ has the emptyset as minimal direct cover, and
2. $\langle B, \tilde{C}\rangle \in \mathcal{C}$ whenever $B \notin \tilde{C}$, and $\tilde{C}$ is of the form

$$\{C_1, \ldots, C_k\} \cup \tilde{D}_1 \cup \ldots \cup \tilde{D}_k$$

such that $\{C_1, \ldots, C_k\}$ is a minimal direct cover of $B$ in $Q$, and for $i \in [1..k]$, $\langle C_i, \tilde{D}_i\rangle \in \mathcal{C}$.

The notion of cover of an atom in a clause is defined analogously. □

One can easily prove that the cover relation is

- 'acyclic', in the sense that if $B$ is in a cover of $A$ then $A$ is not in any cover of $B$,
- 'monotone', in the sense that if $\tilde{C}$ is a cover of $A$ then for all $\theta$ a subset of $\tilde{C}\theta$ is a cover of $A\theta$, and
- 'well-founded', in the sense that if there exists an atom $A$ in $Q$ such that $A$ has a cover then there exists an atom $B$ in $Q$ such that $B$ has an empty cover.

Using the notion of covers, we can define the class of *delay recurrent* programs.

**Definition 3.6 (delay recurrent program)** Let $|\ |$ be a level mapping and $I$ an interpretation for a program $P$.

- A clause $c : H \leftarrow Q$ is *delay-recurrent* with respect to $|\ |$ and $I$ if $I$ is a model for $c$ and for every atom $A$ in $Q$, for every cover $\tilde{B}$ for $A$, and for every $(H, \tilde{B})$-ground instance $H' \leftarrow Q'$ of $c$ such that $H'$ is bounded and $I \models \tilde{B}'$, we have that

$$|[H']| > |[A']|$$

- A program $P$ is delay-recurrent w.r.t. $||$ and $I$ if every clause is delay-recurrent with respect to $||$ and $I$. □

Knowing that a selected atom is bounded is useful, because it implies that one of the covers of that atom has been partially resolved. However, it is not enough. We need to be sure that a cover of the selected atom has been resolved completely. In order to be able to ensure this, we have to use a *local selection rule*. Local selection rules were extensively studied by Vieille in [Vie89].

**Definition 3.7 (local selection rule)** Let $Q$ be a query in a derivation $\eta$, containing atoms $A$ and $B$. Then $A$ is introduced *more recently* than $B$, if the derivation step introducing $A$ comes before the derivation step introducing $B$, in $\eta$. $A$ is introduced *most recently*, if no atom $B$ is introduced more recently than $A$.

A *local selection rule* is a selection rule that only selects *most recently* introduced atoms. □

Note that, if in a query $Q$ none of the most recently introduced atoms satisfies its delay declaration, then a local delay selection rule should deadlock on $Q$.

Using local selection rules, we have the following result.

**Theorem 3.8** *Let $P$ be a logic program with delay declarations. Let $||$ be a level mapping and let $I$ be an interpretation. Suppose that:*

1. *$P$ is delay-recurrent w.r.t. $||$ and $I$, and*
2. *the delay declarations are safe w.r.t. $||$.*

*Then for every query $Q$, every delay SLD-derivation for $Q$ which uses a local selection rule is finite.*

Note that we do not assume $Q$ to be delay recurrent. We don't need to, because with the local selection rule, the atoms in $Q$ will be resolved one at a time, without coroutining.

We conclude this section by showing that the notion of delay-recurrent program is a generalization of the notion of *recurrent programs*. This notion is due to Bezem [Bez93]. A program $P$ is recurrent if for some level mapping $||$, every ground instance $H \leftarrow A_1, \ldots, A_n$ of a clause of $P$ satisfies the test

$$|H| > |A_i|$$

for every $i \in [1, n]$. Then we have the following result.

**Lemma 3.9** *If a program $P$ is recurrent with respect to $||$ then $P$ is delay recurrent with respect to $||$ and $I$, for any model $I$ of $P$.*

## 4 An example: Quicksort

In this section, we illustrate the application of our method by means of an example. To help the reader to focus more on the method than on the example, we have chosen the well-known program *quicksort*/2, defined by the following set of clauses:

$qs([x|xs], ys) \leftarrow$
$\quad part(xs, x, ls, bs),\ qs(ls, sls),\ qs(bs, sbs),\ app(sls, [x|sbs], ys).$
$qs([], []).$

$part([x|xs], y, [x|ls], bs) \leftarrow x \leq y,\ part(xs, y, ls, bs).$
$part([x|xs], y, ls, [x|bs]) \leftarrow x > y,\ part(xs, y, ls, bs).$
$part([], y, [], []).$

augmented with the clauses for *append*/3 given in the Introduction. Usually, the intended use of the predicate *qs* is that of giving it a list as first argument, in order to get a sorted permutation of that list as output in the second argument. This usage of *quicksort*/2 was proven to be safe (with respect to termination) e.g. in [AL95], where a proper delay declaration is chosen. Here we will show that, one can also use safely the program in its reverse, i.e. give *qs* a sorted list in its second argument, and it will produce all permutations of that list in its first argument. Observe that when the Prolog selection rule is used, this alternative usage of the program yields non-termination. This is the main reason why the approach of Apt and Luitjes cannot deal with this case.

We now give a level mapping for the predicates in the program, and a model. It would go too far to give a detailed account of the way we arrived at this specific level mapping. For those who are interested in techniques for finding level mappings, we refer e.g. to [DSF93]. Let $t1, \ldots, t4$ be ground terms. Then:

$|qs(t1, t2)| = tsize(t2) + 1,$
$|part(t1, t2, t3, t4) = tsize(t3) + tsize(t4),$
$|app(t1, t2, t3)| = tsize(t3),$
$|t1 > t2| = 0,$
$|t1 \leq t2| = 0,$

where

$$tsize(t) = \begin{cases} \text{the length of } t & \text{if } t \text{ is a list} \\ 0 & \text{otherwise} \end{cases}$$

Moreover, consider the following interpretation $I$:

$$\begin{array}{lll} I = & \{qs(t1, t2) \mid tsize(t1) = tsize(t2)\} & \cup \\ & \{part(t1, t2, t3, t4) \mid tsize(t1) = tsize(t3) + tsize(t4)\} & \cup \\ & \{app(t1, t2, t3) \mid tsize(t3) = tsize(t1) + tsize(t2)\} & \end{array}$$

| atom | minimal direct cover | cover |
|---|---|---|
| $part(xs, x, ls, bs)$ | $\{qs(ls, sls), qs(bs, sbs)\}$ | $\left\{ \begin{array}{l} qs(ls, sls), qs(bs, sbs), \\ app(sls, [x\|sbs], ys) \end{array} \right\}$ |
| $qs(ls, sls)$ | $\{app(sls, [x\|sbs], ys)\}$ | $\{app(sls, [x\|sbs], ys)\}$ |
| $qs(bs, sbs)$ | $\{app(sls, [x\|sbs], ys)\}$ | $\{app(sls, [x\|sbs], ys)\}$ |
| $app(sls, [x\|sbs], ys)$ | $\emptyset$ | $\emptyset$ |

Figure 1: Computing covers for $qs$

It is easy to check that $I$ is a model of $quicksort/2$.

We have to prove that the clauses of $quicksort/2$ are delay recurrent with respect to this level mapping and this model. For $app$ and $part$, this is easy to check, because they are recurrent with respect to the given level mapping. Hence the result follows from Lemma 3.9.

So, to prove the program delay recurrent, we have to check the two clauses for $qs$. The second clause is trivial, because it is a fact. To check the first clause, we actually have to do some work. First, we compute the minimal direct covers and covers for the atoms in the body. These are given in Figure 1. As we see, in this case every atom has a single minimal direct cover and a single cover.

Having found the covers, we can prove that the clause is delay recurrent. First of all, consider $app(sls, [x|sbs], ys)$. A $qs(xs, ys)$-ground instance of the clause binds $xs$ and $ys$ to ground terms, say $t1$ and $t2$. It follows directly from the level mappings of $qs$ and $app$ that:

$$|[qs([x|t1], t2)]| = tsize(t2) + 1 > tsize(t2) = |[app(sls, [x|sbs], t2)]|$$

Secondly, $qs(ls, sls)$ has $\tilde{B} = \{app(sls, [x|sbs], ys)\}$ as cover. A $(\tilde{B}, qs(xs, ys))$-ground instance of the clause binds $xs, ys, x, sls, sbs$ to ground terms, say $t1, \ldots, t5$, respectively. Suppose that

$$I \models app(t4, [t3|t5], t2).$$

Then $tsize(t2) > tsize(t4)$. But then, we have that

$$|[qs([t3|t1], t2)]| = tsize(t2) + 1 > tsize(t4) + 1 = |[qs(ls, t4)]|$$

The proof for $qs(bs, sbs)$ is similar. Finally, $part(xs, x, ls, bs)$ has cover $\tilde{B} = \{qs(ls, sls), qs(bs, sbs), app(sls, [x|sbs], ys)\}$. A $(\tilde{B}, qs(xs, ys))$-ground instance of the clause binds $xs, ys, x, sls, sbs, ls, bs$ to ground terms, say $t1, \ldots, t7$, respectively. Suppose that

$$I \models qs(t6, t4), qs(t7, t5), app(t4, [t3|t5], t2).$$

Then $tsize(t2) > tsize(t6) + tsize(t7)$. But then we have that

$$|[qs([t3|t1], t2)]| = tsize(t2) + 1 > tsize(t6) + tsize(t7) = |[part(t1, t3, t6, t7)]|$$

So, we have proven that *quicksort*/2 is delay recurrent with respect to $|\ |$ and $I$. As a result, we have that all queries will terminate, provided that a local delay selection rule is used and the delay declarations are safe. Thus, we now have to translate the boundedness information given by the level mapping into delay declarations, i.e. find delay declarations for *qs*, *part* and *app* such that if an atom is not delayed, it is bounded. For this, the following delay declarations suffice:

DELAY $qs(xs, ys)$ UNTIL $list(ys)$

DELAY $part(xs, y, ls, bs)$ UNTIL $list(ls) \ \wedge \ list(bs)$

DELAY $app(xs, ys, zs)$ UNTIL $list(ys)$

# 5 Observations

In this section we discuss some aspects of our approach, and possible extensions. More precisely, we investigate the role of local selection rules in proving termination, the class of delay declarations that can be expressed using our method, and when the delay declarations do not affect the declarative semantics of the program.

## 5.1 Why Local Selection Rules?

In the soundness result on our method (Theorem 3.8), we restrict ourselves to local selection rules. The reason for this is that we want to use the semantic information provided by the model $I$. In the proof of Theorem 3.8, we use this semantic information as follows. First we observe that, when an atom $A$ becomes selectable, some cover $\tilde{B}$ of $A$ in the input clause that introduced $A$ has been partially instantiated. By using the fact that a local selection rule is used, we can conclude that this cover $\tilde{B}$ has been resolved completely. As a result, we have that $I \models \forall\, \tilde{B}\theta$, where $\theta$ is the composition of substitutions between the node where (a generalization of) $A$ was introduced and the node where $A$ is selected. Finally, we use this fact to prove that the level mapping of $A$ is strictly smaller that the level mapping of the selected atom in the resolution step that introduced (a generalization of) $A$.

Thus, we need to restrict ourselves to the local selection rule in order to conclude that $I \models \forall\, \tilde{B}\theta$, which allows us to use the semantic information contained in $I$. This implies that our method cannot be used directly e.g. with Gödel. In fact, the Gödel selection rule selects the leftmost atoms of a

query, among those which satisfy their delay declaration, even if this atom is not most recently introduced.

There is one strong argument against the use of local selection rules: they do not allow any form of coroutining. In order to prove termination with respect to selection rules that allow coroutining, we have to get rid of the restriction to local selection rules. An approach which seems quite promising, is restricting oneself to programs that do not use *speculative bindings*, a notion introduced by Naish in [Nai92]. This is something which deserves further investigation. However, we do have the impression that any method for proving termination with *full* coroutining will be either very complex, or very restrictive in its applications.

### 5.2 On Completeness of Delay Declarations

We have seen how delay declarations can be used to ensure termination of a logic program. One could choose strong delay declarations, like for instance DELAY $p(\tilde{x})$ UNTIL *false*, which certainly imply termination. However, the resulting program would not be very interesting, since it yields no c.a.s.'s. To ensure that the delay declaration is not too strong, one has to guarantee that the declarative semantics of the program is preserved. This is specified in the following definition.

**Definition 5.1 (complete delay declaration)** Let $P$ be a program and let $I$ be the least Herbrand model for $P$. Let $\mathcal{D}$ be a set of delay declarations for $P$. We say that $\mathcal{D}$ is *complete w.r.t.* $P$ if every atom in $I$ has a successful delay SLD-derivation in $P \cup \mathcal{D}$.

A sufficient condition for completeness of a delay declaration w.r.t. $P$ is that every ground atom which is in $I$ is deadlock free. An atom is deadlock free if all its finite derivations do not end in a non-empty query which contains only atoms that do not satisfy their delay declarations. Then, the following result holds.

**Lemma 5.2** *Let $P$ be a program and let $I$ be the least Herbrand model for $P$. Let $\mathcal{D}$ be a set of delay declarations for $P$. Suppose that every atom $A$ of $I$ is deadlock-free. Then $\mathcal{D}$ is complete with respect to $P$.*

Recently, the topic of deadlock-freedom of programs with delay declarations has been studied in [AL95] and [MT95]. The methods there introduced can be applied to prove that every atom of $I$ is deadlock-free.

### 5.3 On Expressiveness of Delay Declarations

In Gödel, one can use the predicate *nonvar* in delay declarations. For instance, the following delay declaration is used for the predicate *app* defined by the program given in the Introduction:

$$\text{DELAY } app(xs, ys, zs) \text{ UNTIL } nonvar(xs) \ \vee \ nonvar(zs)$$

When this delay declaration is used, an atom $app(s,t,u)$ is not selected until either $s$ or $u$ is a non-variable term.

We cannot deal with these kinds of delay declarations. The reason is that in our definition of delay recurrent programs, the notion of level mapping we use is the one used in the definition of recurrent programs. In this definition, the level mapping $|A|$ for ground atoms $A$ is defined by a (total) function from $\mathcal{B}_P$ to $\mathbb{N}$, whereas the level mapping $||[B]||$ for non-ground atoms $B$ is defined as the maximum of the level mappings of all its ground instances. Thus $||\ ||$ is a partial function, because the set of level mappings of ground instances can be unbounded. As a consequence, when taking the level mapping of an atom $p(l)$ to be the length of list $l$, the atom $p([x|xs])$ contains a non-variable term, but $|[p([x|xs])]|$ is undefined because $xs$ can be instantiated with an arbitrary large ground list. Thus, an atom $app([x|xs],ys,zs)$ is not bounded, while it satisfies the condition of the delay declaration. Terms which behave well with respect to a level mapping have been studied for instance in [BCF94], where they are called rigid.

As the *append*/3 example given in the Introduction shows, the termination behaviour of "delay until nonvar" is poorly understood. As far as we can see now, a method handling the *nonvar* delay predicate would also be significantly more complex (or, alternatively, weaker), than our method. All in all, the problems with the *nonvar* delay predicate were enough for us to decide not to deal with it at this point. As a final remark we would like to note that, if one browses through the Gödel manual, it seems that our method is severely handicapped by not being able to handle *nonvar*, because most delay declarations in example programs use *nonvar*. One should note however, that these programs are not guaranteed to terminate for all goals (not even when the leftmost undelayed selection rule is used). To be fair, the Gödel manual only states that the delay declarations can be used to *assist* termination. On the other hand, our method guarantees termination, be it that the delay declarations will be more restrictive.

### 5.4 On programs with negation

It seems that our method can be easily extended to deal with logic programs with negation. We sketch briefly how this could be done. One can extend the procedure for resolving negated atoms to the case of delay selection rules, simply considering a form of (abnormal) termination, which arises when a tree for $\neg A$ is finite but contains at least one leaf consisting of delayed literals. In such a case $\neg A$ has no resolvent; it ends in deadlock. Then the definition of level mapping can be extended to negated atoms, simply by defining $|\neg A| = |A|$. Finally, in Definition 3.6 of delay-recurrent program, the model $I$ should be replaced by some model containing suitable semantic information.

## 6 Related Work

Let us now relate our approach to other work on termination with respect to dynamic selection rules.

The paper which helped us to understand the problems in reasoning about the termination of logic programs with delay declarations, is [Nai92]. In this paper, L. Naish investigates how termination of a conjunction of queries can be established, under the hypothesis that the execution of each query does terminate. However, he does not propose ready to use methods for proving programs terminating. In his paper, Naish argues that the use of modes is crucial to reasoning about termination. To support this claim, he gives a number of useful observations on the termination behaviour of a program with delay declarations, which emphasize how subtle is this behaviour, and how difficult it is to prove termination, when dealing with general coroutining. Towards the end of the paper, Naish suggests that the existence of 'speculative bindings' are an important complicating factor when reasoning about termination. It might be the case that in absence of these speculative bindings, we can generalize our method to non-local delay selection rules.

Another recent contribution to the subject of termination with respect to delay declarations, is [LK92]. In this paper, S. Lüttringhaus-Kappel discusses a non-deterministic scheme for finding delay declarations that ensure termination. First, he presents an algebra of 'when' declarations. This algebra is more expressive than the class of delay declarations we can handle, basically because we cannot handle *nonvar* predicates. The scheme itself is very general; it is meant as a basis for practical implementations, using heuristics and partial evaluation to replace non-deterministic choices. The results of an existing implementation look quite promising. On the other hand, as the scheme is very general, it does not give much insight in the problem of termination itself. Another problem is that one has to prove that a program is 'safe' (not the notion used in this paper), which is quite difficult, the more because there are no methods for doing this.

A very recent paper by K.R. Apt and I. Luitjes [AL95], stimulated us to work on our approach. In this paper, they discuss verification of logic programs with respect to dynamic selection rules. In one section they discuss the problem of termination. The approach they take is more general than ours, in the sense that they do not restrict to local selection rules. As a consequence, they need to impose strong restrictions on the class of programs they consider. One restriction in this work is that the termination results are stated in terms of termination with respect to LD-resolution. Thus, it can only discuss termination with respect to dynamic selection rule of programs which are known to terminate with respect to leftmost selection rule.

It is clear that most of the programs we can prove terminating with our method, can also be proven to be terminating by a static reordering of bodies of program clauses. We think however that the use of covers has a number of advantages. First of all, with covers we have a systematic approach for

finding static orderings that ensure termination, which is more efficient than simply checking all permutations of body atoms. Secondly, our method does not impose an order on body atoms. If one fixes the order of body atoms in order to ensure termination, one looses the freedom to let a compiler or optimizer fix some order. Instead, the covers computed in our method form a concise representation of *all* orderings of body atoms that ensure termination. This information can be fed to a compiler or optimizer, as a constraint on the orderings of bodies it may choose. Finally, there exist programs that can be proven terminating with our approach, which are not (easily) proven terminating with a static approach.

# 7 Conclusion

In this paper we introduced a simple method for proving termination of logic programs with delay declarations. The method is based on the new notion of cover, which is used to describe the inter-relation among the atoms of a clause that can be caused by the dynamic scheduling. Covers are used to define the class of delay recurrent programs. We proved that all derivations of a delay-recurrent program are finite, when the selection rule is local delay, i.e. it selects at each resolution step one atom which satisfies its delay declaration, among those atoms most recently introduced. We discussed advantages and limitations of this last condition on the selection rule. We intend to continue investigating other conditions under which we can relax the restriction to local selection rules, although we think that such methods are necessarily either much more complex or applicable to much smaller classes of programs.

# Acknowledgements

We would like to thank Krszysztof Apt for helpful discussions on the subject of the delay declarations. Also we would like to thank the referees for their useful comments. The research of the first author was partially supported by the ESPRIT Basic Research Action 6810 (Compulog 2). The second author was partially supported by SION, a department of the NWO, the National Foundation for Scientific Research.

# References

[AL95] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST'95)*, Lecture Notes in Computer Science, Berlin, 1995. Springer-Verlag. Invited Lecture.

[BCF94] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of logic programs. *Theoretical Computer Science*, 124:297–328, 1994.

[Bez89] M. Bezem. Characterizing termination of logic programs with level mappings. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 69–80. The MIT Press, 1989.

[Bez93] M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 1 & 2(15):79–97, 1993.

[Cav89] L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the International Conference on Logic Programming*, pages 571–584. The MIT Press, 1989.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 8:69–116, 1987.

[DSF93] S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms: a missing link in automatic termination analysis. In *Proceedings of the International Logic Programming Symposium*, pages 420–436, 1993.

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

[LK92] Stefan Lüttringhaus-Kappel. *Laziness in Logic Programming*. PhD thesis, Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms Universität Bonn, Bonn, 1992.

[MNL90] K. Marriott, L. Naish, and J.L. Lassez. Most specific logic programs. *Annals of mathematics and artificial intelligence*, 1(2), 1990.

[MT95] E. Marchiori and F. Teusink. Compositional proof methods for logic programs with dynamic scheduling. Technical report, CWI, Amsterdam, The Netherlands, 1995. to appear.

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

[TJ86] J. Thom and J.Zobel. NU-prolog reference manual. Technical Report 86/10, University of Melbourne, 1986.

[Vie89] L. Vieille. Recursive query processing: the power of logic. *Theoretical Computer Science*, 69(1):1–53, 1989.