Preventing Instantiation Errors and Loops for Logic Programs with Several Modes Using `block` Declarations*

Jan–Georg Smaus†‡ Pat Hill Andy King

Extended abstract

1 Introduction

Delay declarations are provided in logic programming languages to allow for more flexible control, as opposed to the left-to-right selection rule of Prolog. An atom in a query is selected for resolution only when its arguments are instantiated to a specified degree. This is essential to prevent run-time errors produced by built-in predicates (e.g. `>/2`), and to ensure termination.

We assume that delay declarations are used to enable programs to run in several modes. Other authors have not explicitly made this assumption, but their work only becomes fully relevant under it, since assuming single-moded predicates, there is often no reason for using delay declarations in the first place.

Our contributions are: showing how type and instantiation errors related to built-in predicates (*built-ins*) can be prevented; showing when delay declarations for built-ins can be omitted completely; and proving termination.

For all of the above, we show that under realistic assumptions, `block` declarations, which declare that certain arguments of an atom must be at least *non-variable* before that atom can be selected, are sufficient. In SICStus [2], `block` declarations are efficiently implemented; the instantiation test has hardly any impact on performance. Thus such constructs are the most frequently used delay declarations in practice.

For arithmetic built-ins, we exploit that for numbers, being non-variable implies being ground, and show how to prevent *instantiation* and *type* errors. Sometimes, it is not even necessary to have any delay declarations at all for arithmetic built-ins.

Preventing a predicate from using its own output as input (*circular modes*) is crucial for termination [6]. We generalise this to multi-moded predicates.

Another source of loops [6] is *speculative output bindings*, i.e. bindings made before it is known that a solution exists. We propose two methods for dealing with this problem and thus proving (or ensuring) termination. Which method must be applied will depend on the program and on the mode being considered. The first method exploits that a program does not *make* any speculative bindings. The second method exploits that a program does not *use* any speculative bindings, by ensuring that no atom ever delays. In this latter case, any method for showing termination for programs without delay declarations can be applied.

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

†University of Kent at Canterbury, Canterbury, CT2 7NF, United Kingdom, j.g.smaus@ukc.ac.uk, telephone xx44/1227/827553, fax xx44/1227/762811.

‡Jan–Georg Smaus was supported by EPSRC Grant No. GR/K79635.

2 Preliminaries

We use the notation of [1, 3]. For the examples we use SICStus [2] notation. We recall some important notions. A syntactic object is called **linear** if every variable occurs in it at most once. A **flat** term is a variable or a term of the form $f(x_1, \dots, x_n)$, where $n \geq 0$ and the x_i are distinct variables. Atoms are denoted by h, queries by B, Q, R.

A `block` declaration for a predicate p/n is a set of atoms each of which has the form $p(b_1, \dots, b_n)$ where $b_i \in \{?, -\}$ for $i \in \{1, \dots, n\}$. A **program** consists of a set of clauses and a set of `block` declarations, one for each predicate defined by the clauses. If P is a program, an atom $p(t_1, \dots, t_n)$ is **selectable in** P if for each atom $p(b_1, \dots, b_n)$ in the `block` declaration for p, there is some $i \in \{1, \dots, n\}$ such that t_i is non-variable and $b_i = -$.

A **derivation step** for a program P is a pair $Q; R$, where Q and R are queries, Q is non-empty, and R is obtained by resolving Q (using an atom in Q, called the **selected atom**) with a clause in P. A **derivation** is a (possibly infinite) sequence $Q_0; Q_1; Q_2; \dots$, where each successive pair $Q_i; Q_{i+1}$ is a derivation step.

An **LD-derivation** is a derivation where the selected atom is always the leftmost atom in a query. A **delay-respecting derivation** for a program P is a derivation where the selected atom is always selectable in P. A derivation $Q_0; Q_1; Q_2; \dots$ is **left-based** if the following holds for all $i \geq 0$ where Q_i is non-empty: If $Q_0; \dots; Q_i$ is an LD-derivation and the leftmost atom in Q_i is selectable, then it is selected in $Q_i; Q_{i+1}$. To the best of our knowledge, derivations in most Prolog implementations meet this requirement.

For a predicate p/n, a **mode** is an atom $p(m_1, \dots, m_n)$, where $m_i \in \{I, O\}$ for $i \in \{1, \dots, n\}$. Positions with I are called **input positions**, and positions with O are called **output positions** of p. A **mode of a program** is a set containing one mode for each of its predicates. A program can have several modes, so whenever we refer to the input and output positions, this is always with respect to the particular mode which is clear from the context. To simplify the notation, an atom written as $p(\mathbf{s}, \mathbf{t})$ means: $\mathbf{s}$ is the vector of terms filling the input positions, and $\mathbf{t}$ is the vector of terms filling the output positions.

A **type** is a set of terms closed under substitution. A type is called **variable type** if it contains variables and **non-variable type** otherwise (there is only *one* variable type, containing all terms). A type is called **constant type** if it contains only (possibly infinitely many) constants. We write $t : T$ for "t is in type T". A type is associated with each predicate p/n and each of its argument positions, by writing $p(T_1, \dots, T_n)$ where $T_1, \dots, T_n$ are types. In the examples, we use the following types: *any* is the type containing all terms, *list* is the type of all (nil-terminated) lists, *num* the (constant) type of numbers, and *numlist* is the type of all number lists.

3 Permutations and Modes

This section defines some concepts needed in Sects. 4 and 5. In [1], each predicate has a single mode. The idea is that in a query, every piece of data is produced (i.e. output) before it is consumed (i.e. input), and every piece of data is produced only once. "Before" refers to the textual position.

We generalise this by associating, with each query and each clause in a program, a permutation π of the (body) atoms, such that the "reordered" program meets the requirements in [1]. For a different mode, the permutations would be different.[1] In examples, an actual permutation is written in the form $\langle \pi(1), \dots, \pi(n) \rangle$. Given a sequence $o_1, \dots, o_n$ and a permutation π, we write $\pi(o_1, \dots, o_n)$ for $o_{\pi^{-1}(1)}, \dots, o_{\pi^{-1}(n)}$, i.e. the sequence obtained by applying

[1] This explicit treatment of several modes becomes relevant in Subsec. 5.2, where the result depends on derivations being left-based. Elsewhere, we would not lose generality if we assumed, merely for notational simplicity, that the permutations are always the identity.

π to $o_1, \ldots, o_n$. These permutations are used to compare a program with the (theoretically) "reordered" program; it is not intended that the clauses in a programs are *actually* changed so that delay declarations are not required, since this would commit us to a single mode.

In a *nicely moded* query, a variable in an input position does not occur (textually) later in an output position, and each variable in an output position occurs only once.

Definition 3.1 [permutation nicely moded] Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query and π a permutation on $\{1, \ldots, n\}$. Q is **π-nicely moded** if $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a linear vector of terms and for all $i \in \{1, \ldots, n\}$

$$vars(\mathbf{s}_i) \cap \bigcup_{\pi(j) \geq \pi(i)} vars(\mathbf{t}_j) = \emptyset.$$

The query $\pi(Q)$ is a **nicely moded query corresponding to** Q.

The clause $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **π-nicely moded** if Q is π-nicely moded and $\mathbf{t}_0, \ldots, \mathbf{t}_n$ is a linear vector of terms. The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow \pi(Q)$ is a **nicely moded clause corresponding to** C.

A query (clause) is **permutation nicely moded** if it is π-nicely moded for some π. A program P is **permutation nicely moded** if all of its clauses are. A **nicely moded program corresponding to** P is a program obtained from P by replacing every clause C in P with a nicely moded clause corresponding to C.

Example 3.1

```
:- block permute(-,-).
permute([], []).
permute([U | X], Y) :-
  permute(X, Z),
  delete(U, Y, Z).

:- block delete(?,-,-).
delete(X,[X|Z],Z).
delete(X,[U|Y],[U|Z]) :- delete(X,Y,Z).
```

This program is nicely moded in mode $\{\texttt{permute}(I, O), \texttt{delete}(I, O, I)\}$, and permutation nicely moded in $\{\texttt{permute}(O, I), \texttt{delete}(O, I, O)\}$ (the second clause is $\langle 2, 1\rangle$-nicely moded).

Following [1], we show a persistence property of permutation nicely-modedness.

Lemma 3.1 Every SLD-resolvent of a permutation nicely moded query Q and a permutation nicely moded clause C is permutation nicely moded.

Proof sketch: For Q and $C = h \leftarrow B$, there are permutations π and ρ such that $\pi(Q)$ and $h \leftarrow \rho(B)$ are nicely moded. By [1, Lemma 11] every resolvent of $\pi(Q)$ and $h \leftarrow \rho(B)$ is nicely moded. Thus it follows that every resolvent of Q and C is permutation nicely moded. ∎

Permutation nicely-modedness could be used to show that the occur-check can be omitted.

Example 3.2

```
length(L,N) :- len_aux(L,0,N).

len_aux([],N,N).
len_aux([_|Xs],M,N) :-
  less(M,N),
  M2 is M + 1,
  len_aux(Xs,M2,N).

:- block less(?,-), less(-,?).
less(A,B) :- A < B.
```

This program is permutation nicely moded for mode $\{\texttt{length}(I, O), \texttt{len\_aux}(I, I, O)\}$ (the third clause is $\langle 3, 1, 2\rangle$-nicely moded). For mode $\{\texttt{length}(O, I), \texttt{len\_aux}(O, I, I)\}$, it is *not* permutation nicely moded, since the input in `len_aux([],N,N)` is not linear.

The second concept we generalise from [1] is *well-typedness*. As with modes, we assume that the types are given. In the examples, they will be the obvious ones.

Definition 3.2 [permutation well typed] Let $n \geq 0$ and π be a permutation such that $\pi(i) = i$ whenever $i \notin \{1, \ldots, n\}$. Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query. Suppose that $p_i(\mathbf{S}_i, \mathbf{T}_i)$[2] is the type of p_i for all $i \in \{1, \ldots, n\}$. Then Q is **π-well typed** if for all $i \in \{1, \ldots, n\}$ and every substitution σ

$$(\bigwedge_{\pi(j)<\pi(i)} \mathbf{t}_j\sigma : \mathbf{T}_j) \Rightarrow \mathbf{s}_i\sigma : \mathbf{S}_i. \qquad (*)$$

The clause $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$, where $p(\mathbf{T}_0, \mathbf{S}_{n+1})$ is the type of p, is **π-well typed** if $(*)$ holds for all $i \in \{1, \ldots, n+1\}$ and every substitution σ.

A **permutation well typed** query (clause, program) and a **well typed** query (clause, program) **corresponding to** a query (clause, program) are defined in analogy to Def. 3.1.

Example 3.3 Assume the types $\texttt{permute}(list, list)$, $\texttt{delete}(any, list, list)$. The program in Ex. 3.1 is well typed for mode $\{\texttt{permute}(I, O), \texttt{delete}(I, O, I)\}$, and permutation well typed for mode $\{\texttt{permute}(O, I), \texttt{delete}(O, I, O)\}$, with the same permutations as Ex. 3.1. The same holds assuming types $\texttt{permute}(numlist, numlist)$, $\texttt{delete}(num, numlist, numlist)$.

The following lemma is shown as Lemma 3.1, using [1, Lemma 23].

Lemma 3.2 Every SLD-resolvent of a permutation well typed query and a permutation well typed clause is permutation well typed.

Permutation well-typedness could be used to show that no derivation flounders.

4 Errors related to built-ins

Some built-ins produce an error if an argument has a wrong type. E.g. `X is foo` results in a type error. Some built-ins produce an error if certain arguments are insufficiently instantiated. E.g. `X is V` results in an instantiation error. We take two approaches to ensure freedom from instantiation errors and type errors. Under certain circumstances, we even show that no delay declarations are needed at all. For different programs, different approaches are applicable.

4.1 Exploiting constant types

The first approach aims at preventing instantiation *and type* errors for built-ins which require arguments to be ground, in particular arithmetic built-ins. It is proposed in [1] to equip these predicates with delay declarations such that they are only executed when the input is ground. The advantage is that one can reason about arbitrary arithmetic expressions, e.g. `quicksort([1+1,3-8],M)`. The disadvantage is that `block` declarations cannot be used. In contrast, we assume that the type of arithmetic built-ins is the constant type *num*, rather than arithmetic *expressions*. Then we show that `block` declarations are sufficient. The following lemma is a straightforward variation of [1, Lemma 27].

[2] $\mathbf{S}_i$ and $\mathbf{T}_i$ are the vectors of types of the input and output arguments, respectively.

Lemma 4.1 Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a π-well typed query, where $p_i(\mathbf{S}_i, \mathbf{T}_i)$ is the type of p_i for all $i \in \{1, \ldots, n\}$. Assume for some $i \in \{1, \ldots, n\}$, $\mathbf{S}_i$ is a vector of constant types, $\mathbf{s}_i$ is a vector of non-variable terms, and there is a substitution θ such that $\mathbf{t}_j\theta : \mathbf{T}_j$ for all j with $\pi(j) < \pi(i)$. Then $\mathbf{s}_i$ is correctly typed (and thus ground).
Proof: By Def. 3.2, $\mathbf{s}_i\theta$ is correctly typed, and thus a vector of constants. Since $\mathbf{s}_i$ is already a vector of non-variable terms, it follows that $\mathbf{s}_i$ is a vector of constants and thus $\mathbf{s}_i\theta = \mathbf{s}_i$. Thus $\mathbf{s}_i$ is ground and correctly typed. ■

We define **permutation simply-typedness**. In a permutation simply typed query, it is always possible to instantiate the output arguments so they are correctly typed.

Definition 4.1 [permutation simply typed] A query $p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is **permutation simply typed** if for some permutation π, it is π-nicely moded and π-well typed, and $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a vector of *variables*.

A clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is **permutation simply typed** if for some permutation π, it is π-nicely moded and π-well typed, $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a vector of *variables*, and $\mathbf{t}_0$ has a variable in each position of variable type and a flat term in each position of non-variable type.

A program is **permutation simply typed** if all its clauses are. A **simply typed** query (clause, program) **corresponding to** a query (clause, program) is defined in analogy to Def. 3.1.

Lemma 4.2 Every SLD-resolvent of a permutation simply typed query and a permutation simply typed clause, where the selected atom is non-variable in all input positions of non-variable type[3], is permutation simply typed.
Proof idea: By Lemmas 3.1 and 3.2, the resolvent is permutation nicely moded and permutation well typed. The linearity of the output positions in the query and the input positions in the clause implies that unifying the clause head with an atom in the query does not instantiate any output positions, except the output positions in the selected atom. This implies that the output arguments in the resolvent are again a linear vector of variables. ■

By Def. 4.1, for every permutation simply typed query Q, there is a θ such that $Q\theta$ is correctly typed in its output positions. Thus by Lemma 4.1, if a program is permutation simply typed, where the arithmetic built-ins have type *num* in all input positions, then it is enough to have `block` declarations such that these built-ins are only selected when the input positions are non-variable. Groundness is implied.

Example 4.1 The following is an excerpt of a version of `quicksort`, where the missing clauses are defined as usual, and `leq` is defined in analogy to Ex. 3.2.

```
:- block qs(-,-).
qs([X|Xs],Ys) :-
  append(As2,[X|Bs2],Ys),
  part(Xs,X,As,Bs),
  qs(As,As2),
  qs(Bs,Bs2).

:- block part(?,-,?,?), part(-,?,-,?), part(-,?,?,-).
part([X|Xs],C,[X|As],Bs):-
  leq(X,C),
  part(Xs,C,As,Bs).
```

The program is permutation simply typed wrt. to the obvious types for $\{$`qs`(I, O), `append`(I, I, O), `part`$(I, I, O, O)\}$, and thus there are no instantiation errors. The program is not permutation simply typed for $\{$`qs`(O, I), `append`(O, O, I), `part`$(O, I, I, I)\}$, because of `[X|Bs2]` in an output position.

[3] This is similar to "the delay declarations imply matching" [1].

4.2 Atomic positions

Sometimes, when the above method does not work because a program is not permutation simply typed, it is still possible to show absence of instantiation errors for arithmetic predicates. The idea is to declare certain argument positions to be *atomic*, which means that they can only be ground or free, but not partially instantiated. For atomic positions, it is sufficient to have a `block` declaration such that a predicate delays until the argument in this position is non-variable. It is then automatically ground. Typically one would declare that the atomic positions are the positions of type *num*.

Definition 4.2 [respects atomic positions] A query (clause) **respects atomic positions** if each term in an atomic position is ground or a variable which *only* occurs in atomic positions.

Sometimes we have to interpret the argument positions of a built-in in an unconventional way for a program to respect atomic positions. E.g. in Ex. 3.2, we have to regard the atom `M2 is M + 1` as an atom with three arguments: `M2`, `M`, and `1`.

Lemma 4.3 Every SLD-resolvent between a clause C and a query Q that both respect atomic positions, respects atomic positions.
Proof idea: Since atomic and non-atomic positions do not share variables, an atomic position can never be instantiated to a term other than a variable or constant. ∎

This works for programs where the arithmetic arguments are variable-disjoint from any other arguments, such as Ex. 3.2. Declaring all arithmetic arguments to be atomic, the program respects atomic positions. The `block` declaration on the built-in `<` is realised with an auxiliary predicate `less`. Note that no auxiliary predicate, and thus no `block` declaration, is introduced for `is`. Since in all modes, the input for `is` comes from the clause head rather than from another body atom, it is sufficient to have a `block` declaration for the predicate of the head. This is formalised in the following lemma.

Lemma 4.4 Let P be a program which respects atomic positions, $\mathcal{B}$ a set of built-ins whose input positions are all atomic, and $\mathcal{P}$ the set of all other predicates used in P. Suppose an atom using a predicate in $\mathcal{P}$ is selectable only if it is non-variable in its atomic input positions, and that for each clause $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$, if $p_i \in \mathcal{B}$, then the variables in $\mathbf{s}_i$ are a subset of the variables in the atomic positions in $\mathbf{t}_0$. Let Q be a query respecting atomic positions, where each atom using a predicate in $\mathcal{B}$ has constants in its input positions.

Let ξ be a delay-respecting derivation for Q. Then in all queries in ξ, an atom using a predicate in $\mathcal{B}$ is always ground in its input positions.

5 Termination

We always assume that termination for the corresponding nicely moded (or well typed) programs has been shown by some existing method for LD-derivations. Our approach is simple, and the conditions are easy to check. Termination can be proven for Ex. 5.1, but not for Ex. 4.1. There is a more sophisticated method [7], but it requires more complex checks.

Example 5.1 The query `permute(V,[1])` (Ex. 3.1) loops because `delete` produces a *speculative output binding* [6]: The output variable `Y` is bound before it is known that this binding will never have to be undone. Assuming left-based derivations, termination in both modes can be ensured by replacing the second clause with

```
permute([U | X1], Y) :-
  delete(U, Y, Z),
  permute(X1, Z).
```

This technique can be described as *putting recursive calls last* [6]. To explain termination, we have to apply a different reasoning for the different modes. In mode `permute(`I, O`)`, `delete` is used in mode `delete`(I, O, I), and in this mode it does not *make* speculative bindings.

In mode `permute(`O, I`)`, the speculative output is produced **textually before** it is consumed (used as input). This means that the consumer has to wait until the producer has completed (undone the speculative binding). The program does not *use* speculative bindings.

5.1 Termination by not making speculative bindings

Definition 5.1 [non-speculative] A permutation simply typed program P is **non-speculative** if

(a) every permutation simply typed atom (i.e. query of length 1) using a predicate in P is unifiable with some clause head in P, and

(b) an atom in a permutation simply typed query is selectable in P if and only if all input positions of non-variable type are non-variable.

Example 5.2 Both versions of the `permute` program, assuming any type given in Ex. 3.3, are non-speculative in mode {`permute`(I, O), `delete`(I, O, I)}, but are *not* non-speculative in mode {`permute`(O, I), `delete`(O, I, O)}, because the atom `delete(A,[],B)` is not unifiable with any clause head.

A delay-respecting derivation for a non-speculative program P and a permutation simply typed query can neither flounder nor fail. However it could still be infinite. The following lemma says that this can only happen if the simply typed program corresponding to P (Def. 4.1) also has an infinite (LD) derivation for this query.

Lemma 5.1 Let P be a non-speculative program and P' the simply typed program corresponding to P. Let Q be a permutation simply typed query and Q' the simply typed query corresponding to Q. If there is an infinite delay-respecting derivation for $P \cup \{Q\}$, then there is an infinite LD-derivation for $P' \cup \{Q'\}$.
Proof sketch: Let ξ be an infinite delay-respecting derivation for $P \cup \{Q\}$. Construct an LD-derivation ξ' for $P' \cup \{Q'\}$ which uses the "same" clauses as ξ where possible, and an arbitrary clause otherwise. Since every selected atom matches at least one clause head, the only reason why it might not always be possible is that in ξ, an atom might never be resolved. It turns out that by this construction, ξ' is infinite. ∎

Lemma 5.1 says that for non-speculative programs, atom order in clause bodies is irrelevant for termination.

5.2 Termination by not using speculative bindings

In LD-derivations, speculative bindings are never used [6]. Assuming left-based derivations, all derivations are LD-derivations, provided the leftmost atom in a query is always selectable. This immediately implies the following lemma.

Lemma 5.2 Assume left-based derivations, and let Q be a well typed query and P a well typed program such that an atom is selectable if its input positions of non-variable type are non-variable. Then every derivation for $P \cup \{Q\}$ is an LD-derivation.

By the above lemma, all derivations for `permute`(O, I) are finite. A comparison of Exs. 3.1 and 5.1 makes it clear that no termination guarantees can be given without such strong assumptions about the selection rule.

6 Discussion and Related Work

We have shown methods of preventing instantiation and type errors related to built-ins, and ensuring termination, for logic programs with `block` declarations.

It is often justified not to have any delay declarations for (arithmetic) built-ins, and even if delay declarations are required, `block` declarations are often sufficient. This is useful because it aims at the way arithmetic built-ins are used in practice: it is awkward having to introduce auxiliary predicates to implement delay declarations for built-ins.

For proving termination, we have presented two methods based on not *making* and not *using* speculative bindings, respectively. For Ex. 5.1, it turns out that in one mode, the first method applies, and in the second mode, the other method applies. We envisage a program development tool which would help a programmer to verify the conditions and to reorder atoms in clause bodies to ensure that one of the methods applies for each mode.

This work was inspired by [1]. For arithmetic built-ins, [1] requires declarations which delay an atom until the arguments are ground. Such declarations are usually implemented not as efficiently as `block` declarations. Little attention is given to termination, proposing a method limited to deterministic programs.

The good intuitive explanations for loops in [6] guided our search for further ideas and their formalisation. To ensure termination, some heuristics are proposed, without formal proof. Predicates have a single mode, suggesting that multiple modes, if required, should be achieved by having several versions of a predicate.[4] This bears the question: Why, for the mentioned examples `permute` and `quicksort`, if only one mode is considered, would one want to use delay declarations? In our opinion, the discussion on termination and delay declarations only becomes fully relevant when several modes are assumed.

In [4], a method is proposed to generate control automatically to ensure termination, giving practical evidence that control declarations were generated for many programs. The method assumes arbitrary delay-respecting derivations and hence does not work for programs where termination depends on *left-based* derivations.

The results of [5] are not comparable to ours because they assume a *local selection rule* (always selects an atom which was introduced in the most recent resolution step).

References

[1] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *AMAST'95*, LNCS, Berlin, 1995. Springer Verlag. Invited Lecture.

[2] Intelligent Systems Laboratory, SICS, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 1997. `http://www.sics.se/isl/sicstus/sicstus_toc.html`.

[3] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[4] S. Lüttringhaus-Kappel. Control generation for logic programs. In D. Warren, editor, *Proceedings of ICLP*, pages 478–495. MIT Press, 1993.

[5] E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. Lloyd, editor, *Proceedings of ILPS*, pages 447–461. MIT Press, 1995.

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

[7] J.-G. Smaus, P. M. Hill, and A. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamadessi, editor, *Proceedings of PLILP/ALP*, LNCS. Springer Verlag, 1998.

[8] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, November 1996.

[4] Mercury [8] takes the same approach, and the versions are all generated by the compiler.