Termination of Logic Programs with `block` Declarations Running in Several Modes

Jan–Georg Smaus[1], Pat Hill[2], and Andy King[1]

[1] University of Kent at Canterbury, Canterbury, CT2 7NF, United Kingdom, {j.g.smaus, a.m.king}@ukc.ac.uk
[2] University of Leeds, Leeds, LS2 9JT, United Kingdom, hill@scs.leeds.ac.uk

Abstract We show how termination of logic programs with delay declarations can be proven. Three features are distinctive of this work: (a) we assume that predicates can be used in several modes; (b) we show that `block` declarations, which are a very simple delay construct, are sufficient; (c) we take the selection rule into account, assuming it to be as in most Prolog implementations. Our method is based on identifying the so-called *robust* predicates, for which the textual position of an atom using this predicate is irrelevant. The method can be used to verify existing programs, and to assist in writing new programs. As a byproduct, we also show how programs can be proven to be free from occur-check and floundering.

1 Introduction

Delay declarations are provided in several logic programming languages to allow for more user-defined control [7, 8, 18] as opposed to the standard left-to-right selection rule. An atom in a query is selected for resolution only if its arguments are instantiated to a specified degree.

In this paper we present a method of ensuring termination of programs with delay declarations. As far as possible, we translate the problem to showing termination for a corresponding program with ordinary left-to-right execution. We assume that for the corresponding program, termination has been shown using some existing technique [1].

Three distinctive features of this work make its contribution: (a) it is assumed that procedures may run in more than one mode; (b) we concentrate on `block` declarations, which are a particularly simple and efficient delay construct; (c) the selection rule is taken into account.

(a) Apart from the test-and-generate paradigm (coroutining) [15], allowing procedures to run in more than one mode is probably the most important application of delay declarations. Although other authors have not explicitly assumed multiple modes, their theory and examples only become fully relevant under that assumption. Whether this is a better approach than generating multiple versions of each predicate [18] is an ongoing discussion [6].

(b) The `block` declarations declare that certain arguments of an atom must be *non-variable* before that atom can be selected. Insufficiently instantiated atoms are delayed. As demonstrated in SICStus [8], `block` declarations can be efficiently implemented; the test whether the arguments are non-variable has negligible impact on performance. Termination clearly depends on the instantiation of the arguments of the query. For example, the `append` predicate has infinitely many answers when called with uninstantiated arguments and therefore does not terminate, but it terminates when either the first *or* the third argument is a list of bounded length. Although it cannot be tested in a single step *which* of these arguments is a list of bounded length, `block` declarations are still sufficient.

(c) The property of termination may critically depend on the selection rule, that is the rule which determines, for a derivation, the order in which atoms are selected. We assume that derivations are *left-based*, which are derivations where (allowing for some exceptions, concerning the execution order of two literals woken up simultaneously) the left-most non-delayed atom is selected. This is intended to model derivations in the common implementations of Prolog with `block` declarations. Other authors have avoided the issue by abstracting from a particular selection rule [2, 10]; considering left-based selection rules on a heuristic basis [15]; or making the very restrictive assumption of *local selection rules* [11].

Circular modes (when a predicate uses its own output as input) and *speculative output bindings* (bindings made before it is known that a solution exists) are known sources of loops [15]. We develop this explanation further by identifying predicates which have the undesirable property of looping when they are called with *insufficient* (that is, non-variable but still insufficiently instantiated) input. For instance, the query `permute(A,[1|B])` loops, although the query `permute(A,[1,2])` terminates. The idea of our method for proving termination is that, for such predicates, calls with insufficient input should never arise. This can be ensured by appropriate ordering of atoms in the clause bodies. This actually works in several modes, provided not too many predicates have this undesirable property.

This paper is organised as follows. The next section defines some essential concepts and notations. Sect. 3 introduces some concepts needed later, which are also useful for proving programs free from occur-check and floundering. Sect. 4 is about termination. Sect. 5 investigates related work. Sect. 6 concludes with a summary and a look at ongoing and future work.

2 Essential Concepts and Notations

We base the notation on [2, 9]. For the examples we use SICStus notation [8]. The set of variables in a syntactic object o is denoted by $vars(o)$. A syntactic object is **linear** if every variable occurs in it at most once. A **flat** term is a variable or a term $f(x_1, \ldots, x_n)$, where $n \geq 0$ and the x_i are distinct variables. The **domain** of a substitution σ is $dom(\sigma) = \{x \mid x\sigma \neq x\}$.

For a predicate p/n, a **mode** is an atom $p(m_1, \ldots, m_n)$, where $m_i \in \{I, O\}$ for $i \in \{1, \ldots, n\}$. 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 of modes, 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: $\mathbf{s}$ and $\mathbf{t}$ are the vectors of terms filling the input and output positions, respectively.

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. In the examples, we use the following types: *any* is the type containing all terms, *list* is the type of all (nil-terminated) lists, *int* the type of integers, and *il* is the type of all integer lists. We write $t : T$ for "t is in type T". It is assumed that each argument position of each predicate p/n has a type associated with it. These types are indicated by writing the atom $p(T_1, \ldots, T_n)$ where $T_1, \ldots, T_n$ are types. The type of a program is a set of such atoms, one for each predicate. A term t is **typeable wrt.** T if there is a substitution θ such that $t\theta : T$. A term t occurring in an atom in some position is **typeable** if it is typeable wrt. the type of that position.

A `block` declaration [8] for a predicate p/n is a set of atoms of the form $p(b_1, \ldots, b_n)$, where $b_i \in \{?, -\}$ for $i \in \{1, \ldots, n\}$. 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 $p(t_1, \ldots, t_n)$ is **selectable in** P if for each atom $p(b_1, \ldots, b_n)$ in the `block` declaration for p, there is some $i \in \{1, \ldots, n\}$ such that t_i is non-variable and $b_i = -$.

A **query** is a finite sequence of atoms. A **derivation step** for a program P is a pair $\langle Q, \theta \rangle; \langle R, \theta\sigma \rangle$, where $Q = Q_1, a, Q_2$ and $R = Q_1, B, Q_2$ are queries; θ is a substitution; a an atom; $h \leftarrow B$ (a variant of) a clause in P and σ the most general unifier of $a\theta$ and h. We call $a\theta$ the **selected atom** and $R\theta\sigma$ the **resolvent** of $Q\theta$ and $h \leftarrow B$.

A **derivation** ξ **for** a program P is a sequence $\langle Q_0, \theta_0 \rangle; \langle Q_1, \theta_1 \rangle; \ldots$, where $\theta_0 = \emptyset$ and each successive pair $\langle Q_i, \theta_i \rangle; \langle Q_{i+1}, \theta_{i+1} \rangle$ in ξ is a derivation step. Alternatively, we also say that ξ is a **derivation of** $P \cup \{Q_0\}$. We also denote ξ by $Q_0; Q_1\theta_1; \ldots$. A derivation is an **LD-derivation** if 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. We say that it **flounders** if it ends with a non-empty query where no atom is selectable.

If Q, a, R; $(Q, B, R)\theta$ is a step in a derivation, then each atom in $B\theta$ is a **direct descendant** of a, and $b\theta$ is a **direct descendant** of b for all $b \in Q, R$. We say b is a **descendant of** a if (b, a) is in the reflexive, transitive closure of the relation *is a direct descendant*. The descendants of a *set* of atoms are defined in the obvious way. If, for a derivation $\ldots Q; \ldots; Q'; Q'' \ldots$, the selected atom in $Q'; Q''$ is a descendant of an atom a in Q, then $Q'; Q''$ is called an a**-step**.

Consider a delay-respecting derivation $Q_0; \ldots; Q_i; Q_{i+1}$, where $Q_i = R_1, a, R_2$, and R_1 contains no selectable atom, and a is *not* the selected atom in $Q_i; Q_{i+1}$. Then a is **delayed** in $Q_i; Q_{i+1}$. An atom is **waiting** if it is the descendant of a delayed atom. A delay-respecting derivation $Q_0; Q_1 \ldots$ is **left-**

based if in each Q_i, a non-waiting atom is selected only if there is no selectable atom to the left of it in Q_i.

3 Permutations and Modes

In [2], the concepts of *nicely moded* and *well typed* are introduced, assuming that each predicate has a single mode. They are used to show that the occur-check can safely be omitted and that derivations do not flounder. 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. Here "before" refers to the textual position in a query.

We generalise these concepts and results by considering a permutation of the atoms in each clause body in a program (and in each query), such that an LD-derivation for the reordered program is automatically delay-respecting, and thus, `block` declarations are effectively unnecessary. These permutations are used to compare a program with the (theoretically) reordered program; it is not intended that the program is *actually* changed. Since the permutations are different in each mode, this would commit the program to a particular mode.

3.1 Permutation Nicely Moded Programs

In a *nicely moded* query, a variable occurring in an input position does not occur later in an output position, and each variable in an output position occurs only once. We generalise this to *permutation nicely moded.*

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[1] $\pi(Q)$ 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.

Note that in the clause head, the letter t is used for input and s is used for output, whereas in the body atoms it is vice versa.

[1] Given a sequence $o_1, \ldots, o_n$, we write $\pi(o_1, \ldots, o_n)$ for $o_{\pi^{-1}(1)}, \ldots, o_{\pi^{-1}(n)}$, i.e. the sequence obtained by applying π to $o_1, \ldots, o_n$.

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).
```

In mode $\{\texttt{permute}(I,O),\texttt{delete}(I,O,I)\}$, this program is nicely moded. In mode $\{\texttt{permute}(O,I),\texttt{delete}(O,I,O)\}$, it is permutation nicely moded, since the second clause for `permute` is $\langle 2,1\rangle$-nicely moded, and the other clauses are nicely moded.

Note that the problem of *finding* a mode for a program so that it is nicely moded is considered in [4]. We are not concerned with this here.

We show that there is a persistence property for permutation nicely-modedness similar to that for nicely-modedness in [2].

Lemma 3.1. Every resolvent of a permutation nicely moded query Q and a permutation nicely moded clause C, where $vars(Q) \cap vars(C) = \emptyset$, is permutation nicely moded.

Proof. Let $Q = a_1,\ldots,a_n$ be a π-nicely moded query and $h \leftarrow b_1,\ldots,b_m$ be a ρ-nicely moded clause, and suppose for some $k \in \{1,\ldots,n\}$, h and a_k are unifiable with unifier θ. By Def. 3.1, $a_{\pi^{-1}(1)},\ldots,a_{\pi^{-1}(n)}$ and $h \leftarrow b_{\rho^{-1}(1)},\ldots,b_{\rho^{-1}(m)}$ are nicely moded. Thus by [2, Lemma 11][2]

$$a_{\pi^{-1}(1)},\ldots,a_{\pi^{-1}(\pi(k)-1)},b_{\rho^{-1}(1)},\ldots,b_{\rho^{-1}(m)},a_{\pi^{-1}(\pi(k)+1)},\ldots,a_{\pi^{-1}(n)}\ \theta$$

is nicely moded. This implies that $a_1,\ldots,a_{k-1},b_1,\ldots,b_m,a_{k+1},\ldots,a_n\ \ \theta$ is ϱ-nicely moded, where $\varrho(i)$ is defined as:

$$\begin{array}{ll}
\pi(i) & \text{if } i<k,\ \pi(i)<\pi(k)\\
\pi(i)+m-1 & \text{if } i<k,\ \pi(i)>\pi(k)\\
\pi(k)-1+\rho(i-k+1) & \text{if } k\le i\le k+m-1\\
\pi(i-m+1) & \text{if } k+m\le i\le n+m-1,\ \pi(i-m+1)<\pi(k)\\
\pi(i-m+1)+m-1 & \text{if } k+m\le i\le n+m-1,\ \pi(i-m+1)>\pi(k)
\end{array}$$

□

Fig. 1 illustrates ϱ when $Q = a_1,a_2,a_3,a_4$, $\pi = \langle 4,3,1,2\rangle$, $C = h \leftarrow b_1,b_2$, $\rho = \langle 2,1\rangle$, and $k = 2$. Thus $\varrho = \langle 5,4,3,1,2\rangle$. The following corollary generalises this from a single derivation step to derivations.

[2] Unlike [2], we included the condition that $\mathbf{t}_0$ is linear in Def. 3.1.

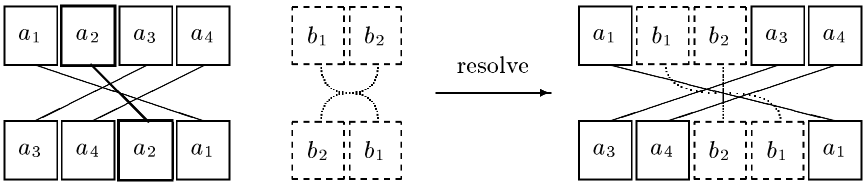

Figure1. The permutation ϱ for the resolvent

Corollary 3.2. Let P be a permutation nicely moded program, $Q = a_1, \ldots, a_n$ be a π-nicely moded query and $i, j \in \{1, \ldots, n\}$ such that $\pi(i) < \pi(j)$. Let $Q; \ldots; R$ be a derivation for P and suppose $R = b_1, \ldots, b_m$ is ρ-nicely moded. If for some $k, l \in \{1, \ldots, m\}$, b_k is a descendant of a_i and b_l is a descendant of a_j, then $\rho(k) < \rho(l)$. (*Proof [17]*)

As an aside, we now use permutation nicely-modedness to show when the occur-check can safely be omitted.

Definition 3.2. A derivation is called **occur-check free** [2, 3] if no execution of the Martelli-Montanari unification algorithm [13] performed during this derivation ends with a system of term equations including an equation $x = t$, where x is not t, but x occurs in t.

If P and Q are nicely moded, then all derivations of $P \cup \{Q\}$ are occur-check free [2, Thm. 13]. The following theorem is a trivial consequence of this and Lemma 3.1.

Theorem 3.3 (occur check). Let P and Q be permutation nicely moded. Then all derivations of $P \cup \{Q\}$ are occur-check free.

3.2 Permutation Well Typed Programs

To show that derivations do not flounder, [2] defines *well-typedness*, which is a generalisation of a simpler concept called *well-modedness*. The idea is that given a query H, a, F, if H is resolved away, then a becomes sufficiently instantiated to be selected. As with the modes, we assume that the types are given. In the examples, they will be the obvious ones.

Definition 3.3 (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, where $p_i(\mathbf{S}_i, \mathbf{T}_i)$[3] 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

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

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

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 3.2. Consider Ex. 3.1 and assume the type $\{$`permute`$(list, list)$, `delete`$(any, list, list)\}$. The program is well typed for mode $\{$`permute`(I, O), `delete`$(I, O, I)\}$, and permutation well typed for mode $\{$`permute`(O, I), `delete`$(O, I, O)\}$, with the same permutations as in Ex. 3.1. The same holds assuming type $\{$`permute`(il, il), `delete`$(int, il, il)\}$.

We now give a statement analogous to Lemma 3.1. The proof is like that of Lemma 3.1, using Lemma 23 instead of 11 in [2].

Lemma 3.4. Every resolvent of a permutation well typed query Q and a permutation well typed clause C, where $vars(Q) \cap vars(C) = \emptyset$, is permutation well typed.

Theorem 3.5. Let P be a permutation well typed program and Q be a permutation well typed query. Assume that an atom is selectable if it is non-variable in all input positions of non-variable type. Then no delay-respecting derivation of $P \cup \{Q\}$ flounders. *(Proof [17])*

For the program in Ex. 3.2, the above lemma shows that no permutation well typed query can flounder.

4 Termination

So far we have introduced two useful concepts of “modedness” and “typedness”. In this section, we will build on these to show termination.

We are interested in termination in the sense that *all* derivations of a query are finite. Therefore the clause order in a program is irrelevant. Furthermore, we are concerned with how delay declarations can affect the termination of a program. Thus it is assumed that termination for the corresponding nicely moded and well typed programs has been shown by some existing method for LD-derivations [1]. We first give some examples to illustrate the issues.

Example 4.1. The `permute` predicate (Ex. 3.1) loops for the query `permute(V,[1])` because `delete` produces a *speculative output binding* [15]: 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 heuristic is called *putting recursive calls last* [14]. The example suggests that one cannot give reasonable termination guarantees without making such strong assumptions about the selection rule.

However, the heuristic of putting recursive calls last cannot explain the appropriate atom order in the following example.

Example 4.2. This program for the n-queens problem shows an application of `block` declarations other than enabling multiple modes: implementing the test-and-generate paradigm. Here `permute` is defined as in Ex. 4.1.

```
nqueens(N,Sol) :-
  sequence(N,Seq),
  safe(Sol),
  permute(Sol,Seq).

:- block sequence(-,?).
sequence(0,[]).
sequence(N,[N|Seq]):-
  0 < N,
  N1 is N-1,
  sequence(N1,Seq).

:- block safe(-).
safe([]).
safe([N|Ns]) :-
  safe_aux(Ns,1,N),
  safe(Ns).

:- block safe_aux(-,?,?), safe_aux(?,-,?),
                          safe_aux(?,?,-).
safe_aux([],_,_).
safe_aux([M|Ms],Dist,N) :-
  no_diag(N,M,Dist),
  Dist2 is Dist+1,
  safe_aux(Ms,Dist2,N).

:- block no_diag(-,?,?), no_diag(?,-,?).
no_diag(N,M,Dist) :-
  Dist =\= N-M,
  Dist =\= M-N.
```

With the mode $\{\mathtt{nqueens}(I,O), \mathtt{safe}(I), \mathtt{sequence}(I,O), \mathtt{permute}(O,I), \mathtt{is}(O,I), {<}(I,I)\}$ and the type $\{\mathtt{nqueens}(int, il), \mathtt{sequence}(int, il), \mathtt{safe}(il), \mathtt{permute}(il, il)\}$, the first clause is $\langle 1,3,2\rangle$-nicely moded and $\langle 1,3,2\rangle$-well typed. Moreover, the query `nqueens(4,Sol)` terminates.

However, if in the first clause, the atom order is changed by moving `sequence(N,Seq)` to the end, then `nqueens(4,Sol)` loops. This is because resolving `sequence(4,Seq)` with the second clause for `sequence` makes a (not speculative!) binding which triggers the call `permute(Sol,[4|T])`. This call results in a loop. Note that `[4|T]`, although non-variable, is insufficiently instantiated for `permute(Sol,[4|T])` to be correctly typed in its input position: `permute` is called with *insufficient input*.

To ensure termination, atoms in a clause body that loop when called with insufficient input should be placed so that all atoms which produce the input for these atoms occur textually earlier.

In the following three subsections, we first define *permutation robustly typed*, which is an elementary property a program must have for our method to be applicable. We then identify the *robust* predicates, which terminate for every delay-respecting selection rule. Finally, we show how predicates which are not robust must be placed in clause bodies to ensure termination.

4.1 Preventing Instantiation of Own Input

A prerequisite of our formalism is that no call arising in a derivation can ever instantiate its own input arguments.

Example 4.3. Consider the following version[4] of delete(O, I, O).

```
:- block delete(?,-,-).
delete(X,[U|[H|T]],[U|Z]) :-
  delete(X,[H|T],Z).
delete(A,[A|B],B).
```

Consider the query `delete(A,L,R), delete(B,[1,2],L)`. The second atom produces `L`, which is used by the first atom as input. The query loops, since the second atom partially binds `L`, which wakes up the first atom, which then instantiates `L` further (i.e. the call instantiates its own input), resulting in a recursive call to `delete`, and so forth.

To prevent a call from instantiating its input, the `block` declarations must enforce that an atom is *only* selected if all input positions of non-variable type are non-variable. As the previous example shows, this is not enough. It also has to be ensured that each input argument in the clause head is flat (which the clause head `delete(X,[U|[H|T]],[U|Z])` violates). The next example shows that even that is not enough.

Example 4.4. Consider the following program in mode p(I, O).

```
:- block p(-,?).
p(g(Y),Y).
```

A call to `p(g(X),3)` instantiates `X` to `3`, and thus instantiates its own input.

The easiest solution seems to be to require that the output positions in a query are always filled by variables. In mode p(I, O), the query `p(g(X),3)` should not arise, since its output is already instantiated. This is considered in [2] (*simply-modedness*). However, it is often too restrictive.

Example 4.5. The following is an excerpt from a version of `quicksort`.

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

For the mode {qs(O, I), append(O, O, I), partition(O, I, I, I)}, the non-variable term `[X|Bs2]` occurs in an output position.

[4] It is part of the *most specific program* [12] corresponding to Ex. 3.1, proposed in [15] to prevent looping for permute(O, I). However, it does not work. The query `permute(A,[1])` indeed terminates, but `permute(A,[1,2])` still loops.

In the sequel, we assume that a label *free* or *bound* is associated with each output position of each predicate. Non-variable terms in output positions in a query are allowed only in bound positions. The bound positions must be of non-variable type. As with assigning the mode and the type to a predicate, we do not propose a method of deciding which positions should be free or bound. In the examples however, an output position of a predicate p is bound if and only if there is a clause body with an atom using p, which has a non-variable term in that position.

For notational convenience, we use the notion of free and bound positions also for *input* positions. An input position is free if and only if it is of variable type. We denote the projection of a vector of arguments $\mathbf{r}$ onto its free positions as $\mathbf{r}^{\mathsf{f}}$, and the projection onto its bound positions as $\mathbf{r}^{\mathsf{b}}$.

Definition 4.1 (permutation robustly typed). Let π be a permutation such that $\pi(i) = i$ whenever $i \notin \{1, \ldots, n\}$. A query $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is **π-robustly typed** if it is π-nicely moded and π-well typed, $\mathbf{t}_1^{\mathsf{f}}, \ldots, \mathbf{t}_n^{\mathsf{f}}$ is a vector of variables, and $\mathbf{t}_1^{\mathsf{b}}, \ldots, \mathbf{t}_n^{\mathsf{b}}$ is a vector of flat typeable terms.

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **π-robustly typed** if it is π-nicely moded and π-well typed, and

1. $\mathbf{t}_0^{\mathsf{f}}, \ldots, \mathbf{t}_n^{\mathsf{f}}$ is a vector of variables, and $\mathbf{t}_0^{\mathsf{b}}, \ldots, \mathbf{t}_n^{\mathsf{b}}$ is a vector of flat typeable terms.
2. if a position in $\mathbf{s}_{n+1}^{\mathsf{b}}$ of type τ is filled with a variable x, then x also fills a position of type τ in $\mathbf{t}_0^{\mathsf{b}}, \ldots, \mathbf{t}_n^{\mathsf{b}}$.

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

Example 4.6. The `permute`-program of Ex. 4.1, for any of the types in Ex. 3.2, assuming all output positions are free, is robustly typed in mode `permute`(O, I) and permutation robustly typed in mode `permute`(I, O).

Consider Ex. 4.5 with the usual definition for the missing clauses, with type $\{\mathtt{qs}(il, il), \mathtt{append}(il, il, il), \mathtt{partition}(il, int, il, il)\}$. This program is permutation robustly typed in mode `qs`(O, I), assuming the second position of `append` is the only bound output position. It is also permutation robustly typed in mode `qs`(I, O), assuming that all output positions are free.

Definition 4.2 (input selectability). Let P be a permutation robustly typed program. P has **input selectability** if for every permutation robustly typed query Q, an atom in Q is selectable in P if and only if it is non-variable in all input positions of non-variable type.

Example 4.7. Consider `append`(O, O, I) where the second position is the only bound output position (Exs. 4.5, 4.6), and the `block` declaration is

```
:- block append(-,?,-), append(?,-,-).
```

This program has input selectability. $Q = \texttt{append(A,[B|Bs],[1])}$ is a permutation robustly typed query, and its only atom is selectable. The atom `append([],[],C)` is also selectable, although its input position is variable. This does not contradict Def. 4.2, since this atom cannot be an atom in a permutation robustly typed query with respect to mode $\texttt{append}(O, O, I)$.

Looking at Def. 4.1, one is tempted to think that it is best to associate the label *bound* with *all* output positions, because that would make the definition less restrictive. However, we require a program to have input selectability in each of its modes. Since input selectability is defined with respect to atoms in permutation robustly typed queries, and permutation robustly typed queries are defined with respect to given free and bound positions, it turns out that the choice of free and bound positions constrains the possible set of modes. For reasons of space, we cannot explain this in detail. Anyway, we have not encountered a case where a "natural" mode of a program was ruled out.

The following lemma shows a persistence property of permutation robustly typedness, and shows furthermore that a derivation step cannot instantiate the input arguments of the selected atom.

Lemma 4.1. Let P be a permutation robustly typed program with input selectability, $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a permutation robustly typed query and $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be a clause in P such that $vars(Q) \cap vars(C) = \emptyset$. Suppose $\langle Q, \emptyset \rangle; \langle R, \sigma \rangle$ is a derivation step with clause C and selected atom $p_k(\mathbf{s}_k, \mathbf{t}_k)$.

Then $R\sigma$ is permutation robustly typed, and $dom(\sigma) \cap vars(\mathbf{s}_k) = \emptyset$ and $vars(\mathbf{s}_k) \cap vars((\mathbf{v}_1, \ldots, \mathbf{v}_m)\sigma) = \emptyset$. *(Proof [17])*

4.2 Robust Predicates

In this subsection, derivations are not required to be left-based. Therefore we do not need to consider arbitrary permutations and we can, without loss of generality, assume that the programs and queries are robustly typed (rather than *permutation* robustly typed). This simplifies the notation. In Subsect. 4.3, we go back to allowing for arbitrary permutations.

Definition 4.3 (robust). A predicate p in a robustly typed program P is **robust** if, for each robustly typed query $p(\mathbf{s}, \mathbf{t})$, any delay-respecting derivation of $P \cup \{p(\mathbf{s}, \mathbf{t})\}$ is finite. An atom is **robust** if its predicate is.

This means that for queries consisting of robust atoms, termination does not depend on left-based derivations. Thus the *position* of a robust atom in a clause body or query does not affect termination. The following lemma says that a robust atom cannot proceed indefinitely unless it is repeatedly "fed" by some other atom.

Lemma 4.2. Let P be a robustly typed program with input selectability and F, a, H a robustly typed query where a is a robust atom. A delay-respecting derivation of $P \cup \{F, a, H\}$ can have infinitely many a-steps only if it has infinitely many b-steps, for some $b \in F$. *(Proof [17])*

The following lemma is a simple consequence and states that the robust atoms in a query on their own cannot produce an infinite derivation.

Lemma 4.3. Let P be a robustly typed program with input selectability and Q a robustly typed query. A delay-respecting derivation of $P \cup \{Q\}$ can be infinite only if there are infinitely many steps where a non-robust atom is resolved. *(Proof [17])*

For LD-derivations, termination proofs usually rely on some norm to measure the size of a term or atom $[1, 5]$. For a query F, a, H, the query F is resolved away before a is resolved, and thus a is sufficiently instantiated to be *bounded* with respect to the norm. In contrast, for arbitrary derivations, the decrease in argument size must be independent of the order in which atoms are selected. We assume a simple norm where a term is smaller than another term if it is a proper subterm. This method could be enhanced by considering other norms.

Example 4.8. Consider Ex. 4.2, where all arguments are input, and the type is $\{$`safe`(il), `safe_aux`(il, int, int), `no_diag`$(int, int, int)\}$. All delay-respecting derivations of a permutation robustly typed query `safe_aux`(l, n, m) terminate, because in the first argument of `safe_aux`, there is a strict decrease with respect to the "subterm" norm.

The following definition is adapted from [1].

Definition 4.4 (depends on). Let p, q be predicates in a program P. We say that p **refers to** q if there is a clause in P with p in its head and q in its body, and p **depends on** q (written $p \sqsupseteq q$) if (p, q) is in the reflexive, transitive closure of *refers to*. We write $p \sqsupset q$ if $p \sqsupseteq q$ and $q \not\sqsupseteq p$, and $p \approx q$ if $p \sqsupseteq q$ and $q \sqsupseteq p$.

To show robustness, one has to find argument positions, one for each predicate, such that there is a decrease in argument size in that position.

Definition 4.5 (decreasing clause). Assume that for each predicate p in a program P, there is a designated position called **decreasing position**. Let $C = q(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \dots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be a clause in P. Suppose that for each $\mu \in \{1, \dots, m\}$ where $q_\mu \approx q$, $q_\mu(\mathbf{u}_\mu, \mathbf{v}_\mu)$ has a variable in its decreasing position which is a proper subterm of the term in the decreasing position of $q(\mathbf{v}_0, \mathbf{u}_{m+1})$. Then C is **decreasing**.

To show that a predicate p is robust, we assume that all predicates q with $p \sqsupset q$ have already been shown to be robust.

Lemma 4.4. Let P be a robustly typed program with input selectability and p a predicate in P. Suppose that for each predicate q, where $p \sqsupseteq q$, either:

1. $p \sqsupset q$ and q is robust.
2. $p \approx q$ and each clause defining q is decreasing.

Then p is robust. *(Proof [17])*

Of course, a predicate in a permutation robustly typed program is not always robust, and so the technique given by the above lemma cannot always be applied. Often there is no decreasing position for a predicate.

Example 4.9. We demonstrate for Ex. 4.8 how Lemma 4.4 is used. Given that the built-in `=\=` terminates, it follows that `no_diag` is robust. We show that the second clause for `safe_aux` meets assumption 2 of Lemma 4.4. With the first position of `safe_aux` as decreasing position, the recursive call to `safe_aux` has `Ms` in the decreasing position, which is a proper subterm of `[M|Ms]`. Similar arguments can be applied for the other clauses, showing that `safe` and `safe_aux` are robust.

4.3 Well Fed Programs

So far we have shown for some predicates that all delay-respecting derivations of queries with these predicates terminate. As $\texttt{permute}(O, I)$ shows, this does not work for all predicates. In a program which uses such predicates, the selection rule must be taken into account. We assume left-based derivations. Consequently we now also give up the assumption, made to simplify the notation, that the clauses and query are robustly typed, rather than just *permutation* robustly typed. All statements from the previous subsection generalise in the obvious way.

A query is called *well fed* if each atom has been shown to be robust or occurs in such a position that all atoms which "feed" the atom occur earlier. Of course, since robustness is undecidable, we must assume a predicate to be non-robust if it has not been shown to be robust.

Definition 4.6 (well fed). Let P be a permutation robustly typed program. For a π-robustly typed query $p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$, an atom $p_i(\mathbf{s}_i, \mathbf{t}_i)$ is **well fed** if it is robust, or $\pi(j) < \pi(i)$ implies $j < i$ for all j. A π-robustly typed query (clause) is **well fed** if all of its (body) atoms are. P is **well fed** if all of its clauses are well fed and it has input selectability.

Example 4.10. The programs mentioned in Ex. 4.6 are well fed in the given modes. The program in Ex. 4.2 is well fed in the given mode. It is not well fed in mode $\{\texttt{nqueens}(O, I), \texttt{permute}(I, O), \texttt{sequence}(O, I), <(I, I), \texttt{is}(O, I)\}$, because it is not permutation nicely moded in this mode: in the second clause for `sequence`, `N1` occurs twice in an output position.

Lemma 4.5. Every resolvent of a well fed query Q and a well fed clause C, where $vars(Q) \cap vars(C) = \emptyset$, is well fed.

Proof. By obvious analogy, Corollary 3.2 also holds if *nicely moded* is replaced with *robustly typed*. The result then follows from Lemma 4.1. □

The following theorem reduces the problem of showing termination of left-based derivations for well fed programs to showing termination of LD-derivations for the corresponding robustly typed program.

Theorem 4.6. Let P and Q be a well fed program and query, and P' and Q' a robustly typed program and query corresponding to P and Q. If every LD-derivation of $P' \cup \{Q'\}$ is finite, then every left-based derivation of $P \cup \{Q\}$ is finite. *(Proof [17])*

Given that for the programs of Ex. 4.10, the corresponding robustly typed programs terminate for robustly typed queries, it follows from the above theorem that the former programs terminate for well fed queries.

5 Related Work

In using "modedness" and "typedness", we follow **Apt** and **Luitjes** [2], and also adopt their notation. Our results on occur-check freedom and non-floundering are straightforward variations of their results. For termination, they propose a method limited to deterministic programs.

Naish [15] gives excellent intuitive explanations why programs loop, which directed our own search for further ideas and their formalisation. To ensure termination, he proposes some heuristics, without any formal proof.

Predicates are assumed to have a single mode. Naish suggests that alternative modes should be achieved by multiple versions of a predicate.[5] However, under that assumption, why have delay declarations in the first place? For instance, in the mentioned example `permute`, if we only consider `permute`(O, I), then Ex. 4.1 does not loop for the plain reason that no atom ever delays, and thus the program behaves as if there were no delay declarations. In this case, the interpretation that one should "put recursive calls last" is misleading. If we only consider `permute`(I, O), then the version of Ex. 4.1 is much less efficient than Ex. 3.1. In short, the whole discussion on delay declarations makes little sense when only one mode is assumed.

Lüttringhaus-Kappel [10] proposes a method of generating control automatically, and has applied it successfully to many programs. However, rather than pursuing a formalisation of some intuitive understanding of why programs loop, and imposing appropriate restrictions on programs, he attempts a high degree of generality. This has certain disadvantages.

The method only finds *acceptable* delay declarations, ensuring that the most general selectable atoms have finite SLD-trees. What is required however are *safe* delay declarations, ensuring that *instances* of most general selectable atoms have finite SLD-trees. A *safe* program is a program for which every acceptable delay declaration is safe. No hint is given as to how it is shown that a program is safe. This is a missing link.

The delay declarations for some programs such as `quicksort` require an argument to be a nil-terminated list before an atom can be selected. As Lüttringhaus-Kappel points out himself, "in NU-Prolog [*or SICStus*] it is not possible to express such conditions". We have shown here that, with a knowledge of modes and types, `block` declarations are sufficient.

[5] This is also the approach taken in Mercury [18], where these versions are generated by the compiler.

Floundering cannot be ruled out systematically, but only be avoided on a heuristic basis. Thus in principle, the method sometimes enforces termination by floundering. This lies in the nature of the weak assumptions made, and thus is sometimes unavoidable, but there is no way of knowing whether for a particular program, it was unavoidable or not.

Marchiori and **Teusink** [11] base termination on norms and the *covering* relation between subqueries of a query. This is loosely related to well-typedness. However, their results are not comparable to ours because they assume a *local selection rule*, that is a rule which always selects an atom which was introduced in the most recent step. We are not aware of an existing language that uses a local selection rule. The authors state that programs that do not use *speculative bindings* deserve further investigation, and that they expect any method for proving termination with *full* coroutining either to be very complex, or very restrictive in its applications.

6 Discussion and Future Work

We have presented a method of proving termination for programs with `block` declarations. This was both a refinement and a formalisation of the heuristics presented in [15].

We required programs to be *permutation robustly typed*, a property which ensured that no call instantiates its own input. In the next step, we identified when a predicate is *robust*, which means that every delay-respecting derivation for a query using the predicate terminates. Robust atoms could be placed in clause bodies arbitrarily. Non-robust atoms had to be placed such that their input is sufficient when they are called.

The main purpose of this work is software development, and it is envisaged that an implementation should take the form of a program development tool. The programmer would provide mode and type information for the predicates in the program. The tool would then generate the `block` declarations and try to reorder the atoms in clause bodies so that the program is well fed with respect to these modes and types. Finding the free and bound positions, as well as the decreasing position used to prove robustness, should be done by the tool. As already indicated, these choices are very constrained anyway, which suggests that this should be feasible.

In [16] we discuss how to prevent errors related to built-ins, in particular arithmetic built-ins. Another interesting issue is how achieving multiple modes using `block` declarations affects the efficiency of programs.

Acknowledgements

We thank the anonymous referees for their helpful suggestions and comments. Jan-Georg Smaus was supported by EPSRC Grant No. GR/K79635.

References

1. K. R. Apt. *From Logic Programming to Prolog*, chapter 6. Prentice Hall, 1997.
2. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *AMAST'95*, LNCS, Berlin, 1995. Springer Verlag. Invited Lecture.
3. K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.
4. R. Chadha and D.A. Plaisted. Correctness of unification without occur check in Prolog. Technical report, University of North Carolina, 1991.
5. Stefaan Decorte and Danny De Schreye. Automatic inference of norms: a missing link in automatic termination analysis. In D. Miller, editor, *Proceedings of ILPS*, pages 420–436. MIT Press, 1993.
6. P. M. Hill, editor. *ALP Newsletter*, `http://www-lp.doc.ic.ac.uk/alp/`, February 1998. Pages 17,18.
7. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
8. 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`.
9. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
10. S. Lüttringhaus-Kappel. Control generation for logic programs. In D. Warren, editor, *Proceedings of ICLP*, pages 478–495. MIT Press, 1993.
11. 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.
12. K. Marriott, L. Naish, and J. L. Lassez. Most specific logic programs. *Annals of mathematics and artificial intelligence*, 1(2), 1990. Also in proceedings of the Fifth JICSLP.
13. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
14. L. Naish. Automatic control of logic programs. *Journal of Logic Programming*, 2(3):167–183, 1985.
15. L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, University of Melbourne, 1992.
16. J.-G. Smaus, P. M. Hill, and A. King. Preventing instantiation errors and loops for logic programs with several modes using `block` declarations. In Pierre Flener, editor, *Pre-proceedings of LOPSTR*. University of Manchester, 1998. Extended abstract.
17. J.-G. Smaus, P. M. Hill, and A. King. Verification of logic programs with `block` declarations running in several modes. Technical Report 7-98, University of Kent at Canterbury, Canterbury, CT2 7NF, United Kingdom, July 1998.
18. 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.