Theory and Practice of Logic Programming **1** (4): 447–486, July 2001.
Printed in the United Kingdom © 2001 Cambridge University Press

Verifying termination and error-freedom of logic programs with `block` *declarations*

JAN-GEORG SMAUS*
CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
(*e-mail:* `jan.smaus@cwi.nl`)

PATRICIA M. HILL
School of Computer Studies, University of Leeds, Leeds LS2 9JT, UK
(*e-mail:* `hill@scs.leeds.ac.uk`)

ANDY KING
University of Kent at Canterbury, Canterbury, CT2 7NF, UK
(*e-mail:* `a.m.king@ukc.ac.uk`)

Abstract

We present verification methods for logic programs with delay declarations. The verified properties are termination and freedom from errors related to built-ins. Concerning termination, we present two approaches. The first approach tries to eliminate the well-known problem of *speculative* output bindings. The second approach is based on identifying the predicates for which the textual position of an atom using this predicate is irrelevant with respect to termination. Three features are distinctive of this work: it allows for predicates to be used in several modes; it shows that `block` declarations, which are a very simple delay construct, are sufficient to ensure the desired properties; it takes the selection rule into account, assuming it to be as in most Prolog implementations. The methods can be used to verify existing programs and assist in writing new programs.

KEYWORDS: verification, delay declarations, termination, modes, types, selection rule, built-ins, errors

1 Introduction

The standard selection rule in logic programming states that the leftmost atom in a query is selected in each derivation step. However, there are some applications for which this rule is inappropriate, e.g. multiple modes, the test-and-generate paradigm (Naish, 1992) or parallel execution (Apt and Luitjes, 1995). To allow for more user-defined control, several logic programming languages provide *delay declarations* (Hill and Lloyd, 1994; SIC, 1998). An atom in a query is selected for resolution only if its arguments are instantiated to a specified degree. This is essential to ensure termination and to prevent runtime errors produced by built-in predicates (*built-ins*).

* Supported by EPSRC Grant No. GR/K79635 and the ERCIM fellowship programme.

In this paper we present methods for verifying programs with delay declarations. We consider two aspects of verification: Programs should terminate, and there should be no type or instantiation errors related to the use of built-ins.

Three distinctive features of this work make its contribution:

(a) it is assumed that predicates may run in more than one mode;
(b) we concentrate on `block` declarations, which are a particularly simple and efficient delay construct; and
(c) the selection rule is taken into account.

We now motivate these features.

(a) Allowing predicates to run in more than one mode is one application of delay declarations. Although other authors (Apt and Luitjes, 1995; Naish, 1992) have not explicitly assumed multiple modes, they mainly give examples where delay declarations are clearly used for that purpose. Whether allowing multiple modes is a good approach or whether it is better to generate multiple versions of each predicate (Somogyi *et al.*, 1996) is an ongoing discussion (Hill, 1998). Our theory allows for multiple modes, but of course this does not exclude other applications of delay declarations.

(b) The `block` declarations declare that certain arguments of an atom must be *non-variable* before that atom can be selected for resolution. Insufficiently instantiated atoms are delayed. As demonstrated in SICStus (SIC, 1998), `block` declarations can be efficiently implemented; the test whether arguments are non-variable has a negligible impact on performance. Therefore, such constructs are the most frequently used delay declarations. Note that most results in this paper also hold for other delay declarations considered in the literature. This is discussed in Sec. 9.

(c) 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*. These are derivations where (allowing for some exceptions concerning the execution order of two literals woken up simultaneously) the leftmost selectable 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 (Apt and Luitjes, 1995; Lüttringhaus-Kappel, 1993); considering left-based selection rules on a heuristic basis (Naish, 1992); or making the very restrictive assumption of *local selection rules* (Marchiori and Teusink, 1999).

The main contribution concerns termination. We have isolated some of the causes of non-termination that are related to the use of delay declarations and identified conditions for programs to avoid those causes. These conditions can easily be checked at compile-time. The termination problem for a program with delay declarations is then translated to the same problem for a corresponding program executed left-to-right. It is assumed that, for the corresponding program, termination can be shown using some existing technique (Apt, 1997; De Schreye and Decorte, 1994; Etalle *et al.*, 1999).

One previously studied cause of non-termination associated with delay declarations is *speculative output bindings* (Naish, 1992). These are bindings made before it is known that a solution exists. We present two complementing 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 *use* any speculative bindings, by ensuring that no atom ever delays. The second method exploits that a program does not *make* any speculative bindings.

However, these two methods are quite limited. As an alternative approach to the termination problem, we identify certain predicates that may loop when called with *insufficient* (that is, non-variable but still insufficiently instantiated) input. For instance, with the predicate `permute/2` where the second argument is input, the query `permute(A,[1|B])` has insufficient input and loops.[1] However, the query `permute(A,[1,2])` has sufficient input and terminates. The idea for proving termination is that, for such predicates, calls with insufficient input must 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.

Our work on built-ins focuses on *arithmetic* built-ins. By exploiting the fact that for numbers, being non-variable implies being ground, we show how both *instantiation* and *type* errors can be prevented.

Finally, we consider two other issues related to delay declarations. First, we identify conditions so that certain `block` declarations can be omitted without affecting the runtime behaviour. Secondly, to verify programs with delay declarations, it is often necessary to impose a restriction on the modes that forbids tests for identity between the input arguments of an atom. We explain how this rather severe restriction is related to the use of delay declarations and how it can be weakened.

This paper is organised as follows. The next section defines some essential concepts and notations. Section 3 introduces four concepts of 'modedness' and 'typedness' that are needed later. Section 4, which is based on previously published work (Smaus *et al.*, 1999), presents the first approach to the termination problem. Section 5, which is also based on previously published work (Smaus *et al.*, 1998), presents the second approach. Section 6 is about errors related to built-ins. Section 7 considers ways of simplifying the `block` declarations. Section 8 investigates related work. Section 9 concludes with a summary and a look at ongoing and future work.

2 Essential concepts and notations

2.1 Standard notions

We base the notation on (Apt and Luitjes, 1995; Lloyd, 1987). For the examples we use SICStus notation (SIC, 1998). A term u occurs *directly* in a vector of terms $\mathbf{t}$ if u is one of the terms of $\mathbf{t}$. (For example, a occurs directly in (a, b) but not in

[1] The program for `permute/2` is given in figure 5.

$(f(a), b)$.) We also say that *u fills a position in* $\mathbf{t}$. To refer to the predicate symbol of an atom, we say that an atom $p(\ldots)$ is an atom *using* p. 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. Otherwise it is *non-linear*. A *flat* term is a variable or a term $f(x_1, \ldots, x_n)$, where $n \geqslant 0$ and the x_i are distinct variables. The *domain* of a substitution σ is $dom(\sigma) = \{x \mid x\sigma \neq x\}$. The variables in the range of σ are denoted as $ran(\sigma) = \{y \mid y \in vars(x\sigma), y \neq x\}$.

A *query* is a finite sequence of atoms. Atoms are denoted by a, b, h, queries by B, F, H, Q, R. Sometimes we say 'atom' instead of 'query consisting of an atom'. 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, renamed apart from $Q\theta$, and σ the most general unifier (MGU) of $a\theta$ and h. We call $a\theta$ (or a)[2] 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 each pair $\langle Q_i, \theta_i \rangle ; \langle Q_{i+1}, \theta_{i+1} \rangle$ in ξ is a derivation step for P. Alternatively, we also say that ξ is a *derivation of* $P \cup \{Q_0\theta_0\}$. We also denote ξ by $Q_0\theta_0 ; Q_1\theta_1 ; \ldots$. A derivation is an *LD-derivation* if the selected atom is always the leftmost atom in a query.

If $F, a, H ; (F, B, H)\theta$ is a step in a derivation, then each atom in $B\theta$ (or B)[2] is a *direct descendant* of a, and $b\theta$ (or b)[2] is a *direct descendant* of b for all b in F, H. We say that b is a *descendant of* a, or a is an *ancestor of* b, 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. Consider a derivation $Q_0 ; \ldots ; Q_i ; \ldots ; Q_j ; Q_{j+1} ; \ldots$. We call $Q_j ; Q_{j+1}$ an *a-step* if a is an atom in Q_i ($i \leqslant j$) and the selected atom in $Q_j ; Q_{j+1}$ is a descendant of a.

2.2 Modes

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. 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 of p, respectively.

An atom $p(\mathbf{s}, \mathbf{t})$ is *input-linear* if $\mathbf{s}$ is linear. A clause is *input-linear* if its head is input-linear. A program is *input-linear* if all of its clauses are input-linear and it contains no uses of $=(I, I)$.[3]

We claim that the techniques we describe are suitable for programs that can run in several modes. Throughout most of the presentation, this is not explicit, since we always consider one mode at a time. Therefore, whenever we refer to the input and output positions, this is always with respect to one particular mode. However, we will see in several examples that *one single* program can be 'mode correct', in a well-defined sense, with respect to several different modes. In particular, *one single* delay declaration for a predicate can allow for this predicate to be used in different modes.

[2] Whether or not the substitution has been applied is always clear from the context.

[3] Conceptually, one can think of each program containing the fact clause `X = X`.

This is different from the assumption made by some authors (Apt and Etalle, 1993; Apt and Luitjes, 1995; Etalle *et al.*, 1999; Naish, 1992) that if a predicate is to be used in several modes, then multiple (renamed) versions of this predicate should be introduced, which may differ concerning the delay declarations and the order of atoms in clause bodies.

Note that our notion of modes could easily be generalised further by assigning a mode to predicate *occurrences* rather than predicates (Smaus, 1999).

2.3 Types

A *type* is a set of terms closed under instantiation (Apt and Luitjes, 1995; Boye, 1996). *The variable type* is the type that contains variables and hence, as it is closed under instantiation, all terms. Any other type is a *non-variable type*. A type is a *ground type* if it contains only ground terms. A type is a *constant type* if it is a ground type that contains only (possibly infinitely many) constants. In the examples, we use the following types: *any* is the variable type, *list* the non-variable type of (nil-terminated) lists, *int* the constant type of integers, *il* the ground type of integer lists, *num* the constant type of numbers, *nl* the ground type of number lists, and finally, *tree* is the non-variable type defined by the context-free grammar $\{tree \rightarrow \texttt{leaf}; tree \rightarrow \texttt{node}(tree, any, tree)\}$.

We write $t : T$ for 't is in type T'. We use $\mathbf{S}$, $\mathbf{T}$ to denote vectors of types, and write $\models \mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$ if for all substitutions σ, $\mathbf{s}\sigma : \mathbf{S}$ implies $\mathbf{t}\sigma : \mathbf{T}$. 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 $p(T_1, \dots, T_n)$ where $T_1, \dots, T_n$ are types. The *type of a program* P is a set of such atoms, one for each predicate defined in P. An atom is *correctly typed* in a position if the term filling this position has the type that is associated with this position. A term t is *type-consistent with respect to* T (Deransart and Małuszyński, 1998) if there is a substitution θ such that $t\theta : T$. A term t occurring in an atom in some position is *type-consistent* if it is type-consistent with respect to the type of that position.

2.4 `block` ***declarations***

A `block` declaration (SIC, 1998) for a predicate p/n is a (possibly empty) set of atoms each of which has the form $p(b_1, \dots, b_n)$, where $b_i \in \{\texttt{?}, \texttt{-}\}$ 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 *blocked in* P if there is an atom $p(b_1, \dots, b_n)$ in the `block` declaration for p such that for all $i \in \{1, \dots, n\}$ with $b_i = \texttt{-}$, we have that t_i is variable. An atom is *selectable in* P if it is not blocked in P.

Example 2.1
Consider a program containing the `block` declaration

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

Then the atoms append(X, Y, Z), append([1|X], Y, Z), and append(X, [2|Y], Z) are all blocked in P, whereas the atoms append([1|X], [2|Y], Z), append(X, Y, [1|Z]) and the atom append(X, [2|Y], [1|Z]) are selectable in P. $\lhd$

Note that equivalent delay constructs are provided in several logic programming languages, although there may be differences in the syntax.

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.

2.5 Left-based derivations

We now formalise the sort of derivations that arise in practice using almost any existing Prolog implementations. Some authors have considered a selection rule stating that in each derivation step, the leftmost selectable atom is selected (Apt and Luitjes, 1995; Boye, 1996; Naish, 1992). We are not aware of an existing language that uses this selection rule, contradicting Boye's claim (1996, page 123) that several modern Prolog implementations and even Gödel (Hill and Lloyd, 1994) use this selection rule. In fact, Prolog implementations do not usually guarantee the order in which two simultaneously woken atoms are selected.

Definition 2.2
[left-based derivation] Consider a delay-respecting derivation $Q_0; \ldots; Q_i; \ldots$, where $Q_i = R_1, R_2$, and R_1 contains no selectable atom. Then every descendant of every atom in R_1 is *waiting*. A delay-respecting derivation $Q_0; Q_1 \ldots$ is *left-based* if for each step $Q_i; Q_{i+1}$, the selected atom is either waiting in Q_i, or it is the leftmost selectable atom in Q_i. $\lhd$

Example 2.3
Consider the following program:

```
:- block a(-).          :- block b(-)
a(1).                   b(X) :- b2(X).

c(1).                   b2(1).                  d.
```

The following is a left-based derivation. Waiting atoms are underlined.

$$\underline{\mathtt{a(X)}}, \underline{\mathtt{b(X)}}, \mathtt{c(X)}, \mathtt{d}; \quad \underline{\mathtt{a(1)}}, \underline{\mathtt{b(1)}}, \mathtt{d}; \quad \underline{\mathtt{a(1)}}, \underline{\mathtt{b2(1)}}, \mathtt{d}; \quad \underline{\mathtt{a(1)}}, \mathtt{d}; \quad \mathtt{d}; \quad \square.$$

Note that b(1) and b2(1) are waiting and selectable, and therefore they can be selected although there is the selectable atom a(1) to the left. $\lhd$

We do not believe that it would be useful or practical to try to specify the selection rule precisely, but from our research, it appears that derivations in most Prolog implementations are left-based.

Note that the definition of left-based derivations for a program and query depends both on the textual order of the atoms in the query and clauses and on the block declarations. In order to maintain the textual order while considering different orders of selection of atoms, it is often useful to associate, with a query, a permutation π of the atoms.

Let π be a permutation on $\{1,\dots,n\}$. We assume that $\pi(i) = i$ for $i \notin \{1,\dots,n\}$. In examples, π is written as $\langle \pi(1),\dots,\pi(n)\rangle$. We write $\pi(o_1,\dots,o_n)$ for the application of π to the sequence $o_1,\dots,o_n$, that is $o_{\pi^{-1}(1)},\dots,o_{\pi^{-1}(n)}$.

3 Correctness conditions for verification

Apt and Luitjes (1995) consider three correctness conditions for programs: *nicely moded*, *well typed*, and *simply moded*. Apt (1997) and Boye (1996) propose a generalisation of these conditions that allows for permutations of the atoms in each query. Such correctness conditions have been used for various verification purposes: occur-check freedom, flounder freedom, freedom from errors related to built-ins (Apt and Luitjes, 1995), freedom from failure (Bossi and Cocco, 1999), and termination (Etalle *et al.*, 1999). In this section we introduce four such correctness conditions and show some important statements about them. The correctness conditions will then be used throughout the paper.

The idea of these correctness conditions is that in a query, every piece of data is produced (output) before it is consumed (input), and every piece of data is produced only once. The definitions of these conditions have usually been aimed at LD-derivations, which means that an output occurrence of a variable must always be to the left of any input occurrence of that variable.

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*. Note that the use of the letters s and t is reversed for clause heads. We believe that this notation naturally reflects the data flow within a clause. This will become apparent in Definition 3.5.

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

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

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

$$vars(\mathbf{t}_0) \cap \bigcup_{j=1}^{n} vars(\mathbf{t}_j) = \emptyset.$$

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.

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

$$M_1 = \{\texttt{permute}(I,O), \texttt{delete}(I,O,I)\}$$
$$M_2 = \{\texttt{permute}(O,I), \texttt{delete}(O,I,O)\}$$

Fig. 1. The `permute` program.

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. $\lhd$

In Lemma 3.3, on which many results of this paper depend, we require a program not only to be permutation nicely moded, but also input-linear (see section 2.2).

Example 3.2
The program in figure 1 is nicely moded and input-linear in mode M_1.[4] In mode M_2 it is permutation nicely moded and input-linear. In particular, the second clause for `permute` is $\langle 2,1\rangle$-nicely moded. In 'test mode', that is, $\{\texttt{permute}(I,I)$, $\texttt{delete}(I,I,O)\}$, it is permutation nicely moded, but not input-linear, because the first clause for `delete` is not input-linear. $\lhd$

We show that there is a persistence property for permutation nicely-modedness similar to that for nicely-modedness (Apt and Luitjes, 1995).

Lemma 3.3
Let $Q = a_1,\dots,a_n$ be a π-nicely moded query and $C = h \leftarrow b_1,\dots,b_m$ be a ρ-nicely moded, input-linear clause where $vars(Q) \cap vars(C) = \emptyset$. Suppose for some $k \in \{1,\dots,n\}$, h and a_k are unifiable. Then the resolvent of Q and C with selected atom a_k is ϱ-nicely moded, where the *derived permutation* ϱ on $\{1,\dots,n+m-1\}$ is defined by $\varrho(i) =$

$$\begin{cases} \pi(i) & \text{if } i < k,\ \pi(i) < \pi(k) \\ \pi(i) + m - 1 & \text{if } i < k,\ \pi(i) > \pi(k) \\ \pi(k) + \rho(i-k+1) - 1 & \text{if } k \leqslant i < k+m \\ \pi(i-m+1) & \text{if } k+m \leqslant i < n+m,\ \pi(i-m+1) < \pi(k) \\ \pi(i-m+1) + m - 1 & \text{if } k+m \leqslant i < n+m,\ \pi(i-m+1) > \pi(k). \end{cases}$$

Proof
Let θ be the MGU of h and a_k. By Def. 3.1, we have that $a_{\pi^{-1}(1)},\dots,a_{\pi^{-1}(n)}$ and

[4] For convenient reference, the modes are included in the figure. Also, the program contains `block` declarations. We will refer to those later; they should be ignored for the moment.

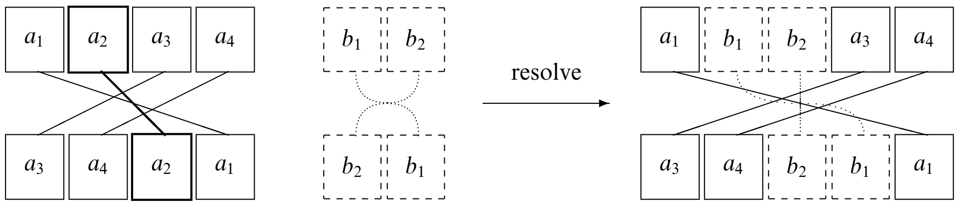

Fig. 2. The derived permutation ϱ for the resolvent.

$h \leftarrow b_{\rho^{-1}(1)}, \ldots, b_{\rho^{-1}(m)}$ are nicely moded and h is input-linear. Thus by Lemma 11 (Apt and Luitjes, 1995)

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

is nicely moded, and hence $(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\,\theta$ is ϱ-nicely moded. □

Figure 2 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 $\varrho = \langle 5,4,3,1,2\rangle$. Observe that, at each step of a derivation, the relative order of atoms given by the derived permutation is preserved. By a straightforward induction on the length of a derivation, using the definition of ϱ for the base case, we obtain the following corollary.

Corollary 3.4
Let P be a permutation nicely moded, input-linear 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 $\rho(k) < \rho(l)$.

Note that derivations of a permutation nicely moded query and a permutation nicely moded, input-linear program are *occur-check free*. This is is a trivial consequence of Theorem 13 (Apt and Luitjes, 1995). In section 7.3, we discuss ways in which the condition of input-linearity in Lemma 3.3 can be weakened.

3.2 Permutation well typed programs

In a *well typed* query (Apt and Luitjes, 1995; Apt and Pellegrini, 1994; Bronsard *et al.*, 1992), the first atom is correctly typed in its input positions. Furthermore, given a well typed query Q, a, Q' and assuming LD-derivations, if Q is resolved away, then a becomes correctly typed in its input positions. We generalise this to *permutation well typed* (previously called *properly typed* (Apt, 1997)). As with the modes, we assume that the type associated with each argument position is given. In the examples, the types will be the natural ones that would be expected.

Definition 3.5
[Permutation well typed] Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query, where $p_i(\mathbf{S}_i, \mathbf{T}_i)$ is the type of p_i for each $i \in \{1, \ldots, n\}$. Let π be a permutation on $\{1, \ldots, n\}$. Then Q

is π-*well typed* if for all $i \in \{1,\dots,n\}$ and $L = 1$

$$\models \left(\bigwedge_{L \leqslant \pi(j) < \pi(i)} \mathbf{t}_j : \mathbf{T}_j \right) \Rightarrow \mathbf{s}_i : \mathbf{S}_i. \tag{1}$$

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$, where $p(\mathbf{T}_0, \mathbf{S}_{n+1})$ is the type of p, is π-*well typed* if (1) holds for all $i \in \{1,\dots,n+1\}$ and $L = 0$.

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 Definition 3.1. $\lhd$

Example 3.6
Consider the program in figure 1 with type $\{$`permute`$(list, list)$, `delete`$(any, list, list)\}$. It is well typed for mode M_1, and permutation well typed for mode M_2, with the same permutations as in Example 3.2. The same holds assuming type $\{$`permute`(nl, nl), `delete`$(num, nl, nl)\}$. $\lhd$

Permutation well-typedness is also a persistent condition. The proof is analogous to Lemma 3.3, but using Lemma 23 instead of Lemma 11 (Apt and Luitjes, 1995).

Lemma 3.7
Let $Q = a_1,\dots,a_n$ be a π-well typed query and $C = h \leftarrow b_1,\dots,b_m$ be a ρ-well typed clause where $vars(Q) \cap vars(C) = \emptyset$. Suppose for some $k \in \{1,\dots,n\}$, h and a_k are unifiable. Then the resolvent of Q and C with selected atom a_k is ϱ-well typed, where ϱ is the derived permutation (see Lemma 3.3).

Generalising Theorem 26 (Apt and Luitjes, 1995), permutation well-typedness can be used to show that derivations do not flounder (Smaus, 1999).

3.3 Permutation simply typed programs

We now define *permutation simply-typedness*. The name *simply typed* is a combination of *simply moded* (Apt and Luitjes, 1995) and *well typed*. In a permutation simply typed query, the output positions are filled with variables, and therefore they can always be instantiated so that all atoms in the query are correctly typed.

Definition 3.8
[Permutation simply typed] Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1),\dots,p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query and π a permutation on $\{1,\dots,n\}$. Then Q is π-*simply typed* if it is π-nicely moded and π-well typed, and $\mathbf{t}_1,\dots,\mathbf{t}_n$ is a vector of *variables*.

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is π-*simply typed* if it is π-nicely moded and π-well typed, $\mathbf{t}_1,\dots,\mathbf{t}_n$ is a vector of *variables* and $\mathbf{t}_0$ is a vector of flat type-consistent terms that has a variable in each position of variable type.

A *permutation simply typed* query (clause, program) and a *simply typed* query (clause, program) *corresponding to* a query (clause, program) are defined in analogy to Definition 3.1. $\lhd$

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

:- block append(-,?,-).
append([],Y,Y).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

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

:- block leq(?,-), leq(-,?).
leq(A,B) :- A =< B.

:- block grt(?,-), grt(-,?).
grt(A,B) :- A > B.
```

$$M_1 = \{\texttt{qsort}(I,O), \texttt{append}(I,I,O), \texttt{leq}(I,I), \texttt{grt}(I,I), \texttt{part}(I,I,O,O)\}$$
$$M_2 = \{\texttt{qsort}(O,I), \texttt{append}(O,O,I), \texttt{leq}(I,I), \texttt{grt}(I,I), \texttt{part}(O,I,I,I)\}$$

Fig. 3. The quicksort program.

Example 3.9
Figure 3 shows a version of the quicksort program. Assume the type $\{\texttt{qsort}(nl, nl)$, $\texttt{append}(nl, nl, nl)$, $\texttt{leq}(num, num)$, $\texttt{grt}(num, num)$, $\texttt{part}(nl, num, nl, nl)\}$. The program is permutation simply typed for mode M_1. It is not permutation simply typed for mode M_2, due to the non-variable term [X|Bs2] in an output position. $\lhd$

The persistence properties stated in Lemmas 3.3 and 3.7 are independent of the selectability of an atom in a query. For permutation simply typed programs, this persistence property only holds if the selected atom is sufficiently instantiated in its input arguments. This motivates the following definition.

Definition 3.10
[Bound/free] Let P be a permutation well typed program. An input position of a predicate p in P is *bound* if there is a clause head $p(\ldots)$ in P that has a non-variable term in that position. An output position of a predicate p in P is *bound* if there is an atom $p(\ldots)$ in a clause body in P that has a non-variable term in that position. A position is *free* if it is not bound.

We denote the projection of a vector of arguments $\mathbf{r}$ onto its free positions as $\mathbf{r}^{\mathrm{f}}$, and onto its bound positions as $\mathbf{r}^{\mathrm{b}}$. $\lhd$

Note that for a permutation simply typed program, there are no bound output positions, and bound input positions must be of non-variable type.

Lemma 3.11
Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a π-simply typed query, and let $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ a ρ-simply typed, input-linear clause where $vars(C) \cap$

$vars(Q) = \emptyset$. Suppose that for some $k \in \{1, \ldots, n\}$, $\mathbf{s}_k$ is non-variable in all bound input positions[5] and θ is the MGU of $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$. Then

1. there exist substitutions θ_1, θ_2 such that $\theta = \theta_1\theta_2$ and
   (a) $\mathbf{v}_0\theta_1 = \mathbf{s}_k$ and $dom(\theta_1) \subseteq vars(\mathbf{v}_0)$,
   (b) $\mathbf{t}_k\theta_2 = \mathbf{u}_{m+1}\theta_1$ and $dom(\theta_2) \subseteq vars(\mathbf{t}_k)$;
2. $dom(\theta) \subseteq vars(\mathbf{t}_k) \cup vars(\mathbf{v}_0)$;
3. $dom(\theta) \cap vars(\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n) = \emptyset$;
4. the resolvent of Q and C with selected atom $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is ϱ-simply typed, where ϱ is the derived permutation (see Lemma 3.3). (For proof, see the Appendix.)

The following corollary of Lemma 3.11 (4) holds, since by Definition 3.5, the leftmost atom in a simply typed query is non-variable in its input positions of non-variable type.

Corollary 3.12
Every LD-resolvent of a simply typed query Q and a simply typed, input-linear clause C, where $vars(C) \cap vars(Q) = \emptyset$, is simply typed.[6]

Before studying permutation simply typed programs any further, we now introduce a generalisation of this class.

3.4 Permutation robustly typed programs

The program in figure 3 is not permutation simply typed in mode M_2, due to the non-variable term `[X|Bs2]` in an output position. It has been acknowledged previously that it is difficult to reason about queries where non-variable terms in output positions are allowed, but on the other hand, there are natural programs where this occurs (Apt and Etalle, 1993).

We define permutation *robustly-typedness*, which is a carefully crafted extension of permutation simply-typedness, allowing for non-variable but flat terms in output positions. It has been designed so that a persistence property analogous to Lemmas 3.3, 3.7 and 3.11 holds.

Definition 3.13
[Permutation robustly typed] Assume a permutation well typed program P where the bound positions are of non-variable type. Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query (using predicates from P) and π a permutation on $\{1, \ldots, n\}$. Then Q is π-*robustly typed* if it is π-nicely moded and π-well typed, $\mathbf{t}^{\mathsf{f}}_1, \ldots, \mathbf{t}^{\mathsf{f}}_n$ is a vector of variables, and $\mathbf{t}^{\mathsf{b}}_1, \ldots, \mathbf{t}^{\mathsf{b}}_n$ is a vector of flat type-consistent terms.

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

1. $\mathbf{t}^{\mathsf{f}}_0, \ldots, \mathbf{t}^{\mathsf{f}}_n$ is a vector of variables, and $\mathbf{t}^{\mathsf{b}}_0, \ldots, \mathbf{t}^{\mathsf{b}}_n$ is a vector of flat type-consistent terms; and

[5] This is similar to the assumption 'the delay declarations imply matching' (Apt and Luitjes, 1995).
[6] This even holds without requiring C to be input-linear (Smaus, 1999, Lemma 7.3), but here we do not need the stronger result, and it is not a corollary of Lemma 3.11 (4).

```
:- block treeList(-,-).                   :- block append(-,?,-).
treeList(leaf,[]).                        append([],Y,Y).
treeList(node(L,Label,R),List) :-         append([X|Xs],Ys,[X|Zs]) :-
  append(LList,[Label|RList],List),         append(Xs,Ys,Zs).
  treeList(L,LList),
  treeList(R,RList).
```

$$M_1 = \{\texttt{treeList}(I,O), \texttt{append}(I,I,O)\}$$
$$M_2 = \{\texttt{treeList}(O,I), \texttt{append}(O,O,I)\}$$

Fig. 4. Converting trees to lists, or vice versa.

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

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 Definition 3.1. $\lhd$

Note that any permutation simply typed program is permutation robustly typed, where all output positions are free.

Example 3.14
Recall the program in figure 3. It is permutation robustly typed in mode M_2, and the second position of append is the only bound output position. Note in particular that Condition 2 of Definition 3.13 is met for the recursive clause of append: the variable Ys fills an output position of the head and also an output position of the body. Moreover, the program is trivially permutation robustly typed in mode M_1. $\lhd$

Example 3.15
The program in figure 4 converts binary trees into lists and vice versa. Assuming type $\{\texttt{treeList}(\mathit{tree}, \mathit{list}), \texttt{append}(\mathit{list}, \mathit{list}, \mathit{list})\}$, the program is permutation robustly typed in mode M_2, and the second position of append is the only bound output position. It is also permutation robustly typed in mode M_1, where all output positions are free. $\lhd$

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 3.16
Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ a π-robustly typed query and $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ a ρ-robustly typed, input-linear clause where $\mathit{vars}(Q) \cap \mathit{vars}(C) = \emptyset$. Suppose that for some $k \in \{1, \ldots, n\}$, $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is non-variable in all bound input positions and θ is the MGU of $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$.

Then the resolvent of Q and C with selected atom $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is ϱ-robustly typed, where ϱ is the derived permutation (see Lemma 3.3). Moreover, $dom(\theta) \cap vars(\mathbf{s}_k) = \emptyset$. (For proof, see the Appendix.)

We now define programs where the `block` declarations fulfill a natural minimum (1) and maximum (2) requirement. The minimum requirement states that selected atoms must fulfill the assumption of Lemmas 3.11 and 3.16. The maximum requirement is needed in section 5.2. In other words, we define programs where the 'static' concept of modes and the 'dynamic' concept of `block` declarations correspond in the natural way.

Definition 3.17
[Input selectability] Let P be a permutation robustly typed program. P has *input selectability* if an atom using a predicate in P that has variables in all free output positions is selectable in P

1. only if it is non-variable in all bound input positions; and
2. if it is non-variable in all input positions of non-variable type.

$\lhd$

Note that the above definition is aimed at atoms in permutation robustly typed queries, since these atoms have variables in all free output positions.

Example 3.18
Consider `append`(O, O, I) where the second position is the only bound output position, as used in the programs in figure 3 in mode M_2 and figure 4 in mode M_2. The program for `append` has input selectability.

Now consider `append`(I, I, O) where the output position is free, as used in the programs in figure 3 in mode M_1 and figure 4 in mode M_1. The program for `append` has input selectability. Note that the `block` declaration for `append` is the one that is usually given (Hill and Lloyd, 1994; Lüttringhaus-Kappel, 1993; Marchiori and Teusink, 1999). $\lhd$

The following is a corollary of Lemma 3.16 needed to prove Lemma 5.4.

Corollary 3.19
Let P be a permutation robustly typed, input-linear program with input selectability, $Q = a_1, \ldots, a_n$ a π-robustly typed query and $i, j \in \{1, \ldots, n\}$ such that $\pi(i) < \pi(j)$. Let

$$Q; \ldots; (b_1, \ldots, b_m); (b_1, \ldots, b_{l-1}, B, b_{l+1}, \ldots, b_m)\theta$$

be a delay-respecting derivation and $k \in \{1, \ldots, m\}$, such that b_k is a descendant of a_i and b_l is a descendant of a_j. Then $dom(\theta) \cap vars(b_k) = \emptyset$.

Proof
Suppose that $b_1, \ldots, b_m$ is ρ-robustly typed. By Corollary 3.4, we have $\rho(k) < \rho(l)$. Suppose $b_l = p_l(\mathbf{s}_l, \mathbf{t}_l)$.

Since θ is obtained by unifying b_l with a head of a clause C, and $vars(C) \cap vars(b_1, \ldots, b_m) = \emptyset$, it follows that $dom(\theta) \cap vars(b_1, \ldots, b_m) \subseteq vars(b_l)$. By Lemma 3.16, $dom(\theta) \cap vars(\mathbf{s}_l) = \emptyset$. Since $b_1, \ldots, b_m$ is ρ-nicely moded, $vars(b_k) \cap vars(\mathbf{t}_l) = \emptyset$ and so $dom(\theta) \cap vars(b_k) = \emptyset$. $\square$

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

$$M_1 = \{\texttt{permute}(I,O), \texttt{delete}(I,O,I)\}$$
$$M_2 = \{\texttt{permute}(O,I), \texttt{delete}(O,I,O)\}$$

Fig. 5. Putting recursive calls last in the `permute` program.

Intuitively, the corollary says that if $\pi(i) < \pi(j)$, then no a_j-step will ever instantiate a descendent of a_i.

We conclude this section with a statement about permutation *simply* typed programs, which we could not present earlier since it relies on the definition of input selectability. It says that in a derivation for a permutation simply typed program and query, it can be assumed without loss of generality that the output positions in each query are filled with variables that occur in the initial query or in some clause body used in the derivation.

Corollary 3.20
Let P be a permutation simply typed program with input selectability and Q_0 be a permutation simply typed query. Let $\theta_0 = \emptyset$ and $\xi = \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle; \ldots$ be a delay-respecting derivation of $P \cup \{Q_0\}$. Then for all $i \geqslant 0$, if x is a variable in an output position in Q_i, then $x\theta_i = x$.

Proof
The proof is by induction on the position i in the derivation. The base case $i = 0$ is trivial since $\theta_0 = \emptyset$. Now suppose the result holds for some i and Q_{i+1} exists. By Lemma 3.11 (4), $Q_i\theta_i$ is permutation simply typed. Thus the result follows for $i+1$ by Lemma 3.11 (3). □

4 Termination and speculative bindings

Like most approaches to the termination problem (De Schreye and Decorte, 1994), we are interested in ensuring that *all* derivations of a query are finite. Therefore the clause order in a program is irrelevant. Furthermore, we do not prove termination as such, but rather reduce the problem of proving termination for a program and query with left-based derivations to that with LD-derivations.

In this section, we present two complementing methods of showing termination. These are explained in the following example.

Example 4.1
Assuming left-based derivations, the program given in figure 1 loops for the query `permute(V,[1])` (hence, in mode M_2) because `delete` produces a *speculative output binding* (Naish, 1992): the third argument of `delete` is bound before it is known

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

Fig. 6. Most specific version of `delete`(O,I,O).

that this binding will never have to be undone. Termination in modes M_1 and M_2 can be ensured by swapping the atoms in the second clause, as shown in Fig. 5. This technique has been described as *putting recursive calls last* (Naish, 1992). To explain why the program terminates, we have to apply a different reasoning for the different modes.

In mode M_2, the atom that produces the speculative output occurs *textually before* the atom that consumes it. This means that the consumer waits until the producer has completed (undone the speculative binding). The program does not *use* speculative bindings. In mode M_1, the program does not *make* speculative bindings.

Note that termination for this example depends on *left-based* derivations, and thus any method that abstracts from the selection rule must fail. $\lhd$

The methods presented in this section can be used to prove that the programs in figures 4–7 terminate, but they do not work for the programs in figures 3 and 8. They formalise previous heuristics (Naish, 1985; Naish, 1992) and rely on conditions that are easy to check.

4.1 Termination by not using speculative bindings

In LD-derivations, speculative bindings are never used (Naish, 1992). A left-based derivation *is* an LD-derivation, provided the leftmost atom in each query is always selectable. Hence by Lemma 3.7, we have the following proposition.

Proposition 4.2
Let Q be a well typed query and P a well typed program such that an atom is selectable in P whenever its input positions of non-variable type are non-variable. Then every left-based derivation of $P \cup \{Q\}$ is an LD-derivation.

We now give two examples of programs where by Proposition 4.2, we can use any method for LD-derivations to show termination for any well typed query. Note that the method of section 5 is not applicable for the program in Example 4.4 (because it is is not permutation robustly typed).

Example 4.3
Consider the program in figure 5 with mode M_2 and either of the types given in Example 3.6. This program is well-typed. $\lhd$

Example 4.4
Consider the version of `delete`(O,I,O) given in figure 6. Assuming either of the types given in Example 3.6, this program is well typed. $\lhd$

Regarding this subsection, one may wonder: what is the point in considering derivations for programs with `block` declarations where in effect we show that those `block` declarations are redundant, that is, the program is executed left-to-right? However, one has to bear in mind that a program might also be used in another mode, and therefore, the `block` declarations may be necessary.

4.2 Termination by not making speculative bindings

Some programs and queries have the nice property that there cannot be any failing derivations. Bossi and Cocco (1999) have identified a class of programs called *noFD* having this property. *Non-speculative* programs are similar, but there are two differences: the definition of noFD programs only allows for *LD*-derivations, but on the other hand, the definition of non-speculative programs requires that the clause heads are input-linear.

Definition 4.5
[non-speculative] A program P is *non-speculative* if it is permutation simply typed and input-linear, and every simply typed atom using a predicate in P is unifiable with some clause head in P. $\lhd$

Example 4.6
Both versions of the `permute` program (figures 1 and 5), with either type given in Example 3.6, are non-speculative in mode M_1. Every simply typed atom is unifiable with at least one clause head. Both versions are *not* non-speculative in mode M_2, because `delete(A,[],B)` is not unifiable with any clause head. $\lhd$

Example 4.7
The program in figure 4 is non-speculative in mode M_1. However, it is not non-speculative in mode M_2 because it is not permutation simply typed, due to the non-variable term `[Label|List]` in an output position. $\lhd$

A delay-respecting derivation for a non-speculative program P with input selectability and a permutation simply typed query cannot fail.[7] However it could still be infinite. The following theorem says that this can only happen if the simply typed program corresponding to P has an infinite *LD*-derivation for this query.

Theorem 4.8
Let P be a non-speculative program with input selectability and P' a simply typed program corresponding to P. Let Q be a permutation simply typed query and Q' a simply typed query corresponding to Q. If there is an infinite delay-respecting derivation of $P \cup \{Q\}$, then there is an infinite LD-derivation of $P' \cup \{Q'\}$. (For proof, see the Appendix.)

Theorem 4.8 says that for non-speculative programs, the atom order in clause bodies is irrelevant for termination.

[7] It can also not *flounder* (Smaus, 1999).

```
:- block is_list(-).                  :- block equal_list(-,?).
is_list([]).                          equal_list([],[]).
is_list([X|Xs]):-                     equal_list([X|Xs],[X|Ys]):-
  is_list(Ys),                          equal_list(Xs,Ys).
  equal_list(Xs,Ys).
```

Fig. 7. The `is_list` program.

Note that any program that uses *tests* cannot be non-speculative. In figure 3, assuming mode M_1, the atoms `leq(X,C)` and `grt(X,C)` are tests. These tests are *exhaustive*, i.e. at least one of them succeeds (Bossi and Cocco, 1999). This suggests a generalisation of non-speculative programs (Pedreschi and Ruggieri, 1999) (see section 8).

We now give an example of a program for which termination can be shown using Theorem 4.8 but not using the method of section 5 (see also Example 5.11).

Example 4.9
Consider the program in figure 7, where the mode is $\{$`is_list`(I), `equal_list`$(I,O)\}$ and the type is $\{$`is_list`(list), `equal_list`$(\mathit{list}, \mathit{list})\}$. The program is permutation simply typed (the second clause is $\langle 2,1\rangle$-simply typed) and non-speculative, and all LD-derivations for the corresponding simply typed program terminate. Therefore all delay-respecting derivations of a permutation simply typed query and this program terminate. $\lhd$

5 Termination and insufficient input

We now present an alternative method for showing termination that overcomes some of the limitations of the methods presented in the previous section. In particular, the methods can be used for the programs in figures 3 and 8 as well as figures 4 and 5. In practice, we expect the method presented here to be more useful, although, as figures 6 and 7 show, it does not subsume the method of the previous section.

As explained in Example 4.1, termination of `permute`(O,I) can be ensured by applying the heuristic of putting recursive calls last (Naish, 1992). The following example however shows that even this version of `permute`(O,I) can cause a loop depending on how it is called within some other program.

Example 5.1
Figure 8 shows a program for the n-queens problem, which uses `block` declarations to implement the test-and-generate paradigm. With the mode M_1 and the type T, the first clause is $\langle 1,3,2\rangle$-nicely moded and $\langle 1,3,2\rangle$-well typed. Moreover, all left-based derivations for the query `nqueens(4,Sol)` terminate.

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 binding (which is *not* speculative) that triggers the call `permute(Sol,[4|T])`. This call results in a loop. Note that `[4|T]`, although non-variable, is insufficiently instantiated

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

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

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

$$
\begin{aligned}
M_1 &= \{\texttt{nqueens}(I,O), \texttt{sequence}(I,O), \texttt{safe}(I), \texttt{permute}(O,I), \texttt{<}(I,I),\\
&\qquad \texttt{is}(O,I), \texttt{safe\_aux}(I,I,I), \texttt{no\_diag}(I,I,I), \texttt{=\textbackslash=}(I,I)\}\\
M_2 &= \{\texttt{nqueens}(O,I), \texttt{sequence}(O,I), \texttt{permute}(I,O), \texttt{is}(O,I), \ldots\}\\
T &= \{\texttt{nqueens}(\mathit{int},\mathit{il}), \texttt{sequence}(\mathit{int},\mathit{il}), \texttt{safe}(\mathit{il}), \texttt{permute}(\mathit{il},\mathit{il}),\\
&\qquad \texttt{<}(\mathit{int},\mathit{int}), \texttt{is}(\mathit{int},\mathit{int}), \texttt{safe\_aux}(\mathit{il},\mathit{int},\mathit{int}), \texttt{no\_diag}(\mathit{int},\mathit{int},\mathit{int}),\\
&\qquad \texttt{=\textbackslash=}(\mathit{int},\mathit{int})\}
\end{aligned}
$$

Fig. 8. A program for *n*-queens.

for `permute(Sol,[4|T])` to be correctly typed in its input position: `permute` is called with *insufficient input*. ◁

To ensure termination, each atom that may loop when called with insufficient input should be placed sufficiently late; all producers of input for that atom must occur textually earlier. This assumes left-based derivations. Note that this explains in particular why in the recursive clause for `permute`, the recursive call should be placed last, and hence we are effectively refining the heuristic proposed by Naish (1992). Note also that in nicely and well moded programs, *all* atoms are placed sufficiently late in this sense.

In the next subsection, we identify the *robust* predicates, which are predicates for which all delay-respecting derivations are finite. In section 5.2, we prove termination for programs where the atoms using non-robust predicates are selected ‘sufficiently late’.

5.1 Robust predicates

In this subsection, derivations are *not* required to be left-based. The statements hold for arbitrary delay-respecting derivations, and thus the textual position of an atom in a query is irrelevant. Therefore we can, for just this subsection, assume that the programs and queries are robustly typed (rather than just *permutation* robustly typed). This simplifies the notation. In section 5.2, we go back to allowing for arbitrary permutations.

Definition 5.2
[robust] Let P be a robustly typed, input-linear program with input selectability. A predicate p in 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. $\lhd$

By definition, a delay-respecting derivation for a query consisting of *one* robust atom terminates. We will see shortly however that this extends to queries of *arbitrary* length. To prove this, we first need the following simple lemma.

Lemma 5.3
Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a robustly typed query. Then there exists a substitution σ such that $dom(\sigma) = vars(\mathbf{t}_1, \dots, \mathbf{t}_{n-1})$, and $p_n(\mathbf{s}_n, \mathbf{t}_n)\sigma$ is robustly typed.

Proof
Since Q is robustly typed and types are closed under instantiation, there exists a substitution σ such that $dom(\sigma) = vars(\mathbf{t}_1, \dots, \mathbf{t}_{n-1})$, $ran(\sigma) = \emptyset$, and $(\mathbf{t}_1, \dots, \mathbf{t}_{n-1})\sigma$ is correctly typed.

Since Q is nicely moded, $dom(\sigma) \cap vars(\mathbf{t}_n) = \emptyset$. Since $ran(\sigma) = \emptyset$, it follows that $vars(\mathbf{s}_n\sigma) \cap vars(\mathbf{t}_n\sigma) = \emptyset$ and hence $p_n(\mathbf{s}_n, \mathbf{t}_n)\sigma$ is nicely moded.

Since Q is well typed, it follows by Def. 3.5 that $p_n(\mathbf{s}_n, \mathbf{t}_n)\sigma$ is well typed.

Therefore, as Q is robustly typed and $\mathbf{t}_n\sigma = \mathbf{t}_n$, it follows that $p_n(\mathbf{s}_n, \mathbf{t}_n)\sigma$ is robustly typed. $\square$

The following lemma says that a robust atom cannot proceed indefinitely unless it is repeatedly ‘fed’ by some other atom.

Lemma 5.4
Let P be a robustly typed, input-linear program with input selectability and F, b, H a robustly typed query where b is a robust atom. A delay-respecting derivation of $P \cup \{F, b, H\}$ can have infinitely many b-steps only if it has infinitely many a-steps, for some $a \in F$. (For proof, see the Appendix.)

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

Lemma 5.5
Let P be a robustly typed, input-linear 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
Let ξ be an infinite delay-respecting derivation of $P \cup \{Q\}$. Assume, for the purpose of deriving a contradiction, that ξ contains only finitely many steps where a non-robust atom is resolved. Then there exists an infinite suffix $\tilde{\xi}$ of ξ containing no steps where a non-robust atom is resolved. Consider the first query $\tilde{Q}$ of $\tilde{\xi}$. Then there is at least one atom in $\tilde{Q}$ that has infinitely many descendants. Let $\tilde{a}$ be the leftmost of these atoms. Then as $\tilde{a}$ is robust, we have a contradiction to Lemma 5.4. $\square$

Approaches to termination usually rely on measuring the size of the *input* in a query. We agree with Etalle *et al.* (1999) that it is reasonable to make this dependency explicit. This gives rise to the notion of *moded level mapping*, which is an instance of *level mapping* introduced by Bezem (1993) and Cavedon (1989). Since we use well typed programs instead of well moded ones (Etalle *et al.*, 1999), we have to generalise the concept further.

In the following definition, $\mathbf{B}_P^{Inp}$ denotes the set of atoms using predicates occurring in P, that are correctly typed in their input positions.

Definition 5.6
[moded typed level mapping] Let P be a program. A function $|.| : \mathbf{B}_P^{Inp} \rightarrow \mathbb{N}$ is a *moded typed level mapping* if for each $p(\mathbf{s},\mathbf{t}) \in \mathbf{B}_P^{Inp}$

- for any $\mathbf{u}$, we have $|p(\mathbf{s},\mathbf{t})| = |p(\mathbf{s},\mathbf{u})|$;
- for any substitution θ, $|p(\mathbf{s},\mathbf{t})| = |p(\mathbf{s}\theta,\mathbf{t})|$.

For $a \in \mathbf{B}_P^{Inp}$, $|a|$ is the *level* of a. $\lhd$

Thus, the level of an atom in $\mathbf{B}_P^{Inp}$ only depends on the terms in the input positions. Moreover, all *instances* of an atom in $\mathbf{B}_P^{Inp}$ have the same level. Here our concept differs from moded level mappings. Also, our concept is defined for atoms in $\mathbf{B}_P^{Inp}$ that are not necessarily ground, but this difference only concerns the presentation.

Since we only consider moded typed level mappings, we will simply call them *level mappings*.

The following standard concept is widely used in the termination literature (Apt, 1997).

Definition 5.7
[Depends on] Let p,q be predicates in a program P. Then *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 $q \sqsupseteq p$.

Abusing notation, we shall also use the above symbols for *atoms*, where $p(\mathbf{s},\mathbf{t}) \sqsupseteq q(\mathbf{u},\mathbf{v})$ stands for $p \sqsupseteq q$, and likewise for $\sqsupset$ and $\approx$. Furthermore, we denote the equivalence class of a predicate p with respect to $\approx$ as $[p]_\approx$. $\lhd$

The following concept is used to show robustness.

Definition 5.8
[well-recurrent] Let P be a program and $|.|$ a level mapping. A clause $C = h \leftarrow B$ is *well-recurrent (with respect to* $|.|$*)* if, for every a in B such that $a \approx h$, and every substitution θ such that $a\theta, h\theta \in \mathbf{B}_P^{Inp}$, we have $|h\theta| > |a\theta|$.

A program (set of clauses) is *well-recurrent with respect to* $|.|$ if each clause is well-recurrent with respect to $|.|$. $\lhd$

Well-recurrence resembles *well-acceptability* (Etalle *et al.*, 1999) in that only for atoms $a \approx h$ there has to be a decrease, and that it assumes moded level mappings. It differs from well-acceptability, but also from *delay-recurrence* (Marchiori and Teusink, 1999), in that it does not refer to a model of the program.

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 5.9
Let P be a robustly typed, input-linear program with input selectability and p a predicate in P. Suppose all predicates q with $p \sqsupset q$ are robust, and all clauses defining predicates $q \in [p]_{\approx}$ are well-recurrent with respect to some level mapping $|.|$. Then p is robust. (For proof, see the Appendix.)

Example 5.10
We demonstrate for the program in figure 8, with mode M_1 and type T, how Lemma 5.9 is used.[8] Given that the built-in `=\=` terminates, it follows that `no_diag` is robust. With the list length of the first argument of `safe_aux` as level mapping, the clauses defining `safe_aux` are well-recurrent so that `safe_aux` is robust. In a similar way, we can show that `safe` is robust. $\lhd$

Example 5.11
Consider again Example 4.9. We conjecture that `is_list` is robust, but Lemma 5.9 cannot show this. While Example 4.9 is contrived, it suggests that the method of section 4.2 might be useful whenever Lemma 5.9 fails to prove that a predicate is robust. On the other hand, one could envisage to improve the method for showing robustness, for example by exploiting information given by a *model* of the program (Etalle *et al.*, 1999). $\lhd$

5.2 *Well fed programs*

As seen in Example 5.1, there are predicates for which requiring delay-respecting derivations is not sufficient for termination. In general, the selection rule must be taken into account. We assume left-based derivations. Consequently, we now 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 to permutation robustly typed in the obvious way.

A *safe* position in a query is a position that is 'sufficiently late'.

[8] We assume that the built-ins used here meet the conditions of Definition 3.17. We will see in section 7.1 why this is a safe assumption.

Definition 5.12
[Safe position] For a permutation π, i is a called *safe position for* π if for all j, $\pi(j) < \pi(i)$ implies $j < i$. $\lhd$

Whenever we simply speak of an *atom in a safe position*, we mean that this atom occurs in a π-robustly typed query Q in the ith position and i is a safe position for π, where Q and π are clear from the context.

The next lemma says that in a left-based derivation, atoms whose ancestors are all in safe positions can never be waiting (see Definition 2.2).

Lemma 5.13
Let P be a permutation robustly typed program with input selectability, Q_0 a permutation robustly typed query and $\xi = Q_0;\ldots;Q_i\ldots$ a left-based derivation of $P \cup \{Q_0\}$. Then no atom in Q_i for which all ancestors are in safe positions is waiting.

Proof
Suppose $Q_i = a_1,\ldots,a_n$ is π_i-robustly typed (note that π_i exists by Lemma 3.16). Let a_k be an atom in Q_i with all its ancestors in safe positions. By Def. 3.5, $a_{\pi_i^{-1}(1)}$ is correctly typed in its input positions, and hence selectable. Moreover, since k is a safe position, ${\pi_i}^{-1}(1) \leqslant k$. It follows that if the *proper ancestors* of a_k are not waiting, then a_k is not waiting.

The result follows by induction on i. When $i = 0$, a_k has no proper ancestors and hence, by the above paragraph, a_k is not waiting. When $i > 0$, then all proper ancestors of a_k are in safe positions (by hypothesis) and hence, by the inductive hypothesis, they are not waiting. Thus, by the above paragraph, a_k is not waiting. $\square$

To show Theorem 5.18, we need the following corollary of Lemma 5.13.

Corollary 5.14
Make the same assumptions as in Lemma 5.13. If $Q_i = a_1,\ldots,a_n$ is π_i-robustly typed and the atom a_k selected in $Q_i;Q_{i+1}$ has only ancestors in safe positions, then $\pi_i(k) = 1$ (and hence a_k is correctly typed in its input positions).

A permutation robustly typed query is called *well fed* if each atom is robust or in a safe position. Note that if a predicate p can be shown to be robust using Lemma 5.9, then all predicates q with $p \sqsupset q$ are also robust. However, this is a property of the method for showing robustness, not of robustness itself. To simplify the proof of Theorem 5.18, we want to exclude the pathological situation that p is robust but some predicate q with $p \sqsupset q$ is not.

Definition 5.15
[Well fed] A π-robustly typed query is *well fed* if for each of its atoms $p(\mathbf{s},\mathbf{t})$, either $p(\mathbf{s},\mathbf{t})$ is in a safe position for π, or all predicates q with $p \sqsupseteq q$ are robust. A clause is well fed if its body is. A program P is *well fed* if all of its clauses are well fed and input-linear, and P has input selectability. $\lhd$

The following proposition is a direct consequence of the definition of the derived permutation (see Lemma 3.3).

Proposition 5.16
Let P and Q be a well fed program and query, and ξ a derivation of $P \cup \{Q\}$. Then each atom in each query in ξ is either robust, or all its ancestors are in safe positions.

Example 5.17
The program in figure 4 is well fed in both modes. The program in figure 8 is well fed in mode M_1. It is not well fed in mode M_2, because it is not permutation nicely moded in this mode: in the second clause for `sequence`, `N1` occurs twice in an output position. $\lhd$

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

Theorem 5.18
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 see Appendix*)

Given that for the programs of figures 3, 5, 4 and 8, the corresponding robustly typed programs terminate for robustly typed queries, it follows by the above theorem that the original programs terminate for well fed queries.

For the program of figure 8, our method can only show termination for the mode M_1, but not for M_2, although the program actually terminates for M_2 (provided the `block` declarations are modified to allow for M_2).

6 Freedom from errors related to built-ins

One problem with built-ins is that their implementation may not be written in Prolog. Thus, for the purposes of this paper, it is assumed that each built-in is conceptually defined by possibly infinitely many (fact) clauses (Sterling and Shapiro, 1986). For example, there could be facts '`0 is 0+0.`', '`1 is 0+1.`', and so forth.

To prove that a program is free from errors related to built-ins, we require it to be permutation simply typed. This applies also to the conceptual clauses for the built-ins.

Some built-ins produce an error if certain arguments have a wrong type, and others produce an error if certain arguments are insufficiently instantiated. For example, `X is foo` results in a type error and `X is V` results in an instantiation error.

The approach described here aims at preventing instantiation and type errors for built-ins, for example arithmetic built-ins, that require arguments to be ground. It has been proposed (Apt and Luitjes, 1995) that these predicates be equipped with delay declarations to ensure that they are only executed when the input is *ground*. This has the advantage that one can reason about arbitrary arithmetic expressions, say `qsort([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 similar to and based on Lemma 27 (Apt and Luitjes, 1995).

Lemma 6.1
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 each $i \in \{1, \ldots, n\}$. Suppose, for some $k \in \{1, \ldots, n\}$, $\mathbf{S}_k$ is a vector of constant types, $\mathbf{s}_k$ 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 $\pi(j) < \pi(k)$. Then $\mathbf{s}_k : \mathbf{S}_k$.

Proof
By Definition 3.5, $\mathbf{s}_k\theta : \mathbf{S}_k$, and thus $\mathbf{s}_k\theta$ is a vector of constants. Since $\mathbf{s}_k$ is already a vector of non-variable terms, it follows that $\mathbf{s}_k$ is a vector of constants and thus $\mathbf{s}_k\theta = \mathbf{s}_k$. Therefore $\mathbf{s}_k : \mathbf{S}_k$. □

Note that if $\mathbf{s}_k$ is of type $\mathbf{S}_k$, then $\mathbf{s}_k$ is ground. By Definition 3.8, 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 6.1, if 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. This is stated in the following theorem which is a consequence of Lemma 6.1.

Theorem 6.2
Let P be a permutation simply typed, input-linear program with input selectability and Q be a permutation simply typed query. Let p be a predicate whose input positions are all bound and of constant type. Then in any delay-respecting derivation of $P \cup \{Q\}$, an atom using p will be selected only when its input arguments are correctly typed.

When we say that the input positions of a built-in are bound, we imply that the conceptual clause heads have non-variable terms in those positions.

Example 6.3
For the program in figure 3 in mode M_1, no delay-respecting derivation for a permutation simply typed query and this program can result in an instantiation or type error related to the arithmetic built-ins.

7 `block` declarations and equality tests

Runtime testing for instantiation has an overhead, and in the case of built-ins, can only be realised by introducing an auxiliary predicate (see figure 3). Therefore, in the following two subsections, we describe ways of simplifying the `block` declarations of a program. An additional benefit is that in some cases, we can even ensure that arguments are ground, rather than just non-variable. We will see in section 7.3 that this is useful in order to weaken the restriction that every clause head must be input-linear. We have postponed these considerations so far in order to avoid making the main arguments of this paper unnecessarily complicated.

7.1 Avoiding `block` *declarations for permutation simply typed programs*

In the program in figure 8, there are no `block` declarations and hence no auxiliary predicates for `<`, `is` and `=\=`. This is justified because the input for those predicates is always provided by the clause heads. For example, it is not necessary to have a `block` declaration for `<` because when an atom using `sequence` is called, the first argument of this atom is already ground. We show here how this intuition can be formalised. In the following definition, we consider a set $\mathcal{B}$ containing the predicates for which we want to omit the `block` declarations.

Definition 7.1
[$\mathcal{B}$-ground] Let P be a permutation simply typed program and $\mathcal{B}$ a set of predicates whose input positions are all of constant type.

A query is *$\mathcal{B}$-ground* if it is permutation simply typed and each atom using a predicate in $\mathcal{B}$ has ground terms in its input positions.

An argument position k of a predicate p in P is a *$\mathcal{B}$-position* if there is a clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ in P such that for some i where $p_i \in \mathcal{B}$, some variable in $\mathbf{s}_i$ also occurs in position k in $p(\mathbf{t}_0, \mathbf{s}_{n+1})$.

The program P is *$\mathcal{B}$-ground* if every $\mathcal{B}$-position of every predicate in P is an input position of constant type, and an atom $p(\mathbf{s}, \mathbf{t})$, where $p \notin \mathcal{B}$, is selectable only if it is non-variable in the $\mathcal{B}$-positions of p. $\triangleleft$

Note that since a constant type is, in particular, a *non-variable* type, it is always possible to find `block` declarations such that both the requirement on selectability in the above definition and in Definition 3.17 (2) are fulfilled.

Example 7.2
The program in figure 8 is $\mathcal{B}$-ground, where $\mathcal{B} = \{$`<`, `is`, `=\=`$\}$. The first position of `sequence`, the second position of `safe_aux`, and all positions of `no_diag` are $\mathcal{B}$-positions. $\triangleleft$

The following theorem says that for $\mathcal{B}$-ground programs, the input of all atoms using predicates in $\mathcal{B}$ is always ground.

Theorem 7.3
Let P be a $\mathcal{B}$-ground, input-linear program with input selectability, Q a $\mathcal{B}$-ground query, and ξ a delay-respecting derivation of $P \cup \{Q\}$. Then each query in ξ is $\mathcal{B}$-ground.

Proof
The proof is by induction on the length of ξ. Let $Q_0 = Q$ and $\xi = Q_0; Q_1; \dots$. The base case holds by the assumption that Q_0 is $\mathcal{B}$-ground.

Now consider some Q_j where $j \geqslant 0$ and Q_{j+1} exists. By Lemma 3.11 (4), Q_j and Q_{j+1} are permutation simply typed and hence type-consistent in all argument positions. The induction hypothesis is that Q_j is $\mathcal{B}$-ground.

Let $p(\mathbf{u}, \mathbf{v})$ be the selected atom, $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be the clause and θ the MGU used in the step $Q_j; Q_{j+1}$. Consider an arbitrary $i \in \{1, \dots, n\}$ such that $p_i \in \mathcal{B}$.

If $p \notin \mathcal{B}$, then by the condition on selectability in Def. 7.1, $p(\mathbf{u}, \mathbf{v})$ is non-variable in the $\mathcal{B}$-positions of p and hence, since the $\mathcal{B}$-positions are of constant type, $p(\mathbf{u}, \mathbf{v})$ is *ground* in the $\mathcal{B}$-positions of p. If $p \in \mathcal{B}$, then $p(\mathbf{u}, \mathbf{v})$ is ground in all input positions by the induction hypothesis, and hence $p(\mathbf{u}, \mathbf{v})$ is a fortiori ground in all $\mathscr{B}$-positions of p.

Thus it follows that $\mathbf{s}_i\theta$ is ground. Since the choice of i was arbitrary, and because of the induction hypothesis, it follows that Q_{j+1} is $\mathscr{B}$-ground. $\square$

In Theorem 7.3, the assumption that the predicates in $\mathscr{B}$ have input selectability is redundant. Atoms using predicates in $\mathscr{B}$ are only selected when their input is ground, simply because their input is ground at all times during the execution.

Example 7.4
In the program in figure 8, there are no `block` declarations, and hence no auxiliaries, for the occurrences of `is`, `<` and `=\=`, but there are `block` declarations on `safe_aux` and `no_diag` that ensure the condition on selectability in Definition 7.1. $\lhd$

7.2 Simplifying the `block` *declarations using atoms in safe positions*

By a simple observation, we can simplify the `block` declarations for predicates that are only used in atoms occurring in safe positions. Consider a permutation robustly typed program P with input selectability and a permutation robustly typed query Q. Suppose we have a predicate p such that for all q with $q \sqsupseteq p$, all atoms using q in Q and clause bodies in P are in safe positions.

Then by Lemma 5.13, in any left-based derivation of $P \cup \{Q\}$, an atom using p is never waiting. Thus, the `block` declarations do not delay the selection of atoms using p. Suppose we modify P by replacing the `block` declaration for p with the empty `block` declaration. Then the modified program has the same set of left-based derivations of Q as the original program. For example, the `block` declaration for `sequence` in the program in figure 8 can be omitted.

7.3 Weakening input-linearity of clause heads

The requirement that clause heads are input-linear is needed to show the persistence of permutation nicely-modedness (Lemma 3.3). This is analogous to the same statement restricted to nicely-modedness (Apt and Luitjes, 1995, Lemma 11). However, the clause head does not have to be input-linear when the statement is further restricted to *LD*-resolvents (Apt and Pellegrini, 1994, Lemma 5.3). The following example by Apt (personal communication) demonstrates this difference.

Example 7.5
Consider the program

```
q(A).              r(1).              eq(A,A).
```

where the mode is $\{\mathtt{q}(I), \mathtt{r}(O), \mathtt{eq}(I, I)\}$. Note that `eq`/2 is equivalent to the built-in =/2. This program is nicely moded but not input-linear. The query

$$\mathtt{q(X),\ r(Y),\ eq(X,Y)}$$

is nicely moded. The query $\mathtt{q(X), r(X)}$ is a resolvent of the above query, and it is *not* nicely moded. $\lhd$

Requiring clause heads to be input-linear is undoubtedly a severe restriction. It means that it is not possible to check two input arguments for equality. However, this also indicates the reason why in the above example, resolving `eq(X,Y)` is harmful: `eq` is meant to be a check, clearly indicated by its mode `eq`(I,I), but in the given derivation step, it actually is not a check, since it binds variables.

It is easy to see that Lemma 3.3 still holds if Definition 3.1 is weakened by allowing `=` to be used in mode `=`(I,I), provided atoms using `=` are only resolved when both arguments are ground. Resolving the permutation nicely moded query $Q_1, s{=}t, Q_2$ selecting $s{=}t$, where s and t are ground, will yield the resolvent Q_1, Q_2, which is permutation nicely moded.

The mode `=`(I,I) can be realised with a delay declaration such that an atom $s{=}t$ is selected only when s and t are ground. In SICStus, this can be done using the built-in `when` (SIC, 1998). However we do not follow this line because this paper focuses on `block` declarations, and because it would commit a particular occurrence of $s{=}t$ to be a test in all modes in which the program is used.

Nevertheless, there are at least two situations when clause heads that are not input-linear can be allowed. First, one can exploit the fact that atoms are in safe positions, and secondly, that the arguments being checked for equality are of constant type.

In the first case, we assume left-based derivations. We could allow for clause heads $p(\mathbf{t},\mathbf{s})$ where a variable x occurs in several input positions, provided that

- all occurrences of x in $\mathbf{t}$ are in positions of ground type, and
- for each clause body and initial query for the program, each atom using a predicate q with $q \sqsupseteq p$ is in a safe position.

By Corollary 5.14, it is then ensured that multiple occurrences of a variable in the input of a clause head implement an equality *check* between input arguments. Therefore, Lemmas 3.3, 3.11 and 3.16 hold assuming this weaker definition of 'input-linear'.

Example 7.6

Consider the program in figure 1 in mode $\{$`permute`(I,I), `delete`$(I,I,I)\}$. This program is not input-linear. Nevertheless, the program can be used in this mode provided that all arguments are of ground type and calls to `permute` and `delete` are always in safe positions. ◁

In the second case, it is sufficient to assume delay-respecting derivations. We can use Theorem 6.2. This time, we have to allow for clause heads $p(\mathbf{t},\mathbf{s})$ where a variable x occurs in several input positions, provided that

- x only occurs *directly* and in positions of constant type in $\mathbf{t}$, and
- an atom using p is selectable only if these positions are non-variable.

It is then ensured that when an atom $p(\mathbf{u},\mathbf{v})$ is selected, $\mathbf{u}$ has constants in each position where $\mathbf{t}$ has x.

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

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

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

Fig. 9. The `length` program.

Example 7.7
Consider the program shown in figure 9. It can be used in mode $\{$`length`(O,I), `len_aux`$(O,I,I)\}$ (it is simply typed) in spite of the fact that `len_aux([],N,N)` is not input-linear, using either of the two explanations above. The first explanation relies on all atoms using predicates $q \sqsupseteq$ `len_aux` being in safe positions. This is somewhat unsatisfactory since imposing such a restriction impedes modularity. Therefore, the second explanation is preferable. $\lhd$

8 Related work

First of all, note that our work implicitly relies on previous work on termination for LD-derivations (Apt, 1997; De Schreye and Decorte, 1994), since we reduce the problem of termination of a program with `block` declarations to the classical problem of termination for LD-derivations.

In using modes and types, we follow Apt and Luitjes (1995), and also adopt their notation. They show occur-check freedom for nicely moded programs and non-floundering for well typed programs. For arithmetic built-ins they require delay declarations such that an atom is delayed until the arguments are ground. Such declarations are usually implemented not as efficiently as `block` declarations. For termination, they propose a method limited to deterministic programs.

Naish (1992) gives good intuitive explanations (without proof) why programs loop, which directed our own search for further ideas and their formalisation. Predicates are assumed to have a single mode. It is suggested that alternative modes should be achieved by multiple versions of a predicate. This approach is quite common (Apt and Etalle, 1993; Apt and Luitjes, 1995; Etalle *et al.*, 1999) and is also taken in Mercury (Somogyi *et al.*, 1996), where these versions are generated by the compiler. While it is possible to take that approach, this is clearly a loss of generality since two different versions of a predicate is not the same thing as a single one which can be used in several modes. Naish uses examples where, under the above assumption, delay declarations are unnecessary. For `permute`, if we only consider the mode M_2, then the program in figure 5 does not loop simply because no atom is ever delayed, and thus the program behaves as if there were no delay declarations. In this case, the interpretation that one should ‘place recursive calls last’ is misleading. If we only consider the mode M_1, then the version of figure 5 is much less efficient than figure 1. In short, his discussion on delay declarations lacks motivation when only one mode is assumed.

Lüttringhaus-Kappel (1993) proposes a method for 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 aims for 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. Lüttringhaus-Kappel states that all programs he has considered are safe, but he gives no hint as to how this might be shown in general.

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, "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.

Furthermore, the method assumes arbitrary delay-respecting derivations and hence does not work for programs where termination depends on derivations being left-based.

Marchiori and Teusink (1999) 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 that always selects an atom that was introduced in the most recent step. No existing language using a local selection rule (other than the LD selection rule) is mentioned, and we are not aware that there is one. 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.

Martin and King (1997) ensure termination by imposing a depth bound on the SLD tree. This is realised by a program transformation introducing additional argument positions for each predicate, which are counters for the depth of the computation. The difficulty is of course to find an appropriate depth bound that does not compromise completeness. It is hard to compare their work to ours since they transform the programs substantially to obtain programs for which it is easier to reason about termination, whereas we show termination for much more 'traditional' programs.

Recently, Pedreschi and Ruggieri (1999) have shown that for programs that have no failing derivations, termination is independent of the selection rule. They consider *guarded* clauses, and the execution model is such that the evaluation of guards is never considered as a failure. For example, even the `quicksort` program is non-failing in this sense, since the tests `leq(X,C)` and `grt(X,C)` (see figure 3) would be guards. In contrast to the method presented in section 4.2, they can show termination for this program.

The verification methods used here can also be used to show that programs are free from (full) unification, occur-check, and floundering. These relatively straightforward generalisations of previous results (Apt and Etalle, 1993; Apt and Luitjes, 1995; Apt and Pellegrini, 1994) are discussed in Smaus' PhD thesis (1999).

9 Discussion and future work

We have presented verification methods for programs with `block` declarations. The verified properties were termination and freedom from errors related to built-ins. These methods refine and formalise previous work in this area (Apt and Etalle, 1993; Apt and Luitjes, 1995; Naish, 1992).

In the introduction, we have said that this work has three distinctive features: (a) assuming multiple modes, (b) using `block` declarations, (c) formalising the 'default left-to-right' selection rule. While the significance of (a) can be argued (see below), at least features (b) and (c) mean that we are addressing existing programs and existing language implementations. This is further strengthened by the fact that, using the results of section 7, we can verify programs where only some of the predicates are equipped with `block` declarations.

In the literature, we also find other types of delay declarations: In Gödel (Hill and Lloyd, 1994), delay declarations can test for non-variableness of sub-arguments up to a certain depth (e.g. `DELAY P([x|xs]) UNTIL NONVAR(xs)`) or for groundness of arguments; also, in theory, one can consider delay declarations that test arguments for being instantiated to a list or similar structure (Lüttringhaus-Kappel, 1993). Most of our results require that an atom is selected only if certain arguments are *at least* non-variable, and so they trivially also hold for those delay declarations. On the other hand, the results in section 5.2 require that an atom is definitely selectable whenever it is correctly typed in its input positions. We claim that this is a natural requirement which should also be fulfilled by most programs using other kinds of delay declarations, but to substantiate this claim, we would have to specify precisely the delay declarations and the underlying modes and types.

For proving termination, we have presented two approaches. The first approach (Smaus *et al.*, 1999) consists of two complementing methods based on not *using* and not *making* speculative bindings, respectively. For figures 4 and 5, it turns out that in one mode, the first method applies, and in the other mode, the second method applies. This approach is simple to understand and to apply. However it is rather limited. Termination cannot be shown for the programs of figures 3 and 8.

In the second approach (Smaus *et al.*, 1998), we required programs to be *permutation robustly typed*, a condition that ensures 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 can be placed in clause bodies arbitrarily. Non-robust atoms must be placed such that their input is sufficiently instantiated when they are called.

Concerning built-ins, we have shown that even though some built-ins require their input arguments to be ground, it is still sometimes sufficient to use `block` declarations.

We have also considered how some of the `block` declarations can be omitted if it can be guaranteed that the instantiation tests they implement are redundant. This is useful because even for programs containing `block` declarations, it is rare that *all* predicates have `block` declarations. In particular, it is awkward having to introduce auxiliary predicates to implement delay declarations for built-ins.

It is an ongoing discussion whether it is reasonable to assume predicates that work in several modes (Hill, 1998). We have argued that a formalism dealing with delay declarations should at least *allow* for multiple modes. This does not exclude in any way other applications of delay declarations, such as implementing the test-and-generate paradigm (coroutining). As seen in the program of figure 8, our results apply to such programs as well.

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 mode and type requirements are met. Where applicable, finding the free and bound positions, as well as the level mapping used to prove robustness, should be done by the tool.

Acknowledgements

We are indebted to the anonymous referees whose comments helped improve this paper. We also thank the referees of PLILP/ALP'98 and LOPSTR'98, as well as Krzysztof Apt, Sandro Etalle, Lee Naish, Salvatore Ruggieri and Sofie Verbaeten for useful discussions.

A Proofs

Lemma 3.11
Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a π-simply typed query and $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ a ρ-simply typed, input-linear clause where $vars(C) \cap vars(Q) = \emptyset$. Suppose that for some $k \in \{1, \ldots, n\}$, $\mathbf{s}_k$ is non-variable in all bound input positions, and θ is the MGU of $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$. Then

1. there exist substitutions θ_1, θ_2 such that $\theta = \theta_1\theta_2$ and
   (a) $\mathbf{v}_0\theta_1 = \mathbf{s}_k$ and $dom(\theta_1) \subseteq vars(\mathbf{v}_0)$,
   (b) $\mathbf{t}_k\theta_2 = \mathbf{u}_{m+1}\theta_1$ and $dom(\theta_2) \subseteq vars(\mathbf{t}_k)$,
2. $dom(\theta) \subseteq vars(\mathbf{t}_k) \cup vars(\mathbf{v}_0)$,
3. $dom(\theta) \cap vars(\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n) = \emptyset$,
4. the resolvent of Q and C with selected atom $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is ϱ-simply typed, where ϱ is the derived permutation (see Lemma 3.3).

Proof
By assumption $\mathbf{s}_k$ is non-variable in all bound positions, and $\mathbf{v}_0$ is a linear vector having flat terms in all bound positions, and variables in all other positions. Thus there is a substitution θ_1 such that $\mathbf{v}_0\theta_1 = \mathbf{s}_k$ and $dom(\theta_1) \subseteq vars(\mathbf{v}_0)$, which shows (1a).

Since $\mathbf{t}_k$ is a linear vector of variables, there is a substitution θ_2 such that $dom(\theta_2) \subseteq vars(\mathbf{t}_k)$ and $\mathbf{t}_k\theta_2 = \mathbf{u}_{m+1}\theta_1$, which shows (1b).

Since Q is π-nicely moded, $vars(\mathbf{t}_k) \cap vars(\mathbf{s}_k) = \emptyset$, and therefore $vars(\mathbf{t}_k) \cap vars(\mathbf{v}_0\theta_1) = \emptyset$. Thus it follows by (1b) that $\theta = \theta_1\theta_2$ is a unifier of $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$. (2) follows from (1a) and (1b), and (3) follows from (2) because of linearity.

By Lemma 3.3 and 3.7, the resolvent is ϱ-nicely moded and ϱ-well typed. By (3), the vector of the output arguments of the resolvent is a linear vector of variables, and hence (4) follows. □

Lemma 3.16

Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ a π-robustly typed query and $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \dots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ a ρ-robustly typed, input-linear clause where $vars(Q) \cap vars(C) = \emptyset$. Suppose that for some $k \in \{1, \dots, n\}$, $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is non-variable in all bound input positions and θ is the MGU of $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$.

Then the resolvent of Q and C with selected atom $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is ϱ-robustly typed, where ϱ is the derived permutation (see Lemma 3.3). Moreover $dom(\theta) \cap vars(\mathbf{s}_k) = \emptyset$.

Proof

We show how θ is computed, where we consider three stages. In the first, $\mathbf{s}_k$ and $\mathbf{v}_0$ are unified. In the second, the output positions are unified where the bindings go from C to Q. In the third, the output positions are unified where the bindings go from Q to C. Figure A 1 illustrates which variables are bound in each stage. The first three parts of the proof correspond to the three stages of the unification.

Part 1 (unifying $\mathbf{s}_k$ and $\mathbf{v}_0$). By Definition 3.13, $\mathbf{v}_0$ is a vector of flat terms, where $\mathbf{v}_0^{\mathsf{f}}$ is a vector of variables, and by assumption, $\mathbf{v}_0$ is linear. By assumption, $\mathbf{s}_k^{\mathsf{b}}$ is a vector of non-variable terms and, since $vars(C) \cap vars(Q) = \emptyset$, $vars(\mathbf{v}_0) \cap vars(\mathbf{s}_k) = \emptyset$. Thus there is a (minimal) substitution θ_1 such that $\mathbf{v}_0\theta_1 = \mathbf{s}_k$. We show that the following hold:

(1a) $dom(\theta_1) \cap vars(\mathbf{s}_k) = \emptyset$.

(1b) $dom(\theta_1) \cap vars(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_n) = \emptyset$.

(1c) Let x be a variable occurring directly in a position of type τ in $\mathbf{u}_{m+1}^{\mathsf{b}}\theta_1$. Then $x \notin vars(\mathbf{s}_k)$. Moreover, x can only occur in $\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_n$ in a bound position of type τ, and the occurrence must be direct.

(1d) $vars(\mathbf{u}_{m+1}\theta_1) \cap vars(\mathbf{t}_k) = \emptyset$.

(1a) holds by the construction of θ_1.

(1b) holds since by Definition 3.13 and since C is input-linear, $\mathbf{v}_0, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_n$ is linear.

Let x be a variable occurring directly in a position of type τ in $\mathbf{u}_{m+1}^{\mathsf{b}}\theta_1$. Let y be the variable in the same position in $\mathbf{u}_{m+1}^{\mathsf{b}}$. Suppose, for the purpose of deriving a contradiction, that $y \in vars(\mathbf{v}_0)$. Then by Definition 3.13, y occurs *directly* in $\mathbf{v}_0^{\mathsf{b}}$, and since $\mathbf{s}_k^{\mathsf{b}}$ is a vector of non-variable terms, $y\theta_1$ is not a variable, which is a contradiction. Therefore, $y \notin vars(\mathbf{v}_0)$. Hence $y \notin dom(\theta_1)$ and thus $x = y$ and $x \notin vars(\mathbf{s}_k)$. Furthermore, it follows by Definition 3.13 that x can only occur in

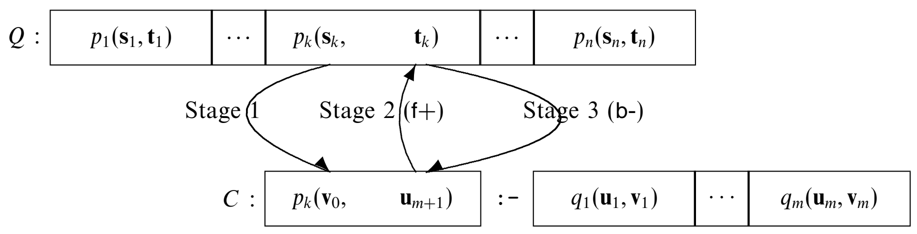

Fig. A 1. Data flow in the unification.

$\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_n$ in a bound position of type τ, and the occurrence must be direct. Thus (1c) holds.

Since Q is permutation nicely moded, $vars(\mathbf{s}_k) \cap vars(\mathbf{t}_k) = \emptyset$ and hence $ran(\theta_1) \cap vars(\mathbf{t}_k) = \emptyset$. Thus (1d) holds.

Part 2 (unifying $\mathbf{t}_k$ and $\mathbf{u}_{m+1}\theta_1$ in each position where either the argument in $\mathbf{t}_k$ is a variable, or the arguments in $\mathbf{t}_k$ and $\mathbf{u}_{m+1}\theta_1$ are both non-variable). Note that this includes all positions in $\mathbf{t}_k^{\mathsf{f}}$ and $\mathbf{u}_{m+1}^{\mathsf{f}}\theta_1$, but may also include positions in $\mathbf{t}_k^{\mathsf{b}}$ and $\mathbf{u}_{m+1}^{\mathsf{b}}\theta_1$. Since, by (1b), $\mathbf{t}_k\theta_1 = \mathbf{t}_k$, Part 2 covers precisely the output positions where the binding 'goes from $\mathbf{u}_{m+1}\theta_1$ to $\mathbf{t}_k\theta_1$' (see figure A 1). We denote by $\mathbf{t}_k^{\mathsf{f+}}$ the projection of $\mathbf{t}_k$ onto the positions where the argument in $\mathbf{t}_k$ is a variable, or the arguments in $\mathbf{t}_k$ and $\mathbf{u}_{m+1}\theta_1$ are both non-variable, and by $\mathbf{t}_k^{\mathsf{b-}}$ the projection onto the other positions, and likewise for $\mathbf{u}_{m+1}\theta_1$.

By (1d), $vars(\mathbf{u}_{m+1}^{\mathsf{f+}}\theta_1) \cap vars(\mathbf{t}_k^{\mathsf{f+}}) = \emptyset$. Thus there is a minimal substitution θ' such that $\mathbf{t}_k^{\mathsf{f+}}\theta' = \mathbf{u}_{m+1}^{\mathsf{f+}}\theta_1$. Let $\theta_2 = \theta_1\theta'$. Then by (1b), $\mathbf{t}_k^{\mathsf{f+}}\theta_2 = \mathbf{u}_{m+1}^{\mathsf{f+}}\theta_2$. We show the following:

(2a) $dom(\theta_2) \cap vars(\mathbf{s}_k) = \emptyset$.

(2b) $dom(\theta_2) \cap vars(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}_k^{\mathsf{b-}}, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n) = \emptyset$.

(2c) Let x be a variable occurring directly in a position of type τ in $\mathbf{u}_{m+1}^{\mathsf{b-}}\theta_2$. Then $x \notin vars(\mathbf{s}_k)$. Moreover, x can only occur in $\mathbf{v}_1, \dots, \mathbf{v}_m$, $\mathbf{t}_1, \dots, \mathbf{t}_{k-1}$, $\mathbf{t}_k^{\mathsf{b-}}$, $\mathbf{t}_{k+1}, \dots, \mathbf{t}_n$ in a bound position of type τ, and the occurrence must be direct.

(2d) $vars(\mathbf{u}_{m+1}\theta_2) \cap vars(\mathbf{t}_k^{\mathsf{b-}}) = \emptyset$.

Since $vars(\mathbf{s}_k) \cap vars(\mathbf{t}_k) = \emptyset$, $dom(\theta') \cap vars(\mathbf{s}_k) = \emptyset$. This and (1a) imply (2a).

(2b) holds because (1b) holds and $\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_n$ is linear.

By (1d), $dom(\theta') \cap vars(\mathbf{u}_{m+1}^{\mathsf{b-}}\theta_1) = \emptyset$. This together with (1c) implies (2c). Furthermore, because of the linearity of $\mathbf{t}_k$, (2d) follows.

Part 3 (unifying $\mathbf{t}_k^{\mathsf{b-}}$ and $\mathbf{u}_{m+1}^{\mathsf{b-}}\theta_2$). By (1d), $dom(\theta') \cap vars(\mathbf{u}_{m+1}^{\mathsf{b-}}\theta_1) = \emptyset$, and thus $\mathbf{u}_{m+1}^{\mathsf{b-}}\theta_2 = \mathbf{u}_{m+1}^{\mathsf{b-}}\theta_1$. Therefore, by the definition of the superscript $\mathsf{b-}$ in Part 2, $\mathbf{u}_{m+1}^{\mathsf{b-}}\theta_2$ is a vector of variables. By (2d), $vars(\mathbf{u}_{m+1}^{\mathsf{b-}}\theta_2) \cap vars(\mathbf{t}_k^{\mathsf{b-}}) = \emptyset$, so that there is a minimal substitution θ'' such that $\mathbf{u}_{m+1}^{\mathsf{b-}}\theta_2\theta'' = \mathbf{t}_k^{\mathsf{b-}}$. Let $\theta_3 = \theta_2\theta''$. Then, by (2b), we have $\mathbf{u}_{m+1}^{\mathsf{b-}}\theta_3 = \mathbf{t}_k^{\mathsf{b-}}\theta_3$. We show (3a) and (3b).

(3a) $dom(\theta_3) \cap vars(\mathbf{s}_k) = \emptyset$.

(3b) $(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n)\theta_3$ is linear and has flat type-consistent terms in all bound positions and variables in all free positions.

By (2c), $dom(\theta'') \cap vars(\mathbf{s}_k) = \emptyset$. This and (2a) imply (3a).

Suppose x is a variable in $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2$ occurring in a position i of type τ, and x also occurs in $\mathbf{v}_1, \ldots, \mathbf{v}_m$, $\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}$, $\mathbf{t}_{k+1}, \ldots, \mathbf{t}_n$. By (2c), the latter occurrence of x is in a bound position of type τ, and is the *only* occurrence of x in $\mathbf{v}_1, \ldots, \mathbf{v}_m$, $\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}$, $\mathbf{t}_{k+1}, \ldots, \mathbf{t}_n$. Let I be the set of positions where x occurs in $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2$, and let T be the set of terms occurring in $\mathbf{t}^{\mathsf{b}-}_k$ in positions in I. Then T is a set of variable-disjoint, flat terms. Therefore, their most general common instance $x\theta''$ is a flat term and $x\theta''$ is type-consistent with respect to τ. Moreover, since $(\mathbf{v}_1, \ldots, \mathbf{v}_m$, $\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}$, $\mathbf{t}^{\mathsf{b}-}_k, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)$ is linear, we have $vars(x\theta'') \cap vars(\mathbf{v}_1, \ldots, \mathbf{v}_m$, $\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}$, $\mathbf{t}_{k+1}, \ldots, \mathbf{t}_n) = \emptyset$ and therefore it follows that $(\mathbf{v}_1, \ldots, \mathbf{v}_m$, $\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}$, $\mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\theta''$ is a linear vector of type-consistent terms. This and (2b) imply (3b).

Part 4: Defining $\theta = \theta_3$ it follows that $p_k(\mathbf{s}_k, \mathbf{t}_k)\theta = p_k(\mathbf{v}_0, \mathbf{u}_{m+1})\theta$. By (3b) and Lemmas 3.3 and 3.7, the resolvent of Q and C is ϱ-robustly typed. By (3a), we have $\mathbf{s}_k\theta = \mathbf{s}_k$. □

Theorem 4.8

Let P be a non-speculative program with input selectability and P' a simply typed program corresponding to P. Let Q be a permutation simply typed query and Q' a simply typed query corresponding to Q. If there is an infinite delay-respecting derivation of $P \cup \{Q\}$, then there is an infinite LD-derivation of $P' \cup \{Q'\}$.

Proof

For simplicity assume that Q and each clause body do not contain two identical atoms. Let $Q_0 = Q$, $\theta_0 = \emptyset$ and $\xi = \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle; \ldots$ be a delay-respecting derivation of $P \cup \{Q\}$. The idea is to construct an LD-derivation ξ' of $P' \cup \{Q'\}$ such that whenever ξ uses a clause C, then ξ' uses the corresponding clause C' in P'. It will then turn out that if ξ' is finite, ξ must also be finite.

We call an atom *a resolved in ξ at i* if a occurs in Q_i but not in Q_{i+1}. We call *a resolved in ξ* if for some i, a is resolved in ξ at i. Let $Q'_0 = Q'$ and $\theta'_0 = \emptyset$. We construct an LD-derivation $\xi' = \langle Q'_0, \theta'_0\rangle; \langle Q'_1, \theta'_1\rangle; \ldots$ of $P' \cup \{Q'\}$ showing that for each $i \geqslant 0$ the following hold:

(1) If $q(\mathbf{u}, \mathbf{v})$ is an atom in Q'_i that is not resolved in ξ, then $vars(\mathbf{v}\theta'_i) \cap dom(\theta_j) = \emptyset$ for all $j \geqslant 0$.
(2) Let x be a variable such that, for some $j \geqslant 0$, $x\theta_j = f(\ldots)$. Then $x\theta'_i$ is either a variable or $x\theta'_i = f(\ldots)$.

We first show these properties for $i = 0$. Let $q(\mathbf{u}, \mathbf{v})$ be an atom in Q'_0 that is not resolved in ξ. Since $\theta'_0 = \emptyset$, $\mathbf{v}\theta'_0 = \mathbf{v}$. Furthermore, by Corollary 3.12 and Corollary 3.20 and since $q(\mathbf{u}, \mathbf{v})$ is not resolved in ξ, we have $\mathbf{v}\theta_j = \mathbf{v}$ for all j. Thus (1) holds. (2) holds because $\theta'_0 = \emptyset$.

Now assume that for some i, $\langle Q'_i, \theta'_i\rangle$ is defined, Q'_i is not empty, and (1) and (2) hold. Let $p(\mathbf{s}, \mathbf{t})$ be the leftmost atom of Q'_i. We define a derivation step $\langle Q'_i, \theta'_i\rangle; \langle Q'_{i+1}, \theta'_{i+1}\rangle$ with $p(\mathbf{s}, \mathbf{t})$ as the selected atom, and show that (1) and (2) hold for $\langle Q'_{i+1}, \theta'_{i+1}\rangle$.

Case 1: $p(\mathbf{s}, \mathbf{t})$ is resolved in ξ at l for some l. Consider the simply typed clause $C' = h \leftarrow B'$ corresponding to the *uniquely renamed* clause (using the same renaming) used in ξ to resolve $p(\mathbf{s}, \mathbf{t})$. Since $p(\mathbf{s}, \mathbf{t})$ is resolved in ξ at l, $p(\mathbf{s}, \mathbf{t})\theta_l$ is non-variable in all bound input positions. Thus each bound input position of $p(\mathbf{s},\mathbf{t})$ must be filled by a non-variable term or a variable x such that $x\theta_l = f(\ldots)$ for some f. Moreover, $p(\mathbf{s},\mathbf{t})\theta'_i$ must have non-variable terms in all bound input positions since $Q'_i\theta'_i$ is well typed. Thus it follows by (2) that in each bound input position, $p(\mathbf{s},\mathbf{t})\theta'_i$ has the same top-level functor as $p(\mathbf{s},\mathbf{t})\theta_l$, and since h has flat terms in the bound input positions, there is an MGU ϕ'_i of $p(\mathbf{s},\mathbf{t})\theta'_i$ and h. We use C' for the step $\langle Q'_i,\theta'_i\rangle;\langle Q'_{i+1},\theta'_{i+1}\rangle$.

We must show that (1) and (2) hold for $i+1$. Consider an atom $q(\mathbf{u},\mathbf{v})$ in Q'_i other than $p(\mathbf{s},\mathbf{t})$. By Lemma 3.11 (3), $vars(\mathbf{v}\theta'_i) \cap dom(\phi'_i) = \emptyset$. Thus for the atoms in Q'_{i+1} that occur already in Q'_i, (1) is maintained. Now consider an atom $q(\mathbf{u},\mathbf{v})$ in B' that is not resolved in ξ. By Corollary 3.20, $\mathbf{v}\theta'_{i+1} = \mathbf{v}$. Since $q(\mathbf{u},\mathbf{v})$ is not resolved in ξ, for all $j > l$ we have that $q(\mathbf{u},\mathbf{v})$ occurs in Q_j and thus by Corollary 3.20, $\mathbf{v}\theta_j = \mathbf{v}$. Thus (1) follows. (2) holds since it holds for i and $p(\mathbf{s},\mathbf{t})$ is resolved using the same clause head as in ξ.

Case 2: $p(\mathbf{s},\mathbf{t})$ is not resolved in ξ. Since P' is non-speculative, there is a (uniquely renamed) clause $C' = h \leftarrow B'$ in P' such that h and $p(\mathbf{s},\mathbf{t})\theta'_i$ have an MGU ϕ'_i. We use C' for the step $\langle Q'_i,\theta'_i\rangle;\langle Q'_{i+1},\theta'_{i+1}\rangle$.

We must show that (1) and (2) hold for $i+1$. Consider an atom $q(\mathbf{u},\mathbf{v})$ in Q'_i other than $p(\mathbf{s},\mathbf{t})$. By Lemma 3.11 (3), $vars(\mathbf{v}\theta'_i) \cap dom(\phi'_i) = \emptyset$. Thus for the atoms in Q'_{i+1} that occur already in Q'_i, (1) is maintained. Now consider an atom $q(\mathbf{u},\mathbf{v})$ in B'. Clearly, $q(\mathbf{u},\mathbf{v})$ is not resolved in ξ. Since $vars(C') \cap vars(Q_j\theta_j) = \emptyset$ for all j and since by Corollary 3.20, we have $\mathbf{v}\theta'_{i+1} = \mathbf{v}$, (1) holds for $i+1$.

By (1) for i, we have $vars(\mathbf{t}\theta'_i) \cap dom(\theta_j) = \emptyset$ for all j. By Lemma 3.11 (2), we have $dom(\phi'_i) \subseteq vars(\mathbf{t}\theta'_i) \cup vars(C')$. Thus we have $dom(\phi'_i) \cap dom(\theta_j) = \emptyset$ for all j. Moreover, (2) holds for i. Thus (2) holds for $i+1$.

Since this construction can only terminate when the query is empty, either Q'_n is empty for some n, or ξ' is infinite.

Thus we show that if ξ' is finite, then every atom resolved in ξ is also resolved in ξ'. So let ξ' be finite of length n. Assume for the sake of deriving a contradiction that j is the smallest number such that the atom a selected in $\langle Q_j,\theta_j\rangle;\langle Q_{j+1},\theta_{j+1}\rangle$ is never selected in ξ'. Then $j \neq 0$ since Q_0 and Q'_0 are permutations of each other and all atoms in Q'_0 are eventually selected in ξ'. Thus there must be a $k < j$ such that a does not occur in Q_k but does occur in Q_{k+1}. Consider the atom b selected in $\langle Q_k,\theta_k\rangle;\langle Q_{k+1},\theta_{k+1}\rangle$. Then by the assumption that j was minimal, b must be the selected atom in $\langle Q'_i,\theta'_i\rangle;\langle Q'_{i+1},\theta'_{i+1}\rangle$ for some $i \leqslant n$. Hence a must occur in Q'_{i+1}, since the clause used to resolve b in ξ' is a simply typed clause corresponding to the clause used to resolve b in ξ. Thus a must occur in Q'_n, contradicting that ξ' terminates with the empty query.

Thus ξ can only be infinite if ξ' is also infinite. □

Lemma 5.4

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

Proof
In this proof, by an *F-step* we mean an a-step, for some $a \in F$; likewise we define an *H-step*. By Corollary 3.19, no H-step can instantiate any descendant of F or b. Thus, the H-steps can be disregarded, and without loss of generality, we assume H is empty. Suppose ξ is a delay-respecting derivation for $P \cup \{F, b\}$ containing only finitely many F-steps.

All F-steps are contained in a finite prefix of ξ. Moreover, by Cor. 3.19, no b-step can instantiate any descendant of F. Therefore, we can repeatedly apply the Switching Lemma (Lloyd, 1987, Lemma 9.1) to this prefix of ξ to obtain a delay-respecting derivation

$$\xi_2 = \langle F,b,\ \emptyset\rangle;\ldots;\langle F',b,\ \rho\rangle;\xi'$$

such that $\langle F,b,\ \emptyset\rangle;\ldots;\langle F',b,\ \rho\rangle$ contains only F-steps and $\langle F',b,\ \rho\rangle;\xi'$ contains only b-steps. Now construct the delay-respecting derivation

$$\xi_3 = \langle b,\rho\rangle;\xi_3'$$

by removing the prefix F' in each query in $\langle F',b,\ \rho\rangle;\xi'$.

By Lemma 3.16, $(F',b)\rho$ is robustly typed. Thus by Lemma 5.3, there exists a substitution σ such that $b\rho\sigma$ is robustly typed, and $dom(\sigma) = V$, where V is the set of variables occurring in the output arguments of $F'\rho$.

By Corollary 3.19, no b-step in ξ_2, and hence no derivation step in ξ_3, can instantiate a variable in V. Since $dom(\sigma) = V$, it thus follows that we can construct a delay-respecting derivation

$$\xi_4 = \langle b,\rho\sigma\rangle;\xi_3'\sigma$$

by applying σ to each query in ξ_3.

Since $b\rho\sigma$ is a robustly typed query and b is robust, ξ_4 is finite. Therefore, ξ_3, ξ_2, and finally ξ are finite. $\square$

Lemma 5.9
Let P be a robustly typed, input-linear program with input selectability and p a predicate in P. Suppose all predicates q with $p \sqsupset q$ are robust, and all clauses defining predicates $q \in [p]_\approx$ are well-recurrent with respect to some level mapping $|.|$. Then p is robust.

Proof
If a is an atom using a predicate in $[p]_\approx$ such that the set $S = \{|a\theta| \mid a\theta \in \mathbf{B}_P^{Inp}\}$ is non-empty and bounded, we define $||a|| = sup(S)$. Thus, for each atom a and substitution θ such that $||a||$ and $||a\theta||$ are defined

$$||a\theta|| \leqslant ||a|| \tag{A 1}$$

To measure the size of a query, we use the multiset containing the level of each atom whose predicate is in $[p]_\approx$. The multiset is formalised as a function *Size*, which takes as arguments a query and a natural number:

$$Size(Q)(n) = \#\{q(\mathbf{u},\mathbf{v}) \mid q(\mathbf{u},\mathbf{v}) \in Q,\ q \approx p \text{ and } ||q(\mathbf{u},\mathbf{v})|| = n\}.$$

Note that if a query contains several identical atoms, each occurrence must be counted. We define $Size(Q) < Size(R)$ if and only if there is $l \in \mathbb{N}$ such that $Size(Q)(l) < Size(R)(l)$ and $Size(Q)(l') = Size(R)(l')$ for all $l' > l$. Intuitively, there is a decrease when an atom in a query is replaced with a finite number of smaller atoms. All descending chains with respect to $<$ are finite (Dershowitz, 1987).

Let $Q_0 = p(\mathbf{s}, \mathbf{t})$ be a robustly typed query. Then $p(\mathbf{s}, \mathbf{t}) \in \mathbf{B}_P^{Inp}$ and thus $||Q_0||$ is defined. Let $\xi = Q_0; Q_1; \ldots$ be a delay-respecting derivation of $P \cup \{Q_0\}$.

Since all predicates q with $p \sqsupset q$ are robust, it follows by Lemma 5.5 that there cannot be an infinite suffix of ξ without any steps where an atom $q(\mathbf{u}, \mathbf{v})$ such that $q \approx p$ is resolved. We show that for all $i \geqslant 0$, if the selected atom in $Q_i; Q_{i+1}$ is $q(\mathbf{u}, \mathbf{v})$ and $q \approx p$, then $Size(Q_{i+1}) < Size(Q_i)$, and otherwise $Size(Q_{i+1}) \leqslant Size(Q_i)$. This implies that ξ is finite, and as the choice of the initial query $Q_0 = p(\mathbf{s}, \mathbf{t})$ was arbitrary, p is robust.

By Lemma 3.16, each position in each atom in Q_{i+1} is filled with a type-consistent term. (*)

Consider $i \geqslant 0$ and let $C = q(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be the clause, $q(\mathbf{u}, \mathbf{v})$ the selected atom and θ the MGU used in $Q_i; Q_{i+1}$.

If $p \sqsupset q$, then $p \sqsupset q_j$ for all $j \in \{1, \ldots, m\}$, and hence by (A 1) and (*) it follows that $Size(Q_{i+1}) \leqslant Size(Q_i)$. Intuitively, the set of atoms that are measured by $Size$ does not change in this step (although the level of each atom might decrease).

Now consider $q \approx p$. Since C is well-recurrent and because of (*), we have $||q(\mathbf{v}_0, \mathbf{u}_{m+1})\theta|| > ||q_j(\mathbf{u}_j, \mathbf{v}_j)\theta||$ for all j with $q_j \approx p$. This together with (A 1) implies $Size(Q_{i+1}) < Size(Q_i)$. Intuitively, one atom has been replaced by smaller atoms in this step, but apart from that, the set of atoms that are measured by $Size$ does not change. □

Theorem 5.18

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

Suppose there is an infinite left-based derivation ξ of $P \cup \{Q\}$. Then letting $Q_0 = Q$, $\theta_0 = \emptyset$, we can write

$$\xi = \langle Q_0, \theta_0 \rangle; \ldots; \langle R_1, \sigma_1 \rangle; \langle Q_1, \theta_1 \rangle; \ldots; \langle R_2, \sigma_2 \rangle; \langle Q_2, \theta_2 \rangle \ldots$$

where $R_1, R_2, \ldots$ are the queries in ξ where a non-robust atom is selected. By Lemma 5.5, there are infinitely many such queries. We derive a contradiction.

By Proposition 5.16, the non-robust atoms in each query in ξ have only ancestors in safe positions. Thus by Cor. 5.14, for each $i > 1$, where R_i is ρ_i-robustly typed, the ${\rho_i}^{-1}(1)$'th atom in R_i is selected in $\langle R_i, \sigma_i \rangle; \langle Q_i, \theta_i \rangle$.

Now consider an arbitrary query $\tilde{Q}$ in ξ and assume it is $\tilde{\pi}$-robustly typed. By Corollary 3.4 and the previous paragraph it follows that there exists a query in ξ that contains no descendants of the $\tilde{\pi}^{-1}(1)$'th atom $\tilde{Q}$. Intuitively, for each query in ξ, the atom that is 'leftmost according to its permutation' will eventually be resolved completely.

By repeatedly applying the Switching Lemma to prefixes of ξ, we can construct a derivation ζ of $P \cup \{Q\}$ such that in each query $\tilde{Q}$ in ζ that is $\tilde{\pi}$-robustly typed, the $\tilde{\pi}^{-1}(1)$'th atom is selected using the same clause (copy) used in ξ. Note that this construction is possible by the previous paragraph. Also note that ζ is infinite.

Now consider the derivation ζ' obtained from ζ by replacing each $\tilde{\pi}$-robustly typed query $\tilde{Q}$ with $\tilde{\pi}(\tilde{Q})$, i.e. the robustly typed query corresponding to $\tilde{Q}$. The derivation ζ' is an LD-derivation of $P' \cup \{Q'\}$, and it is infinite. This is a contradiction. □

References

Apt, K. R. (1997) *From Logic Programming to Prolog*. Prentice Hall.

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

Apt, K. R. and Luitjes, I. (1995) Verification of logic programs with delay declarations. In: V. S. Alagar and M. Nivat (eds.), *Proceedings of AMAST'95: Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Apt, K. R. and Pellegrini, A. (1994) On the occur-check free Prolog programs. *ACM Trans. Programming Lang. and Syst.* **16**(3), 687–726.

Bezem, M. (1993) Strong termination of logic programs. *J. Logic Programming*, **15**(1 & 2), 79–97.

Bossi, A. and Cocco, N. (1999) Successes in logic programs. In: P. Flener (ed.), *Proceedings of the 8th International Workshop on Logic Program Synthesis and Transformation: Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 219–239.

Boye, J. (1996) *Directional Types in Logic Programming*. PhD thesis, Linköpings Universitet.

Bronsard, F., Lakshman, T. K. and Reddy, U. S. (1992) A framework of directionality for proving termination of logic programs. In: K. R. Apt (ed.), *Proceedings of the 9th Joint International Conference and Symposium on Logic Programming*, MIT Press, pp. 321–335.

Cavedon, L. (1989) Continuity, consistency and completeness properties for logic programs. In: G. Levi and M. Martelli (eds.), *Proceedings of the 6th International Conference on Logic Programming*, MIT Press, pp. 571–584.

De Schreye, D. and Decorte, S. (1994) Termination of logic programs: The never-ending story. *J. Logic Programming*, **19/20**, 199–260.

Deransart, P. and Małuszyński, J. (1998) Towards soft typing for CLP. In: F. Fages (ed.), *JICSLP'98 Post-Conference Workshop on Types for Constraint Logic Programming*, École Normale Supérieure. (Available at `http://discipl.inria.fr/TCLP98/`.)

Dershowitz, N. (1987) Termination of rewriting. *J. Symbolic Computation*, **3**(1 & 2), 69–115. (Corrigendum **4**(3), 409–410.)

Etalle, S., Bossi, A. and Cocco, N. (1999) Termination of well-moded programs. *J. Logic Programming*, **38**(2), 243–257.

Hill, P. M. and Lloyd, J. W. (1994) *The Gödel Programming Language*. MIT Press.

Hill, P. M. (ed.) (1998) *ALP Newsletter*. `http://www-lp.doc.ic.ac.uk/alp/`. pp. 17, 18.

Lloyd, J. W. (1987) *Foundations of Logic Programming*. Springer-Verlag.

Lüttringhaus-Kappel, S. (1993) Control generation for logic programs. In: D. S. Warren (ed.), *Proceedings of the 10th International Conference on Logic Programming*, MIT Press, pp. 478–495.

Marchiori, E. and Teusink, F. (1999) Termination of logic programs with delay declarations. *J. of Logic Programming*, **39**(1–3), 95–124.

Martin, J. C. and King, A. M. (1997) Generating efficient, terminating logic programs. In:

M. Bidoit and M. Dauchet (eds.), *Proceedings of TAPSOFT'97: Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 273–284.

Naish, L. (1985) Automatic control of logic programs. *J. Logic Programming*, **2**(3), 167–183.

Naish, L. (1992) Coroutining and the construction of terminating logic programs. *Technical Report 92/5*, University of Melbourne.

Pedreschi, D. and Ruggieri, S. (1999) On logic programs that do not fail. In: S. Etalle and J.-G. Smaus (eds.), *Proceedings of the Workshop on Verification, organised within ICLP'99: Electronic Notes in Theoretical Computer Science 30*, Elsevier.

SIC (1998) *SICStus Prolog User's Manual.* `http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_toc.html`.

Smaus, J.-G. (1999) *Modes and Types in Logic Programming*. PhD thesis, University of Kent at Canterbury. (Available from `www.cs.ukc.ac.uk/pubs/1999/986`.)

Smaus, J.-G., Hill, P. M. and King, A. M. (1998) Termination of logic programs with `block` declarations running in several modes. In: C. Palamidessi (ed.), *Proceedings of the 10th Symposium on Programming Language Implementations and Logic Programming: Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Smaus, J.-G., Hill, P. M. and King, A. M. (1999) Preventing instantiation errors and loops for logic programs with multiple modes using `block` declarations. In: P. Flener (ed.), *Proceedings of the 8th International Workshop on Logic-based Program Synthesis and Transformation: Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 289–307.

Somogyi, Z., Henderson, F. and Conway, T. (1996) The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Logic Programming*, **29**(1–3), 17–64

Sterling, L. and Shapiro, E. (1986) *The Art of Prolog*. MIT Press.