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

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

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

Abstract. We present several verification methods for logic programs with delay declarations, dealing with five aspects of verification: occur-check freedom, flounder freedom, preventing errors related to built-ins, unification freedom, and termination. These methods can be used to verify existing programs, and to assist in writing new programs. Three features are distinctive of this work: we assume that predicates can be used in several modes; we show that `block` declarations, which are a very simple delay construct, are sufficient; we take the selection rule into account, assuming it to be as in most Prolog implementations. Our results on the first two aspects are straightforward generalisations of previous results. The most important contribution of this work is about termination. The method for proving termination is based on identifying the so-called *robust* predicates, for which the textual position of an atom using this predicate is irrelevant.

Note: This report comprises two conference papers [25, 26] and the proofs that have been omitted there for space reasons. Sections 5 and 8 contain new results.

1 Introduction

Delay declarations are provided in several logic programming languages to allow for more user-defined control [15, 16, 27] as opposed to the standard left-to-right selection rule. An atom in a query is selected for resolution only if its arguments are instantiated to a specified degree. This is essential to prevent runtime errors produced by built-in predicates (built-ins), to ensure termination, and can sometimes be exploited to replace unification with matching. In this paper we present various methods for verifying logic programs with delay declarations. Following [2, 3], we consider five aspects of verification: The omission of the occur check should be safe; programs should not flounder (delay indefinitely); there should be no type or instantiation errors with the use of built-ins; programs should (ideally) be unification free; and programs should terminate.

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

(a) Apart from the test-and-generate paradigm (coroutining) [24], allowing procedures to run in more than one mode is probably the most important application of delay declarations. Although other authors [3, 24] have not explicitly assumed multiple modes, their theory and examples only become fully relevant under that assumption, since assuming single-moded predicates, there is often no reason for using delay declarations in the first place. Whether this is a good approach or whether it is better to realise multiple modes by generating multiple versions of each predicate [27] is an ongoing discussion [14].

(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 until they become sufficiently instantiated. As demonstrated in SICStus [16], `block` declarations can be efficiently implemented; the test whether the arguments are non-variable has a negligible impact on performance. Thus such constructs are the most frequently used delay declarations in practice. Termination clearly depends on the instantiation of the arguments of the query. For example, the `append` predicate has infinitely many answers when called with uninstantiated arguments and therefore does not terminate, but it terminates when either the first *or* the third argument is a list of bounded length. Although it cannot be tested in a single step *which* of these arguments is a list of bounded length, we argue that `block` declarations are still sufficient.

(c) The property of termination may critically depend on the selection rule, that is the rule which determines, for a derivation, the order in which atoms are selected. (For the other aspects of verification the selection rule is not crucial.) We assume that derivations are *left-based*, which are derivations where (allowing for some exceptions, concerning the execution order of two literals woken up simultaneously) the left-most non-delayed atom is selected. This is intended to model derivations in the common implementations of Prolog with `block` declarations. Other authors have avoided the issue by abstracting from a particular selection rule [3, 18]; considering left-based selection rules on a heuristic basis [24]; or making the very restrictive assumption of *local selection rules* [19].

Our results on occur-check and flounder freedom are straightforward generalisations of previous work [3]. Our work on built-ins focuses on *arithmetic* built-ins. We exploit that for numbers, being non-variable implies being ground, and show how to prevent *instantiation* and *type* errors. Sometimes, it is not even necessary to have any delay declarations at all for arithmetic built-ins. As a byproduct, we will also be able to show that certain programs are *unification free*, that is, unification can be replaced by matching.

Our most substantial contribution is on termination. As far as possible, we translate the problem to showing termination for a corresponding program with ordinary left-to-right execution. We assume that for the corresponding program, termination has been shown using some existing technique [1, 8, 12]. This way we attempt to isolate the problems for termination that are related to the use of delay declarations, as opposed to any other problems for termination.

A well-known source of loops in programs with delay declarations is *speculative output bindings* [24], that is bindings made before it is known that a solution exists. We propose two simple methods for dealing with this problem and thus proving (or ensuring) termination, which have previously appeared in [25]. Which method must be applied will depend on the program and on the mode being considered. The first method exploits that a program does not *make* any speculative bindings. The second method exploits that a program does not *use* any speculative bindings, by ensuring that no atom ever delays.

However, these two simple methods are quite limited, and therefore we develop them further [26]. We identify predicates which have the undesirable property of looping when they are called with *insufficient* (that is, non-variable but still insufficiently instantiated) input. For instance, the query `permute(A,[1|B])` loops, although the query `permute(A,[1,2])` terminates. The idea for proving termination is that, for such predicates, calls with insufficient input should never arise. This can be ensured by appropriate ordering of atoms in the clause bodies. This actually works in several modes, provided not too many predicates have this undesirable property.

This paper is organised as follows. The next section defines some essential concepts and notations. Section 3 introduces two concepts of "modedness" and "typedness" that are needed later, but which are also useful for proving programs free from occur-check and floundering. Section 4 is about errors related to built-ins. Section 5 shows when programs are unification free. Sections 6 and 7 are about termination. Section 8 considers ways of simplifying the `block` declarations. Section 9 investigates related work. Section 10 concludes with a summary and a look at ongoing and future work.

2 Essential Concepts and Notations

We base the notation on [3, 17]. For the examples we use SICStus notation [16]. 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 $(f(a), b)$.) The set of variables in a syntactic object o is denoted by $vars(o)$. A syntactic object is **linear** if every variable occurs in it at most once. A **flat** term is a variable or a term $f(x_1, \ldots, x_n)$, where $n \geq 0$ and the x_i are distinct variables. The **domain** of a substitution σ is $dom(\sigma) = \{x \mid x\sigma \neq x\}$. The **range** of σ is $ran(\sigma) = \{y \mid y \in vars(x\sigma), y \neq x\}$.

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

A **type** is a set of terms closed under substitution. A type is called **variable type** if it contains variables and **non-variable type** otherwise. A type is called **ground type** if it contains only ground terms. A type is called **constant type** if it contains only (possibly infinitely many) constants. In the examples, we use the following types: *any* is the type containing all terms, *list* the type of all (nil-terminated) lists, *int* the type of integers, *il* the type of all integer lists, and finally, *tree* is the 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". It is assumed that each argument position of each predicate p/n has a type associated with it. These types are indicated by writing the atom $p(T_1, \ldots, T_n)$ where $T_1, \ldots, T_n$ are types. The type of a program is a set of such atoms, one for each predicate. A term t is **typeable with respect to** T if there is a substitution θ such that $t\theta : T$. A term t occurring in an atom in some position is **typeable** if it is typeable with respect to the type of that position.

A `block` declaration [16] for a predicate p/n is a set of atoms of the form $p(b_1, \ldots, b_n)$, where $b_i \in \{?, -\}$ for $i \in \{1, \ldots, n\}$. A **program** consists of a set of clauses and a set of `block` declarations, at most one for each predicate defined by the clauses. If P is a program, an atom $p(t_1, \ldots, t_n)$ is **selectable in** P if either P contains no `block` declaration for p, or for each atom $p(b_1, \ldots, b_n)$ in the `block` declaration for p, there is some $i \in \{1, \ldots, n\}$ such that t_i is non-variable and $b_i = -$.

A **query** is a finite sequence of atoms. Atoms are denoted by a, b, h, queries by B, F, H, Q, R. If $a_1, \ldots, a_n$ is a query, then $a_{i_1}, \ldots, a_{i_m}$, where $1 \leq i_1 < \ldots < i_m \leq n$, is a **subquery** of $a_1, \ldots, a_n$. A **derivation step** for a program P is a pair $\langle Q, \theta\rangle; \langle R, \theta\sigma\rangle$, where $Q = Q_1, a, Q_2$ and $R = Q_1, B, Q_2$ are queries; θ is a substitution; a an atom; $h \leftarrow B$ (a variant of) a clause in P and σ the most general unifier of $a\theta$ and h. We call $a\theta$ (or a) the **selected atom**[1] and $R\theta\sigma$ the **resolvent** of $Q\theta$ and $h \leftarrow B$.

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

If Q, a, R; $(Q, B, R)\theta$ is a step in a derivation, then each atom in $B\theta$ (or B)[1] is a **direct descendant** of a, and $b\theta$ (or b)[1] is a **direct descendant** of b for all $b \in Q, R$. We say b is a **descendant of** a if (b, a) is in the reflexive, transitive closure of the relation *is a direct descendant*. The descendants of a *set* of atoms are defined in the obvious way. If, for a derivation $\ldots Q; \ldots; Q'; Q'' \ldots$, the selected atom in $Q'; Q''$ is a descendant of an atom a in Q, then $Q'; Q''$ is called an a**-step**.

Consider a delay-respecting derivation $Q_0; \ldots; Q_i; Q_{i+1}$, where $Q_i = R_1, a, R_2$, and R_1 contains no selectable atom, and a is *not* the selected

[1] This abuse of terminology of not distinguishing whether the substitution is applied is never a source of ambiguity.

atom in $Q_i; Q_{i+1}$. Then a is **delayed** in $Q_i; Q_{i+1}$. An atom is **waiting** if it is the descendant of a delayed atom. A delay-respecting derivation $Q_0; Q_1 \ldots$ is **left-based** if in each Q_i, a non-waiting atom is selected only if there is no selectable atom to the left of it in Q_i.

3 Permutations and Modes

In [3], *nicely moded* and *well typed* programs are introduced, assuming that each predicate has a single mode. It is then shown that the occur-check can safely be omitted and that derivations do not flounder. Other authors have also used such concepts for various purposes, for example to show that programs are unification free [2], successful [6], and terminating [12].

The idea of these concepts 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. Here "before" refers to the textual position in a query.

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

In examples, an actual permutation is written in the form $\langle \pi(1), \ldots, \pi(n) \rangle$. Given a sequence $o_1, \ldots, o_n$ and a permutation π, we write $\pi(o_1, \ldots, o_n)$ for $o_{\pi^{-1}(1)}, \ldots, o_{\pi^{-1}(n)}$, that is the sequence obtained by applying π to $o_1, \ldots, o_n$.

The formalism of these permutations, used to characterise that one fixed program can run in several modes, is closely linked to the fact that we make the selection rule explicit. Other authors [3, 18] have abstracted away from the selection rule in that the textual position of an atom in a query had no relevance for its selection. Clearly, if textual position is irrelevant, then there is no loss of generality in assuming that the atoms are always "in the right order", or in other words, the permutations are always the identity. In this paper, the textual position is irrelevant for many results, but not for the results on termination, which are the main contribution. For the sake of a uniform presentation we use the permutations throughout. In a first reading of the definitions, it is probably helpful to ignore the permutations.

3.1 Permutation Nicely Moded Programs

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

Definition 3.1 (permutation nicely moded). Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query and π a permutation on $\{1, \ldots, n\}$. Q is **π-nicely moded**

if $\mathbf{t}_1, \dots, \mathbf{t}_n$ is a linear vector of terms and for all $i \in \{1, \dots, n\}$

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

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

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

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

Note that in the clause head, the letter t is used for input and s is used for output, whereas in the body atoms it is vice versa. Also note that in [3], it is not assumed that clause heads are input linear. The implications of this difference are discussed in Sec. 8.

Example 3.1.

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

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

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

Example 3.2.

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

:- 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).

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

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

The problem of *finding* a mode for a program so that it is nicely moded is considered in [7]. We are not concerned with this here.

We show that there is a persistence property for permutation nicely-modedness similar to that for nicely-modedness in [3].

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

Proof. Let $Q = a_1, \ldots, a_n$ be a π-nicely moded query and $h \leftarrow b_1, \ldots, b_m$ be a ρ-nicely moded clause, and suppose for some $k \in \{1, \ldots, n\}$, h and a_k are unifiable with unifier θ. By Def. 3.1, $a_{\pi^{-1}(1)}, \ldots, a_{\pi^{-1}(n)}$ and $h \leftarrow b_{\rho^{-1}(1)}, \ldots, b_{\rho^{-1}(m)}$ are nicely moded. Thus by [3, Lemma 11]

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

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

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

□

Fig. 1 illustrates ϱ when $Q = a_1, a_2, a_3, a_4$, $\pi = \langle 4,3,1,2\rangle$, $C = h \leftarrow b_1, b_2$, $\rho = \langle 2,1\rangle$, and $k = 2$. Thus $\varrho = \langle 5,4,3,1,2\rangle$. The following corollary generalises Lemma 3.1 from a single derivation step to derivations.

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

Proof. Inspection of the definition of ϱ in the proof of Lemma 3.1 shows that the result holds for derivations of length 1. The general result follows by a straightforward induction on the length. □

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

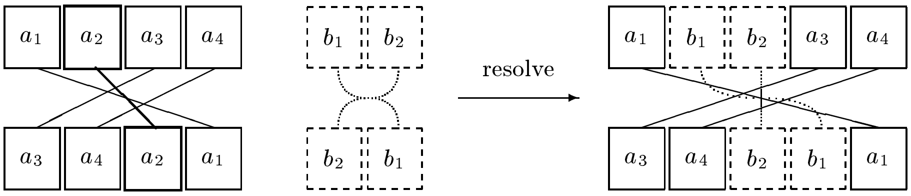

Fig. 1. The permutation ϱ for the resolvent

Definition 3.2 (occur-check free). A derivation is **occur-check free** [3, 5] if no execution of the Martelli-Montanari unification algorithm [21] performed during this derivation ends with a system of term equations including an equation $x = t$, where x is not t, but x occurs in t.

If P and Q are nicely moded, then all derivations of $P \cup \{Q\}$ are occur-check free [3, Thm. 13]. The following theorem is a trivial consequence of this and Lemma 3.1.

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

3.2 Permutation Well Typed Programs

In a *well typed* query, 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*. As with the modes, we assume that the types are given. In the examples, they will be the obvious ones.

Definition 3.3 (permutation well typed). Let $n \geq 0$ and π be a permutation such that $\pi(i) = i$ whenever $i \notin \{1, \dots, n\}$. Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query, where $p_i(\mathbf{S}_i, \mathbf{T}_i)$[2] is the type of p_i for all $i \in \{1, \dots, n\}$. Then Q is **π-well typed** if for all $i \in \{1, \dots, n\}$ and every substitution σ

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

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

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

Note that well-modedness [3] is well-typedness for the special case that the only type is the type containing all ground terms.

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

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

We now give a statement analogous to Lemma 3.1.

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

Proof. Analogous to Lemma 3.1, using Lemma 23 instead of 11 in [3]. $\square$

Permutation well-typedness is useful to show that derivations do not flounder.

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

Proof. Let $\xi = Q_0; Q_1; \ldots$ be a delay-respecting derivation for $P \cup \{Q\}$. Consider an arbitrary $Q_i = a_1, \ldots, a_m$ where $m > 0$. By Lemma 3.4, Q_i is π-well typed for some π. By Def. 3.3, the atom a_j, where $\pi(j) = 1$, is correctly typed in its input positions, and thus is non-variable in its input positions of non-variable type. Therefore a_j is selectable. Thus every non-empty query in ξ contains a selectable atom, and so ξ does not flounder. $\square$

For the program in Ex. 3.3, the above lemma shows that no permutation well typed query can flounder.

4 Errors Related to 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. We take two approaches to ensure freedom from instantiation and type errors. For different programs, different approaches are applicable. Under certain conditions, we even show that no delay declarations are needed at all.

4.1 The Connection between Delay Declarations and Type Errors

At first sight, the problem of type errors seems to have little bearing with delay declarations. Delay declarations cannot enforce that arguments are correctly typed, neither would one expect that a non left-to-right selection rule is to *blame* for wrongly typed arguments.

This is probably true in practice, but in principle, there is an aspect of type errors that is specifically related to delay declarations. Consider the program consisting of the fact clause '`four(4).`' and the built-in `is`, with type $\{\texttt{four}(int)$, $\texttt{is}(int, int)\}$ and mode $\{\texttt{four}(O), \texttt{is}(O, I)\}$. The query

```
X is foo, four(foo)
```

is $\langle 2,1\rangle$-well typed since $\models$ `foo` $\sigma : int \Rightarrow$ `foo` $\sigma : int$ for all substitutions σ. In other words, correctly typed output of `four` implies correctly typed input for `is`, simply because there is no way of making the output correctly typed. The above query results in a type error.

For LD-derivations this problem does not arise. The well typed query corresponding to the above query is `four(foo), X is foo`. Since the type of `four` is *int* and the program is well typed, the atom `four(foo)` can never be resolved away, and therefore the derivation fails without ever reaching `X is foo`.

4.2 Exploiting Constant Types

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

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

Proof. By Def. 3.3, $\mathbf{s}_i\theta : \mathbf{S}_i$, and thus $\mathbf{s}_i\theta$ is a vector of constants. Since $\mathbf{s}_i$ is already a vector of non-variable terms, it follows that $\mathbf{s}_i$ is a vector of constants and thus $\mathbf{s}_i\theta = \mathbf{s}_i$. Therefore $\mathbf{s}_i : \mathbf{S}_i$. □

We define **permutation simply-typedness**. The name is a combination of *simply moded* [3] and *well typed*. In a permutation simply typed query, the output arguments can always be instantiated so that they are correctly typed.

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

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

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

We want to show a persistence property similar to Lemmas 3.1 and 3.4. To this end it is necessary to ensure that a derivation step cannot instantiate the input arguments of the selected atom. This motivates the following definitions.

Definition 4.2 (bound position). Let P be a permutation well typed program. An input position of a predicate p in P is **bound** if there is some clause defining p whose head has a non-variable term in that position, or if p is a built-in[3].

Definition 4.3 (input selectability). Let P be a permutation simply typed program. P has **input selectability** if for every permutation simply typed query Q, an atom in Q is selectable in P if and only if and only it is non-variable in all bound input positions.

Note that by Def. 4.1, in a permutation *simply* typed program, a bound position must be of non-variable type.

In a permutation simply typed query Q, the output positions of each atom are filled with variables. This gives us a way of checking input selectability: the `block` declarations must be such that an atom whose output positions are all variable is selectable if and only if all bound input positions are non-variable.

Lemma 4.2. Every resolvent of a permutation simply typed query Q and a permutation simply typed clause C, where $vars(C) \cap vars(Q) = \emptyset$ and the selected atom is non-variable in all bound input positions[4], is permutation simply typed.

Proof. Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a permutation simply typed query and $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be a permutation simply typed clause. By Lemma 3.1, the resolvent is permutation nicely moded. By Lemma 3.4, the resolvent is permutation well typed.

Let $p_k(\mathbf{s}_k, \mathbf{t}_k)$ be the selected atom and assume it is unifiable with $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$. 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. Furthermore $\mathbf{v}_0, \ldots, \mathbf{v}_m$ is linear and $vars(C) \cap vars(Q) = \emptyset$. Thus there is a substitution θ_1 such that $\mathbf{v}_0\theta_1 = \mathbf{s}_k$, and $dom(\theta_1) \cap vars(Q) = \emptyset$ and $dom(\theta_1) \cap vars(\mathbf{v}_1, \ldots, \mathbf{v}_m) = \emptyset$.

Since Q is permutation nicely moded, $vars(\mathbf{t}_k) \cap vars(\mathbf{s}_k) = \emptyset$. Therefore $vars(\mathbf{t}_k) \cap vars(C\theta_1) = \emptyset$. Furthermore $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a linear vector of variables. Thus there is a substitution θ_2 such that

$$\begin{aligned} \mathbf{t}_k\theta_2 &= \mathbf{u}_{m+1}\theta_1 \\ dom(\theta_2) \cap vars(C\theta_1) &= \emptyset \\ dom(\theta_2) \cap vars(\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n) &= \emptyset. \end{aligned}$$

[3] This definition is inappropriate for built-ins such as `var/1`, but programs using these are probably not permutation simply typed anyway.

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

Therefore $\theta = \theta_1\theta_2$ is a unifier of $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$, and

$$(\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\,\theta$$

is a linear vector of variables.

So the resolvent is permutation nicely moded, permutation well typed, and the vector of its output arguments is a linear vector of variables, and hence the resolvent is permutation simply typed. □

The following two corollaries of Lemma 4.2 are needed in Sections 5 and 6. The first holds since by Def. 3.3, the leftmost atom in a simply typed query is non-variable in its bound input positions.

Corollary 4.3. Every LD-resolvent of a simply typed query Q and a simply typed clause C, where $vars(C) \cap vars(Q) = \emptyset$, is simply typed.

The second corollary 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. Furthermore, each derivation step only instantiates the output variables of the selected atom. The proof is a trivial generalisation of the proof of Lemma 4.2 from a single derivation step to a derivation.

Corollary 4.4. 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 $P \cup \{Q_0\}$ which is either infinite or terminates with the empty goal. Then for all $i \geq 0$ such that Q_i is not empty:

(a) If x is a variable occurring in an output position in Q_i, then $x\theta_i = x$.
(b) If C is the clause, the atom $p(\mathbf{s}, \mathbf{t})$ in Q_i is the selected atom, and ϕ_i is the most general unifier in step $\langle Q_i, \theta_i\rangle; \langle Q_{i+1}, \theta_{i+1}\rangle$, then $dom(\phi_i) \subseteq vars(\mathbf{t}\theta_i) \cup vars(C)$.

Note that for Lemma 4.2, it is sufficient that the selected atom is non-variable in the *bound* input positions. It is not necessary that *all* input positions, or even all input positions of non-variable type, are non-variable.

Example 4.1. Consider Ex. 3.1. In mode `delete`(I, O, I), the third position is the only bound input position. In mode `delete`(O, I, O), the second position is the only bound input position. The `block` declarations ensure input selectability. This is true assuming any type given in Ex. 3.3.

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

Example 4.2. The following is a version of `quicksort`, where `leq` and `greater` are defined in analogy to `less` in Ex. 3.2.

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

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

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

Assume the type $\{$`qs`(il, il), `append`(il, il, il), `leq`(int, int), `greater`(int, int), `part`$(il, int, il, il)\}$. The program is permutation simply typed for mode $\{$`qs`(I, O), `append`(I, I, O), `leq`(I, I), `greater`(I, I), `part`$(I, I, O, O)\}$, and thus there are no instantiation errors. It is not permutation simply typed for mode $\{$`qs`(O, I), `append`(O, O, I), `leq`(I, I), `greater`(I, I), `part`$(O, I, I, I)\}$, due to the non-variable term `[X|Bs2]` in an output position.

Note the `block` declaration for `append`. Given that the possible modes are `append`(I, I, O) and `append`(O, O, I), one might expect a general theory to say that an atom using `append` should be selectable if either the first two or the third argument are non-variable. This would correspond to the `block` declaration

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

However, the simpler `block` declaration is justified by Lemma 4.2 since the second position is not bound. The simpler `block` declarations is the one which is usually given [15, 18, 19], but to the best of our knowledge, it has never been explained on such an abstract level why it is sufficient.

4.3 Atomic Positions

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

Definition 4.4 (respects atomic positions). A query (clause) **respects atomic positions** if each term in an atomic position is ground or a variable which *only* occurs in atomic positions. A program respects atomic positions if each of its clauses does.

A program need not be permutation nicely moded or permutation well typed in order to respect atomic positions.

Note that in Ex. 3.2, we are regarding the atom `M2 is M + 1` as an atom with three arguments `M2`, `M`, and `1`. The program then respects atomic positions.

Lemma 4.5. Every resolvent between a clause C and a query Q that both respect atomic positions, where $vars(C) \cap vars(Q) = \emptyset$, respects atomic positions.

Proof. Let $Q = a_1, \dots, a_n$ be the query and $C = h \leftarrow b_1, \dots, b_m$ be the clause. Let a_k be the selected atom and assume it is unifiable with h using mgu θ. We must show that

$$Q' = (a_1, \dots, a_{k-1}, b_1, \dots, b_m, a_{k+1}, \dots, a_n)\theta$$

respects atomic positions.

Let x be a variable which fills an atomic position in a_k or h. Since Q and C respect atomic positions, $x\theta$ is either a variable which only occurs in atomic positions in Q', or a ground term $(*)$.

First consider a term s filling an atomic position in $a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n$. If s is a ground term, then $s\theta$ is also a ground term. If s is a variable and $s \in dom(\theta)$, then s must fill an atomic position in a_k. By $(*)$ $s\theta$ is either a variable which only occurs in atomic positions in Q', or a ground term.

Now consider a term s filling an atomic position in $b_1, \dots, b_m$. In analogy to the previous paragraph, $s\theta$ is either a variable which only occurs in atomic positions in Q', or a ground term. Thus Q' respects atomic positions. $\square$

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

Definition 4.5 ($\mathcal{B}$-ground). Let P be a program which respects atomic positions, $\mathcal{B}$ a set of predicates whose input positions are all atomic, and $\overline{\mathcal{B}}$ the set of all other predicates used in P.

A query Q respecting atomic positions is **$\mathcal{B}$-ground** if each atom using a predicate in $\mathcal{B}$ has ground terms in its input positions. A clause $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is **$\mathcal{B}$-ground** if for all i, if $p_i \in \mathcal{B}$, then the variables in $\mathbf{s}_i$ are a subset of the variables in the atomic positions in $\mathbf{t}_0$. The program P is **$\mathcal{B}$-ground** if all of its clauses are and an atom using a predicate in $\overline{\mathcal{B}}$ is selectable only if it is non-variable in its atomic input positions.

Intuitively, for $\mathcal{B}$-ground programs it is always guaranteed that the input of all atoms in $\mathcal{B}$ is ground, since that input originates either from the initial query or the clause head.

Lemma 4.6. Let P be a $\mathcal{B}$-ground program and Q be a $\mathcal{B}$-ground clause. Let ξ be a delay-respecting derivation of $P \cup \{Q\}$. Then in all queries in ξ, an atom using a predicate in $\mathcal{B}$ is always ground in its input positions.

Proof. The proof is by induction on the length of ξ. Let $Q_0 = Q$ and $Q_0; Q_1; \ldots$ be a delay-respecting derivation of $P \cup \{Q_0\}$. By assumption, each atom in Q_0 using a predicate in $\mathcal{B}$ is ground in its input positions.

Now consider some Q_i where $i \geq 0$. If the selected atom in $Q_i; Q_{i+1}$ uses a predicate in $\overline{\mathcal{B}}$, then it is non-variable in its atomic input positions, and since Q_i respects atomic positions, it is *ground* in its atomic input positions. If the selected atom in $Q_i; Q_{i+1}$ uses a predicate in $\mathcal{B}$, then it is ground in its atomic input positions by the hypothesis. Since the clause used in $Q_i; Q_{i+1}$ is $\mathcal{B}$-ground, it follows that in Q_{i+1}, all atoms using a predicate in $\mathcal{B}$ are ground in their input positions. □

The program of Ex. 3.2 is $\{$`is`$\}$-ground, and thus by Lemma 4.6, it is justified that there is no `block` declarations for `is`.

5 Unification Free Programs

This short section deals with another useful verification aspect. It is not essential for understanding the subsequent sections.

A program is *unification free* if unification can be replaced by matching. Knowing that a program has this property can improve the efficiency of the compiled code. In [2], unification freedom is shown for LD-derivations. In this section we generalise these results to arbitrary (delay-respecting) derivations. The main difference is that they assume well typed programs and LD-derivations and can therefore rely on the selected atom always being correctly typed in its input positions. This is different for arbitrary derivations. Nevertheless, it is not very difficult to see that permutation simply typed programs, defined in the previous section, are unification free. We briefly recall some definitions [2].

Definition 5.1 (match, left-right disjoint). Given two vectors of terms $\mathbf{s} = s_1, \ldots, s_n$ and $\mathbf{t} = t_1, \ldots, t_n$ we use $\{\mathbf{s} = \mathbf{t}\}$ as abbreviation for the set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$. Consider a set of equations $E = \{\mathbf{s} = \mathbf{t}\}$. A substitution θ such that $dom(\theta) \subseteq vars(\mathbf{s})$ and $\mathbf{s}\theta = \mathbf{t}$, or $dom(\theta) \subseteq vars(\mathbf{t})$ and $\mathbf{t}\theta = \mathbf{s}$, is a **match** for E. E is **left-right disjoint** if $vars(\mathbf{s}) \cap vars(\mathbf{t}) = \emptyset$.

The following is a special case of *iterated matching* [2].

Definition 5.2 (double matching). Let E be a left-right disjoint set of equations. E is **solvable by double matching** if E is unifiable implies that there are equation sets E_1 and E_2 and substitutions θ_1 and θ_2 such that

- $E = E_1 \cup E_2$,
- $E_2\theta_1$ is left-right disjoint, and
- θ_1 is a match for E_1 and θ_2 is a match for $E_2\theta_1$.

Definition 5.3 (unification free). Let P be a permutation simply typed program where an atom is selectable in P only if it is non-variable in all bound positions, and let Q be a permutation simply typed query. Let ξ be a derivation of $P \cup \{Q\}$. Let a be a selected atom and h be the head of the clause used to resolve a in ξ. Then the system $a = h$ is **considered in** ξ.

Suppose that all systems of equations considered in all delay-respecting derivations of $P \cup \{Q\}$ are solvable by double matching. Then $P \cup \{Q\}$ is **unification free**.

Note that the above definition differs from the corresponding definition in [2] in that we do not assume LD-derivations.

In [2], it is exploited that many programs have *generic expressions* in their input positions. A generic expression for T is a term t such that if s is a term of type T and s is unifiable with t, then s is an instance of t. It is easy to see that in permutation simply typed programs, the input positions of each clause head are filled with generic expressions. Moreover, these generic expressions are *flat* terms, which is crucial since, unlike [2], we cannot rely on the input positions of the selected atom being correctly typed in its input positions. We conclude with the following theorem.

Theorem 5.1. Let P be a permutation simply typed program where an atom is selectable in P only if it is non-variable in all bound positions, and Q be a permutation simply typed query. Then $P \cup \{Q\}$ is unification free.

Proof. Consider a derivation step in a delay-respecting derivation of $P \cup \{Q\}$, where $p(\mathbf{s}, \mathbf{t})$ is the selected atom and $p(\mathbf{v}, \mathbf{u})$ is the head of the clause used to resolve it. Taking $E_1 = \{\mathbf{s} = \mathbf{v}\}$ and $E_2 = \{\mathbf{t} = \mathbf{u}\}$, it is clear by the proof of Lemma 4.2 that $E = \{\mathbf{s} = \mathbf{v}, \mathbf{t} = \mathbf{u}\}$ is solvable by double matching. □

6 Termination

This section corresponds to the results on termination of [25]. We are interested in termination in the sense that *all* derivations of a query are finite. Therefore the clause order in a program is irrelevant. Furthermore, we always assume that termination for the corresponding nicely moded and well typed programs has been shown using some existing method for LD-derivations [1, 8].

The methods presented here are quite limited. Termination can be proven for Exs. 6.1 and 6.2, but not for Exs. 4.2 and 7.1. In Sec. 7 we present a much more sophisticated method which can show termination for all these examples. This bears the question why we do not present this method straightaway. The reasons are the following:

- The methods presented here rely on conditions which are easy to check.
- The simplicity of the methods is appealing, they formalise previous heuristics [23, 24], and at the same time their limitations motivate Sec. 7,
- In a formal sense, the method of Sec. 7 is not more powerful than the methods presented here. There are programs for which the methods presented here are applicable but the method of Sec. 7 are not.

Example 6.1. The `permute` predicate of Ex. 3.1 loops for the query `permute(V,[1])` because `delete` produces a *speculative output binding* [24]: The output variable `Y` is bound before it is known that this binding will never have to be undone. Assuming left-based derivations, termination in both modes can be ensured by replacing the second clause with

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

This technique has been described as *putting recursive calls last* [24]. To explain termination, we have to apply a different reasoning for the different modes. In mode `permute`(I,O), `delete` is used in mode `delete`(I,O,I), and in this mode it does not *make* speculative bindings.

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

Clearly in this example termination depends on derivations being left-based, and therefore any method which abstracts from the selection rule must fail.

6.1 Termination by not Making Speculative Bindings

Some programs and queries have the nice property that there cannot be any failing derivations. In [6], such programs are called *noFD*. The following definition of *non-speculative* programs is similar to this, although there are two differences: in [6], clause heads that are not input-linear are allowed, but on the other hand, the results only hold for LD-derivations.

Definition 6.1 (non-speculative). A program P is **non-speculative** if it is permutation simply typed, has input selectability, and every simply typed atom[5] using a predicate in P is unifiable with some clause head in P.

[5] We say "atom" for "query of length 1".

Example 6.2. The following program converts binary trees into lists and vice versa. The append predicate is defined as in Ex. 4.2. The type of the program is $\{\texttt{ttl}(tree, list),\ \texttt{append}(list, list, list)\}$. It is non-speculative in mode $\{\texttt{ttl}(+,-), \texttt{append}(+,+,-)\}$. However it is not non-speculative in mode $\{\texttt{ttl}(-,+), \texttt{append}(-,-,+)\}$ because it is not permutation simply typed (there is the non-variable term [Label|List] in an output position).

```
:- block ttl(-,-).
ttl(leaf,[]).
ttl(node(L,Label,R),List) :-
  append(First,[Label|Last],List),
  ttl(L,First),
  ttl(R,Last).
```

Example 6.3. Both versions of the permute program (Exs. 3.1 and 6.1), assuming any type given in Ex. 3.3, are non-speculative in mode $\{\texttt{permute}(I, O), \texttt{delete}(I, O, I)\}$. Every simply typed atom is unifiable with at least one clause head, and the block declarations ensure input selectability.

Both versions of the permute program are *not* non-speculative in mode $\{\texttt{permute}(O, I), \texttt{delete}(O, I, O)\}$, because delete(A,[],B) is not unifiable with any clause head.

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

Theorem 6.1. Let P be a non-speculative program 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.[6] 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 $P' \cup \{Q'\}$ which uses the "same" clauses as ξ wherever possible. 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}, and we call a **resolved in** ξ if a is resolved in ξ at i for some i. Let $Q'_0 = Q'$ and $\theta'_0 = \emptyset$. We construct an LD-derivation $\xi' = \langle Q'_0, \theta'_0\rangle; \langle Q'_1, \theta'_1\rangle; \ldots$ inductively, showing that for all i where Q'_i is not empty, the following properties hold:

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

[6] Otherwise the proof is notationally more complex.

Note that by Corollary 4.3, Q'_i must be simply typed for all i and thus Corollary 4.4 is applicable for ξ'.

We first show these properties for $i = 0$. Clearly $\mathbf{v}\theta'_0 = \mathbf{v}$. Furthermore, by Corollary 4.4 (a) 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, $\langle Q'_0, \theta'_0\rangle; \ldots; \langle Q'_i, \theta'_i\rangle$ has already been constructed, Q'_i is not empty, and (1) and (2) hold. Let $p(\mathbf{s}, \mathbf{t})$ be the leftmost atom of Q'_i.

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 a most general unifier 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 Corollary 4.4 (b) and since the output arguments of $Q'_i\theta'_i$ are linear, we have $vars(\mathbf{v}\theta'_i) \cap dom(\phi'_i) = \emptyset$. 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' which is not resolved in ξ. By Corollary 4.4 (a) $\mathbf{v}\theta'_{i+1} = \mathbf{v}$. 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 4.4 (a) $\mathbf{v}\theta_j = \mathbf{v}$. 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 a most general unifier 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 Corollary 4.4 (b) and since the output arguments of $Q'_i\theta'_i$ are linear, $vars(\mathbf{v}\theta'_i) \cap dom(\phi'_i) = \emptyset$. 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 4.4 (a), 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 Corollary 4.4 (b), 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 $i + 1$.

Clearly by this construction, either ξ' terminates with the empty goal or is infinite.

Finally, 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 $i \leq n$. Hence a must occur in Q'_{i+1} since the clause used to resolve b in ξ' is the 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 goal.

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

Theorem 6.1 says that for non-speculative programs, atom order in clause bodies is irrelevant for termination.

Clearly a program cannot be non-speculative if it uses *tests*, such as `part`(I, I, O, O) in Ex. 4.2, where `leq(X,C)` and `greater(X,C)` are so-called *exhaustive tests* [6]. We are confident that the result of this subsection could be improved by allowing for such tests. However, we do not follow this line of research because we consider the approach of Sec. 7 more promising.

We now give an example of a program for which termination can be shown using non-speculative programs but not using the method of Sec. 7.

Example 6.4. The mode of the following program is $\{$`plus_one`(I), `minus_two`(I), `minus_one`(I), `two`(O), `one`(O), `zero`$(O)\}$, and the type of all arguments is il.

```
:- block plus_one(-).
plus_one(X) :- minus_two([1|X]).

:- block minus_two(-).
minus_two([A|X]) :- minus_one(X).
minus_two([]).

:- block minus_one(-).
minus_one([A|X]) :- plus_one(X).
minus_one([]).

two([2|X]) :- one(X).

one([1|X]) :- zero(X).

zero([]).
```

This program is non-speculative, and the query

```
plus_one(X), two(X).
```

is $\langle 2, 1\rangle$-simply typed. We can therefore show termination for this program using Thm. 6.1. (Section 7 is not applicable because we cannot show that `plus_one` is robust.)

6.2 Termination by not Using Speculative Bindings

In LD-derivations, speculative bindings are never used [24]. Assuming left-based derivations, all derivations are LD-derivations, provided the leftmost atom in a query is always selectable. This immediately implies the following theorem.

Theorem 6.2. Let Q be a well typed query and P a well typed program such that an atom is selectable in P if its input positions of non-variable type are non-variable. Then every left-based derivation of $P \cup \{Q\}$ is an LD-derivation.

By this theorem, all derivations of $\texttt{permute}(O, I)$ as defined in Ex. 6.1 are finite. As Ex. 3.1 shows, no termination guarantees can be given without such strong assumptions about the selection rule.

We now give an example of a program for which termination can be shown exploiting that a program is well typed, but not using the method of Sec. 7.

Example 6.5. Consider the following version of $\texttt{delete}(O, I, O)$.

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

This program is well typed in mode $delete(O, I, O)$ and thus we can show termination for all well typed queries. (Section 7 is not applicable because the program is not robustly typed.)

7 Termination Revisited

This section corresponds to the method for showing termination which was presented in [26]. As in the previous section, the order of clauses in a program is irrelevant, and it is assumed that termination for the corresponding nicely moded and well typed programs has been shown using some existing method for LD-derivations [1, 8].

In Ex. 6.1, termination could be ensured by applying the heuristic of putting recursive calls last [24]. However, this cannot explain the appropriate atom order in the following example.

Example 7.1. Figure 2 shows a program for the n-queens problem, which is an application of `block` declarations other than enabling multiple modes: implementing the test-and-generate paradigm. Here `permute` is defined as in Ex. 6.1.

With the mode $\{\texttt{nqueens}(I, O), \texttt{sequence}(I, O), \texttt{safe}(I), \texttt{permute}(O, I), \texttt{<}(I, I), \texttt{is}(O, I), \texttt{safe\_aux}(I, I, I), \texttt{no\_diag}(I, I, I), \texttt{=\textbackslash=}(I, I)\}$ and the type $\{\texttt{nqueens}(int, il), \texttt{sequence}(int, il), \texttt{safe}(il), \texttt{permute}(il, il), \texttt{<}(int, int), \texttt{is}(int, int), \texttt{safe\_aux}(il, int, int), \texttt{no\_diag}(int, int, int), \texttt{=\textbackslash=}(int, int)\}$, the first clause is $\langle 1, 3, 2\rangle$-nicely moded and $\langle 1, 3, 2\rangle$-well typed. Moreover, the query `nqueens(4,Sol)` terminates.

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

```
:- 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.
```

Fig. 2. A program for n-queens

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

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

7.1 Preventing Instantiation of Own Input

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

Example 7.2. Consider `delete`(O, I, O) as defined in Ex. 6.5 and the query

```
delete(A,L,R), delete(B,[1,2],L).
```

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

This version of `delete` is part of the *most specific program* [20] corresponding to Ex. 3.1, proposed in [24] to prevent looping for `permute`(O, I). However, it does not work. The query `permute(A,[1])` indeed terminates, but `permute(A,[1,2])` still loops.

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

Example 7.3. Consider the following program in mode $\mathtt{p}(I, O)$.

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

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

The easiest solution seems to be to require that the output positions in a query are always filled by variables. In mode $\mathtt{p}(I, O)$, the query `p(g(X),3)` should not arise, since its output is already instantiated. This requirement is part of the definition of non-speculative programs (Def. 6.1). However, it is often too restrictive.

Example 7.4. Consider Ex. 4.2. For the mode $\{\mathtt{qs}(O, I), \mathtt{append}(O, O, I), \mathtt{part}(O, I, I, I)\}$, the non-variable term `[X|Bs2]` occurs in an output position. Example 6.2 in mode $\{\mathtt{ttl}(O, I), \mathtt{append}(O, O, I)\}$ is similar.

It has been acknowledged previously that it is difficult to reason about programs where non-variable terms in output positions are allowed, but on the other hand, there are natural programs where this occurs [2]. Apt/Etalle assume that output positions in a query are always filled with variables, but consider allowing for non-variable terms as a direction for future work.

We define *permutation robustly-typedness*, which is a carefully crafted extension of permutation simply-typedness, allowing for non-variable but flat terms in certain output positions. The difficulty in designing such a concept is in ensuring that a natural persistence property such as Lemma 4.2 holds.

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

For *input* positions, a *bound* position has been defined in Def. 4.2. 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 $\mathbf{r}^{\mathsf{f}}$, and the projection onto its bound positions as $\mathbf{r}^{\mathsf{b}}$.

Definition 7.1 (permutation robustly typed). Let π be a permutation such that $\pi(i) = i$ whenever $i \notin \{1, \dots, n\}$. A query $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is **π-robustly typed** if it is π-nicely moded and π-well typed, $\mathbf{t}_1^{\mathsf{f}}, \dots, \mathbf{t}_n^{\mathsf{f}}$ is a vector of variables, and $\mathbf{t}_1^{\mathsf{b}}, \dots, \mathbf{t}_n^{\mathsf{b}}$ is a vector of flat typeable terms.

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

1. $\mathbf{t}^{\mathsf{f}}_0, \dots, \mathbf{t}^{\mathsf{f}}_n$ is a vector of variables, and $\mathbf{t}^{\mathsf{b}}_0, \dots, \mathbf{t}^{\mathsf{b}}_n$ is a vector of flat typeable terms.
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 $\mathbf{t}^{\mathsf{b}}_0, \dots, \mathbf{t}^{\mathsf{b}}_n$.

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 Def. 3.1.

Example 7.5. The `permute`-program of Ex. 6.1, for any of the types in Ex. 3.3, assuming all output positions are free, is robustly typed in mode `permute`(O, I) and permutation robustly typed in mode `permute`(I, O).

The program of Ex. 4.2 is permutation robustly typed in mode `qs`(O, I), assuming the second position of `append` is the only bound output position. It is also permutation robustly typed in mode `qs`(I, O), assuming that all output positions are free.

The following definition is in analogy to Def. 4.3.

Definition 7.2 (input selectability). Let P be a permutation robustly typed program. P has **input selectability** if for every permutation robustly typed query Q, an atom in Q is selectable in P if and only if it is non-variable in all bound input positions.

Input selectability can be achieved in a similar way as for permutation simply typed programs: the `block` declarations must be such that an atom whose free output positions are all variable is selectable if and only if all bound input positions are non-variable.

Example 7.6. Consider `append`(O, O, I) where the second position is the only bound output position (Exs. 4.2, 7.5). This program has input selectability. $Q =$ `append(A,[B|Bs],[1])` is a permutation robustly typed query, and its atom is selectable. The atom `append([],[],C)` is also selectable, although its input position is variable. This does not contradict Def. 7.2, since the first position is free, and thus this atom cannot be an atom in a permutation robustly typed query with respect to mode `append`(O, O, I).

Looking at Def. 7.1, one is tempted to think that it is best to associate the label *bound* with *all* output positions, because that would make the definition less restrictive. However, we require a program to have input selectability in each of its modes. Since input selectability is defined with respect to atoms in permutation robustly typed queries, and permutation robustly typed queries are defined with respect to given free and bound positions, it turns out that the choice of free and bound positions constrains the possible set of modes.

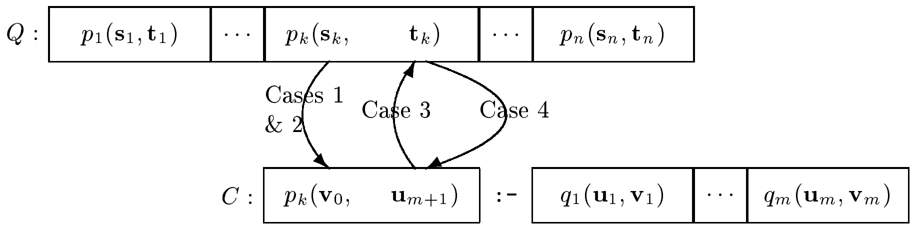

Fig. 3. Data flow in the unification

Example 7.7. Consider `append`(O, O, I), where *both* output positions are bound, and the `block` declaration is as in Ex. 4.2. Note that this `block` declaration is intended to allow for the current mode `append`(O, O, I), but also alternatively for mode `append`(I, I, O). Now consider the query

```
append([A|As],[B|Bs],Cs), append(Cs,Ds,[1,2,3])
```

This query is $\langle 2, 1\rangle$-robustly typed with respect to the current mode. The first atom is selectable although it is variable in its only bound input position. Therefore the program does not have input selectability. This could be rectified by replacing the `block` declaration with

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

but then the program could not be used in mode `append`(I, I, O) anymore. Recall that it is a distinctive feature of this paper that it is shown that one predicate can work in several modes. We have not encountered a case where a "natural" mode of a program was ruled out because of this problem.

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

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

Proof. We show how σ is computed, where we consider four stages. The first two stages are unifying $\mathbf{s}_k^{\mathsf{b}}$ and $\mathbf{v}_0^{\mathsf{b}}$, and unifying $\mathbf{s}_k^{\mathsf{f}}$ and $\mathbf{v}_0^{\mathsf{f}}$. The third stage is unifying all output positions where the binding goes from C to Q. The fourth stage is unifying all output positions where the binding goes from Q to C. The last stage is the most difficult one to reason about. Figure 3 illustrates which variables are bound in each case. Note that by Lemma 3.3, we can assume that this unification is occur-check free.

Part 1 (unifying $\mathbf{s}^{\mathsf{b}}_k$ and $\mathbf{v}^{\mathsf{b}}_0$). By assumption $\mathbf{s}^{\mathsf{b}}_k$ is a vector of non-variable terms. By Def. 7.1, $\mathbf{v}^{\mathsf{b}}_0$ is a vector of flat terms. By assumption, $vars(\mathbf{v}_0) \cap vars(\mathbf{s}_k) = \emptyset$. Thus there is a (minimal) substitution σ_1 such that $\mathbf{v}^{\mathsf{b}}_0\sigma_1 = \mathbf{s}^{\mathsf{b}}_k$. We show that the following hold:

(1a) $dom(\sigma_1) \cap vars(\mathbf{s}_k) = \emptyset$.
(1b) Every variable occurring directly in $\mathbf{u}^{\mathsf{b}}_{m+1}\sigma_1$, does not occur in $\mathbf{s}_k$.
(1c) $vars(\mathbf{s}_k) \cap vars(\mathbf{t}_k\sigma_1) = \emptyset$.
(1d) $(\mathbf{v}^{\mathsf{f}}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_n)\sigma_1$ is linear and has flat typeable terms in all bound positions and variables in all free positions.
(1e) Every variable occurring directly in a position of type τ in $\mathbf{u}^{\mathsf{b}}_{m+1}\sigma_1$, can only occur in $(\mathbf{v}^{\mathsf{f}}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_n)\,\sigma_1$ in a bound position of type τ, and the occurrence must be direct.
(1f) $vars(\mathbf{s}_k) \cap vars((\mathbf{v}^{\mathsf{f}}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m)\,\sigma_1) = \emptyset$.
(1g) $vars(\mathbf{u}_{m+1}\sigma_1) \cap vars(\mathbf{t}_k\sigma_1) = \emptyset$.

(1a) holds by the construction of σ_1.

For each variable x occurring directly in both $\mathbf{v}^{\mathsf{b}}_0$ and $\mathbf{u}^{\mathsf{b}}_{m+1}$, (1b) holds because $x\sigma_1$ is a non-variable term. If w is a non-variable term in $\mathbf{v}^{\mathsf{b}}_0$, then by Def. 7.1, w cannot contain a variable which occurs directly in $\mathbf{u}^{\mathsf{b}}_{m+1}$. Thus in this case, (1b) also holds.

(1c) holds because of Q is permutation nicely moded and because $dom(\sigma_1) \cap vars(Q) = \emptyset$.

By Def. 7.1, and since C and Q are variable-disjoint

$$dom(\sigma_1) \cap vars(\mathbf{v}^{\mathsf{f}}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_n)) = \emptyset.$$

Thus (1d) holds.

Let x be a variable occurring directly in $\mathbf{u}^{\mathsf{b}}_{m+1}$. By Def. 7.1, x does not occur indirectly in $\mathbf{v}^{\mathsf{b}}_0$ (however it might be that x fills a position in $\mathbf{v}^{\mathsf{b}}_0$) and thus $x\sigma_1$ is not a variable. Since $ran(\sigma_1) \subseteq vars(\mathbf{s}_k)$, since $vars(C) \cap vars(G) = \emptyset$, $x \notin ran(\sigma_1)$. Thus (1e) holds.

(1f) holds since by Def. 7.1, $dom(\sigma_1) \cap vars(\mathbf{v}^{\mathsf{f}}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m) = \emptyset$.

By Def. 7.1, $dom(\sigma_1) \cap vars(\mathbf{t}_k) = \emptyset$. Furthermore, since Q is permutation nicely moded, $ran(\sigma_1) \cap vars(\mathbf{t}_k) = \emptyset$. Thus (1g) holds.

Part 2 (unifying $\mathbf{s}^{\mathsf{f}}_k\sigma_1$ and $\mathbf{v}^{\mathsf{f}}_0\sigma_1$). By (1d), $\mathbf{v}^{\mathsf{f}}_0\sigma_1$ is a vector of variables. Thus there is a (minimal) substitution σ' such that $\mathbf{v}^{\mathsf{f}}_0\sigma_1\sigma' = \mathbf{s}^{\mathsf{f}}_k\sigma_1$. We show that (2a-g) are true for $\sigma_2 = \sigma_1\sigma'$.

(2a) $dom(\sigma_2) \cap vars(\mathbf{s}_k) = \emptyset$.
(2b) Every variable occurring directly in $\mathbf{u}^{\mathsf{b}}_{m+1}\sigma_2$, does not occur in $\mathbf{s}_k$.
(2c) $vars(\mathbf{s}_k) \cap vars(\mathbf{t}_k\sigma_2) = \emptyset$.
(2d) $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_n)\sigma_2$ is linear and has flat typeable terms in all bound positions and variables in all free positions.
(2e) Every variable occurring directly in a position of type τ in $\mathbf{u}^{\mathsf{b}}_{m+1}\sigma_2$ can only occur in $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_n)\,\sigma_2$ in a bound position of type τ, and the occurrence must be direct.

(2f) $vars(\mathbf{s}_k) \cap vars((\mathbf{v}_1, \ldots, \mathbf{v}_m)\,\sigma_2) = \emptyset$.
(2g) $vars(\mathbf{u}_{m+1}\sigma_2) \cap vars(\mathbf{t}_k\sigma_2) = \emptyset$.

(2a) holds by (1f) and since $dom(\sigma') \subseteq vars(\mathbf{v}^{\mathsf{f}}\sigma_1)$.

(2b) holds since by (1e), no variable occurring directly in $\mathbf{v}_0^{\mathsf{f}}$ occurs directly in $\mathbf{u}_{m+1}^{\mathsf{b}}\sigma_1$.

(2c) and (2d) are similar to Case 1.

By (1e), if x is a variable occurring directly in $\mathbf{v}_0^{\mathsf{f}}$, then x does not occur directly in $\mathbf{u}_{m+1}^{\mathsf{b}}\sigma_1$, and thus no variable occurring directly in $\mathbf{u}_{m+1}^{\mathsf{b}}\sigma_1$ is bound by σ'. Since $ran(\sigma') \subseteq vars(\mathbf{s}_k)$, by (1b), $ran(\sigma') \cap vars(\mathbf{u}_{m+1}^{\mathsf{b}}\sigma_1) = \emptyset$. Thus (2e) holds.

(2f) and (2g) are shown as in Part 1.

Part 3 (unifying $\mathbf{t}_k\sigma_2$ and $\mathbf{u}_{m+1}\sigma_2$ in all positions where either $\mathbf{t}_k\sigma_2$ is variable, or $\mathbf{t}_k\sigma_2$ and $\mathbf{u}_{m+1}\sigma_2$ are both non-variable). Note that by (2d), this includes all positions in $\mathbf{t}_k^{\mathsf{f}}\sigma_2$ and $\mathbf{u}_{m+1}^{\mathsf{f}}\sigma_2$, but may also include positions in $\mathbf{t}_k^{\mathsf{b}}\sigma_2$ and $\mathbf{u}_{m+1}^{\mathsf{b}}\sigma_2$. Case 3 covers precisely the output positions where the binding "goes from $\mathbf{u}_{m+1}\sigma_2$ to $\mathbf{t}_k\sigma_2$" (see Fig. 3). We denote by $\mathbf{t}_k^{\mathsf{f}+}\sigma_2$ the projection of $\mathbf{t}_k\sigma_2$ onto the positions where $\mathbf{t}_k\sigma_2$ is variable, or $\mathbf{t}_k\sigma_2$ and $\mathbf{u}_{m+1}\sigma_2$ are both non-variable, and by $\mathbf{t}_k^{\mathsf{b}-}\sigma_2$ the projection onto the other positions, and likewise for $\mathbf{u}_{m+1}\sigma_2$.

By (2g), $\mathbf{u}_{m+1}\sigma_2$ and $\mathbf{t}_k\sigma_2$ are variable-disjoint. There is a minimal substitution σ'' such that $\mathbf{t}_k^{\mathsf{f}+}\sigma_2\sigma'' = \mathbf{u}_{m+1}^{\mathsf{f}+}\sigma_2$. We show (3a-g) for $\sigma_3 = \sigma_2\sigma''$.

(3a) $dom(\sigma_3) \cap vars(\mathbf{s}_k) = \emptyset$.
(3b) Every variable occurring directly in $\mathbf{u}_{m+1}^{\mathsf{b}-}\sigma_3$, does not occur in $\mathbf{s}_k$.
(3c) $vars(\mathbf{s}_k) \cap vars(\mathbf{t}_k^{\mathsf{b}-}\sigma_3) = \emptyset$.
(3d) $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_k^{\mathsf{b}-}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\sigma_3$ is linear and has flat typeable terms in all bound positions and variables in all free positions.
(3e) Every variable occurring directly in a position of type τ in $\mathbf{u}_{m+1}^{\mathsf{b}-}\sigma_3$ can only occur in $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_k^{\mathsf{b}-}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\,\sigma_3$ in a bound position of type τ, and the occurrence must be direct.
(3f) $vars(\mathbf{s}_k) \cap vars((\mathbf{v}_1, \ldots, \mathbf{v}_m)\,\sigma_3) = \emptyset$.
(3g) $vars(\mathbf{u}_{m+1}\sigma_3) \cap vars(\mathbf{t}_k^{\mathsf{b}-}\sigma_3) = \emptyset$.

(3a) holds since by (2a), $dom(\sigma'') \cap vars(\mathbf{s}_k) = \emptyset$.

By (2g), $dom(\sigma'') \cap vars(\mathbf{u}_{m+1}\sigma_2) = \emptyset$. By (3a), $dom(\sigma'') \cap vars(\mathbf{s}_k) = \emptyset$. Thus (3b) holds.

By (2d), $dom(\sigma'') \cap vars(\mathbf{t}_k^{\mathsf{b}-}\sigma_2) = \emptyset$. By (3a), $dom(\sigma'') \cap vars(\mathbf{s}_k) = \emptyset$. Thus (3c) holds.

By (2d)

$$dom(\sigma'') \cap vars(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_k^{\mathsf{b}-}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\sigma_2) = \emptyset.$$

Thus (3d) holds. Furthermore by (2g), $dom(\sigma'') \cap vars(\mathbf{u}_{m+1}\sigma_2) = \emptyset$. Thus (3e) holds.

By (2d), $dom(\sigma'') \cap vars((\mathbf{v}_1, \ldots, \mathbf{v}_m)\sigma_2) = \emptyset$. By (3a), $dom(\sigma'') \cap vars(\mathbf{s}_k) = \emptyset$. Thus (3f) holds.

By (2g), $dom(\sigma'') \cap vars(\mathbf{u}_{m+1}\sigma_2) = \emptyset$. By (2d), $dom(\sigma'') \cap vars(\mathbf{t}_k^{\mathsf{b}-}) = \emptyset$. Thus (3g) holds.

Part 4 (unifying $\mathbf{t}_k^{\mathsf{b}-}\sigma_3$ and $\mathbf{u}_{m+1}^{\mathsf{b}-}\sigma_3$). By (2g), $dom(\sigma'') \cap vars(\mathbf{u}_{m+1}\sigma_2) = \emptyset$, and thus $\mathbf{u}_{m+1}^{\mathsf{b}-}\sigma_3 = \mathbf{u}_{m+1}^{\mathsf{b}-}\sigma_2$. Therefore $\mathbf{u}_{m+1}^{\mathsf{b}-}\sigma_3$ is a vector of variables, and there is a minimal substitution σ''' such that $\mathbf{u}_{m+1}^{\mathsf{b}-}\sigma_3\sigma''' = \mathbf{t}_k^{\mathsf{b}-}\sigma_3$.

We show (4a), (4d), (4f) for $\sigma_4 = \sigma_3\sigma'''$.

(4a) $dom(\sigma_4) \cap vars(\mathbf{s}_k) = \emptyset$.
(4d) $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\sigma_4$ is linear and has flat terms in all bound positions and variables in all free positions.
(4f) $vars(\mathbf{s}_k) \cap vars((\mathcal{R}_l(\mathbf{v}_0), \mathbf{v}_1, \ldots, \mathbf{v}_m)\, \sigma_4) = \emptyset$.

(4a) holds since by (3b), $dom(\sigma''') \cap vars(\mathbf{s}_k) = \emptyset$.

Suppose x is a variable in $\mathbf{u}_{m+1}^{\mathsf{b}-}\sigma_3$ occurring in a position i of type τ, and x also occurs in $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\sigma_3$. By (3e), the latter occurrence of x is in a bound position of type τ, and by (3d), it is the *only* occurrence of x in $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\sigma_3$. Let I be the set of positions where x occurs in $\mathbf{u}_{m+1}^{\mathsf{b}-}\sigma_3$, and Let T be the set of terms occurring in $\mathbf{t}_k^{\mathsf{b}-}\sigma_3$ in positions in I. By (3d), T is a set of variable-disjoint, flat terms. Therefore their most general common instance $x\sigma'''$ is a flat term. Since position $i \in I$ is of type τ, it follows that $x\sigma'''$ is typeable wrt. τ. The term $x\sigma'''$ occurs in $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\sigma_3\sigma'''$ in a position of type τ. By (3d), $x\sigma'''$ is variable-disjoint from $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\sigma_3$, and thus $(\mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\sigma_3\sigma'''$ is linear. Thus (4d) holds.

By (4a), $dom(\sigma''') \cap vars(\mathbf{s}_k) = \emptyset$. By (3c), $ran(\sigma''') \cap vars(\mathbf{s}_k) = \emptyset$. Thus (4f) holds.

Part 5: Defining $\sigma = \sigma_4$ it follows that $p_k(\mathbf{s}_k, \mathbf{t}_k)\sigma = p_k(\mathbf{v}_0, \mathbf{u}_{m+1})\sigma$. By (4d) and Lemmas 3.1 and 3.4, $Q\sigma$ is permutation robustly typed. By (4a), $\mathbf{s}_k\sigma = \mathbf{s}_k$, and by (4f), $vars(\mathbf{s}_k) \cap vars((\mathbf{v}_1, \ldots, \mathbf{v}_m)\sigma) = \emptyset$. □

The following are corollaries of Lemma 7.1. They are needed to prove Lemma 7.4.

Corollary 7.2. Make the same assumptions as in Lemma 7.1. Then for all i with $\pi(i) < \pi(k)$, $dom(\sigma) \cap vars(p_i(\mathbf{s}_i, \mathbf{t}_i)) = \emptyset$.

Proof. Since σ is obtained by unifying $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$, and $vars(C) \cap vars(Q) = \emptyset$, it follows that $dom(\sigma) \cap vars(Q) \subseteq vars(p_k(\mathbf{s}_k, \mathbf{t}_k))$. By Lemma 7.1, $dom(\sigma) \cap vars(\mathbf{s}_k) = \emptyset$. Since Q is nicely moded, $vars(p_i(\mathbf{s}_i, \mathbf{t}_i)) \cap vars(\mathbf{t}_k) = \emptyset$ for all i where $\pi(i) < \pi(k)$, and so $dom(\sigma) \cap vars(p_i(\mathbf{s}_i, \mathbf{t}_i)) = \emptyset$. □

Corollary 7.3. Let P be a permutation robustly typed program with input selectability. Let $Q = a_1, \ldots, a_n$ be 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 suppose that $b_1, \dots, b_m$ is ρ-robustly typed, and let $k, l \in \{1, \dots, m\}$ 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. By Corollary 3.2 we have $\rho(k) < \rho(l)$. The selected atom in the last derivation step is b_l, so the result follows from Corollary 7.2. □

7.2 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, without loss of generality, assume that the programs and queries are robustly typed (rather than *permutation* robustly typed). This simplifies the notation. In Subsec. 7.3, we go back to allowing for arbitrary permutations.

Definition 7.3 (robust). A predicate p in a robustly typed program P is **robust** if, for each robustly typed query $p(\mathbf{s}, \mathbf{t})$, any delay-respecting derivation of $P \cup \{p(\mathbf{s}, \mathbf{t})\}$ is finite. An atom is **robust** if its predicate is.

For queries consisting of robust atoms, termination does not depend on derivations being left-based.

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

Lemma 7.4. Let P be a robustly typed 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, we call an a-step for some $a \in F$ an F**-step**, and likewise for H. By Corollary 7.3, any H-step does not instantiate any descendant of b. Thus the H-steps can be disregarded, and without loss of generality, we assume H is empty. Let

$$\xi = \langle F, b, \ \emptyset\rangle; \dots ; \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle \dots$$

be a delay-respecting derivation such that $\langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle \dots$ contains no F-steps (that is, ξ contains only finitely many F-steps). Since by Corollary 7.3, no b-step can instantiate any descendant of F, there exists a delay-respecting derivation

$$\xi_2 = \langle F, b, \ \emptyset\rangle; \dots ; \langle R, \rho\rangle; \dots ; \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle \dots$$

such that $\langle F, b, \ \emptyset\rangle; \dots ; \langle R, \rho\rangle$ contains only F-steps and $\langle R, \rho\rangle; \dots ; \langle Q_0, \theta_0\rangle$ contains only b-steps (that is, the F-steps are moved forward using the Switching Lemma [17]). Since $R = R', b$ for some R', there exists a delay-respecting derivation

$$\xi_3 = \langle b, \rho\rangle; \dots ; \langle I_0, \theta_0\rangle; \langle I_1, \theta_1\rangle \dots$$

obtained from $\langle R, \rho\rangle; \dots ; \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle \dots$ by removing R' in each query.

Let $\mathbf{t}_1, \dots, \mathbf{t}_m$ be the vector of output arguments of $R'\rho$. Since $R'\rho$ is robustly typed, it follows that there is a substitution σ such that $R'\rho\sigma$ is correctly typed in its output positions and $b\rho\sigma$ is a robustly typed query.

By Corollary 7.3, no b-step instantiates $\mathbf{t}_1, \dots, \mathbf{t}_m$. Therefore from ξ_3 we can construct a delay-respecting derivation

$$\xi_4 = \langle b, \rho\sigma \rangle; \dots; \langle I_0, \theta_0\sigma \rangle; \langle I_1, \theta_1\sigma \rangle \dots$$

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. □

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

Lemma 7.5. Let P be a robustly typed program with input selectability and Q a robustly typed query. A delay-respecting derivation of $P \cup \{Q\}$ can be infinite only if there are infinitely many steps where a non-robust atom is resolved.

Proof. Let $Q = F, b, H$ where b is a robust atom and ξ an infinite derivation for Q. We show that ξ can have infinitely many b-steps only if ξ has infinitely many steps where a non-robust atom is resolved.

We prove this by induction on the number of atoms in F. If F is empty, the result follows immediately from Lemma 7.4. Now suppose F contains at least one atom. By Lemma 7.4, ξ can have infinitely many b-steps only if for some $a \in F$, ξ has infinitely many a-steps. If a is not robust, the result follows immediately. If a is robust, let $F = H, a, K$. Since H contains fewer atoms than F, the result follows from the inductive hypothesis. □

For LD-derivations, termination proofs usually rely on some norm to measure the size of a term or atom [10, 11]. For a query F, a, H, the query F is resolved away before a is selected, and thus a is sufficiently instantiated to be *bounded* with respect to the norm. In contrast, for arbitrary derivations, the decrease in argument size must be independent of the order in which atoms are selected. We assume a simple norm where a term is smaller than another term if it is a proper subterm. One could enhance this method by considering other norms.

Example 7.8. Consider Ex. 7.1. All delay-respecting derivations of a robustly typed query `safe_aux`(l, n, m) terminate, because in the first argument of `safe_aux`, there is a strict decrease with respect to the "subterm norm".

The following definition is adapted from [1].

Definition 7.4 (depends on). Let p, q be predicates in a program P. We say that p **refers to** q if there is a clause in P with p in its head and q in its body, and p **depends on** q (written $p \sqsupseteq q$) if (p, q) is in the reflexive, transitive closure of *refers to*. We write $p \sqsupset q$ if $p \sqsupseteq q$ and $q \not\sqsupseteq p$, and $p \approx q$ if $p \sqsupseteq q$ and $q \sqsupseteq p$.

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

Definition 7.5 (decreasing clause). Assume that for each predicate p in a program P, there is a designated position called **decreasing position**. Let $C = q(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be a clause in P. Suppose that for each $\mu \in \{1, \ldots, m\}$ where $q_\mu \approx q$, $q_\mu(\mathbf{u}_\mu, \mathbf{v}_\mu)$ has a variable in its decreasing position which is a proper subterm of the term in the decreasing position of $q(\mathbf{v}_0, \mathbf{u}_{m+1})$. Then C is **decreasing**.

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

Lemma 7.6. Let P be a robustly typed program with input selectability and p a predicate in P. Suppose that for each predicate q, where $p \sqsupseteq q$, either:

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

Then p is robust.

Proof. In this proof, we say "the decreasing term of an atom" as abbreviation for "the term in the decreasing position of an atom". For a set of atoms B, define

$$D(B) = \{s \mid s \text{ is the decreasing term of some } a \in B\}.$$

For a query R, define[7]

$$A(R) = \{q(\mathbf{u}, \mathbf{v}) \mid q(\mathbf{u}, \mathbf{v}) \text{ is in } R \text{ and } q \approx p\}.$$

Furthermore define $A(R') < A(R)$ if for some $a \in A(R)$, some finite set B' of atoms whose decreasing terms are *proper* subterms of the decreasing term of a, and some substitution θ such that $dom(\theta) \cap vars(D(A(R))) = \emptyset$

$$A(R') = ((A(R) \setminus \{a\}) \cup B')\theta.$$

Intuitively, a decrease with respect to $<$ is obtained when some decreasing term is replaced with a finite number of smaller decreasing terms. It is easy to see (König's Lemma [13]) that all descending chains with respect to $<$ are finite. Furthermore define $A(R') \equiv A(R)$ if $A(R') = A(R)\theta$, where θ is defined as above, and define $A(R') \leq A(R)$ if $A(R') < A(R)$ or $A(R') \equiv A(R)$.

Let $Q_0 = p(\mathbf{s}, \mathbf{t})$ be a robustly typed query and $\xi = Q_0; Q_1; Q_2 \ldots$ be a delay-respecting derivation of $P \cup \{Q_0\}$.

Part 1: We show that for each $i \geq 0$

I(i) If $Q_i = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ and, for some $\nu \in \{1, \ldots, n\}$, $p_\nu \approx p$ and s is the decreasing term of $p_\nu(\mathbf{s}_\nu, \mathbf{t}_\nu)$, then $vars(s) \cap vars(\mathbf{t}_1, \ldots, \mathbf{t}_n) = \emptyset$.

[7] We assume that a query does not contain two identical atoms. The statement also holds otherwise, but the proof is notationally more complex.

I(0) holds because Q_0 contains only one atom and is nicely moded.

Now suppose I(i) holds for some $i \geq 0$. Let $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be the clause, $p_k(\mathbf{s}_k, \mathbf{t}_k)$ the selected atom and θ the most general unifier used in $Q_i; Q_{i+1}$. We show I($i+1$).

Consider first an atom already in Q_i, that is, consider $\nu \in \{1, \ldots, n\}$ such that $p_\nu \approx p$. Let s be the decreasing term of $p_\nu(\mathbf{s}_\nu, \mathbf{t}_\nu)$. By Lemma 7.1,

$$vars((\mathbf{v}_1, \ldots, \mathbf{v}_m)\theta) \cap vars(\mathbf{s}_k) = \emptyset$$

and thus

$$vars((\mathbf{v}_1, \ldots, \mathbf{v}_m)\theta) \subseteq vars(C) \cup vars(\mathbf{t}_k)$$

and so by I(i),

$$vars((\mathbf{v}_1, \ldots, \mathbf{v}_m)\theta) \cap vars(s) = \emptyset. \tag{1}$$

Again by Lemma 7.1, $dom(\theta) \cap vars(Q_i) \subseteq vars(\mathbf{t}_k)$ and so by I(i)

$$dom(\theta) \cap vars(s) = \emptyset. \tag{2}$$

Thus, since $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is linear and by (1), (2)

$$vars(s\theta) \cap vars((\mathbf{t}_1, \ldots \mathbf{t}_{k-1}, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\theta) = \emptyset. \tag{3}$$

Now consider a newly added atom, that is, consider $\mu \in \{1, \ldots, m\}$ such that $q_\mu \approx p$. Let s' be the decreasing term in $q_\mu(\mathbf{u}_\mu, \mathbf{v}_\mu)\theta$. Then by assumption 2, s' is a proper subterm of s, where s is the decreasing term in $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})\theta$, and $p_k \approx p$. However $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})\theta = p_k(\mathbf{s}_k, \mathbf{t}_k)\theta$, and thus by (3)

$$vars(s) \cap vars((\mathbf{t}_1, \ldots \mathbf{t}_{k-1}, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\theta) = \emptyset,$$

and therefore

$$vars(s') \cap vars((\mathbf{t}_1, \ldots \mathbf{t}_{k-1}, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\theta) = \emptyset. \tag{4}$$

(3) and (4) together imply I($i+1$).

Part 2: Since by assumption 1, all predicates q with $p \sqsupset q$ are robust, it follows by Lemma 7.5 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 $i \geq 0$, if the selected atom in $Q_i; Q_{i+1}$ is $q(\mathbf{u}, \mathbf{v})$ and $q \approx p$, then $A(Q_{i+1}) < A(Q_i)$, and otherwise $A(Q_{i+1}) \equiv A(Q_i)$. 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.

Consider $i \geq 0$ and let $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be the clause, $p_k(\mathbf{s}_k, \mathbf{t}_k)$ the selected atom and θ the most general unifier used in $Q_i; Q_{i+1}$.

If $p \sqsupset p_k$, then $p \sqsupset q_\mu$ for all $\mu \in \{1, \ldots, m\}$. Thus $A(Q_{i+1}) = A(Q_i)\theta$. By I($i$), $dom(\theta) \cap D(A(Q_i)) = \emptyset$, and so $A(Q_{i+1}) \equiv A(Q_i)$.

Now consider the case $p_k \approx p$. Let s be the decreasing term in $p_k(\mathbf{s}_k, \mathbf{t}_k)$. The term s cannot be a variable because P has input selectability and thus $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is non-variable in its input positions of non-variable type.

Now assume s is a constant. By assumption 2, the decreasing term in $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$ can only be a constant if $p \sqsupset q_\mu$ for all $\mu \in \{1, \ldots, m\}$. Thus $A(Q_{i+1}) = (A(Q_i) \setminus \{q_\mu(\mathbf{u}_\mu, \mathbf{v}_\mu)\})\theta$. By I($i$), $dom(\theta) \cap D(A(Q_i)) = \emptyset$, and so $A(Q_{i+1}) < A(Q_i)$.

Now assume $s = f(r_1, \ldots, r_j)$. By Def. 7.1 and assumption 2, the decreasing term in $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$ is of the form $f(x_1, \ldots, x_j)$. Suppose $q_\mu \approx p$ for some $\mu \in \{1, \ldots, m\}$. Then, for some $\iota \in \{1, \ldots, j\}$, x_ι is the decreasing term in $q_\mu(\mathbf{u}_\mu, \mathbf{v}_\mu)$. Thus $x_\iota \theta = r_\iota$ and the decreasing term in $q_\mu(\mathbf{u}_\mu, \mathbf{v}_\mu)\theta$ is r_ι. Thus

$$A(Q_{i+1}) = ((A(Q_i) \setminus \{q_\mu(\mathbf{u}_\mu, \mathbf{v}_\mu)\}) \cup B)\theta$$

where B is a set of atoms whose decreasing terms are proper subterms of the decreasing term of $q_\mu(\mathbf{u}_\mu, \mathbf{v}_\mu)$. By I($i$), $dom(\theta) \cap D(A(Q_i)) = \emptyset$, and thus $A(Q_{i+1}) < A(Q_i)$. □

Example 7.9. We demonstrate for Ex. 7.8 how Lemma 7.6 is used. Given that the built-in `=\=` terminates, it follows that `no_diag` is robust. With the first position of `safe_aux` as decreasing position, the recursive call to `safe_aux` in the second clause for `safe_aux` has `Ms` in the decreasing position, which is a proper subterm of `[M|Ms]`. Similar arguments can be applied for the other clauses, showing that `safe` and `safe_aux` are robust.

However, often it is not possible to find a decreasing position for a predicate. This could mean that the predicate is not robust, or just that our method is not powerful enough to show it to be robust.

7.3 Well Fed Programs

So far we have shown for some predicates that all delay-respecting derivations of queries with these predicates terminate. As `permute`(O, I) (Ex. 3.1) shows, this does not work for all predicates. In a program which uses such predicates, the selection rule must be taken into account. We assume left-based derivations. Consequently we now also give up the assumption, made to simplify the notation, that the clauses and query are robustly typed, rather than just *permutation* robustly typed. All statements from the previous subsection generalise in the obvious way. Whenever a statement refers to a robustly typed query $a_1, \ldots, a_n$ and two atoms a_i and a_j such that $i < j$, then this must be replaced by a π-robustly typed query and two atoms a_i and a_j such that $\pi(i) < \pi(j)$.

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

Definition 7.6 (well fed). Let P be a permutation robustly typed program. For a π-robustly typed query $p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$, an atom $p_i(\mathbf{s}_i, \mathbf{t}_i)$ is **well fed** if it is robust, or $\pi(j) < \pi(i)$ implies $j < i$ for all j. A π-robustly typed query (clause) is **well fed** if all of its (body) atoms are. P is **well fed** if all of its clauses are well fed and it has input selectability.

Example 7.10. The programs mentioned in Ex. 7.5 are well fed in the given modes. The program in Ex. 7.1 is well fed in the given mode. It is not well fed in mode $\{\texttt{nqueens}(O, I), \texttt{permute}(I, O), \texttt{sequence}(O, I), <(I, I), \texttt{is}(O, I)\}$, because it is not permutation nicely moded in this mode: in the second clause for `sequence`, `N1` occurs twice in an output position.

Lemma 7.7. Every resolvent of a well fed query Q and a well fed clause C, where $vars(Q) \cap vars(C) = \emptyset$, is well fed.

Proof. By obvious analogy, Corollary 3.2 also holds if *nicely moded* is replaced with *robustly typed*. The result then follows from Lemma 7.1. □

The following two statements are simple consequences of Def. 3.3 needed for the proof of Theorem 7.10. The first says that for a π-well typed query Q, every prefix of $\pi(Q)$ is well typed. The second says that if all atoms in $\pi(Q)$ before an atom a are resolved away, then a becomes correctly typed in its input positions.

Corollary 7.8. Let $Q = a_1, \ldots, a_n$ be a π-well typed query. For all $i \in \{1, \ldots, n\}$, the subquery of Q containing all a_j such that $\pi(j) \leq \pi(i)$ is permutation well typed.

Corollary 7.9. Let $Q = a_1, \ldots, a_n$ be a π-well typed query. For all $i \in \{1, \ldots, n\}$, if $Q; \ldots; (F, a_i, H)\theta$ is a derivation for Q and for all j with $\pi(j) < \pi(i)$, $(F, a_i, H)\theta$ contains no descendants of a_j, then $a_i\theta$ is correctly typed in its input positions.

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 7.10. 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. Let $Q_0 = Q$, $\theta_0 = \emptyset$ and

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

be a left-based derivation, where $R_1, R_2, \ldots$ are the queries in ξ where a non-robust atom is selected.

Part 1: We show that in each query in ξ, the non-robust atoms are not waiting, and that for each $i > 0$, the leftmost non-robust atom in R_i is selected in the step $\langle R_i, \sigma_i \rangle; \langle Q_i, \theta_i \rangle$.

Clearly in $\langle Q_0, \theta_0 \rangle$ no atom is waiting. Now consider some $i \geq 0$, let $Q_i = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ and suppose $Q_i\theta_i$ is π-robustly typed, and k is the smallest number such that $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is not robust.

Let F be the subquery of Q_i containing all $p_j(\mathbf{s}_j, \mathbf{t}_j)$ with $\pi(j) < \pi(k)$. By Lemma 7.7, $Q_i\theta_i$ is well fed, and thus $j < k$ for all $p_j(\mathbf{s}_j, \mathbf{t}_j)$ in F. By Corollary 7.8

$$F\theta_i \text{ is permutation well typed.} \tag{1}$$

Consider an arbitrary $\langle\tilde{Q}, \tilde{\theta}\rangle$ in $\langle Q_i, \theta_i\rangle; \ldots; \langle R_{i+1}, \sigma_{i+1}\rangle$ such that $\tilde{Q}$ contains at least one descendant of F, and assume that no non-robust atom in $\tilde{Q}$ is waiting. By (1) and Lemma 3.4, $\tilde{Q}$ contains, in particular, at least one descendant a of F such that $a\tilde{\theta}$ is *selectable*, and moreover a occurs to the left of any non-robust atom in $\tilde{Q}$. Therefore no non-robust atom is selected or delayed in ξ in the step following $\langle\tilde{Q}, \tilde{\theta}\rangle$.

However, since some non-robust atom is selected in $\langle R_{i+1}, \sigma_{i+1}\rangle; \langle Q_{i+1}, \theta_{i+1}\rangle$, there is some (earliest) $\langle\tilde{Q}, \tilde{\theta}\rangle$ in $\langle Q_i, \theta_i\rangle; \ldots; \langle R_{i+1}, \sigma_{i+1}\rangle$ such that $\tilde{Q}$ contains no descendants of F. Thus in particular

$$R_{i+1} \text{ contains no descendants of } F. \tag{2}$$

By Corollary 7.9, $p_k(\mathbf{s}_k, \mathbf{t}_k)\tilde{\theta}$ is correctly typed in its input positions and thus selectable. It follows therefore that in $\langle\tilde{Q}, \tilde{\theta}\rangle; \ldots; \langle R_{i+1}, \sigma_{i+1}\rangle$, no non-robust atom becomes waiting, and the selected atom in $\langle R_{i+1}, \sigma_{i+1}\rangle; \langle Q_{i+1}, \theta_{i+1}\rangle$ is $p_k(\mathbf{s}_k, \mathbf{t}_k)$.

Part 2: For all $i > 0$, let C_i be the uniquely renamed clause used in the step $\langle R_i, \sigma_i\rangle; \langle Q_i, \theta_i\rangle$, and let C'_i be a robustly typed clause corresponding to C_i (using the same renaming). Let $Q'_0 = Q'$ and $\theta'_0 = \emptyset$. We construct an LD-derivation

$$\xi' = \langle Q'_0, \theta'_0\rangle; \ldots; \langle R'_1, \sigma'_1\rangle; \langle Q'_1, \theta'_1\rangle; \ldots; \langle R'_2, \sigma'_2\rangle; \langle Q'_2, \theta'_2\rangle \ldots,$$

where $R'_1, R'_2, \ldots$ are the queries in ξ' where a non-robust atom is selected, such that for all $i > 1$, C'_i is the clause used in $\langle R'_i, \sigma'_i\rangle; \langle Q'_i, \theta'_i\rangle$. Since ξ' is finite by assumption, this implies that ξ is finite. We show the following properties for all $i \geq 0$:

- I1(i) The non-robust atoms of Q_i and Q'_i are identical and occur in the same order.
- I2(i) $\theta_i = \theta'_i\rho_i$ for some substitution ρ_i.
- I3(i) Let $Q_i = a_1, \ldots, a_n$ and assume that $Q_i\theta_i$ is π-robustly typed, and let a_k be a non-robust atom ($k \in \{1, \ldots, n\}$). By I1(i) we can write $Q'_i = (F', a_k, I')$ for some F' and I'. For every a in F', for every a_j ($j \in \{1, \ldots, n\}$) that is a descendant of a in ξ, we have $\pi(j) < \pi(k)$.
- I4(i) The head of C'_i is unifiable with the selected atom in $R'_i\sigma'_i$.

I1(0) and I3(0) follow from Def. 7.6. I2(0) and I4(0) hold since $\theta_0 = \theta'_0 = \emptyset$. We assume now that the properties hold for some $i \geq 0$, and construct

$$\langle Q'_i, \theta'_i\rangle; \ldots; \langle R'_{i+1}, \sigma'_{i+1}\rangle; \langle Q'_{i+1}, \theta'_{i+1}\rangle$$

so that the properties hold for $i + 1$.

Let Q_i, π, k and F be defined as in Part 1. By I1(i), $Q'_i = (F', p_k(\mathbf{s}_k, \mathbf{t}_k), I')$ for some F' and I', where F' contains only robust atoms. By I3(i), for every a in F', for every a_j ($j \in \{1, \dots, n\}$) that is a descendant of a in ξ, we have $\pi(j) < \pi(k)$, and therefore a_j is in F. Thus it follows by (2) that

$$R_{i+1} \text{ contains no descendants of } F'. \tag{3}$$

Let $R'_{i+1} = p_k(\mathbf{s}_k, \mathbf{t}_k), I'$. By (3) and since by I2(i), θ'_i is more general than θ_i, it is possible to construct an LD-derivation $\langle Q'_i, \theta'_i \rangle; \dots; \langle R'_{i+1}, \sigma'_{i+1} \rangle$, such that if $\tilde{C}$ is the uniquely renamed clause used to resolve an atom in ξ, then a robustly typed clause $\tilde{C}'$ corresponding to $\tilde{C}$ (using the same renaming) is used in $\langle Q'_i, \theta'_i \rangle; \dots; \langle R'_{i+1}, \sigma'_{i+1} \rangle$. Furthermore σ'_{i+1} is more general than σ_{i+1}. Hence C'_{i+1} can be used in $\langle R'_{i+1}, \sigma'_{i+1} \rangle; \langle Q'_{i+1}, \theta'_{i+1} \rangle$.

Thus I4($i+1$) holds. Since in the clause body of C'_{i+1}, the non-robust atoms occur in the same order as in C_{i+1}, I1($i+1$) holds. Since σ'_{i+1} is more general than σ_{i+1}, it follows that θ'_{i+1} is more general than θ_{i+1}, so I2($i+1$) holds. For the non-robust atoms in Q_{i+1} which occur in the clause body of C'_{i+1}, I3($i+1$) follows from Def. 7.6. For the non-robust atoms in Q_{i+1} which occur already in R_{i+1}, I3($i+1$) follows from I3(i).

By Lemma 7.5, a left-based derivation for a well fed query can be infinite only if there are infinitely many steps where a non-robust atom is resolved. Since ξ' is finite, ξ cannot have infinitely many steps where a non-robust atom is resolved, and thus ξ is finite. □

Given that for the programs of Exs. 4.2, 6.1, 6.2 and 7.1, the corresponding robustly typed programs terminate for robustly typed queries, it follows by the above theorem that the former programs terminate for well fed queries.

For Ex. 7.1, our method can only show termination for the indicated mode, but not for `nqueens`(O, I), although the program actually terminates for that mode (provided the `block` declarations are modified to allow for both modes). The reason that our method fails is not some insignificant detail of our definitions that could easily be rectified. One can definitely say that the modes in this program "go wrong": every call to `sequence`(O, I) triggers calls to `sequence`(I, O). The consequence is that `nqueens`(O, I) runs in exponential time although it could run in quadratic time.

8 Further Improvements

In this section we consider how the results obtained so far can be further improved by simplifying the `block` declarations and loosening the restriction imposed in Def. 3.1 that every clause head must be input linear. We will see that these issues are related. We have postponed these considerations so far in order not to make the main arguments of this paper unnecessarily complicated. Moreover, the results of this section are rather informal. Formalising them better is a direction of future work.

8.1 Simplifying the `block` Declarations

Sometimes the `block` declarations can be simplified based on the observation that it is not necessary to have a runtime instantiation check if, using the assumptions about the initial query, that check can be shown at compile-time to be always true. This reduces the overhead of testing for instantiation (which is very small however); in the case of built-ins, it saves us from having to introduce an auxiliary predicate; and it helps in loosening the restriction that every clause head must be input linear.

In Lemma 4.6 we have seen that under certain conditions, no `block` declaration for (arithmetic) built-ins was necessary, provided the input of the built-in originated from the clause head. This result holds for all delay-respecting derivations. Here we show that exploiting that derivations are left-based, the `block` declarations can be omitted in even more cases.

Definition 8.1 (well placed). Let P be a permutation well typed program. For a π-well typed query $p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$, an atom $p_i(\mathbf{s}_i, \mathbf{t}_i)$ is **well placed** if $\pi(j) < \pi(i)$ implies $j < i$ for all j.

A predicate p is **well placed** (with respect to P and a query Q) if for all predicates q with $p \sqsubseteq q$, all atoms using q in Q and all clause bodies in P are well placed.

The proof of the following corollary is essentially a part of the proof of Thm. 7.10.

Corollary 8.1. Let P be a well typed program where an atom is selectable in P if all input positions of non-variable type are non-variable. Let $F, p(\mathbf{s}, \mathbf{t}), H$ be a π-well typed query where $p(\mathbf{s}, \mathbf{t})$ is a well placed atom, and let

$$\xi = \langle F, p(\mathbf{s}, \mathbf{t}), H,\ \emptyset\rangle; \ldots; \langle F', p(\mathbf{s}, \mathbf{t}), H',\ \theta\rangle; \langle F', H',\ \sigma\rangle$$

be a left-based derivation of $P \cup \{F, p(\mathbf{s}, \mathbf{t}), H\}$. Then $p(\mathbf{s}, \mathbf{t})\theta$ is correctly typed in its input positions.

Corollary 8.1 says that regardless of selectability of instances of $p(\mathbf{s}, \mathbf{t})$, $p(\mathbf{s}, \mathbf{t})$ will definitely *not* be selected before it is correctly typed in its input positions, since in a left-based derivation, the atoms that "feed" $p(\mathbf{s}, \mathbf{t})$ will always be selected before $p(\mathbf{s}, \mathbf{t})$. This means that if a predicate p is well placed, then it is not necessary to check the input positions using `block` declarations. For example, the `block` declaration for `sequence` (Ex. 7.1) can be omitted.

The problem with Corollary 8.1 is that whether or not a predicate p is well placed depends on the initial query and all predicates q with $p \sqsubseteq q$. For example, `length`(I, O) and `len_aux`(I, I, O) (Ex. 3.2) are well placed, but if Ex. 3.2 is embedded into a larger program, it has to be ensured that every call to `length`(I, O) is well placed.

It is easy to see that it is nonetheless possible to simplify the `block` declaration for `len_aux` to

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

Provided that len_aux is only called via length, the second argument of len_aux is always ground when len_aux is called. However, we do not attempt to formulate a systematic rule using this insight.

8.2 Loosening Input Linearity of Clause Heads

We will now discuss the requirement of input linear clause heads in Def. 3.1. Other authors have not made this requirement in the definition of nicely moded clauses [3, 5, 12]. Instead, it is made explicitly to show that a resolvent of a nicely moded goal and a nicely moded clause is nicely moded [3, Lemma 11]. It is not needed however when the statement is restricted to *LD*-resolvents [5, Lemma 5.3]. The following example by Krzysztof Apt demonstrates this difference.

Example 8.1. Consider the program

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

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 as defined in [5, 3] but not by our definition because eq(A,A) is not input linear. The query

```
q(X), r(Y), eq(X,Y)
```

is nicely moded (by both definitions). The query q(X), r(X) is a resolvent of the above query, and it is *not* nicely moded (by both definitions).

The example shows that in a nicely moded program (according to our definition), =/2 cannot be used in mode $I = I$, since it is (conceptually) defined by the fact clause A = A. Thus it is not possible to circumvent the requirement of input linear clause heads by replacing multiple occurrences of the same variable with renamed variables and introducing equations between those variables. For example, assuming mode $\mathtt{len\_aux}(O, I, I)$, it does not help to replace the clause "len_aux([],N,N)." of Ex. 3.2 with

```
len_aux([],N,M) :- N = M.
```

What this also shows is that in order to weaken the requirement of input linear clause heads, we can focus on the question how =/2 could be used in mode $I = I$. Every derivation step using a clause head that is not input linear can conceptually be regarded as a derivation step using an input linear clause head, followed by a sequence of derivation steps using $I = I$.

Requiring input linear clause heads is undoubtedly a severe restriction. It means that it is not possible to check two input arguments for equality. However, this also indicates the deeper 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 unifies its two arguments.

It is easy to see that Lemma 3.1 still holds if Def. 3.1 is weakened by allowing `=` to be used in mode $I = I$, provided atoms using `=` are only resolved when both arguments are ground. Clearly, 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.

From a semantic point of view, it is reasonable to require that an atom $s = t$, where the mode is $I = I$, is only resolved when s and t are ground. Only when s and t are ground is the usual semantics of $=$ consistent with the requirement that $=$ should be a *test*, that is, not bind variables. One might argue that this is also the case if s and t are bound to identical terms possibly containing variables. However this argument is somewhat flawed in that it states a condition for $s = t$ *being* a test which already implies that the test succeeds.

Of course, one could have an equality test predicate whose operational semantics is as follows: if s and t are identical, it succeeds; if s and t are not unifiable, it fails; otherwise, the test is delayed until s or t become further instantiated. Such a test is used in the *guards* of clauses in concurrent (constraint) logic languages such as FGHC [28], but in ordinary logic programming, it is usually not provided.

Thus one way to allow for $I = I$ is to have a delay declaration such that an atom $s = t$ is selected only when s and t are ground. In SICStus this can be realised using the built-in `when` [16]. However we do not follow this line because this paper focuses on `block` declarations, and because it would commit $s = t$ 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, Definition 3.1 and the subsequent Definitions 4.1 and 7.1 can be generalised by allowing for clause heads $p(\mathbf{v}, \mathbf{u})$ where a variable x occurs in several input positions, provided that

- p is well placed with respect to the program and initial query, and
- all occurrences of x in $\mathbf{v}$ are in positions of ground type.

By Corollary 8.1, 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 3.1, 4.2 and 7.1 hold under the more general definitions.

Secondly, one can use the results of Subsec. 4.2. Definition 4.1 can be generalised by allowing for clause heads $p(\mathbf{v}, \mathbf{u})$ where a variable x occurs in several input positions, provided that

- all occurrences of x in $\mathbf{v}$ are direct and in positions of constant type, and
- an atom using p is selectable only if these positions are non-variable.

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

Example 8.2. Consider Ex. 3.2. This program can be used in both given modes, using any of the two explanations above. However, the second explanation is better because it is valid independent of any program into which Ex. 3.2 might be embedded. It does not rely on all calls to `length` being well placed.

9 Related Work

In using modes and types, we follow **Apt** and **Luitjes** [3], 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 declarations which delay an atom 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.

Our results on unification freedom are based on work by **Apt** and **Etalle** [2]. They assume well typed programs and LD-derivations and can therefore rely on the selected atom always being correctly typed in its input positions.

Naish [24] gives good intuitive explanations why programs loop, which directed our own search for further ideas and their formalisation. To ensure termination, he proposes some heuristics, without any formal proof.

Predicates are assumed to have a single mode. Naish suggests that alternative modes should be achieved by multiple versions of a predicate.[8] However, under that assumption, why have delay declarations in the first place? For instance, in the mentioned example `permute`, if we only consider `permute`(O, I), then Ex. 6.1 does not loop for the plain reason that no atom ever delays, and thus the program behaves as if there were no delay declarations. In this case, the interpretation that one should "put recursive calls last" is misleading. If we only consider `permute`(I, O), then the version of Ex. 6.1 is much less efficient than Ex. 3.1. In short, the whole discussion on delay declarations makes little sense when only one mode is assumed.

Lüttringhaus-Kappel [18] proposes a method of generating control automatically, and has applied it successfully to many programs. However, rather than pursuing a formalisation of some intuitive understanding of why programs loop, and imposing appropriate restrictions on programs, he attempts a high degree of generality. This has certain disadvantages.

The method only finds *acceptable* delay declarations, ensuring that the most general selectable atoms have finite SLD-trees. What is required however are *safe* delay declarations, ensuring that *instances* of most general selectable atoms have finite SLD-trees. A *safe* program is a program for which every acceptable delay declaration is safe. 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. This is a missing link.

The delay declarations for some programs such as `quicksort` require an argument to be a nil-terminated list before an atom can be selected. As Lüttringhaus-

[8] This is also the approach taken in Mercury [27], where these versions are generated by the compiler.

Kappel points out himself, “in NU-Prolog [*or SICStus*] it is not possible to express such conditions”. We have shown here that, with a knowledge of modes and types, `block` declarations are sufficient.

Floundering cannot be ruled out systematically, but only be avoided on a heuristic basis. Thus in principle, the method sometimes enforces termination by floundering. This lies in the nature of the weak assumptions made, and thus is sometimes unavoidable, but there is no notion that would allow to reason about whether for a particular program, it was unavoidable or not. In contrast, we can say that a permutation well typed program should not flounder for a permutation well typed query.

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** [19] base termination on norms and the *covering* relation between subqueries of a query. This is loosely related to well-typedness. However, their results are not comparable to ours because they assume a *local selection rule*, that is a rule which always selects an atom which was introduced in the most recent step. No existing language using a local 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** [22] ensure termination by imposing a depth bound on the SLD tree. This is realised by a 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.

Of course our work implicitly relies on previous work on termination for LD-derivations. Throughout this paper, we have alway assumed that termination for the corresponding nicely moded and well typed program has been shown. Methods for showing termination are treated in [1, 4, 8–12].

10 Discussion and Future Work

We have presented verification methods for programs with `block` declarations. The verified properties were freedom from occur-check, floundering, errors related to built-ins and unification. The main contribution was about termination. These methods refine and formalise previous work in this area [2, 3, 24].

Our results on occur-check freedom and non-floundering are straightforward variations of [3], based on the observation that considering arbitrary delay-respecting derivations one can generalise the mode concepts so that textual order of atoms is less relevant.

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

For showing error-freedom, we have introduced two quite different methods. Whether one of these methods or even both are applicable depends on the program. It would be desirable to find *one* uniform approach which would work for a larger class of programs, but so far we have not accomplished this.

We have shown that permutation simply typed programs are unification free. This result is more general that the corresponding one in [2] in that we allow for arbitrary delay-respecting derivations, but on the other hand, we are more restrictive in that we only allow flat terms in input positions in clause heads.

For proving termination, we have presented two approaches. The first approach [25] is a combination of two methods based on not *making* and not *using* speculative bindings, respectively. For Exs. 6.1 and 6.2, it turns out that in one mode, the first method applies, and in the second mode, the other method applies. This approach is simple to understand and to apply. However it is rather limited. Termination cannot be shown for Exs. 4.2 and 7.1.

In the second approach [26], we required programs to be *permutation robustly typed*, a property which 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 sufficient when they are called.

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

It is an ongoing discussion whether it is reasonable to assume predicates that work in several modes [14]. We have argued that multiple modes is a natural assumption in the context of `block` declarations. Therefore it is not the relevancy of our work that depends on this assumption, but to a large extent the relevancy of `block` declarations as such. However, multiple modes is not the only application of `block` declarations. They can also be used to implement the test-and-generate paradigm (coroutining). As seen in Ex. 7.1, our results apply to such programs as well.

We have discussed that `block` declarations can be omitted or simplified when sufficient instantiation can be guaranteed at compile time (always assuming the initial query is permutation well typed). Another interesting issue is how achieving multiple modes using `block` declarations affects the efficiency of programs. Furthermore we plan to develop a more powerful method of showing that predicates are robust.

Acknowledgements

We thank the anonymous referees of PLILP/ALP'98 and LOPSTR'98, as well as all colleagues who commented on our work during the LOPSTR'98 workshop, in particular Krzysztof Apt. Jan-Georg Smaus was supported by EPSRC Grant No. GR/K79635.

References

1. K. R. Apt. *From Logic Programming to Prolog*, chapter 6. Prentice Hall, 1997.
2. K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science*, LNCS, pages 1–19, Berlin, 1993. Springer-Verlag.
3. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *Proceedings of AMAST'95*, LNCS, Berlin, 1995. Springer-Verlag. Invited Lecture.
4. K. R. Apt and D. Pedreschi. Studies in Pure Prolog: Termination. In J. W. Lloyd, editor, *Proceedings of the Symposium in Computational Logic*, pages 150–176. Springer-Verlag, 1990.
5. K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.
6. A. Bossi and N. Cocco. Successes in logic programs. In Pierre Flener, editor, *Pre-proceedings of LOPSTR*, number UMCS-98-6-1, pages 146–153. University of Manchester, 1998. Extended abstract.
7. R. Chadha and D.A. Plaisted. Correctness of unification without occur check in Prolog. Technical report, University of North Carolina, 1991.
8. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.
9. D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proceedings of FGCS*, pages 481–488. ICOT Tokyo, 1992.
10. S. Decorte and D. De Schreye. Automatic inference of norms: a missing link in automatic termination analysis. In D. Miller, editor, *Proceedings of ILPS*, pages 420–436. MIT Press, 1993.
11. S. Decorte and D. De Schreye. Termination analysis: Some practical properties of the norm and level mapping space. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 233–249. MIT Press, 1998.
12. S. Etalle, A. Bossi, and N. Cocco. Well-terminating programs. *Journal of Logic Programming*, 1998. Accepted for publication.
13. M. Fitting. *First-order Logic and Automated Theorem Proving*. Springer-Verlag, 1996.
14. P. M. Hill, editor. *ALP Newsletter*, `http://www-lp.doc.ic.ac.uk/alp/`, February 1998. Pages 17,18.

15. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
16. Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 1997. `http://www.sics.se/isl/sicstus/sicstus_toc.html`.
17. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
18. S. Lüttringhaus-Kappel. Control generation for logic programs. In D. Warren, editor, *Proceedings of ICLP*, pages 478–495. MIT Press, 1993.
19. E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. W. Lloyd, editor, *Proceedings of ILPS*, pages 447–461. MIT Press, 1995.
20. K. Marriott, L. Naish, and J. L. Lassez. Most specific logic programs. *Annals of mathematics and artificial intelligence*, 1(2), 1990. Also in proceedings of the Fifth JICSLP.
21. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
22. J. C. Martin and A. M. King. Generating efficient, terminationg logic programs. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT'97*, LNCS, pages 273–284. Springer-Verlag, 1997.
23. L. Naish. Automatic control of logic programs. *Journal of Logic Programming*, 2(3):167–183, 1985.
24. L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, University of Melbourne, 1992.
25. J.-G. Smaus, P. M. Hill, and A. M. King. Preventing instantiation errors and loops for logic programs with several modes using `block` declarations. In Pierre Flener, editor, *Pre-proceedings of LOPSTR*, number UMCS-98-6-1, pages 72–29. University of Manchester, 1998. Extended abstract.
26. J.-G. Smaus, P. M. Hill, and A. M. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamadessi, editor, *Proceedings of PLILP/ALP*, LNCS. Springer-Verlag, 1998.
27. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, November 1996.
28. K. Ueda. Guarded Horn clauses. In E. Wada, editor, *Logic Programming '85, Proceedings of the 4th Conference*, volume 221 of *LNCS*, pages 168–179. Springer-Verlag, 1986.