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

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. This paper presents several verification methods for logic programs with delay declarations. It is shown how type and instantiation errors related to built-ins can be prevented, and how termination can be ensured. Three features are distinctive of this work: it is assumed that predicates can be used in several modes; it is shown that `block` declarations, which are a very simple delay construct, are sufficient to ensure the desired properties; the selection rule is taken into account, assuming it to be the rule of most Prolog implementations. The methods can be used both to verify existing programs and to assist in writing new programs.

1 Introduction

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

We assume that delay declarations may be used to enable programs to run in multiple modes. Although other authors [3, 15] have not explicitly assumed multiple modes, they mainly give examples where delay declarations are clearly used for that purpose. However, our results are also useful in the context of other applications of delay declarations, such as the test-and-generate paradigm [15] or parallel execution [3].

Our contributions are: showing how type and instantiation errors related to built-ins can be prevented; showing when delay declarations for built-ins can be omitted completely; and showing how termination can be ensured.

For all of the above, we demonstrate that under realistic assumptions, `block` declarations, which declare that certain arguments of an atom must be at least *non-variable* before that atom can be selected, are sufficient. As demonstrated in SICStus [8], `block` declarations can be efficiently implemented; the instantiation test has hardly any impact on performance. Thus, in practice, such constructs are the most frequently used delay declarations.

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

Predicates which use their own output as input (*circular modes*) are a well-known source of loops which must be eliminated to ensure termination [15]. We generalise the idea of "well-behaved" modes to multi-moded predicates.

Another source of loops is *speculative output bindings*, that is bindings made before it is known that a solution exists [15]. We propose two methods for dealing with this problem and thus proving (or ensuring) termination. Which method must be applied depends on the program and on the mode being considered. The first method exploits that a program does not *use* any speculative bindings, by ensuring that no atom ever delays. The second method exploits that a program does not *make* any speculative bindings. We always reduce the problem to showing termination for programs without delay declarations, such that any method for programs without delay declarations [1, 5] can be applied.

The approach to termination presented here is simple and formalises previous heuristics [14, 15]. Although there are some clear limitations of the approach, it is not subsumed by any method we know of, in particular not by the method we presented in [16].

The rest of this paper is organised as follows. The next section gives some preliminaries. Section 3 defines some concepts of modes and types which are the basis of our verification methods. Section 4 is about errors related to built-ins. Section 5 is about termination. Section 6 investigates related work, and Section 7 concludes.

2 Preliminaries

We base the notation on [3, 11]. For the examples we use SICStus notation [8]. We recall some important notions. The set of variables in a syntactic object o is denoted by $vars(o)$. A syntactic object is called **linear** if every variable occurs in it at most once. A **flat** term is a variable or a term of the form $f(x_1, \dots, x_n)$, where $n \geq 0$ and the x_i are distinct variables. The **domain** of a substitution σ is $dom(\sigma) = \{x \mid x\sigma \neq x\}$.

For a predicate p/n, a **mode** is an atom $p(m_1, \dots, m_n)$, where $m_i \in \{I, O\}$ for $i \in \{1, \dots, n\}$. Positions with I are called **input positions**, and positions with O are called **output positions** of p. A **mode of a program** is a set containing one mode for each of its predicates.[1] 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 of p, respectively.

A **type** is a set of terms closed under instantiation. A **non-variable type** is a type that does not contain variables. *The* **variable type** is the type that

[1] This can easily be extended to allow for different *occurrences* of a predicate within a program to have different modes.

contains variables and hence, as it is instantiation closed, all terms. A **constant type** is a type that contains only (possibly infinitely many) constants. In the examples, we use the following types: *any* is the variable type, *list* the type of (nil-terminated) lists, *num* the type of numbers and *nl* the type of number lists.

We write $t : T$ for "t is in type T". We use $\mathbf{S}$, $\mathbf{T}$ to denote vectors of types, and write $\models \mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$ if for all substitutions σ, $\mathbf{s}\sigma : \mathbf{S}$ implies $\mathbf{t}\sigma : \mathbf{T}$. It is assumed that each argument position of each predicate p/n has a type associated with it. These types are indicated by writing the atom $p(T_1, \ldots, T_n)$ where $T_1, \ldots, T_n$ are types. The **type of a program** P is a set of such atoms, one for each predicate defined in P.

A `block` declaration [8] for a predicate p/n is a (possibly empty) set of atoms each of which has the form $p(b_1, \ldots, b_n)$ where $b_i \in \{\texttt{?}, \texttt{-}\}$ for $i \in \{1, \ldots, n\}$. A **program** consists of a set of clauses and a set of `block` declarations, one for each predicate defined by the clauses. If P is a program, an atom $p(t_1, \ldots, t_n)$ is **selectable in** P if for each atom $p(b_1, \ldots, b_n)$ in the `block` declaration for p, there is some $i \in \{1, \ldots, n\}$ such that t_i is non-variable and $b_i = \texttt{-}$.[2]

A **query** is a finite sequence of atoms. A **derivation step** for a program P is a pair $\langle Q, \theta \rangle; \langle R, \theta\sigma \rangle$, where $Q = Q_1, a, Q_2$ and $R = Q_1, B, Q_2$ are queries; θ is a substitution; a an atom; $h \leftarrow B$ a renamed variant of a clause in P and σ the most general unifier of $a\theta$ and h. We call a (or $a\theta$)[3] the **selected atom** and $R\theta\sigma$ the **resolvent** of $Q\theta$ and $h \leftarrow B$.

A **derivation** ξ **for** a program P is a sequence $\langle Q_0, \theta_0 \rangle; \langle Q_1, \theta_1 \rangle; \ldots$ where each pair $\langle Q_i, \theta_i \rangle; \langle Q_{i+1}, \theta_{i+1} \rangle$ in ξ is a derivation step. Alternatively, we also say that ξ is a **derivation of** $P \cup \{Q_0\theta_0\}$. We also denote ξ as $Q_0\theta_0; Q_1\theta_1; \ldots$.

An **LD-derivation** is a derivation where the selected atom is always the leftmost atom in a query. A **delay-respecting derivation** for a program P is a derivation where the selected atom is always selectable in P. A derivation is **left-based** if it is either an LD-derivation, or it contains at least one non-empty query where the leftmost atom is not selectable. To the best of our knowledge, derivations in most Prolog implementations are left-based.

3 Permutations and Modes

In [3], three correctness properties for programs are introduced: *nicely moded*, *well typed*, and *simply moded*. 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. It is assumed that each predicate has a single mode. Nicely-modedness is used to show that the occur-check can safely be omitted. Well-typedness is used to show that derivations do not flounder. Finally, simply-modedness is a special case of nicely-modedness and is used to show that a program is free from errors related to built-ins. Other authors have

[2] Note that in the (pathological) case that p has a `block` declaration $p(\texttt{?}, \ldots, \texttt{?})$, an atom using p can never be selectable.

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

also used these or similar concepts, for example to show that programs are unification free [2], successful [4], and terminating [6].

In a query (clause body), one can consider three different orderings among the atoms. First, there is the *textual* order. Secondly, there is a conceptual order given by the *producer-consumer relation* [10]. According to this order, the atom that produces a piece of data occurs before atoms that consume it. Thirdly, there is the execution order, which depends on the selection rule.

In the case of LD-derivations, all of these orders are identical. This assumption underlies the above concepts as used in [2, 4, 6].

In the case of arbitrary delay-respecting derivations, none of these orders is necessarily identical to another one. However, since the textual order is irrelevant for the selection of an atom and hence for the execution order, one might just as well assume, for the sake of notational convenience, that the textual order is identical to the producer-consumer order. Although not explicitly stated, this assumption underlies the above concepts as used in [3]. More precisely, any result stated there can be generalised trivially to programs where the atoms in the clause bodies are permuted in an arbitrary way.

In the case of left-based derivations, again none of these orders is necessarily identical to another one. However, the textual order is relevant for the execution order. Therefore it is not correct to make a simplifying assumption about the textual order as for arbitrary delay-respecting derivations. We need a formalism that makes both the textual order and the producer-consumer order explicit.

This is accomplished by associating, with each query and each clause in a program, a permutation π of the (body) atoms, which gives the producer-consumer order. For a different mode, the permutations would be different. Let π be a permutation on $\{1, \ldots, n\}$. For notational convenience we assume that $\pi(i) = i$ for $i \notin \{1, \ldots, n\}$. In examples, π is written as $\langle \pi(1), \ldots, \pi(n) \rangle$. Also, we write $\pi(o_1, \ldots, o_n)$ for the sequence obtained by applying π to the sequence $o_1, \ldots, o_n$, that is $o_{\pi^{-1}(1)}, \ldots, o_{\pi^{-1}(n)}$.

Note that most results in this paper hold for arbitrary delay-respecting derivations, but the result in Subsec. 5.1 assumes left-based derivations. For the sake of consistency we use the permutations throughout.

3.1 Permutation Nicely Moded Programs

In a *nicely moded* query, a variable in an input position does not occur later in an output position, and each variable in an output position occurs only once. This has been generalised in [16].

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 $\{1, \ldots, n\}$. Then Q is **π-nicely moded** if $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a linear vector of terms and for all $i \in \{1, \ldots, n\}$

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

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

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

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

Example 3.1.

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

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

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

We now recall a persistence property of permutation nicely-modedness [16].

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

For permutation nicely moded programs and queries, the occur-check can be omitted [3, 17].

3.2 Permutation Well Typed Programs

Well-typedness is a generalisation of a simpler concept called *well-modedness* [3]. The idea is that given a query Q, a, Q', when Q is resolved away, then the atom a becomes sufficiently instantiated to be selected. As with modes, we assume that the types are given. In the examples, they will be the obvious ones.

Definition 3.2 (permutation well typed [16]). Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query, where $p_i(\mathbf{S}_i, \mathbf{T}_i)$ is the type of p_i for each $i \in \{1, \ldots, n\}$. Let π be a permutation on $\{1, \ldots, n\}$. Then Q is **π-well typed** if for all $i \in \{1, \ldots, n\}$ and $L = 1$

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

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

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

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

The following lemma is analogous to Lemma 3.1 and has also been stated in [16].

Lemma 3.2. Every resolvent of a permutation well typed query Q and a permutation well typed clause C, where $vars(C) \cap vars(Q) = \emptyset$, is permutation well typed.

For permutation well typed programs and queries, no derivation flounders [17].

3.3 Permutation Simply Typed Programs

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

Definition 3.3 (permutation simply typed). 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 $\{1, \ldots, n\}$. Then Q is **π-simply typed** if it is π-nicely moded and π-well typed, and $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a vector of *variables*.

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **π-simply typed** if it is π-nicely moded and π-well typed, $\mathbf{t}_1, \ldots, \mathbf{t}_n$ 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 **permutation simply typed** query (clause, program) and a **simply typed** query (clause, program) **corresponding to** a query (clause, program) are defined in analogy to Def. 3.1.

Example 3.4. The following is a version of `quicksort`, where `leq` and `grt` 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]):-
  grt(X,C),
  part(Xs,C,As,Bs).

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

The program is permutation simply typed with respect to the obvious types for mode $\{\texttt{qs}(I,O), \texttt{append}(I,I,O), \texttt{leq}(I,I), \texttt{grt}(I,I), \texttt{part}(I,I,O,O)\}$. It is not permutation simply typed for mode $\{\texttt{qs}(O,I), \texttt{append}(O,O,I), \texttt{leq}(I,I), \texttt{grt}(I,I), \texttt{part}(O,I,I,I)\}$, due to the non-variable term `[X|Bs2]` in an output position.

The persistence properties stated in Lemmas 3.1 and 3.2 are independent of the selectability of an atom in a query. We show a similar persistence property for permutation simply typed programs. However this property only holds if the selected atom is sufficiently instantiated in its input arguments, since otherwise output positions of the resolvent might become non-variable. This motivates the following definition.

Definition 3.4 (bound position). Let P be a permutation simply 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.

Note that by Def. 3.3, a bound position must be of non-variable type.

Lemma 3.3. 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 $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ a permutation simply typed clause where $vars(C) \cap vars(Q) = \emptyset$. Suppose that for some $k \in \{1, \ldots, n\}$, $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is non-variable in all bound input positions[4] and θ is the

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

mgu of $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$. Then the resolvent of Q and C with selected atom $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is permutation simply typed. Moreover,

a. $dom(\theta) \subseteq vars(\mathbf{t}_k) \cup vars(\mathbf{v}_0)$, and
b. $(\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n)\,\theta =$
$\mathbf{t}_1, \ldots, \mathbf{t}_{k-1}, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{t}_{k+1}, \ldots, \mathbf{t}_n$

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

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

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

By Lemmas 3.1 and 3.2, the resolvent is permutation nicely moded and permutation well typed. By (b), the vector of the output arguments of the resolvent is a linear vector of variables, and hence the resolvent is permutation simply typed. □

The following corollary of Lemma 3.3 holds since by Def. 3.2, the leftmost atom in a simply typed query is non-variable in its bound input positions.

Corollary 3.4. 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 following definition gives a name to the condition that the selected atom must always be non-variable in bound input positions.

Definition 3.5 (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.

In a permutation simply typed query, 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.

Note that for input selectability, 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 3.5. 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.

The next lemma 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.

Lemma 3.5. Let P be a permutation simply typed program with input selectability and Q_0 be a permutation simply typed query. Let $\theta_0 = \emptyset$ and $\xi = \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle; \ldots$ be a delay-respecting derivation of $P \cup \{Q_0\}$. Then for all $i \geq 0$, if x is a variable occurring in an output position in Q_i, then $x\theta_i = x$.

Proof. The proof is by induction on the position i in the derivation. The base case $i = 0$ is trivial since $\theta_0 = \emptyset$. Now suppose the result holds for some i and Q_{i+1} exists. By Lemma 3.3, $Q_i\theta_i$ is permutation simply typed. Thus the result follows for $i + 1$ by Lemma 3.3 (b). $\square$

4 Errors Related to Built-ins

One problem with built-ins is that their implementation may not be written in Prolog. Thus, for the purposes of this paper, it is assumed that each built-in is conceptually defined by possibly infinitely many (fact) clauses. The ISO standard for Prolog [9] does not define the built-in predicates as conceptual clauses, but it is nevertheless so precise that it should generally be possible to verify whether such a definition is correct.

To prove that a program is free from errors related to built-ins, we require it to meet certain correctness conditions some of which have been introduced in the previous section (see Definitions 3.1, 3.2, 3.3). The correctness conditions have to be satisfied by the conceptual clauses for the built-ins as well as by the user-defined part of the program.

For instance, consider the atom `M2 is M + 1` in Ex. 3.2. Conceptually, we regard this as an atom with two arguments `M2` and `M`, and we assume that there are facts "`1 is 0+1.`", "`2 is 1+1.`", and so forth.

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. In this section, we consider two approaches to ensure freedom from instantiation and type errors due to the presence of delay declarations. For different programs and built-ins, 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, it seems that delay declarations do not affect the problem of type errors, be it positively or negatively. Delay declarations cannot enforce arguments to be correctly typed. Also, one would not expect that a selection rule that is not left-to-right could be the *cause* of wrongly typed arguments.

This is probably true in practice, but in theory, there is an aspect of type errors that is specifically related to delay declarations. Consider the program consisting of the fact clause 'two(2).' and the built-in is, with type $\{\texttt{two}(num),$ $\texttt{is}(num, num)\}$ and mode $\{\texttt{two}(O), \texttt{is}(O, I)\}$. The query

$$\texttt{X is foo, two(foo)}$$

is $\langle 2, 1\rangle$-well typed since trivially $\models \texttt{foo} : num \Rightarrow \texttt{foo} : num$. That is, since the output of two is wrongly typed, we can say that correctly typed output of two implies correctly typed input for is. 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 `two(foo), X is foo`. Since the type of two is *num* and the program is well typed, the atom two(foo) can never be resolved, and therefore the derivation fails without ever reaching `X is foo`.

4.2 Exploiting Constant Types

The approach described in this subsection aims at preventing instantiation and type errors for built-ins, for example arithmetic built-ins, that require arguments to be ground. It has been proposed in [3] that these predicates be equipped with delay declarations to ensure that they are only executed when the input is ground. This has the advantage that one can reason about arbitrary arithmetic expressions, say 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 similar to and based on [3, Lemma 27].

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

Proof. By Def. 3.2, $\mathbf{s}_k\theta : \mathbf{S}_k$, and thus $\mathbf{s}_k\theta$ is a vector of constants. Since $\mathbf{s}_k$ is already a vector of non-variable terms, it follows that $\mathbf{s}_k$ is a vector of constants and thus $\mathbf{s}_k\theta = \mathbf{s}_k$. Therefore $\mathbf{s}_k : \mathbf{S}_k$. $\square$

By Def. 3.3, 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. This is stated in the following theorem which is a consequence of Lemma 4.1.

Theorem 4.2. Let P be a permutation simply typed program with input selectability and Q be a permutation simply typed query. Let p be a predicate whose input arguments are all of constant type. Then in any derivation of $P \cup \{Q\}$, an atom using p will be selected only when its input arguments are correctly typed.

Example 4.1. Consider the program in Ex. 3.4 in mode $\mathtt{qs}(I, O)$. No delay-respecting derivation for a permutation simply typed query and this program can result in an instantiation or type error related to the arithmetic built-ins.

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 built-ins. We observe that these built-ins have argument positions of the constant type *num*. Thus, the idea is to declare certain argument positions in a predicate, including the above argument positions of the built-ins, to be *atomic*. This means that they can only be ground or free but not partially instantiated. Then there needs to be a `block` declaration such that an atom delays until the arguments in these positions are non-variable, and hence ground. Just as with types and modes, we assume that the positions which are atomic are already known.

Definition 4.1 (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.

Lemma 4.3. Let C be a clause and Q a query which respect atomic positions, where $vars(C) \cap vars(Q) = \emptyset$. Then every resolvent of C and Q also respects atomic positions.

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

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

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.

Consider a term s filling an atomic position in $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$ or $b_1, \ldots, b_m$. If s is a ground term, then $s\theta$ is also a ground term. Suppose that s is a variable. If $s \notin dom(\theta)$, then $s\theta$ is also a variable. If $s \in dom(\theta)$ then s must fill an atomic position in a_k or h. By the previous paragraph, $s\theta$ is either a variable which only occurs in atomic positions in Q', or a ground term. $\square$

Using Lemma 4.3 we can show freedom from instantiation errors for programs where the arguments of type *num* are variable-disjoint from any other arguments.

Example 4.2. Consider the program for `length` in Ex. 3.2. If the atomic positions are exactly those positions with arithmetic arguments, then the program respects atomic positions. The `block` declaration on the built-in `<` is realised with an auxiliary predicate `less`.

Note that in the second clause for `len_aux`, for each mode, the input for `is` is provided by the clause head. Therefore no `block` declaration and hence no auxiliary predicate is required for `is` — it is sufficient to have a `block` declaration for `len_aux`. This is formalised by the following definition and lemma.

Definition 4.2 ($\mathcal{B}$-ground). Let P be a program which respects atomic positions and $\mathcal{B}$ a set of predicates whose input positions are all atomic.

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

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

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

By the following theorem, for $\mathcal{B}$-ground programs and queries 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.

Theorem 4.4. Let P and Q be a $\mathcal{B}$-ground program and query, and ξ be a delay-respecting derivation of $P \cup \{Q\}$. Then each query in ξ is $\mathcal{B}$-ground.

Proof. The proof is by induction on the length of ξ. Let $Q_0 = Q$ and $Q_0; Q_1; \ldots$ be a delay-respecting derivation of $P \cup \{Q_0\}$. The base case holds by the assumption that Q_0 is $\mathcal{B}$-ground.

Now consider some Q_j where $j \geq 0$ and Q_{j+1} exists. By Lemma 4.3, Q_j and Q_{j+1} respect atomic positions.

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

If $p \notin \mathcal{B}$, then by Def. 4.2, $\mathbf{u}$ is non-variable in the $\mathcal{B}$-positions of p, and hence, since Q_j respects atomic positions, $\mathbf{u}$ is *ground* in the $\mathcal{B}$-positions of p.

If $p \in \mathcal{B}$, then $\mathbf{u}$ is ground in the $\mathcal{B}$-positions of p by the induction hypothesis.

Thus it follows that in both cases, $\mathbf{s}_i\theta$ is ground. Since the choice of i was arbitrary, it follows that Q_{j+1} is $\mathcal{B}$-ground. □

Example 4.3. The program of Ex. 3.2 is $\{$`is`$\}$-ground, and the second position of `len_aux` is an $\{$`is`$\}$-position. By Thm. 4.4, it is justified that there is no `block` declarations for `is`.

5 Termination

We are interested in ensuring that *all* derivations of a query are finite. Therefore the clause order in a program is irrelevant. Furthermore, we do not prove termination as such, but rather reduce the problem of proving termination for a program *with* `block` declarations to the same problem for a corresponding program *without* `block` declarations. We always assume that termination for the latter program can be shown using an existing method for LD-derivations [1, 5]. We present two methods for showing termination, which are illustrated in the following example.

Example 5.1. The query `permute(V,[1])` (Ex. 3.1) loops because `delete` produces a *speculative output binding* [15]: The output variable `Y` is bound before it is known that this binding will never have to be undone. Assuming left-based derivations, termination in both modes can be ensured by replacing the second clause with

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

This technique has been described as *putting recursive calls last* [15].

To explain termination of this program, we have to apply a different reasoning for the different modes. In mode `permute(`O, I`)`, the atom that produces the speculative output occurs *textually before* the atom that consumes it. This means that the consumer waits until the producer has completed (undone the speculative binding). The program does not *use* speculative bindings. In mode `permute(`I, O`)`, `delete` is used in mode `delete(`I, O, I`)`, and in this mode it does not *make* speculative bindings.

Note that termination for this example depends on derivations being left-based, and so any method which abstracts from the selection rule must fail.

Both methods described below formalise previous heuristics [14, 15] and rely on conditions that are easy to check. It can be shown using these methods that Ex. 5.1 will terminate. These methods are not adequate for Ex. 3.4. A more complex method suitable for Ex. 3.4 is described in [16]. However, we will also give two examples of programs for which termination can be shown using the methods described here, but not using the method of [16].

5.1 Termination by not Using Speculative Bindings

In LD-derivations, speculative bindings are never used [15]. A left-based derivation *is* an LD-derivation, provided the leftmost atom in each query in the derivation is always selectable. This implies the following proposition.

Proposition 5.1. Let Q be a well typed query and P a well typed program such that an atom is selectable in P whenever its input positions of non-variable type are non-variable. Then every left-based derivation of $P \cup \{Q\}$ is an LD-derivation.

We now give two examples of programs where by Prop. 5.1, we can use any method for LD-derivations to show termination for any well typed query.

Example 5.2. Consider $\texttt{permute}(O, I)$ as defined in Ex. 5.1 with either of the types given in Ex. 3.3. This program is well-typed.

Example 5.3. Consider the following version of $\texttt{delete}(O, I, O)$.

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

Assuming either of the types given in Ex. 3.3, this program is well typed. Note that for this program, the method of [16] is not applicable.

5.2 Termination by not Making Speculative Bindings

Some programs have the nice property that there cannot be any failing derivations. In [4], such a class of programs called *noFD* was identified. *Non-speculative* programs are similar, but there are two differences: the definition of noFD programs allows only for *LD*-derivations, but on the other hand, the definition of non-speculative programs requires that the clause heads are input linear.

Definition 5.1 (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.

Example 5.4. Both versions of the `permute` program (Examples 3.1 and 5.1), assuming either of the types 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. Theorem 5.2 says that this can only happen if the simply typed program corresponding to P also has an infinite (LD-)derivation for this query.

Theorem 5.2. 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 $P' \cup \{Q'\}$.

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

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

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

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

We first show these properties for $i = 0$. Let $q(\mathbf{u}, \mathbf{v})$ be an atom in Q'_0 that is not resolved in ξ. Since $\theta'_0 = \emptyset$, $\mathbf{v}\theta'_0 = \mathbf{v}$. Furthermore, by Corollary 3.4 and Lemma 3.5 and since $q(\mathbf{u}, \mathbf{v})$ is not resolved in ξ, we have $\mathbf{v}\theta_j = \mathbf{v}$ for all j. Thus (1) holds. (2) holds because $\theta'_0 = \emptyset$.

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

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

We must show that (1) and (2) hold for $i+1$. Consider an atom $q(\mathbf{u}, \mathbf{v})$ in Q'_i other than $p(\mathbf{s}, \mathbf{t})$. By Lemma 3.3 (b), $vars(\mathbf{v}\theta'_i) \cap dom(\phi'_i) = \emptyset$. 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 Lemma 3.5, $\mathbf{v}\theta'_{i+1} = \mathbf{v}$. 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 Lemma 3.5, $\mathbf{v}\theta_j = \mathbf{v}$. Thus (1) follows. (2) holds since it holds for i and $p(\mathbf{s}, \mathbf{t})$ is resolved using the same clause head as in ξ.

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

We must show that (1) and (2) hold for $i+1$. Consider an atom $q(\mathbf{u}, \mathbf{v})$ in Q'_i other than $p(\mathbf{s}, \mathbf{t})$. By Lemma 3.3 (b), $vars(\mathbf{v}\theta'_i) \cap dom(\phi'_i) = \emptyset$. 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 Lemma 3.5, we have $\mathbf{v}\theta'_{i+1} = \mathbf{v}$, (1) holds for $i+1$.

By (1) for i, we have $vars(\mathbf{t}\theta'_i) \cap dom(\theta_j) = \emptyset$ for all j. By Lemma 3.3 (a), we have $dom(\phi'_i) \subseteq vars(\mathbf{t}\theta'_i) \cup vars(C')$. Thus we have $dom(\phi'_i) \cap dom(\theta_j) = \emptyset$ for all j. Moreover, (2) holds for i. Thus (2) holds for $i+1$.

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

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

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

Theorem 5.2 says that for non-speculative programs, atom order in clause bodies is irrelevant for termination. In other words, termination is independent of any particular selection rule, as long as derivations are delay-respecting.

We now give an example of a program for which termination can be shown using non-speculative programs but not using the method of [16].

Example 5.5. The following program has mode $\{$`plus_one`(I), `minus_two`(I), `minus_one`(I), `two`(O), `one`(O), `zero`$(O)\}$, and the type of all arguments is nl.

```
:- 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

$$Q = \texttt{plus\_one(X)},\ \texttt{two(X)}$$

is $\langle 2, 1\rangle$-simply typed. Moreover, it is easy to see that, for the corresponding simply typed query `two(X),plus_one(X)`, any *LD*-derivation is finite. Therefore by Thm. 5.2, all delay-respecting derivations of Q are finite.

6 Related Work

This work was inspired by **Apt** and **Luitjes** [3]. 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. Little attention is given to termination, proposing a method limited to deterministic programs.

Naish [15] 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. Alternative modes are achieved by multiple versions of a predicate. This approach is quite common [2, 3, 6] and is also taken in Mercury [18], where these versions are generated by the compiler. However, generating multiple versions implies code duplication and hence there is a loss of generality in assuming single-moded predicates.

Naish uses examples such as `permute` where under the above assumption, there is no reason for using delay declarations in the first place. Therefore, his discussion on delay declarations lacks motivation.

Lüttringhaus-Kappel [12] proposes a method to generate control automatically to ensure termination, giving practical evidence that control declarations were generated for many programs. The method assumes arbitrary delay-respecting derivations and hence does not work for programs where termination depends on derivations being left-based.

Marchiori and **Teusink** [13] base termination on norms and the *covering* relation between subqueries of a query. This is loosely related to well-typedness. However, their results are not comparable to ours because they assume a *local selection rule*, that is a rule which always selects an atom which was introduced in the most recent step. We are not aware of an existing language that uses a local selection rule. The authors state that programs that do not use *speculative bindings* deserve further investigation, and that they expect any method for proving termination with *full* coroutining either to be very complex, or very restrictive in its applications.

We have previously presented a method of proving termination which is based on identifying the so-called *robust* predicates, for which the textual position of an atom using this predicate is irrelevant [16]. That method is quite complex and not as intuitive as the methods described here. In practice, it is probably more widely applicable than the method of this paper, but we have seen in Examples 5.3 and 5.5 that there are programs for which the methods of this paper work and the method of [16] does not.

7 Conclusion and Future Work

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

To prevent errors due to built-ins, we have provided two methods that show that `block` declarations, as opposed to more complicated delay constructs [3], are often sufficient. In addition, in many cases, particularly for the arithmetic predicates where the arguments must be atomic, we show that there is no need for any form of delay declarations. Whether one of these methods or even both are applicable depends on the program. Finding a more uniform approach to this problem could be a topic of future work.

For proving termination, we have presented two methods based on not *using* and not *making* speculative bindings, respectively. Proposition 5.1 states that for well typed programs meeting a simple condition on selectability, any left-based derivation of a well typed query is an LD-derivation. Theorem 5.2 states that for non-speculative programs, termination is independent of a particular selection rule. We have noted that in some cases such as Ex. 5.1, in one mode, the first method applies, and in the second mode, the other method applies. These methods are appealing because they are simple and formalise heuristics that have been known for some time [14, 15].

It is an ongoing discussion whether it is reasonable to assume predicates that run in several modes [7]. We believe that a formalism dealing with delay declarations should at least *allow* for multiple modes, while not excluding other applications such as the test-and-generate paradigm (coroutining). Our results are still applicable and relevant for programs with just a single mode. For example, our results about built-ins could be used to show absence of errors in a program for the n-queens problem, which is a well-known example for the test-and-generate paradigm [17]. Also, Thm. 5.2 is useful for single-moded programs. However, we admit that Prop. 5.1 lacks motivation without the assumption of multiple modes.

We envisage that an implementation of our work should take the form of a program development tool which would help a programmer to verify the various mode and type conditions introduced in this paper. It should also support transforming a program so that the conditions are met, which would mainly be limited to reordering atoms in clause bodies. Further research concerning the design and development of such a tool is ongoing work.

Acknowledgements

We thank the anonymous referees and all colleagues who commented on our work during the LOPSTR '98 workshop. Jan–Georg Smaus was supported by EPSRC Grant No. GR/K79635.

References

1. K. R. Apt. *From Logic Programming to Prolog.* 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. A. Bossi and N. Cocco. Successes in logic programs. In P. Flener, editor, *Proceedings of the 8th International Workshop on Logic Program Synthesis and Transformation*, LNCS. Springer-Verlag, 1999.
5. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.
6. S. Etalle, A. Bossi, and N. Cocco. Well-terminating programs. *Journal of Logic Programming*, 1998. Accepted for publication.
7. P. M. Hill, editor. *ALP Newsletter*, `http://www-lp.doc.ic.ac.uk/alp/`, February 1998. Pages 17,18.
8. 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`.
9. International Organization for Standardization. *The ISO Prolog Standard*, 1995. `http://www.logic-programming.org/prolog_std.html`.
10. M. R. K. Krishna Rao, D. Kapur, and R. K. Shyamasundar. A transformational methodology for proving termination of logic programs. In *Proceedings of the 5th Conference for Computer Science Logic*, LNCS, pages 213–226. Springer-Verlag, 1991.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
12. S. Lüttringhaus-Kappel. Control generation for logic programs. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 478–495. MIT Press, 1993.
13. E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. W. Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 447–461. MIT Press, 1995.
14. L. Naish. Automatic control of logic programs. *Journal of Logic Programming*, 2(3):167–183, 1985.
15. L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, University of Melbourne, 1992.
16. J.-G. Smaus, P. M. Hill, and A. M. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamidessi, editor, *Proceedings of the 10th Symposium on Programming Language Implementations and Logic Programming*, LNCS. Springer-Verlag, 1998.
17. J.-G. Smaus, P. M. Hill, and A. M. King. Verification of logic programs with `block` declarations running in several modes. Technical Report 7-98, University of Kent at Canterbury, Canterbury, CT2 7NF, United Kingdom, July 1998.
18. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, November 1996.