Termination of Logic Programs for Various Dynamic Selection Rules

Jan–Georg Smaus

Institut für Informatik
Universität Freiburg
Georges-Köhler-Allee 52, 79110 Freiburg im Breisgau, Germany

smaus@informatik.uni-freiburg.de

30th December 2003

Technical Report No. 191

Abstract

We study termination of logic programs with dynamic scheduling, as it can be realised using delay declarations. Following previous work, our minimum assumption is that derivations are *input-consuming*, a notion introduced to define dynamic scheduling in an abstract way. Since this minimum assumption is sometimes insufficient to ensure termination, we consider here various *additional* assumptions on the permissible derivations. In one dimension, we consider derivations parametrised by any property that the selected atoms must have, e.g. being ground in the input positions. In another dimension, we consider both *local* and non-local derivations. In all cases, we give sufficient criteria for termination. The dimensions can freely be combined, yielding the most comprehensive approach so far to termination of logic programs with dynamic scheduling.

1 Introduction

Termination of logic programs has been widely studied for the LD selection rule, i.e., derivations where the leftmost atom in a query is selected [1, 4, 7, 8, 9, 14]. This rule is adequate for many applications, but there are situations, e.g., in the context of parallel executions or the test-and-generate paradigm, that require *dynamic scheduling*, i.e., some mechanism to determine at runtime which atom is selected. Dynamic scheduling can be realised by *delay declarations* [10, 19], specifying that an atom must be instantiated to a certain degree to be selected.

Termination of logic programs with dynamic scheduling has been studied for about a decade [3, 5, 6, 11, 12, 13, 15, 17, 18]; the research started with observations of some surprisingly complex (non-)termination behaviour of simple programs such as `APPEND` or `PERMUTE` with delay declarations [15]. In our own research [6, 17, 18], we found that *modes* (input and output), while arguably compromising the "pure logical" view of logic programming, are the key to understanding this behaviour and achieving or verifying termination. We have proposed *input-consuming derivations* (where in each resolution step, the input arguments of the selected atom do not become instantiated) as a reasonable minimum assumption about the selection rule, which abstracts away from the technical details of delay declarations.

In this report, we study termination of logic programs for input-consuming derivations with various *additional* assumptions about the selection rule, e.g. saying that the selected atom must be ground in its input positions. Some authors have considered termination under such strong assumptions [11, 12, 13, 15, 18], partly because certain programs do not terminate for input-consuming derivations, but also because termination for input-consuming derivations is so hard to show: termination proofs usually build on *level mappings*, which measure the size of an atom; an atom is *bounded* if its level is invariant under instantiation. Now, the usual reasoning that there is a decrease in level mapping between a clause head and the clause body atoms does not readily apply to derivations where selected atoms are not necessarily bounded.

After intensive studies of the semantics [5], and restricting to a class of programs that is well-behaved wrt. the modes, we now have a sufficient and complete criterion for termination of input-consuming derivations [6]. The key concept of that approach is a special notion of *model*, bottom-up computable by a variant of the well-known T_P-operator. The notion reflects the answer substitutions computed by input-consuming derivations. We build on this work here.

We consider additional assumptions in two dimensions. *Certain* additional assumptions about derivations, e.g. the one above, can be formulated in terms of the selected atoms alone. We do this abstractly by saying that each selected atom must have a property $\mathcal{P}$. All that is required is that $\mathcal{P}$ is closed under instantiation. It turns out that the approach of [6] can be parametrised by such a $\mathcal{P}$ quite easily, yielding a sufficient criterion for termination.

Other additional assumptions about derivations cannot be expressed in terms of the selected atoms alone. We consider here one such assumption, that of derivations being *local*, meaning that in a resolution step, the most recently introduced atoms must be resolved first [12]. Clearly, this is not a property of a single atom, but of atoms in the context of a derivation. To deal with local selection rules, we modify the model notion of [6] so that it reflects the substitutions computed by local derivations. Based on such models, we can give a criterion for termination of *local* derivations, parametrised by a $\mathcal{P}$ as before.

We thus present a framework for showing termination of logic programs with dynamic scheduling. The initial motivation for this work was our impression that while stronger assumptions than that of input-consuming derivations are sometimes required, *locality* is too strong. The five main insights we can now confirm, or have gained, are the following:

1. There is a class of recursive clauses, using a natural pattern of programming, that narrowly misses the property of termination for input-consuming derivations. Put simply, theses clauses have the form $p(\mathtt{X}) \leftarrow q(\mathtt{X},\mathtt{Y}),\ p(\mathtt{Y})$, where the mode is $p(I),\ q(I,O)$. Due to the variable in the head, it follows that an atom using p may always be selected, and hence we have non-termination. Sometimes, just requiring the argument of p to be at least non-variable is enough to ensure termination. This can be captured in our approach by setting (the relevant subset of) $\mathcal{P}$ to $\{p(t) \mid t \text{ is non-variable}\}$ (Ex. 4.7).

2. Some programs require for termination that selected atoms must be bounded wrt. a level mapping $|.|$. This is related to *speculative output bindings*, and the `PERMUTE` program is the standard example [15]. This can be captured in our approach by setting $\mathcal{P}$ to the set of bounded atoms (Ex. 4.4).

3. For some programs it is useful to consider "hybrid" selection rules, where differently strong assumptions are made for different predicates. For example, one might require ground input positions for some predicates but no additional assumptions for other predicates. This can be captured by setting $\mathcal{P}$ accordingly (Ex. 4.8).

4. A method for showing termination of programs with delay declarations has been proposed in [12], assuming local selection rules. In our opinion, this assumption is unsatisfactory. No implementation of local selection rules is mentioned. Local selection rules do not permit any coroutining. But most importantly, while "the class of local selection rules [...] supports simple tools for proving termination" [12], in practice, it does not seem to make programs terminate that would not terminate otherwise. In fact, we can show termination for `PERMUTE` without requiring local selection rules (Ex. 4.4).

5. In spite of point 4, there are programs that crucially rely on the assumption of local selection rules for termination. We are only aware of artificial examples, but our treatment of local selection rules helps to understand the role this assumption plays in

proving termination and why this assumption is not required for more realistic examples (Ex. 5.8).

The rest of this report is organised as follows. The next section gives some preliminaries. In Sec. 3, we adapt the semantics approach of [6] to derivations parametrised by a property of the selected atoms. In Sec. 4, we study termination for such derivations. In Sec. 5, we adapt our whole approach to local selection rules. In Sec. 6, we conclude.

2 Preliminaries

We assume familiarity with the basic notions and results of logic programming [1]. For $m, n \in \mathbb{N}_0$, $m < n$, the set $\{m, \ldots, n\}$ is denoted by $[m..n]$. For any kind of object that we commonly denote with a certain letter, we use the same letter in boldface to denote a finite *sequence* of such objects [1].

We denote by $Term$ and $Atom$ the set of terms and atoms of the language in which the programs and queries in question are written. We denote by $Rel(A)$ the predicate symbol of an atom A. The arity n of a predicate symbol p is indicated by writing p/n. We use typewriter font for logical variables, e.g. $\mathtt{X}, \mathtt{Ys}$, and lower case letters for arbitrary terms, e.g. t, s, xs.

For any syntactic object o, we denote by $Vars(o)$ the set of variables occurring in o. A syntactic object is **linear** if every variable occurs in it at most once.

A **substitution** is a finite mapping from variables to terms. The domain (resp., set of variables in the range) of a substitution σ is denoted as $Dom(\sigma)$ (resp., $Ran(\sigma)$). We denote by $\sigma_{|o}$ the restriction of a substitution σ to $Vars(o)$. The result of the application of a substitution σ to a term t is called an **instance** of t and it is denoted by $t\sigma$. We say t is a **variant** of t' if t and t' are instances of each other. A substitution σ is a **unifier** of terms t and t' if $t\sigma = t'\sigma$. We denote by $mgu(t, t')$ any *most general unifier* (*mgu*) of t and t'.

A **query** is a finite sequence of atoms $A_1, \ldots, A_m$, denoted $\Box$ when $m = 0$. A **clause** is a formula $H \leftarrow \mathbf{B}$ where H is an atom (the **head**) and $\mathbf{B}$ is a query (the **body**). $H \leftarrow \Box$ is simply written $H \leftarrow$. A **program** is a finite set of clauses. We denote atoms by H, A, B, C, D, E, queries by $\mathbf{Q}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}$, clauses by c, d and programs by P. We often suppress reference to P.

If $\mathbf{A}, B, \mathbf{C}$ is a query and $c = H \leftarrow \mathbf{B}$ is a fresh variant of a clause, and B and H unify with mgu σ, then $(\mathbf{A}, \mathbf{B}, \mathbf{C})\sigma$ is called a **resolvent of** $\mathbf{A}, B, \mathbf{C}$ **and** $H \leftarrow \mathbf{B}$ **with selected atom** B **and mgu** σ. We call $\mathbf{A}, B, \mathbf{C} \overset{\sigma}{\Longrightarrow}_c (\mathbf{A}, \mathbf{B}, \mathbf{C})\sigma$ a **derivation step**, in short: **step**. If c is irrelevant then we drop the reference. A sequence $\delta = \mathbf{Q}_0 \overset{\sigma_1}{\Longrightarrow}_{c_1} \mathbf{Q}_1 \overset{\sigma_2}{\Longrightarrow}_{c_2} \cdots$ is called a **derivation of** $P \cup \{\mathbf{Q}_0\}$.

If $\delta = \mathbf{Q}_0 \overset{\sigma_1}{\Longrightarrow}_{c_1} \cdots \overset{\sigma_n}{\Longrightarrow}_{c_n} \mathbf{Q}_n$ is a finite derivation, we also denote it as $\delta = \mathbf{Q}_0 \overset{\sigma}{\longrightarrow} \mathbf{Q}_n$ where $\sigma = \sigma_1 \cdots \sigma_n$. We call $len(\delta) = n$ the **length** of δ.

Let $\mathbf{A}, B, \mathbf{C} \overset{\sigma}{\Longrightarrow} (\mathbf{A}, \mathbf{B}, \mathbf{C})\sigma$ be a step. We say each atom in $\mathbf{B}\sigma$ is a **direct descendant of** B, and for each atom D in $(\mathbf{A}, \mathbf{C})$, $D\sigma$ is a **direct descendant** of D. We say that D is a **descendant** of E if the pair (D, E) is in the reflexive, transitive closure of the relation *is a direct descendant of*.

Consider a derivation $\mathbf{Q}_0 \overset{\sigma_1}{\Longrightarrow} \cdots \overset{\sigma_i}{\Longrightarrow} \mathbf{Q}_i \cdots \overset{\sigma_j}{\Longrightarrow} \mathbf{Q}_j \overset{\sigma_{j+1}}{\Longrightarrow} \mathbf{Q}_{j+1} \cdots$. We say that $\mathbf{Q}_j \overset{\sigma_{j+1}}{\Longrightarrow} \mathbf{Q}_{j+1} \cdots$ is a **D-step** if $\mathbf{D}$ is a subquery of $\mathbf{Q}_i$ and the selected atom in $\mathbf{Q}_j$ is a descendant of an atom in $\mathbf{D}$.

A derivation **proceeds left-to-right** if whenever an atom B is selected in a query $\mathbf{A}, B, \mathbf{C}$, then no $\mathbf{A}$-step is performed afterwards.

2.1 Moded Programs

For a predicate p/n, a **mode** is an atom $p(m_1, \ldots, m_n)$, where $m_i \in \{I, O\}$ for $i \in [1..n]$. Positions with I (O) are called **input** (**output**) positions of p. To simplify the notation, an atom $p(\mathbf{s}, \mathbf{t})$ means: $\mathbf{s}$ is the vector of terms filling in the input positions, and $\mathbf{t}$ is the vector of terms filling in the output positions.

Several notions of "modedness" have been proposed, e.g. *nicely-modedness* and *well-modedness* [1]. We assume here *simply* moded programs [2], a special case of nicely moded programs. Most practical programs are simply moded [6].

Definition 2.1 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 **simply moded** if $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a linear vector of variables and for all $i \in [1..n]$

$$\mathit{Vars}(\mathbf{t}_i) \cap \mathit{Vars}(\mathbf{t}_0) = \emptyset \quad \text{and} \quad \mathit{Vars}(\mathbf{t}_i) \cap \bigcup_{j=1}^{i} \mathit{Vars}(\mathbf{s}_j) = \emptyset.$$

A query $\mathbf{B}$ is **simply moded** if the clause $\mathtt{dummy} \leftarrow \mathbf{B}$ is simply moded. A program is **simply moded** if all of its clauses are simply moded.

Thus, a clause is simply moded if the output positions of body atoms are filled in by distinct variables, and every variable occurring in an output position of a body atom does not occur in an earlier input position.

2.2 Norms and Level Mappings

Proofs of termination usually rely on the notions of *norm* and *level mapping* for measuring the size of terms and atoms. These concepts were originally defined for ground objects [9], but here we define them for arbitrary objects (in [16], we call such norms and level mappings *generalised*). To show termination of moded programs, it is natural to use *moded* level mappings, where the level of an atom depends only on its input positions [6].

Definition 2.2 A **norm** is a function $|.| : \mathit{Term} \rightarrow \mathbb{N}_0$, and a **level mapping** is a function $|.| : \mathit{Atom} \rightarrow \mathbb{N}_0$, both invariant under renaming. A **moded level mapping** is a level mapping where for any $\mathbf{s}$, $\mathbf{t}$ and $\mathbf{u}$, $|p(\mathbf{s}, \mathbf{t})| = |p(\mathbf{s}, \mathbf{u})|$.

An atom A is **bounded** wrt. the level mapping $|.|$ if there exists $k \in \mathbb{N}$ such that for every substitution σ, we have $k > |A\sigma|$.

Note that for a non-ground atom A, if we have $|A| = k$, then this should not be confused with A being bounded by k.

One commonly used norm is the term size norm, defined as

$$\begin{array}{rcll} |f(t_1, \ldots, t_n)| & = & 1 + |t_1| + \ldots + |t_n| & \text{if } n > 0, \\ |t| & = & 0 & \text{if } t \text{ constant/variable.} \end{array}$$

Another widely used norm is the list-length function, defined as

$$\begin{array}{rcll} |[t|ts]| & = & 1 + |ts|, & \\ |f(\ldots)| & = & 0 & \text{if } f \neq [\,.\,|\,.\,]. \end{array}$$

For a nil-terminated list $[t_1, \ldots, t_n]$, the list-length is n.

2.3 Selection Rules in the Literature

A selection rule is some rule stating which atom in a query may be selected in each step. We do not give any formal definition here; instead we define various kinds of derivations and state our formal results in terms of those.

The notion of *input-consuming derivation* was introduced in [17] as formalism for describing dynamic scheduling in an abstract way.

Definition 2.3 A step $\mathbf{A}, p(\mathbf{s}, \mathbf{t}), \mathbf{C} \stackrel{\sigma}{\Longrightarrow} (\mathbf{A}, \mathbf{B}, \mathbf{C})\sigma$ is **input-consuming** if $\mathbf{s}\sigma = \mathbf{s}$. A derivation is **input-consuming** if all its steps are input-consuming.

Local derivations were treated in [12]. Consider a query, containing atoms A and B, in a derivation ξ. Then A is **introduced more recently** than B if the step introducing A comes after the step introducing B, in ξ.

Definition 2.4 A derivation is **local** if in each step, there is no more recently introduced atom in the current query than the selected atom.

The situation that a derivation ends in a query where no atom can be selected is called **deadlock**. It is a form of termination.

3 Input-Consuming $\mathcal{P}$-Derivations

Definition 3.1 A $\mathcal{P}$**-derivation** is a derivation where each selected atom is in $\mathcal{P}$, where $\mathcal{P}$ is a set of atoms closed under instantiation.

This abstract way of defining derivations is an original notion of this report. Of course, a $\mathcal{P}$-derivation can be qualified further by saying *input-consuming* $\mathcal{P}$-derivation etc.

We now generalise some results of [5, 6] to $\mathcal{P}$-derivations. We first cite a fundamental proposition about input-consuming steps [5, Remark 9].

Proposition 3.2 If P and $\mathbf{A}, B, \mathbf{C}$ are a simply moded program and query and $\mathbf{A}, B, \mathbf{C} \stackrel{\sigma}{\Longrightarrow} (\mathbf{A}, \mathbf{B}, \mathbf{C})\sigma$ is an input-consuming step in P, then $\mathbf{A}\sigma = \mathbf{A}$.

The Left-Switching Lemma states that steps in an input-consuming derivation can always be switched so that atoms more to the left are selected earlier [5, Lemma 13]. Here we give a generalisation to $\mathcal{P}$-derivations.

Lemma 3.3 Let the program P and the query $\mathbf{Q}$ be simply moded. Let δ be an input-consuming $\mathcal{P}$-derivation of $P \cup \{\mathbf{Q}\}$ of the form (the selected atom is underlined for clarity):

$$\mathbf{Q} \longrightarrow \mathbf{A}, B, \mathbf{C}, \underline{D}, \mathbf{E} \stackrel{\theta}{\Longrightarrow}_c (\mathbf{A}, \underline{B}, \mathbf{C}, \mathbf{D}, \mathbf{E})\theta \stackrel{\sigma}{\Longrightarrow}_d (\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E})\theta\sigma \longrightarrow \dots$$

where $c = \tilde{D} \leftarrow \mathbf{D}$ and $d = \tilde{B} \leftarrow \mathbf{B}$ are clauses from P.

Then, there exists an input-consuming $\mathcal{P}$-derivation δ' of the form

$$\mathbf{Q} \longrightarrow \mathbf{A}, \underline{B}, \mathbf{C}, D, \mathbf{E} \stackrel{\sigma'}{\Longrightarrow}_d (\mathbf{A}, \mathbf{B}, \mathbf{C}, \underline{D}, \mathbf{E})\sigma' \stackrel{\theta'}{\Longrightarrow}_c (\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E})\sigma'\theta' \longrightarrow \dots,$$

where $\theta\sigma = \sigma'\theta'$ and

- δ and δ' coincide up to the resolvent $\mathbf{A}, B, \mathbf{C}, D, \mathbf{E}$,
- $B = B\theta$,
- δ and δ' coincide after the resolvent $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E})\theta\sigma$.

Proof. The statement $B = B\theta$ follows from Prop. 3.2. Moreover, $D\sigma'$ is at least as instantiated as D. Since $\mathcal{P}$ is a set closed under instantiation and δ is a $\mathcal{P}$-derivation, it thus follows that δ' is a $\mathcal{P}$-derivation. Apart from that, the statement is identical to [5, Lemma 13] (but we use a more concise notation). □

The following statement, originally stated in [6, Cor. 2.12] as a corollary of [5, Lemma 10], is also generalised here to $\mathcal{P}$-derivations.

Corollary 3.4 Let the program P and the query $\mathbf{A}, \mathbf{B}$ be simply moded. Suppose that $\delta = \mathbf{A}, \mathbf{B} \stackrel{\sigma}{\longrightarrow} \mathbf{C}$ is an input-consuming $\mathcal{P}$-derivation of $P \cup \{(\mathbf{A}, \mathbf{B})\}$.

Then there exist $\mathbf{C}_1$ and $\mathbf{C}_2$ and an input-consuming $\mathcal{P}$-derivation δ' that proceeds left-to-right of the form

$$\delta' = \mathbf{A}, \mathbf{B} \stackrel{\sigma_1}{\longrightarrow} \mathbf{C}_1, \mathbf{B}\sigma_1 \stackrel{\sigma_2}{\longrightarrow} \mathbf{C}_1, \mathbf{C}_2$$

such that $len(\delta) = len(\delta')$ and $\mathbf{C} = \mathbf{C}_1, \mathbf{C}_2$ and $\sigma = \sigma_1\sigma_2$, all the $\mathbf{A}$-steps are performed in the prefix $\mathbf{A}, \mathbf{B} \stackrel{\sigma_1}{\longrightarrow} \mathbf{C}_1, \mathbf{B}\sigma_1$, all the $\mathbf{B}$-steps are performed in the suffix $\mathbf{C}_1, \mathbf{B}\sigma_1 \stackrel{\sigma_2}{\longrightarrow} \mathbf{C}_1, \mathbf{C}_2$ and $\mathbf{C}_1\sigma_2 = \mathbf{C}_1$.

```
permute([],[]).                     insert(Xs,X,[X|Xs]).
permute([X|Xs],Ys) ←                insert([Y|Xs],X,[Y|Zs]) ←
   insert(Zs,X,Ys),                    insert(Xs,X,Zs), call_me_late.
   permute(Xs,Zs).                  call_me_late.
```

Figure 1: PERMUTE

We now define *simply-local* substitutions, which reflect the way simply moded clauses become instantiated in input-consuming derivations [6]. Given a clause

$$c = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n),$$

first $\mathbf{t}_0$ becomes instantiated, and the range of that substitution contains only variables from outside of c. Then, by resolving $p_1(\mathbf{s}_1, \mathbf{t}_1)$, $\mathbf{t}_1$ becomes instantiated, and the range of that substitution contains variables from outside of c and from $\mathbf{s}_1$. Continuing in the same way, finally, $\mathbf{t}_n$ becomes instantiated, and the range of that substitution contains variables from outside of c and from $\mathbf{s}_1 \ldots \mathbf{s}_n$.

Definition 3.5 The substitution σ is **simply-local** wrt. the clause $c = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ if there exist substitutions $\sigma_0, \sigma_1 \ldots, \sigma_n$ and disjoint sets of variables $V_0, V_1, \ldots, V_n$, fresh wrt. c, such that $\sigma = \sigma_0\sigma_1 \cdots \sigma_n$, where for $i \in [0..n]$,

- $Dom(\sigma_i) \subseteq Vars(\mathbf{t}_i)$,
- $Ran(\sigma_i) \subseteq Vars(\mathbf{s}_i\sigma_0\sigma_1 \cdots \sigma_{i-1}) \cup V_i$.[1]

σ is **simply-local** wrt. a query $\mathbf{B}$ if σ is simply-local wrt. the clause $\mathtt{dummy} \leftarrow \mathbf{B}$.

Note that in the case of a simply-local substitution wrt. a query, σ_0 is the empty substitution. Note also that if $\mathbf{A}, B, \mathbf{C} \Longrightarrow (\mathbf{A}, \mathbf{B}, \mathbf{C})\sigma$ is an input-consuming step using clause $c = H \leftarrow \mathbf{B}$, then $\sigma_{|H}$ is simply-local wrt. the clause $H \leftarrow$ and $\sigma_{|B}$ is simply-local wrt. the atom[2] B [6]. In fact, we may assume that all mgu's employed in an input-consuming derivation for a simply-moded program and query are simply-local [6, Lemma 3.8].

Example 3.6 Consider the recursive clause for `insert` in Fig. 1, with mode $\mathtt{insert}(O, O, I)$. The substitution $\sigma = \{\mathtt{Y/V}, \mathtt{Zs/[W]}, \mathtt{Xs/[]}, \mathtt{X/W}\}$ is simply-local wrt. it: let $\sigma_0 = \{\mathtt{Y/V}, \mathtt{Zs/[W]}\}$, $\sigma_1 = \{\mathtt{X/W}, \mathtt{Xs/[]}\}$, and $\sigma_2 = \emptyset$; then $Dom(\sigma_0) \subseteq \{\mathtt{Y}, \mathtt{Zs}\}$, $Ran(\sigma_0) \subseteq V_0$ where $V_0 = \{\mathtt{V}, \mathtt{W}\}$, $Dom(\sigma_1) \subseteq \{\mathtt{Xs}, \mathtt{X}\}$, and $Ran(\sigma_1) \subseteq Vars(\mathtt{Zs}\,\sigma_0)$.

Based on simply-local substitutions, we define a particular notion of model, which reflects the substitutions that can be computed by input-consuming derivations. According to this notion, a model is a set of *not necessarily ground* atoms.

Definition 3.7 Let $M \subseteq Atom$. We say that M is a **simply-local model** of $c = H \leftarrow B_1, \ldots, B_n$ if for every substitution σ simply-local wrt. c,

$$\text{if } B_1\sigma, \ldots, B_n\sigma \in M \text{ then } H\sigma \in M. \tag{1}$$

M is a **simply-local model** of a program P if M is a simply-local model of each clause of P.

We denote the set of all simply moded atoms[2] for the program P by SM_P.

Least simply-local models, possibly containing SM_P, can be computed by a variant of the well-known T_P-operator [6]: Given a program P and $I \subseteq Atom$, we define

$$\begin{aligned} T_P^{sl}(I) &= \{H\sigma \mid \; \exists\, c = H \leftarrow B_1, \ldots, B_n \text{ variant of a clause in } P, \\ &\qquad\qquad \sigma \text{ is simply-local wrt. } c,\; B_1\sigma, \ldots, B_n\sigma \in I\}, \\ T_P^{SL}(I) &= I \cup T_P^{sl}(I). \end{aligned}$$

We denote the least simply-local model of P containing SM_P by PM_P^{SL}.

[1] Note that $\mathbf{s}_0$ is undefined. By abuse of notation, $Vars(\mathbf{s}_0 \ldots) = \emptyset$.
[2] We sometimes say "atom" for "query containing only one atom".

Example 3.8 Recall Ex. 3.6. SM_P consists of all atoms of the form $\mathtt{insert}(\mathtt{Us}, \mathtt{U}, vs)$ where $\mathtt{Us}, \mathtt{U} \notin \mathit{Vars}(vs)$. To construct PM_P^{SL}, we iterate T_P^{SL} starting from any atom in SM_P (the resulting atoms are written on the l.h.s. below) and the fact clause (r.h.s.). Each line below corresponds to one iteration of T_P^{SL}. We have $PM_P^{SL} =$

$$\begin{array}{lll}\{ & \mathtt{insert}(\mathtt{Us}, \mathtt{U}, vs), & \\ & \mathtt{insert}([y_1|\mathtt{Us}], \mathtt{U}, [y_1|vs]), & \mathtt{insert}(xs_1, x_1, [x_1|xs_1]), \\ & \mathtt{insert}([y_2, y_1|\mathtt{Us}], \mathtt{U}, [y_2, y_1|vs]), & \mathtt{insert}([y_1|xs_1], x_1, [y_1, x_1|xs_1]), \\ & \dots & \dots \\ & \mid vs, xs_1, x_1, y_1, y_2, \dots \text{ arbitrary where } \mathtt{Us}, \mathtt{U} \notin \mathit{Vars}(vs)\}. & \end{array} \tag{2}$$

Observe the variable occurrences of $\mathtt{U}, \mathtt{Us}$ in the atoms on the l.h.s. In Ex. 4.4, we will see the importance of such variable occurrences.

If we are very fussy, the $y_1, y_2, \dots$ are not completely arbitrary: each y_i must not contain any variables occurring in the clause variant used by T_P^{SL} to introduce $\mathtt{insert}([y_i, \dots, y_1|\mathtt{Us}], \mathtt{U}, [y_i, \dots, y_1|vs])$; speaking in terms of Def. 3.5, $\mathit{Vars}(y_i) \subseteq V_0$. But since one can assume that each application of T_P^{SL} always uses a fresh clause copy, there is no need to worry about this point.

The following lemma states that simply-local models indeed capture the partial answer substitutions that can be computed by input-consuming derivations. It is obtained by combining [6, Prop. 4.3 and Lemma 4.4].

Lemma 3.9 Let the program P and the query $\mathbf{A}$ be simply moded and $I \subseteq SM_P$. If there exists an input-consuming derivation $\delta = \mathbf{A} \xrightarrow{\sigma} \mathbf{C}$ with $\mathbf{C} \subseteq I$, then $\mathbf{A}\sigma \subseteq M$ for any simply-local model M of P containing I.

4 Termination without Requiring Local Selection Rules

The following concept is adopted from Apt [1].

Definition 4.1 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 $p \simeq q$ if $p \sqsupseteq q$ and $q \sqsupseteq p$.

We extend this notation to *atoms*, e.g. $p(\mathbf{s}, \mathbf{t}) \simeq q(\mathbf{u}, \mathbf{v})$ if $p \simeq q$.

Definition 4.2 A program is **input $\mathcal{P}$-terminating** if all input-consuming $\mathcal{P}$-derivations starting in a simply-moded query are finite.

We now give a sufficient criterion for input $\mathcal{P}$-termination.

Definition 4.3 Let P be a simply moded program, $|.|$ a moded level mapping and M a simply-local model of P containing SM_P. A clause $A \leftarrow B_1, \dots, B_n$ is **simply $\mathcal{P}$-acceptable by $|.|$ and M**[3] if for every substitution σ simply-local wrt. it, for all $i \in [1..n]$,

$$B_1\sigma, \dots, B_{i-1}\sigma \in M \text{ and } A \simeq B_i \text{ and } B_i\sigma \in \mathcal{P} \quad \text{imply} \quad |A\sigma| > |B_i\sigma|.$$

The program P is **simply $\mathcal{P}$-acceptable by $|.|$ and M** if each clause of P is simply $\mathcal{P}$-acceptable by $|.|$ and M.

[3] This terminology should be regarded as provisional. If the author succeeds in finding a sufficient and *necessary* condition for input $\mathcal{P}$-termination different from the one given here, then it will probably be reasonable to call this condition "simply $\mathcal{P}$-acceptable" rather than inventing a new name.

The only difference to the definition of a *simply acceptable* clause [6] is in the condition $B_i\sigma \in \mathcal{P}$. The remarkable observation here is that simply-local models capture all possible input-consuming derivations of a simply-moded query, including the ones where we impose an additional condition $\mathcal{P}$. Hence this small modification gives us a sufficient criterion for input $\mathcal{P}$-termination.

Note that a decrease between the head and body atoms must be shown only for the atoms where $A \simeq B_i$. The idea is that termination is shown incrementally, so we assume that for the B_i where $A \sqsupset B_i$, termination has been shown already. One can go further and explicitly give *modular* termination results [6, 9], but this is a side issue for us.

The following is the standard example of a program that requires boundedness as additional condition on selected atoms (see point 2 in the introduction).

Example 4.4 Consider `PERMUTE` (Fig. 1) with mode $\mathtt{permute}(O, I)$, $\mathtt{insert}(O, O, I)$. Recall Ex. 3.8. As norm we take the list-length function, and we define the level mapping as $|\mathtt{permute}(xs, zs)| = |zs|$ and $|\mathtt{insert}(zs, x, xs)| = |xs|$. Now for all atoms $\mathtt{insert}(zs, x, ys) \in PM_P^{SL}$, we have $|ys| \geq |zs|$; for the ones on the r.h.s. even $|ys| > |zs|$. Let $\mathcal{P}$ be the set of bounded atoms wrt. $|.|$.

Now let us look at the recursive clause for `permute`. We verify that the second body atom fulfils the requirement of Def. 4.3, where M is PM_P^{SL}. So we have to consider all simply-local substitutions σ such that $\mathtt{insert}(\mathtt{Zs}, \mathtt{X}, \mathtt{Ys})\sigma \in PM_P^{SL}$. For the atoms on the l.h.s. in (2), this means that

$$\sigma \supseteq \{\mathtt{Ys}/[y_n, \ldots, y_1|vs], \mathtt{Zs}/[y_n, \ldots, y_1|\mathtt{Us}], \mathtt{X}/\mathtt{U}\} \qquad (n \geq 0).$$

Clearly, $\mathtt{permute}(\mathtt{Xs}, \mathtt{Zs})\sigma \notin \mathcal{P}$, and hence no proof obligation arises. For the atoms on the r.h.s. in (2), this means that

$$\sigma \supseteq \{\mathtt{Ys}/[y_n, \ldots, y_1, x_1|xs_1], \mathtt{Zs}/[y_n, \ldots, y_1|xs_1], \mathtt{X}/x_1\} \qquad (n \geq 0).$$

But then $|\mathtt{permute}([\mathtt{X}|\mathtt{Xs}], \mathtt{Ys})\sigma| > |\mathtt{permute}(\mathtt{Xs}, \mathtt{Zs})\sigma|$.

The other clauses are trivial to check, and so `PERMUTE` is simply $\mathcal{P}$-acceptable.

The atom `call_me_late` only serves the purpose of allowing for non-local derivations, to emphasise that locality is not needed for termination (see point 4 in Sec. 1). Without this atom, all $\mathcal{P}$-derivations would automatically be local.

In [6], we find a result stating that simply acceptable programs are input terminating. By a slight modification of that result, we will show that simply $\mathcal{P}$-acceptable programs are input $\mathcal{P}$-terminating.

The following statement allows us to restrict our attention to atomic queries. Originally given in [5, Cor. 23], it generalises easily to $\mathcal{P}$-derivations because it essentially rests on Cor. 3.4, which holds for $\mathcal{P}$-derivations.

Corollary 4.5 Let P be a simply moded program. P is input $\mathcal{P}$-terminating if and only if for each simply moded atomic query A all input-consuming $\mathcal{P}$-derivations of $P \cup \{A\}$ are finite.

Theorem 4.6 Let P be a simply moded program. Let M be a simply-local model of P containing SM_P. Suppose that P is simply $\mathcal{P}$-acceptable wrt. M and a moded level mapping $|.|$. Then P is input $\mathcal{P}$-terminating.

Proof. For each predicate p, define $dep(p)$ to be the number of predicates p depends on: $dep(p) = \#\{q | p \sqsupset q\}$. Clearly, $dep(p)$ is always finite. Moreover, if $p \simeq q$ then $dep(p) = dep(q)$, and if $p \sqsupset q$ then $dep(p) > dep(q)$.

By Cor. 4.5, it is sufficient to prove that for any simply moded atomic query A, all input-consuming $\mathcal{P}$-derivations of $P \cup \{A\}$ are finite.

For the purpose of deriving a contradiction, assume that δ is an infinite input-consuming $\mathcal{P}$-derivation of $P \cup \{A\}$. Then

$$\delta = A \stackrel{\sigma_1}{\Longrightarrow} (B_1, \ldots, B_n)\sigma_1 \longrightarrow \cdots$$

where $c = H \leftarrow B_1, \ldots, B_n$ is the clause used in the first step and $\sigma_1 = mgu(A, H)$. Clearly, $(B_1, \ldots, B_n)\sigma_1$ has an infinite input-consuming $\mathcal{P}$-derivation. By Cor. 3.4, for some $i \in [1..n]$ and for some substitution σ_2,

1. there exists an infinite input-consuming $\mathcal{P}$-derivation of $P \cup \{A\}$ of the form

$$A \overset{\sigma_1}{\Longrightarrow} (B_1, \ldots, B_n)\sigma_1 \xrightarrow{\sigma_2} \mathbf{C}, (B_i, \ldots, B_n)\sigma_1\sigma_2 \longrightarrow \cdots;$$

2. there exists an infinite input-consuming $\mathcal{P}$-derivation of $P \cup \{B_i\sigma_1\sigma_2\}$

both employing only simply-local mgu's ([6, Lemma 3.8]).

Let $\sigma = (\sigma_1\sigma_2)_{|c}$. Using [6, Proposition A.1], one can easily show that σ is simply-local wrt. c. By Lemma 3.9, $(B_1, \ldots, B_{i-1})\sigma \in M$.

Consider the ordering $\succ$ defined by: $\langle m, n\rangle \succ \langle m', n'\rangle$ if either $m > m'$ or $m = m'$ and $n > n'$. We will show that $\langle dep(Rel(A)), |A|\rangle \succ \langle dep(Rel(B_i\sigma)), |B_i\sigma|\rangle$. By an obvious inductive argument, this is a contradiction to point 2.

We distinguish two cases:

1. $Rel(H) \sqsupset Rel(B_i)$,
2. $Rel(H) \simeq Rel(B_i)$.

Consider the derivation $A \xrightarrow{\sigma} \mathbf{C}, (B_i, \ldots, B_n)\sigma$. By [5, Lemma 10][4] and since $|.|$ is a moded level mapping, we have $|A| = |A\sigma| = |H\sigma|$.

In case 1 we have $dep(Rel(A)) = dep(Rel(H\sigma)) > dep(Rel(B_i\sigma))$. Therefore,

$$\langle dep(Rel(A)), |A|\rangle = \langle dep(Rel(H\sigma)), |H\sigma|\rangle \succ \langle dep(Rel(B_i\sigma)), |B_i\sigma|\rangle. \tag{3}$$

In case 2, since P is simply $\mathcal{P}$-acceptable wrt. $|.|$ and M, and σ is simply-local wrt. c, and $(B_1, \ldots, B_{i-1})\sigma \in M$, and moreover $B_i\sigma \in \mathcal{P}$, it follows that $|H\sigma| > |B_i\sigma|$. Hence, (3) holds also in this case. □

We conclude the section with two further examples. The first illustrates point 1 in the Introduction.

Example 4.7 Let `PERMUTE2` be the program obtained from `PERMUTE` by replacing the recursive clause for `insert` by its *most specific variant* [15]:

```
insert([Y|Xs],X,[Y,H|T]) ←insert(Xs,X,[H|T]).
```

Assume $|.|$ and the modes as in Ex. 4.4. As in Ex. 3.8 we have $SM_P = \{\mathtt{insert}(\mathtt{Us}, \mathtt{U}, vs) \mid vs \text{ arbitrary}\}$, but when we start applying T_P^{SL} to the atoms in SM_P, then only the atoms of the form $\mathtt{insert}(\mathtt{Us}, \mathtt{U}, [v|vs'])$ contribute; $\mathtt{insert}(\mathtt{Us}, \mathtt{U}, [])$ does not contribute. We get:

$$\begin{array}{lll} PM_P^{SL} = \{ & \mathtt{insert}(\mathtt{Us}, \mathtt{U}, vs), & \\ & \mathtt{insert}([y_1|\mathtt{Us}], \mathtt{U}, [y_1, v, vs']), & \mathtt{insert}(xs_1, x_1, [x_1|xs_1]), \\ & \mathtt{insert}([y_2, y_1|\mathtt{Us}], \mathtt{U}, [y_2, y_1, v|vs']), & \mathtt{insert}([y_1|xs_1], x_1, [y_1, x_1|xs_1]), \\ & \ldots & \ldots \\ & \mid vs, v, vs', xs_1, x_1, y_1, y_2, \ldots \text{ arbitrary where } \mathtt{Us}, \mathtt{U} \notin \mathit{Vars}(vs, v, vs')\}. & \end{array}$$

We show that the program is simply $\mathcal{P}$-acceptable where $\mathcal{P}$ is the set of atoms that are at least non-variable in their input positions. We have to consider all simply-local substitutions σ such that $\mathtt{insert}(\mathtt{Zs}, \mathtt{X}, \mathtt{Ys})\sigma \in PM_P^{SL}$, and moreover $\mathtt{permute}(\mathtt{Xs}, \mathtt{Zs})\sigma \in \mathcal{P}$. It is easy to see that for all such σ, we have $|\mathtt{permute}([\mathtt{X}|\mathtt{Xs}], \mathtt{Ys})\sigma| > |\mathtt{permute}(\mathtt{Xs}, \mathtt{Zs})\sigma|$. The important point is that the atoms $\mathtt{insert}(\mathtt{Us}, \mathtt{U}, vs) \in PM_P^{SL}$ do not give rise to a proof obligation since $\mathtt{permute}(\_, \mathtt{Us}) \notin \mathcal{P}$.

The following is an example of "hybrid" selection rules (point 3 in Sec. 1).

[4] The lemma states that a variable in a query must occur in some output position to become instantiated.

Example 4.8 We only sketch this example. A program for the well-known n-queens problem has the following main clause:

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

We could implement `permute` as in Ex. 4.4 or as in Ex. 4.7. In either case, we have a non-trivial $\mathcal{P}$. In contrast, $\mathcal{P}$ may contain all atoms using `safe`. In fact, for efficiency reasons, atoms using `safe` should be selected as early as possible.

Note that such a hybrid selection rule can effectively be realised by means of the *default left-to-right selection rule* [18]. To this end, the second and third atoms must be swapped. Since any results in this report do not actually depend on the textual position of atoms, they still apply to the thus modified program.

5 Local Selection Rules

In this section, we adapt the results of the two previous sections to local selection rules. First note that local derivations genuinely need special treatment, since one cannot express locality as a property $\mathcal{P}$ of the selected atoms.

Assuming local selection rules is helpful for showing termination, since one can exploit model information almost in the same way as for LD derivations [4]. In fact, some arguments are simpler here than in the previous two sections.

Note here that *local* [12] and *simply-local* [5, 6] are completely different concepts, and strange as it may sound, we shall define *local simply-local* models.

The first important observation is that the Left-Switching Lemma does not hold for local derivations — switching steps may destroy the locality property. Corollary 4.5 rests on the Left-Switching Lemma. But it turns out that for local derivations, the analogous statement is a corollary of the following lemma.

Lemma 5.1 Let the program P and the query $\mathbf{Q} = A_1, \ldots, A_n$ be simply moded. Suppose there is an infinite local input-consuming $\mathcal{P}$-derivation δ of $P \cup \{\mathbf{Q}\}$. Then for some $i \in [1..n]$, δ has the form

$$\mathbf{Q} \xrightarrow{\sigma} (\mathbf{B}, A_i, \mathbf{C})\sigma \longrightarrow \ldots$$

where the suffix after $(\mathbf{B}, A_i, \mathbf{C})\sigma$ contains only A_i-steps, and there exists an infinite local input-consuming $\mathcal{P}$-derivation of $P \cup \{A_i\sigma\}$.

Proof. Since δ is infinite, it contains an infinite number of A_i-steps for some $i \in [1..n]$. Moreover, since δ is local, there can only be one such i. Hence δ contains only a finite number of A_j-steps for all $j \neq i$, and again since δ is local, each A_j, where $j \neq i$, is either resolved completely (and before the first A_i-step) or not at all. We write $A_1, \ldots, A_n$ as $\mathbf{A}_0, A_{j_1}, \mathbf{A}_1, \ldots, A_{j_k}, \mathbf{A}_k$ where $A_{j_1}, \ldots, A_{j_k}$ are the resolved atoms and $\mathbf{A}_0, \ldots, \mathbf{A}_k$ are the sequences of unresolved atoms in between. Note that $i = j_l$ for some $l \in [1..k]$ since A_i is (partially) resolved. Then δ has the desired form

$$A_1, \ldots, A_n \xrightarrow{\sigma} \underbrace{(\mathbf{A}_0, \mathbf{A}_1, \ldots, \mathbf{A}_{l-1}, A_{j_l}, \mathbf{A}_l, \ldots, \mathbf{A}_k)\sigma \longrightarrow \ldots}_{=:\delta'}$$

where δ' contains only A_{j_l}-, i.e., A_i-steps. But then an infinite local input-consuming $\mathcal{P}$-derivation of $A_i\sigma$ is obtained from δ' simply by removing the sequences $\mathbf{A}_0, \ldots, \mathbf{A}_k$. □

To reflect the substitutions that can be computed by local derivations, we need as model the union of a simply-local model (for the completely resolved atoms) and the set of simply moded atoms (for the unresolved atoms).

Definition 5.2 Let $M \subseteq \mathit{Atom}$ and P a simply moded program. M is a **local simply-local model** of P if $SM_P \subseteq M$ and for some simply-local model M' of P, we have $M' \subseteq M$. We denote the least local simply-local model by LM_P^{SL}.

So a local simply-local model contains SM_P, but unlike PM_P^{SL}, does not involve applications of T_P^{SL} to atoms in SM_P.

Example 5.3 Let P be the following program in mode $\mathtt{r}(I), \mathtt{p}(I,O)$:

```
r(X) ←p(X,Y), r(Y).          p(X,s(X))←fail.
r(0).                        p(s(X),X).
```

We have

$$\begin{array}{rcl} SM_P & = & \{\mathtt{r}(x),\ \mathtt{p}(x,\mathtt{Z}),\ \mathtt{fail} \mid x \text{ arbitrary where } \mathtt{Z} \notin \mathit{Vars}(x)\} \\ PM_P^{SL} & = & SM_P \cup \{\mathtt{r}(\mathtt{0}),\ \mathtt{p}(\mathtt{s}(x),x),\ \mathtt{p}(x,\mathtt{s}(x)),\ \mathtt{r}(\mathtt{s}(x)),\ \mathtt{r}(x) \mid x \text{ arbitrary}\}. \end{array}$$

Another simply-local model of P, not containing SM_P, is the following:

$$M' = \{\mathtt{r}(\mathtt{s}^n(\mathtt{0})),\ \mathtt{p}(\mathtt{s}(x),x) \mid n \in \mathbb{N}_0,\ x \text{ arbitrary}\}.$$

$LM_P^{SL} = SM_P \cup M'$ is the least local simply-local model of P. In contrast to PM_P^{SL} we have $\mathtt{p}(x,\mathtt{s}(x)) \notin LM_P^{SL}$. This reflects that in a local derivation, resolving an atom with the clause head $\mathtt{p}(\mathtt{X},\mathtt{s}(\mathtt{X}))$ will definitely lead to finite failure. For this program, locality is crucial for termination (see point 5 in Sec. 1).

The following lemma adapts Lemma 3.9 to local derivations.

Lemma 5.4 Let the program P and the query $\mathbf{A}$ be simply moded and $I \subseteq SM_P$. If there exists an input-consuming derivation $\delta = \mathbf{A} \xrightarrow{\sigma} \mathbf{C}$ with $\mathbf{C} \subseteq I$, then $\mathbf{A}\sigma \subseteq M \cup I$ for any simply-local model M of P.

Proof. Since δ is local, at most one atom of $\mathbf{A}$ is partially resolved; all others are resolved completely or not at all. We write $\mathbf{A}$ as $\mathbf{A}_0, A_1, \mathbf{A}_1, \ldots, A_n, \mathbf{A}_n$ where $A_1, \ldots, A_n$ are the (at least partially) resolved atoms and $\mathbf{A}_0, \ldots, \mathbf{A}_n$ are the sequences of unresolved atoms in between. Let $j \in [1..n]$ be the index such that A_j is partially resolved, arbitrary if no such index exists. Then $\mathbf{C} = (\mathbf{A}_0, \mathbf{A}_1, \ldots, \mathbf{A}_{j-1}, \mathbf{A'}_j, \mathbf{A}_j, \ldots, \mathbf{A}_n)\sigma$, where $\mathbf{A'}_j$ consists of the descendants of A_j. $\mathbf{A'}_j$ may be empty. Let M be any simply-local model of P.

By Cor. 3.4, setting $\mathbf{B}$ and $\mathbf{C}_1$ to $\Box$, there is an input-consuming derivation

$$\begin{array}{rl} \delta' = & \mathbf{A}_0, A_1, \mathbf{A}_1, \ldots, A_n, \mathbf{A}_n \\ \xrightarrow{\sigma_1} & (\mathbf{A}_0, \mathbf{A}_1, A_2, \ldots, A_n, \mathbf{A}_n)\sigma_1 \\ \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_{j-1}} & (\mathbf{A}_0, \ldots, \mathbf{A}_{j-1}, A_j, \mathbf{A}_j, \ldots, A_n, \mathbf{A}_n)\sigma_1 \ldots \sigma_{j-1} \\ \xrightarrow{\sigma_j} & (\mathbf{A}_0, \ldots, \mathbf{A}_{j-1}, \mathbf{A'}_j, \mathbf{A}_j, \ldots, A_n, \mathbf{A}_n)\sigma_1 \ldots \sigma_j \\ \xrightarrow{\sigma_{j+1}} \ldots \xrightarrow{\sigma_{n-1}} & (\mathbf{A}_0, \mathbf{A}_1, \ldots, \mathbf{A'}_j, \ldots, \mathbf{A}_{n-1}, A_n, \mathbf{A}_n)\sigma_1 \ldots \sigma_{n-1} \\ \xrightarrow{\sigma_n} & (\mathbf{A}_0, \mathbf{A}_1, \ldots, \mathbf{A'}_j, \ldots, \mathbf{A}_{n-1}, \mathbf{A}_n)\sigma_1 \ldots \sigma_n \end{array}$$

which proceeds left-to-right, and $\sigma = \sigma_1, \ldots, \sigma_n$. Note that in general, δ' will not be local (due to A_j), but this does not matter.

Now for each $l \in [1..n]$ where $l \neq j$, there exists an input-consuming derivation

$$A_l\sigma_1 \ldots, \sigma_{l-1} \xrightarrow{\sigma_l} \Box,$$

and so by Lemma 3.9, $A_l\sigma_1 \ldots, \sigma_l \in M$. By Prop. 3.2, $A_l\sigma_1 \ldots, \sigma_l = A_l\sigma$, and so $A_l\sigma \in M$. Moreover, by hypothesis, $\mathbf{C} \subseteq I$, i.e., for each $l \in [0..n]$, $\mathbf{A}_l\sigma \in I$, and so the claim follows. $\Box$

We now proceed with the treatment of termination.

Definition 5.5 A program is **local $\mathcal{P}$-terminating** if all input-consuming local $\mathcal{P}$-derivations starting in a simply-moded query are finite.

We now give the promised corollary of Lemma 5.1.

Corollary 5.6 Let P be a simply moded program. P is local $\mathcal{P}$-terminating if and only if for each simply moded atomic query A all local input-consuming $\mathcal{P}$-derivations of $P \cup \{A\}$ are finite.

We now give a sufficient criterion for local $\mathcal{P}$-termination.

Definition 5.7 Let P be a simply moded program, $|.|$ a moded level mapping and M a local simply-local model of P. A clause $A \leftarrow B_1, \ldots, B_n$ is **local $\mathcal{P}$-acceptable by** $|.|$ **and** M[5] if for every substitution σ simply-local wrt. it, for all $i \in [1..n]$,

$$(B_1, \ldots, B_{i-1})\sigma \in M \text{ and } A \simeq B_i \text{ and } B_i\sigma \in \mathcal{P} \quad \text{implies} \quad |A\sigma| > |B_i\sigma|.$$

The program P is **local $\mathcal{P}$-acceptable by** $|.|$ **and** M if each clause of P is local $\mathcal{P}$-acceptable by $|.|$ and M.

Example 5.8 Consider again the program in Ex. 5.3, in particular the recursive clause. Let $\mathcal{P}$ be the set of atoms where all input arguments are non-variable and $|\mathtt{r}(x)| = |\mathtt{p}(x,y)| = |x|$ where $|.|$ is the term size norm. We verify that the second body atom fulfils the requirement of Def. 5.7. We have to consider all simply-local σ such that $\mathtt{p(X,Y)}\sigma$ is an instance of an atom in LM_P^{SL}. So

$$\sigma \supseteq \{\mathtt{X}/x, \mathtt{Y}/\mathtt{Z}\} \quad \text{or} \quad \sigma \supseteq \{\mathtt{X}/\mathtt{s}(x), \mathtt{Y}/x\}$$

In the first case, $\mathtt{r(Y)}\sigma \notin \mathcal{P}$ and hence no proof obligation arises. In the second case, $|\mathtt{r(X)}\sigma| > |\mathtt{r(Y)}\sigma|$. Hence the clause is local $\mathcal{P}$-acceptable. Note that the clause is not simply $\mathcal{P}$-acceptable (due to the atoms $\mathtt{p}(x, \mathtt{s}(x)) \in PM_P^{SL}$).

Observe that unlike [12], we do not explicitly require that the selected atoms must be bounded. In our formalism, the instantiation requirements of the selected atom and the locality issue are two separate dimensions.

Theorem 5.9 Let P be a simply moded program. Let M be a local simply-local model of P. Suppose that P is local $\mathcal{P}$-acceptable wrt. M and a moded level mapping $|.|$. Then P is local $\mathcal{P}$-terminating.

Proof. We define $dep(p)$ as in the proof of Thm. 4.6. By Cor. 5.6, it is sufficient to prove that for any simply moded atomic query A, all local input-consuming $\mathcal{P}$-derivations of $P \cup \{A\}$ are finite.

For the purpose of deriving a contradiction, assume that δ is an infinite local input-consuming $\mathcal{P}$-derivation of $P \cup \{A\}$. Then, spelling out the first step and using Lemma 5.1, for some $i \in [1..n]$, we have

$$\delta = A \overset{\sigma_1}{\Longrightarrow} (B_1, \ldots, B_n)\sigma_1 \xrightarrow{\sigma_2} (\mathbf{A}, B_i, \mathbf{C})\sigma_1\sigma_2 \longrightarrow \ldots,$$

where $c = H \leftarrow B_1, \ldots, B_n$ is the clause used in the first step, and

$$\text{there is an infinite local input-consuming } \mathcal{P}\text{-derivation of } P \cup \{B_i\sigma_1\sigma_2\}. \tag{4}$$

Let $\sigma = (\sigma_1\sigma_2)_{|c}$. Using [6, Proposition A.1], one can easily show that σ is simply-local wrt. c. By Lemma 5.4, since $SM_P \subseteq M$ and $M' \subseteq M$ for some simply-local model M' of P, we have $(B_1, \ldots, B_{i-1})\sigma \in M$.

Define $\succ$ as in the proof of Thm. 4.6. We will show that

$$\langle dep(Rel(A)), |A| \rangle \succ \langle dep(Rel(B_i\sigma)), |B_i\sigma| \rangle.$$

By an obvious inductive argument, this contradicts (4).

As in the proof of Thm. 4.6 we have $|A| = |A\sigma| = |H\sigma|$, and we distinguish two cases: (a) $Rel(H) \sqsupset Rel(B_i)$, and (b) $Rel(H) \simeq Rel(B_i)$.

[5] This terminology should be regarded as provisional (see footnote 3).

In case (a) we have $dep(Rel(A)) = dep(Rel(H\sigma)) > dep(Rel(B_i\sigma))$. Therefore,

$$\langle dep(Rel(A)), |A|\rangle = \langle dep(Rel(H\sigma)), |H\sigma|\rangle \succ \langle dep(Rel(B_i\sigma)), |B_i\sigma|\rangle. \qquad (5)$$

In case (b), from the hypothesis that P is local $\mathcal{P}$-acceptable wrt. $|.|$ and M, σ is simply-local wrt. c and $(B_1, \dots, B_{i-1})\sigma \in M$, and moreover $B_i\sigma \in \mathcal{P}$, it follows that $|H\sigma| > |B_i\sigma|$. Hence, (5) holds also in this case. □

6 Conclusion

We have presented a framework for proving termination of logic programs with dynamic scheduling. We have considered various assumptions about the selection rule, in addition to the assumption that derivations must be input-consuming. On the one hand, derivations can be restricted by giving a property $\mathcal{P}$ that the selected atoms must fulfill. On the other hand, derivations may or may not be required to be local. These aspects can be combined freely. We now refer back to the five points in the introduction.

Some programs terminate under an assumption about the selection rule that is just slightly stronger than assuming input-consuming derivations (point 1). Others need what we call *strong* assumptions: the selected atom must be bounded wrt. a level mapping (point 2). Different versions of `PERMUTE`, which is the standard example of a program that tends to loop for dynamic scheduling [15], are representatives of these program classes. Then there are programs for which one should make hybrid assumptions about the selection rule: depending on the predicate, an atom should be bounded in its input positions or not (point 3). Considering our work together with [6], it is no longer true that "the termination behaviour of 'delay until nonvar' is poorly understood" [12].

The authors of [12] have assumed local selection rules. There are programs for which this assumption is genuinely needed. Abstractly, this is the case for a query $A_1, \dots, A_n$ where for some atom A_i and some clause c, the subderivations associated with A_i and c all fail, but at the same time, the unification between A_i and c's head produces a substitution that may trigger an infinite derivation for some atom A_j, where $j > i$. In this case, locality ensures failure of A_i before the infinite derivation of A_j can happen. The comparison between our model notions (Defs. 3.7 and 5.2) also clarifies the role of locality: substitutions obtained by *partial* resolution of an atom can be disregarded (point 5). But we are not aware of a *realistic* program where this matters (point 4). As an obvious consequence, we have no realistic program that we can show to terminate for local derivations and the method of [12] cannot. But the better understanding of the role of locality may direct the search for such an example.

The concepts of *input-consuming derivations* and $\mathcal{P}$-derivations are both meant to be abstract descriptions of dynamic scheduling. Delay declarations that check for arguments being at least non-variable, or at least non-variable in some sub-argument [10, 19], are often adequate for ensuring input-consuming derivations with $\mathcal{P}$ stating that the input arguments are at least non-variable (see Ex. 4.7). Delay declarations that check for groundness are adequate for ensuring boundedness of atoms (see Ex. 4.4). In general groundness is stronger than boundedness, but we are not aware of delay declarations that could check for boundedness, e.g., check for a list being nil-terminated. This deficiency has been mentioned previously [11]. Hybrid selection rules can be realised with delay declarations combined with the default left-to-right selection rule (see Ex. 4.8).

Concerning automation of our method, the problems are not so different from the ones encountered when proving left-termination: we have to reason about infinite models — to do so, abstract interpretation approaches, where terms are abstracted as their norms, may be useful. It seems that in our case automation is additionally complicated because we have to consider infinitely many simply-local substitutions. But looking at Ex. 3.8, we have terms $y_1, y_2, \dots$ that are arbitrary and whose form does not affect the termination problem. Therefore it might be sufficient to consider *most general* substitutions in applications of T_P^{SL}.

The main topic for future work is to find conditions that are not only sufficient but also *necessary* for the two notions of termination. Our investigations so far suggest that the conditions presented here are *not* necessary conditions. Moreover, it seems that necessary conditions, while being obviously "better" (weaker) than the sufficient conditions given here, are conceptually quite complicated and that the adaptation of [6] must be quite involved. This is in contrast to the results of this report where the adaptation of [6] was relatively straightforward.

Another topic for future work is, of course, a practical evaluation, looking at a larger program suite. Also, we should consider the following issues: (1) possible generalisations of the results in Sec. 5, leaving aside the assumption of input-consuming derivations; (2) a termination criterion that would capture programs that terminate for certain (intended) queries, but not for all queries; (3) relaxing the condition of simply moded programs.

Acknowledgements

The author thanks Lee Naish and Alexander Serebrenik for useful discussions about this work.

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, *Proc. of the 18th International Symposium on Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, pages 1–19. Springer-Verlag, 1993.

[3] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V. S. Alagar and M. Nivat, editors, *Proc. of the 4th International Conference on Algebraic Methodology and Software Technology*, volume 936 of *LNCS*, pages 66–90. Springer-Verlag, 1995.

[4] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[5] A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *Theory and Practice of Logic Programming*, 2(2):125–154, 2002.

[6] A. Bossi, S. Etalle, S. Rossi, and J.-G. Smaus. Semantics and termination of simply moded logic programs with dynamic scheduling. *Transactions on Computational Logic*, 2003. To appear. Available from `http://www.dsi.unive.it/%7Esrossi/papers.html`.

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

[8] N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 2001(1/2):117–156, 2001.

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

[10] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[11] 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.

[12] E. Marchiori and F. Teusink. On termination of logic programs with delay declarations. *Journal of Logic Programming*, 39(1-3):95–124, 1999.

[13] J. Martin and A. King. Generating efficient, terminating logic programs. In M. Bidoit and M. Dauchet, editors, *Proc. of the 7th International Conference on Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 273–284. Springer-Verlag, 1997.

[14] F. Mesnard, É. Payet, and U. Neumerkel. Detecting optimal termination conditions of logic programs. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis Symposium*, volume 2477 of *LNCS*, pages 509–526. Springer-Verlag, 2002.

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

[16] D. Pedreschi, S. Ruggieri, and J.-G. Smaus. Classes of terminating logic programs. *TPLP*, 2(3):369–418, 2002.

[17] J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *Proc. of the International Conference on Logic Programming*, pages 335–349. MIT Press, 1999.

[18] J.-G. Smaus, P. M. Hill, and A. M. King. Verifying termination and error-freedom of logic programs with `block` declarations. *Theory and Practice of Logic Programming*, 1(4):447–486, 2001.

[19] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 2003. `http://www.sics.se/isl/sicstuswww/site/documentation.html`.