Termination of Logic Programs Using 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

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 be combined, yielding the most comprehensive approach so far to termination of logic programs with dynamic scheduling. For non-local derivations, the termination criterion is even necessary.

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, 8–11, 16]. 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* [12, 23], 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, 6, 7, 13–15, 17, 19, 22], starting with observations of surprisingly complex (non-)termination behaviour of simple programs such as `APPEND` or `PERMUTE` with delay declarations [17]. In our own research [7, 19, 22], 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, abstracting away from the technicalities of delay declarations.

In this paper, 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 [13–15, 17, 22],

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 use *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 [6], and restricting to a class of programs that is well-behaved wrt. the modes, we now have a sufficient and necessary criterion for termination of input-consuming derivations [7]. 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 $\mathcal{P}$. There are some natural conditions on $\mathcal{P}$ mentioned later. It turns out that the approach of [7] can be easily adapted to give a *sufficient* criterion for termination of $\mathcal{P}$-derivations [20]. The semantic notions (model) of [7] could be used without change. In this paper we give a criterion that is also *necessary*. To this end, the approach of [7] required some small modifications in many places. More specifically, the model notion had to be modified.

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 [14]. 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 [7] so that it reflects the substitutions computed by local derivations. Based on such models, we can give a sufficient 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. More specifically, by instantiating the framework appropriately, we can now make the following five points:

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 $p(input),\ q(input, output)$. 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 by setting (the relevant subset of) $\mathcal{P}$ to $\{p(t) \mid t \text{ is non-variable}\}$ (Ex. 17).
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 [17]. This can be captured in our approach by setting $\mathcal{P}$ to the set of bounded atoms (Ex. 14).

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. 18).
4. A method for showing termination of programs with delay declarations has been proposed in [14], 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" [14], 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. 14).
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. 23).

The rest of this paper is organised as follows. The next section gives some preliminaries. In Sec. 3, we adapt the semantics approach of [7] to $\mathcal{P}$-derivations. In Sec. 4, we study termination for such derivations. In Sec. 5, we adapt our 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 $m, n \in \mathbb{N}_0$, $m \leq n$, 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. 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 $\mathbf{A}, B, \mathbf{C} \stackrel{\sigma}{\Longrightarrow}_c (\mathbf{A}, \mathbf{B}, \mathbf{C})\sigma$ a **derivation step**, in short: **step**. If c is irrelevant then we drop the reference. A sequence $\delta = \mathbf{Q}_0 \stackrel{\sigma_1}{\Longrightarrow}_{c_1} \mathbf{Q}_1 \stackrel{\sigma_2}{\Longrightarrow}_{c_2} \cdots$ is called a **derivation of** $P \cup \{\mathbf{Q}_0\}$.

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

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.

We assume that the mode of each predicate is unique. One way of ensuring this is to rename predicates whenever multiple modes are desired.

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 [7], although we will also give an example of a clause that is not.

Note that the use of the letters $\mathbf{s}$ and $\mathbf{t}$ is reversed for clause heads. We believe that this notation naturally reflects the data flow within a clause.

Definition 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 $i \in [1..n]$

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

A query $\mathbf{B}$ is **simply moded** if the clause $\texttt{dummy} \leftarrow \mathbf{B}$ 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.

As an example of a clause that is *not* simply moded, consider the clause

```
reverse([X|Xs],Ys) ← append(Zs,[X],Ys), reverse(Xs,Zs).
```

in mode $\texttt{reverse}(O, I), \texttt{append}(O, O, I)$: $[\texttt{X}]$ is not a variable. In Ex. 16, we give a slightly modified version of the `NAIVE_REVERSE` program that is simply moded. *Robustly typed* programs [22] are in some sense a generalisation of simply moded programs, and include the above clause. However, the results of this paper have so far not been generalised to robustly typed programs.

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 [10], but here we define them for arbitrary objects (in [18], 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 [7].

Definition 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 $k > |A\sigma|$.

Our method of showing termination, following [7], inherently relies on measuring the size of atoms that are not bounded. In Def. 13, a decrease in level mapping must be shown (also) for such atoms. So it is important to understand that stating $|A| = k$ is different from stating that A is bounded by k.

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

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

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

$$\begin{array}{rll} |[t|ts]| = 1 + |ts|, & & \\ |t| = 0 & & \text{if } t \neq [_|_] \text{ (in particular, if } t \text{ variable).} \end{array}$$

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 [19] as formalism for describing dynamic scheduling in an abstract way.

Definition 3. A derivation 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 [14]. 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 4. A derivation is **local** if in each step, there is no more recently introduced atom in the current query than the selected atom.

Intuitively, in a local derivation, once an atom is selected, that atom must be resolved away completely before any of its siblings may be selected.

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

We consider derivations restricted by some property $\mathcal{P}$ of the selectable atoms. There are two conditions on $\mathcal{P}$. Some of our results would hold without these conditions, but the conditions are so natural that we do not bother with this.

Definition 5. A **$\mathcal{P}$-derivation** is a derivation such that each selected atom is in $\mathcal{P}$, where

1. $\mathcal{P}$ is a set of atoms closed under instantiation;
2. for any $\mathbf{s}$, $\mathbf{t}$ and $\mathbf{u}$, $p(\mathbf{s}, \mathbf{t}) \in \mathcal{P}$ implies $p(\mathbf{s}, \mathbf{u}) \in \mathcal{P}$.

Note that the atoms of a simply moded query have variables in their output positions, and so it would clearly be pathological to require a particular instantiation of the output arguments of an atom for that atom to be selected.

This is the first published work introducing the concept of $\mathcal{P}$-derivations. Of course, a $\mathcal{P}$-derivation can be qualified further by saying *input-consuming* $\mathcal{P}$-derivation etc.

Input-consuming ($\mathcal{P}$-)derivations may end in a query where no atom can be selected. This situation is called **deadlock**. It is a form of termination.

We now define *simply-local* substitutions, which reflect the way simply moded clauses become instantiated in input-consuming derivations [7]. 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 6. 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 $V_0, V_1, \ldots, V_n$ consisting of fresh (wrt. c) variables such that $\sigma = \sigma_0\sigma_1 \cdots \sigma_n$, where for $i \in [0..n]$,

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

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

In the case of a simply-local substitution wrt. a query, σ_0 is the identity.

Example 7. Consider `DELETE` in Fig. 1, with mode $\texttt{delete}(I, O, O)$. The substitution $\sigma = \{\texttt{Y/V}, \texttt{Zs/[W]}, \texttt{Xs/[]}, \texttt{X/W}\}$ is simply-local wrt. the recursive clause: let $\sigma_0 = \{\texttt{Y/V}, \texttt{Zs/[W]}\}$, $\sigma_1 = \{\texttt{X/W}, \texttt{Xs/[]}\}$, and $\sigma_2 = \emptyset$; then $\mathit{Dom}(\sigma_0) \subseteq \{\texttt{Y}, \texttt{Zs}\}$, $\mathit{Ran}(\sigma_0) \subseteq V_0$ where $V_0 = \{\texttt{V}, \texttt{W}\}$, $\mathit{Dom}(\sigma_1) \subseteq \{\texttt{Xs}, \texttt{X}\}$, $\mathit{Ran}(\sigma_1) \subseteq \mathit{Vars}(\texttt{Zs}\,\sigma_0)$.

[1] Note that $\mathbf{s}_0$ is undefined. By abuse of notation, $\mathit{Vars}(\mathbf{s}_0 \ldots) = \emptyset$.

```
permute([],[]).
permute(Ys,[X|Xs]) ←
   delete(Ys,Zs,X),
   permute(Zs,Xs).
```

```
delete([X|Xs],Xs,X).
delete([Y|Zs],[Y|Xs],X) ←
   delete(Zs,Xs,X), call_late.
call_late.
```

Fig. 1. PERMUTE (DELETE)

We can safely assume that all the mgu's employed in an input-consuming derivation of a simply moded program with a simply moded query are simply-local, that is to say: 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 $\sigma = \sigma_0\sigma_1$ and $\sigma_0 (= \sigma_{|H})$ is simply-local wrt. the clause $H \leftarrow$ and $\sigma_1 (= \sigma_{|B})$ is simply-local wrt. the atom[2] B [7, Lemma 3.8]. This assumption is crucial in the proofs of the results of this paper.

In [7], a particular notion of model is defined, 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. Here, we generalise this notion so that it reflects the substitutions that can be computed by input-consuming $\mathcal{P}$-derivations. This generalisation is crucial for the results in Subsec. 4.3.

Definition 8. Let $M \subseteq Atom$. We say that M is a **simply-local $\mathcal{P}$-model** of $c = H \leftarrow B_1, \ldots, B_n$ if for every substitution σ simply-local wrt. c,

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

M is a **simply-local $\mathcal{P}$-model** of a program P if M is a simply-local $\mathcal{P}$-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 $\mathcal{P}$-models, possibly containing SM_P, can be computed by a variant of the well-known T_P-operator [7].

Definition 9. Given a program P and $I \subseteq Atom$, we define

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

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

Example 10. Consider the program DELETE (see Fig. 1) ignoring the call_late predicate. Recall Ex. 7. Let $\mathcal{P}$ be the set containing *all* atoms using delete. SM_P consists of all atoms of the form $\texttt{delete}(vs, \texttt{Us}, \texttt{U})$ where $\texttt{Us}, \texttt{U} \notin Vars(vs)$. To construct $PM_P^{SL\mathcal{P}}$, we iterate $T_P^{SL\mathcal{P}}$ 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

[2] We sometimes say "atom" for "query containing only one atom".

line below corresponds to one iteration of $T_P^{SL\mathcal{P}}$. We have $PM_P^{SL\mathcal{P}} =$

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

Observe the variable occurrences of $\mathtt{U}, \mathtt{Us}$ in the atoms on the l.h.s. In Ex. 14, we will see the importance of such variable occurrences.

In the above example, we assume that $\mathcal{P}$ is the set of *all* atoms, and so the simply-local $\mathcal{P}$-model is in fact a *simply-local model* [7]. In order to obtain a *necessary* termination criterion, the approach of [7] required some small modifications in many places, one of them being the generalisation of simply-local models to simply-local $\mathcal{P}$-models. However, we are not aware of a practical situation where one has to consider a simply-local $\mathcal{P}$-model that is not a simply-local model.

The model semantics given here is equivalent to the operational semantics. We do not formally state this equivalence here for lack of space, but it is used in the proofs of the termination results of the following sections [21].

4 Termination without Requiring Local Selection Rules

4.1 Simply $\mathcal{P}$-Accceptable Programs

The following concept is adopted from Apt [1].

Definition 11. 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 12. A program is **input $\mathcal{P}$-terminating** if all input-consuming $\mathcal{P}$-derivations starting in a simply-moded query are finite.

Previously, we had defined *input termination*, which is input $\mathcal{P}$-termination for $\mathcal{P}$ being the set of all atoms [7]. We now give a sufficient and necessary criterion for input $\mathcal{P}$-termination.

Definition 13. Let P be a simply moded program, $|.|$ a moded level mapping and M a simply-local $\mathcal{P}$-model of P containing SM_P. A clause $H \leftarrow B_1, \ldots, B_n$ is **simply $\mathcal{P}$-acceptable by** $|.|$ **and** M if for every substitution σ simply-local wrt. it, for all $i \in [1..n]$,

$$B_1\sigma, \ldots, B_{i-1}\sigma \in M \text{ and } H \simeq B_i \text{ and } H\sigma \in \mathcal{P} \text{ and } B_i\sigma \in \mathcal{P} \text{ imply } |H\sigma| > |B_i\sigma|. \tag{3}$$

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

The difference to the definition of a *simply acceptable* clause [7] is in the conditions $B_i\sigma \in \mathcal{P}$ and $H\sigma \in \mathcal{P}$. The condition $B_i\sigma \in \mathcal{P}$ may seem more natural than the condition $H\sigma \in \mathcal{P}$, since σ, due to the model condition, reflects the degree of instantiation that B_i may have when it is selected. But for $i = 1$, σ reflects the degree of instantiation of the entire clause obtained by unification when the clause is used for an input-consuming $\mathcal{P}$-derivation step. Moreover, the condition $H\sigma \in \mathcal{P}$ is important for showing *necessity* (Subsec. 4.3).

Note that a decrease between the head and body atoms must be shown only for the atoms where $H \simeq B_i$. The idea is that termination is shown incrementally, so we assume that for the B_i where $H \sqsupset B_i$, termination has been shown already. One can go further and explicitly give *modular* termination results [7, 10], but this is a side issue for us and we refrain from it for space reasons.

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 14. Consider PERMUTE in mode $\texttt{permute}(I, O)$, $\texttt{delete}(I, O, O)$ (Fig. 1). Recall Ex. 10. As norm we take the list-length function, and we define the level mapping as $|\texttt{permute}(zs, xs)| = |zs|$ and $|\texttt{delete}(xs, zs, x)| = |xs|$. Now for all atoms $\texttt{delete}(ys, zs, x) \in PM_P^{SL\mathcal{P}}$, 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. 13, where M is $PM_P^{SL\mathcal{P}}$. So we have to consider all simply-local substitutions σ such that $\texttt{delete}(\texttt{Ys}, \texttt{Zs}, \texttt{X})\sigma \in PM_P^{SL\mathcal{P}}$. For the atoms on the l.h.s. in (2), this means that

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

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

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

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

The other clauses are trivial to check, and so PERMUTE is simply $\mathcal{P}$-acceptable. Observe that only the model of DELETE played a role in our argument, not the model of PERMUTE.

The atom call_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.

The following infinite derivation (ignoring call_late), input-consuming but not a $\mathcal{P}$-derivation, demonstrates that the program does not input-terminate:

$$\begin{array}{l}
\underline{\texttt{permute}([1], \texttt{W})} \Longrightarrow \underline{\texttt{delete}([1], \texttt{Zs}', \texttt{X}')}, \ \texttt{permute}(\texttt{Zs}', \texttt{Xs}') \Longrightarrow \\
\texttt{delete}([], \texttt{Xs}'', \texttt{X}'), \ \underline{\texttt{permute}([1|\texttt{Xs}''], \texttt{Xs}')} \Longrightarrow \\
\texttt{delete}([], \texttt{Xs}'', \texttt{X}'), \ \underline{\texttt{delete}([1|\texttt{Xs}''], \texttt{Zs}''', \texttt{X}''')}, \ \texttt{permute}(\texttt{Zs}''', \texttt{Xs}') \Longrightarrow \\
\texttt{delete}([], \texttt{Xs}'', \texttt{X}'), \ \texttt{delete}(\texttt{Xs}'', \texttt{Xs}'''', \texttt{X}'''), \ \underline{\texttt{permute}([1|\texttt{Xs}''''], \texttt{Xs}')} \Longrightarrow \ldots
\end{array}$$

```
reverse([X|Xs],Ys) ←                    append_sing([X|Xs],Y,[X|Zs]) ←
   append_sing(Zs,X,Ys),                   append_sing(Xs,Y,Zs).
   reverse(Xs,Zs).                      append_sing([],Y,[Y]).
reverse([],[]).
```

Fig. 2. NAIVE_REVERSE

4.2 Sufficiency of Simply $\mathcal{P}$-Acceptability

In [7], we find a result stating that simply acceptable programs are input terminating. The following theorem generalises this result to $\mathcal{P}$-derivations. The proofs of all results of this paper can be found in [21].

Theorem 15. Let P be a simply moded program. Let M be a simply-local $\mathcal{P}$-model of P containing SM_P. Suppose that P is simply $\mathcal{P}$-acceptable by M and a moded level mapping $|.|$. Then P is input $\mathcal{P}$-terminating.

We give three further examples. The first supports point 2 in the introduction.

Example 16. The program NAIVE_REVERSE (Fig. 2) in mode $\texttt{reverse}(O, I)$, $\texttt{append_sing}(O, O, I)$ is not input terminating, but it is input $\mathcal{P}$-terminating for $\mathcal{P}$ chosen in analogy to Ex. 14.

The next example illustrates point 1 in the introduction.

Example 17. Let PERMUTE2 be the program obtained from PERMUTE by replacing the recursive clause for delete by its *most specific variant* [17]:

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

Assume $|.|$ and the modes as in Ex. 14. As in Ex. 10 we have

$$SM_P = \{\texttt{delete}(vs, \texttt{Us}, \texttt{U}) \mid vs \text{ arbitrary}\},$$

but when we start applying $T_P^{SL\mathcal{P}}$ to the atoms in SM_P, then due to the modified clause above, only the atoms of the form $\texttt{delete}([v|vs'], \texttt{Us}, \texttt{U})$ contribute; $\texttt{delete}([], \texttt{Us}, \texttt{U})$ does not contribute:

$$\begin{aligned} PM_P^{SL\mathcal{P}} = \{&\texttt{delete}(vs, \texttt{Us}, \texttt{U}),\\ &\texttt{delete}([y_1, v, vs'], [y_1|\texttt{Us}], \texttt{U}), && \texttt{delete}([x_1|xs_1], xs_1, x_1),\\ &\texttt{delete}([y_2, y_1, v|vs'], [y_2, y_1|\texttt{Us}], \texttt{U}), && \texttt{delete}([y_1, x_1|xs_1], [y_1|xs_1], x_1),\\ &\ldots && \ldots\\ &\mid vs, v, vs', xs_1, x_1, y_1, y_2, \ldots \text{ arbitrary} && \text{where } \texttt{Us}, \texttt{U} \notin Vars(vs, v, vs')\}. \end{aligned}$$

We show that the program is simply $\mathcal{P}$-acceptable by $|.|$ and $PM_P^{SL\mathcal{P}}$, where $\mathcal{P}$ is the set of atoms that are at least non-variable in their input positions. As in Ex. 14, we focus on the second body atom of the recursive clause for permute. We have to consider all simply-local substitutions σ such that $\texttt{delete}(\texttt{Ys}, \texttt{Zs}, \texttt{X})\sigma \in PM_P^{SL\mathcal{P}}$, and moreover $\texttt{permute}(\texttt{Zs}, \texttt{Xs})\sigma \in \mathcal{P}$. It is easy to see that for all such σ, we have $|\texttt{permute}(\texttt{Ys}, [\texttt{X}|\texttt{Xs}])\sigma| > |\texttt{permute}(\texttt{Zs}, \texttt{Xs})\sigma|$. The important point is that the atoms of the form $\texttt{delete}(vs, \texttt{Us}, \texttt{U}) \in PM_P^{SL\mathcal{P}}$ do not give rise to a proof obligation since $\texttt{permute}(\texttt{Us}, _) \notin \mathcal{P}$.

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

Example 18. For space reasons, 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(Seq,Sol), safe(Sol).
```

We could implement `permute` as in Ex. 14 or as in Ex. 17. 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 be implemented by means of the *default left-to-right selection rule* [22]. To this end, the second and third atoms must be swapped. Since any results in this paper do not actually depend on the textual position of atoms, they still apply to the thus modified program.

4.3 Necessity of Simply $\mathcal{P}$-Acceptability

We now give the converse of Theorem 15, namely that our criterion for proving input $\mathcal{P}$-termination wrt. simply moded queries is also necessary. The level mapping is constructed as a kind of tree that reflects all possible input-consuming $\mathcal{P}$-derivations, following the approach of [7] which in turn is based on [5]. But for space reasons, we only state the main result.

Theorem 19. Let P be a simply moded program and $\mathcal{P}$ a set of atoms according to Def. 5. If P is input $\mathcal{P}$-terminating then P is $\mathcal{P}$-simply acceptable. In particular, P is $\mathcal{P}$-simply acceptable by $PM_P^{SL\mathcal{P}}$.

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. Note also that *local* [14] and *simply-local* [6, 7] are completely different concepts.

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, manifest in the proofs [21]. However, this is also due to the fact that we currently just have a sufficient termination criterion for local derivations. How this criterion must be adapted to become also necessary is a topic for future work.

A *simply-local model* [7] is a simply-local $\mathcal{P}$-model where $\mathcal{P}$ is the set of all atoms. Analogously, we write PM_P^{SL} instead of $PM_P^{SL\mathcal{P}}$ in this case. 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).

Let M be the least simply-local model of P (note: not the least simply-local model of P *containing* SM_P) We define $LM_P^{SL} := SM_P \cup M$. So LM_P^{SL} contains SM_P, but unlike PM_P^{SL}, does not involve applications of T_P^{SL} to atoms in SM_P.

Example 20. Let P be the following program in mode $\texttt{even}(I), \texttt{minus2}(I, O)$:

```
even(X) ← minus2(X,Y), even(Y).        minus2(X,s(X)) ← fail.
even(0).                               minus2(s(s(X)),X).
```

We have

$$\begin{aligned} SM_P &= \{\texttt{even}(x),\ \texttt{minus2}(x, \texttt{Z}),\ \texttt{fail} \mid x \text{ arbitrary where } \texttt{Z} \notin \mathit{Vars}(x)\} \\ PM_P^{SL} &= SM_P \cup \{\texttt{even}(\texttt{0}),\ \texttt{minus2}(\texttt{s}(\texttt{s}(x)), x),\ \texttt{minus2}(x, \texttt{s}(x)), \\ &\qquad \texttt{even}(\texttt{s}(\texttt{s}(x))),\ \texttt{even}(x) \mid x \text{ arbitrary}\}. \end{aligned}$$

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

$$M = \{\texttt{even}(\texttt{s}^{2n}(\texttt{0})),\ \texttt{minus2}(\texttt{s}(\texttt{s}(x)), x) \mid n \in \mathbb{N}_0,\ x \text{ arbitrary}\}.$$

Then $LM_P^{SL} = SM_P \cup M$. In contrast to PM_P^{SL} we have $\texttt{minus2}(x, \texttt{s}(x)) \notin LM_P^{SL}$. This reflects that in a local derivation, resolving an atom with the clause head $\texttt{minus2}(\texttt{X}, \texttt{s}(\texttt{X}))$ will definitely lead to finite failure. For this program, locality is crucial for termination (see point 5 in Sec. 1).

The example[3] is contrived since the first clause for `minus2` is completely unnecessary, yet natural enough to suggest that there might be a "real" example.

We now proceed with the treatment of termination.

Definition 21. 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 a sufficient criterion for local $\mathcal{P}$-termination.

Definition 22. Let P be a simply moded program, $|.|$ a moded level mapping and M a set such that $SM_P \subseteq M$ and for some simply-local model M' of P, $M' \subseteq M$. A clause $A \leftarrow B_1, \ldots, B_n$ is **local $\mathcal{P}$-acceptable by $|.|$ and M**[4] 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 23. Consider again the program in Ex. 20, in particular the recursive clause. Let $\mathcal{P}$ be the set of atoms where all input arguments are non-variable and $|\texttt{even}(x)| = |\texttt{minus2}(x, y)| = |x|$ where $|.|$ is the term size norm. We verify that the second body atom fulfils the requirement of Def. 22, taking $M = LM_P^{SL}$. We have to consider all simply-local σ such that $\texttt{minus2}(\texttt{X}, \texttt{Y})\sigma \in LM_P^{SL}$. So

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

[3] Thanks to Felix Klaedtke for inspiring the example!

[4] This terminology should be regarded as provisional. If a sufficient and *necessary* condition for local $\mathcal{P}$-termination different from the one given here is found eventually, then it should be called "local $\mathcal{P}$-acceptable" rather than inventing a new name.

In the first case, $\texttt{even}(\texttt{Y})\sigma \notin \mathcal{P}$ and hence no proof obligation arises. In the second case, $|\texttt{even}(\texttt{X})\sigma| > |\texttt{even}(\texttt{Y})\sigma|$. Hence the clause is local $\mathcal{P}$-acceptable. Note that the clause is not simply $\mathcal{P}$-acceptable (due to $\texttt{minus2}(x, \texttt{s}(x)) \in PM_P^{SL}$).

Observe that unlike [14], we do not 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 24. Let P be a simply moded program. Let M be a set such that $SM_P \subseteq M$ and for some simply-local model M' of P, $M' \subseteq M$. Suppose that P is local $\mathcal{P}$-acceptable by M and a moded level mapping $|.|$. Then P is local $\mathcal{P}$-terminating.

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 fulfil. 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 [17], 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 [7], it is no longer true that "the termination behaviour of 'delay until nonvar' is poorly understood" [14].

The authors of [14] have assumed local selection rules. There are programs for which this assumption is genuinely needed. Abstractly, this is the case for a query $A_1, \ldots, A_n$ 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 (see Ex. 20) 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 [14] cannot. But the better understanding of the role of locality may direct the search for such an example.

For derivations that are not assumed to be local, we obtain a sufficient and necessary termination criterion. For local derivations, our criterion is sufficient but probably not necessary. Finding a necessary criterion is a topic for future work. This would be an important advance over [14], since the criterion given there is known not to be necessary.

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 [12, 23], are often adequate for ensuring input-consuming derivations with $\mathcal{P}$ stating that the input arguments are at least non-variable (see Ex. 17). Delay declarations that check for groundness are adequate for ensuring boundedness of atoms (see Ex. 14). 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 [13]. Hybrid selection rules can be realised with delay declarations combined with the default left-to-right selection rule (see Ex. 18).

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 [11, 16]. It seems that in our case automation is additionally complicated because we have to consider infinitely many simply-local substitutions. But looking at Ex. 10, we have terms $y_1, y_2, \ldots$ that are arbitrary and whose form does not affect the termination problem. Hence it may be sufficient to consider *most general* substitutions in applications of $T_P^{SL\mathcal{P}}$.

Another topic for future work is, of course, a practical evaluation, looking at a larger program suite. In this context, it would be desirable to infer a $\mathcal{P}$, as unrestrictive as possible, automatically. 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.

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. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.
6. A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *Theory and Practice of Logic Programming*, 2(2):125–154, 2002.
7. 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*, 2004. To appear in summer 2004.
8. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.
9. 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.
10. S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.
11. S. Genaim, M. Codish, J. Gallagher, and V. Lagoon. Combining norms to prove termination. In A. Cortesi, editor, *Proc. of the 3rd International Workshop on Verification, Model Checking, and Abstract Interpretation*, volume 2294 of *LNCS*, pages 126–138. Springer-Verlag, 2002.
12. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
13. 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.
14. E. Marchiori and F. Teusink. On termination of logic programs with delay declarations. *Journal of Logic Programming*, 39(1-3):95–124, 1999.
15. 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.
16. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In P. Cousot, editor, *Proc. of the 8th Static Analysis Symposium*, volume 2126 of *LNCS*, pages 93–110. Springer-Verlag, 2001.
17. L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, Department of Computer Science, University of Melbourne, 1992.
18. D. Pedreschi, S. Ruggieri, and J.-G. Smaus. Classes of terminating logic programs. *Theory and Practice of Logic Programming*, 2(3):369–418, 2002.
19. 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.
20. J.-G. Smaus. Termination of logic programs for various dynamic selection rules. Technical Report 191, Institut für Informatik, Universität Freiburg, 2003.
21. J.-G. Smaus. Termination of logic programs using various dynamic selection rules. Technical Report 203, Institut für Informatik, Universität Freiburg, 2004.
22. 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.
23. Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 2003. `http://www.sics.se/isl/sicstuswww/site/documentation.html`.