On logic programs that always succeed

Dino Pedreschi and Salvatore Ruggieri

Dipartimento di Informatica, Università di Pisa
Via F. Buonarroti 2, 56125 Pisa, ITALY
{pedre, ruggieri}@di.unipi.it

Abstract

We introduce a generalized definition of SLD-resolution admitting restrictions on atom and/or clause selectability. Instances of these restrictions include delay declarations, input-consuming unification and guarded clauses. In the context of such a generalization of SLD-resolution, we offer a theoretical framework to reason about programs and queries such that *all* derivations are successful. We provide a characterization of those programs and queries which allows to reuse existing methods from the literature on termination and verification of Prolog programs.

1 Introduction

There are two main advantages of reasoning on programs and queries that have only successful derivations.

On one hand, *don't care* nondeterminism can be safely adopted in executions of logic programs, without incurring speculative computations. Under the don't care interpretation of nondeterminism, the operated choices are never retracted, i.e. backtracking is not permitted. For instance, the don't care or *committed-choice* nondeterminism is the usual interpretation both in imperative parallel languages, such as Dijkstra's guarded commands [15], and in concurrent logic languages [24]. From a study of the literature, it is apparent that all (correct) parallel programs, in either paradigms, *do not fail*. The reason for this phenomenon is that a don't care interpretation of nondeterministic choices implies that all possible choices are admissible, i.e. any choice leads to some solution. This is particularly striking in the logic paradigm, where *don't know* nondeterministic programming is also supported, and is implemented by means of backtracking.

On the other hand, modern logic programming languages offer control mechanisms, such as delay declarations or some form of commitment at clause level, that allow to restrict atom selectability and/or clause selectability. In this way, the programmer (or the analysis phase of the compiler) provide a specification

of the control. Then the compiler is free to use any execution strategy that meets that specification, possibly together with any other optimization, such as program transformation, coroutining, or parallel execution. While for pure SLD-resolution the class of programs and queries that have successful derivations only is rather small, in some cases (as will be shown in the examples of this paper) the control mechanisms mentioned above are powerful enough to express a specification that leads to successful derivations only.

As a third, more technical, argument in favor of studying programs and queries with only successful derivations, we observe that it is possible to simplify the termination proofs in case there is no failed derivation. To see this point, consider, as an example, the following program, and the query `p`.

```
p ← q, p.
```

A left-to-right selection strategy results in a failure when `q` is selected. On the contrary, a right-to-left strategy produces an infinite derivation. As a consequence, in presence of failures, the termination behavior of a program depends on the particular selection rule adopted. We will show that, in absence of failures, termination w.r.t. a fixed selection rule implies that *every* derivation is finite and successful.

Our object of study is a generalization of SLD-resolution, which is parametric w.r.t. a function $\mathcal{R}$ mapping a program and an atom into either a set of clauses that are candidates for resolution or into the special value `delay`. The function $\mathcal{R}$ models restrictions on atom selectability (by assigning the value `delay`) and/or clause selectability (by mapping the selected atom into a proper subset of the program). Particular instances include modern logic programming languages which allow for restricting executions, e.g. to those respecting delay declarations as in the Gödel language [17], to those respecting block declarations as in Sicstus Prolog [18], to input-consuming derivations as described by Smaus [25], to guarded clause resolution as in the style of SLDG-resolution of Apt and Luitjes [3], and to combinations of them.

In the context of $\mathcal{R}$-SLD resolution, we offer a theoretical framework to reason about programs and queries such that *all* derivations for them are successful. We provide a characterization of those programs and queries which allow to reuse existing relations from the literature on termination and verification of Prolog programs.

The characterization is expressed for generic relations μ which are persistent along derivations, and for a μ-failure free program. μ-failure freedom requires that when $\mu(P, Q)$ holds then some atom in Q is *covered*, i.e. has a justification in the program. A μ-failure free program P and a query Q in the relation μ cannot have deadlocked or failed derivations via any selection rule s that selects covered atoms, if any. We call s a μ-failure free selection rule, and show that for μ-failure free programs, at least one such a rule exists. In addition, if we admit *termination* via s, no derivation of P and Q can be deadlocked or failed. Moreover, if we admit *bounded nondeterminism*, i.e. a maximum length to refutations, we can conclude that all derivations of P and Q are successful. The general framework is then instantiated with reference to the leftmost selection

rule as μ-failure free selection rule, and substantiated with some examples.

Plan of the paper

In the Preliminaries (Section 2) we recall some standard notation of logic programming and several characterizations concerning moding, typing and termination of logic programs. In Section 3, the generalization of SLD-resolution is introduced, together with several instances including derivations respecting delay declarations, input-consuming derivations, arithmetic built-in's, guarded resolution. The characterizations of programs and queries with successful derivations only is offered in Section 4, and instantiated on the leftmost selection rule in Section 5. Some examples are discussed as well. Finally, we compare related works in Section 6.

2 Preliminaries

We adhere to the standard notation of Apt [1] for basic notions such as the (first order) language L of programs and queries, terms, the set of ground terms U_L, atoms, the set of ground atoms B_L, the set of atoms $Atom_L$, Herbrand interpretations and models. $rel(A)$ is the predicate symbol of the atom A. A query $B_1, \ldots, B_n$ is a sequence of atoms. $\Box$ denotes the empty query. A clause is a formula $A \leftarrow Q$ where A is an atom and Q is a query. We write the clause as $A.$ when Q is empty, and call it a *fact* clause. A *program* is a finite set of clauses. Finally, we denote with $\mathcal{P}_L$ the set of all logic programs on language L.

2.1 SLD-derivation step

We recall from [1] the definition of SLD-derivation step. Let us write $mgu(A, H) = \theta$ if θ is the most general unifier (modulo renaming) of atoms A and H.

Definition 2.1 [SLD-derivation step] Consider a non-empty query $Q = A_1, \ldots, A_n$ and a clause c. Let $H \leftarrow B_1, \ldots, B_m$ be a variant of c variable disjoint with Q. Suppose that A_i, with $i \in [1, n]$, and H unify with $mgu(A_i, H) = \theta$. Then the query Q':

$$A_1\theta, \ldots, A_{i-1}\theta, B_1\theta, \ldots, B_m\theta, A_{i+1}\theta, \ldots, A_n\theta$$

is called the *SLD-resolvent* of Q and c w.r.t. A_i with an mgu θ. A_i is called the *selected atom* of Q, and $H \leftarrow B_1, \ldots, B_m$ is called the *input clause*. Also, we write:

$$Q \Rightarrow_c^{i,\theta} Q'$$

and call it an *SLD derivation step*. $\Box$

2.2 Modes

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

In the literature, several correctness criteria concerning the modes have been proposed. In this article, we need *simply* moded programs [2]. We say a sequence of terms is *linear* if each variable occurs at most once in the sequence.

Definition 2.2 [Simply-moding] 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 $i \in [1, n]$:

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

A query Q is *simply-moded* if the clause $q \leftarrow Q$ is simply-moded, where q is any variable-free atom. A program is simply-moded if all of its clauses are. □

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. In particular, every unit clause is simply-moded.

2.3 Well-typing

Among the proposed type systems for logic programs, we recall well-typing [11]. By following the presentation style of [4], a *type* is a non-empty set of terms closed under instantiation. Let us fix the notation for some types: U is the set of all terms; *Ground* is the set of ground terms; *List* is the set of lists, formally defined as: $List ::= \texttt{[]} \mid \texttt{[}U \texttt{ | } List\texttt{]}$; *Gae* is the set of all ground arithmetic expressions (gae's, in short), formally defined[1] as: $Gae ::= \texttt{n} \mid Gae \texttt{ bop } Gae \mid \texttt{uop } Gae$, where `n` is a numeric constant, `bop` is any binary arithmetic operator, and `uop` is any unary arithmetic operator; $List(Gae)$ is the set of lists of gae's, formally defined as: $List(Gae) ::= \texttt{[]} \mid \texttt{[}Gae \texttt{ | } List\texttt{]}$. We write $\mathbf{s} \in \mathbf{S}$ to denote that each term in the sequence $\mathbf{s}$ belongs to the type occurring at the same position in the sequence $\mathbf{S}$.

For a predicate p/n, a *type* is an atom $p(m_1 : t_1, \ldots, m_n : t_n)$, where $p(m_1, \ldots, m_n)$ is a mode for p/n and t_i is a type for $i \in [1, n]$. The t_i's occurring in input positions are called *input types*, otherwise *output types*. To simplify the notation, an atom written as $p(\mathbf{s} : \mathbf{S}, \mathbf{t} : \mathbf{T})$ means: $\mathbf{s}$ is the vector of terms filling in the input positions with types $\mathbf{S}$, and $\mathbf{t}$ is the vector of terms filling in the output positions with types $\mathbf{O}$.

[1] In case the division operator is admitted, the set *Gae* is assumed not to include division by 0, e.g. as in `3/0`.

Definition 2.3 [Well-typing] A clause c is *well-typed* if for every

$$p_0(\mathbf{o}_0 : \mathbf{O}_0, \mathbf{i}_{n+1} : \mathbf{I}_{n+1}) \leftarrow p_1(\mathbf{i}_1 : \mathbf{I}_1, \mathbf{o}_1 : \mathbf{O}_1), \ldots, p_n(\mathbf{i}_n : \mathbf{I}_n, \mathbf{o}_n : \mathbf{O}_n)$$

instance of c,

$$\text{for } i \in [1, n+1],\ \mathbf{o}_0 \in \mathbf{O}_0 \wedge \ \ldots\ \wedge \mathbf{o}_{i-1} \in \mathbf{O}_{i-1} \quad \textit{implies} \quad \mathbf{i}_i \in \mathbf{I}_i.$$

A query Q is *well-typed* if the clause $q \leftarrow Q$ is well-typed, where q is any variable-free atom. A program is *well-typed* if all of its clauses are. □

2.4 Well-assertedness

Types express monotonic (w.r.t. instantiation) properties of terms. In some cases, however, we need to express non-monotonic properties, such as "X is a variable" or "$T = [X \mid Xs]$ with X, Xs variables". We recall the method of Drabent and Małuszyński [16] in a simplified form (from [4]). A *specification* is a pair $(pre, post)$ where $pre, post \subseteq Atom_L$. A *valuation sequence* for a clause $c : A \leftarrow B_1, \ldots, B_n$ and an atom H is a sequence of substitutions $\rho_0, \ldots, \rho_n$ such that: (1) $vars(H) \cap vars(c) = \emptyset$; (2) $\rho_0 = mgu(A, H)$; (3) there exists $\sigma_1, \ldots, \sigma_n$ such that for every $i \in [1, n]$: $\rho_i = \rho_{i-1}\sigma_i$, $dom(\sigma_i) \subseteq vars(B_i\rho_{i-1})$ and $range(\sigma_i) \cap vars((B_1, \ldots, B_n)\rho_{i-1}) \subseteq vars(B_i\sigma_{i-1})$. Intuitively, a valuation sequence is an abstraction of the sequence of computed answer substitutions in a derivation of H using c as input clause at the first SLD-derivation step.

Definition 2.4 [Well-Assertedness] A clause $c : A \leftarrow B_1, \ldots, B_n$ is *well-asserted* w.r.t. $(pre, post)$ if for every $H \in Pre$ and every valuation sequence $\rho_0, \ldots, \rho_n$ for c and H:

$$\text{for } i \in [1, n]\ B_1\rho_1 \in post \wedge \ \ldots\ \wedge B_{i-1}\rho_{i-1} \in post \quad \textit{implies} \quad B_i\rho_{i-1} \in pre$$
$$B_1\rho_1 \in post \wedge \ \ldots\ \wedge B_n\rho_n \in post \quad \textit{implies} \quad A\rho_n \in post.$$

A query Q is *well-asserted* w.r.t. $(pre, post)$ if the clause $q \leftarrow Q$ is well-asserted w.r.t. $(pre \cup \{q\}, post \cup \{q\})$ where q is a fresh variable-free atom. A program is *well-asserted* w.r.t. $(pre, post)$ if all of its clauses are. □

2.5 Acceptability

Acceptability [5] is a sound characterization of programs and queries with only finite derivations via the leftmost selection rule. The characterization uses the well-known concepts of level mappings $|\ | : B_L \to N$, i.e. functions from the Herbrand base B_L to natural numbers, and Herbrand interpretation, i.e. subsets of B_L. Also, let us denote by $ground_L(c)$ the set of ground instances of clause c on language L.

Definition 2.5 [Acceptability] Let I be a Herbrand interpretation, and $|\ | : B_L \to N$ a level mapping. A clause c is *acceptable by* $|\ |$ *and* I if I is a model of c and for every $A \leftarrow B_1, \ldots, B_n$ in $ground_L(c)$:

$$\text{for } i \in [1, n] \quad I \models B_1, \ldots, B_{i-1} \quad \textit{implies} \quad |A| > |B_i|.$$

A query Q is *acceptable*[2] *by* $|\ |$ *and* I if there exists $k \in N$ such that for every ground instance $A_1 , \ldots , A_n$ of it:

$$\text{for } i \in [1,n] \quad I \models A_1, \ldots, A_{i-1} \quad \textit{implies} \quad k > |A_i|.$$

A program is *acceptable by* $|\ |$ *and* I if all of its clauses are. □

3 Generalizing SLD-resolution

In this section, we present a generalization of SLD-resolution which takes into account restrictions on atom and/or clause selectability.

3.1 $\mathcal{R}$-SLD resolution

Throughout this paper, we will consider functions $\mathcal{R}$ mapping a program P and an atom A into a subset of clauses of P that can be selected to resolve A. In order to model non-selectability of A, $\mathcal{R}(P, A)$ may assume the special value `delay`.

Assumption 3.1 *We denote by* $\mathcal{R}$ *(possibly superscripted) a function that maps a program* P *and an atom* A *either into a subset of* P *or into the special value* `delay`*. We assume* $\mathcal{R}$ *is invariant under renaming, i.e.* $\mathcal{R}(P, A) = \mathcal{R}(P, B)$ *for* B *variant of* A*.* □

We extend SLD-resolution by allowing a derivation step only when the input clause is admitted by the $\mathcal{R}$ function.

Definition 3.2 [$\mathcal{R}$-SLD derivation step] Consider a non-empty query $Q = A_1 , \ldots , A_n$. An $\mathcal{R}$*-SLD derivation step* is:

- a SLD-derivation step $Q \Rightarrow_c^{i,\theta} Q'$ for some $i \in [1,n]$ such that $c \in \mathcal{R}(P, A_i) \neq$ `delay`; in this case, we call A_i the *selected atom* and Q' an $\mathcal{R}$-SLD resolvent of P and Q;
- or a *failure* step $Q \Rightarrow {}^i\texttt{fail}$ for some $i \in [1,n]$ such that $\mathcal{R}(P, A_i) \neq$ `delay` and there is no $\mathcal{R}$-SLD resolvent of P and A_i: in this case, we call A_i the *selected atom*;
- or a *deadlock* step $Q \Rightarrow$ `deadlock` otherwise, namely if for every $i \in [1,n]$, $\mathcal{R}(P, A_i) =$ `delay`.

We write $Q \Rightarrow^{\mathcal{R}} Q'$ to denote an $\mathcal{R}$-SLD derivation step from Q into a state Q', where *states* are either queries, or the special symbols `fail` and `deadlock`. □

Derivations can now be defined as the maximal sequences of $\mathcal{R}$-SLD derivation steps.

[2] Apt and Pedreschi [5] call Q *bounded*.

Definition 3.3 [$\mathcal{R}$-SLD derivation] An $\mathcal{R}$-SLD derivation of a program P and a query Q_0 is a maximal sequence (possibly infinite)

$$Q_0 \Rightarrow^{\mathcal{R}} Q_1 \ldots Q_{n-1} \Rightarrow^{\mathcal{R}} Q_n \Rightarrow^{\mathcal{R}} \ldots$$

such that if $Q' \Rightarrow^{\mathcal{R}} Q''$ is a SLD-derivation step in the sequence then the input clause employed is variable disjoint from the set of variables occurring in Q_0 or in the mgu's used at earlier steps or in the input clauses used at earlier steps. □

Renaming apart of input clauses and the assumption that $\mathcal{R}$ is invariant under variable renaming allows us to reason about $\mathcal{R}$-derivations modulo variable renaming. Also, as one could expect, by defining $\mathcal{R}^{sld}(P, A) = P$, $\mathcal{R}^{sld}$-SLD resolution boils down to SLD-resolution.

Differently from SLD-derivations, finite $\mathcal{R}$-SLD derivations may be successful, failed, or *deadlocked*. Deadlock occurs when the $\mathcal{R}$ function labels every atom in the last query as unselectable.

Definition 3.4 Let ξ be a finite $\mathcal{R}$-SLD derivation of P and Q whose last state is Q'. We say that:

- ξ is *successful*, or that ξ is a *refutation*, if Q' is the empty query;
- ξ is *failed* if $Q' = \texttt{fail}$;
- ξ is *deadlocked* if $Q' = \texttt{deadlock}$. □

Starting from this definition, $\mathcal{R}$-SLD computed instances are defined in the standard way [1]. Since $\mathcal{R}$-SLD refutations are actually SLD-refutations, the SLD-resolution Soundness Theorem extends to $\mathcal{R}$-SLD resolution, i.e. every computed instance of P and Q is a logical consequence of P. However, since function $\mathcal{R}$ may cut some SLD-derivations a Completeness Theorem does not hold in general. Consider for instance, the function $\mathcal{R}(P, A) = \texttt{delay}$ for all P and A. $\mathcal{R}$ admits only deadlocked derivations.

It is worth noting that function $\mathcal{R}$ affects both clause selectability ($c \notin \mathcal{R}(P, A) \neq \texttt{delay}$ means c is not selectable) and atom selectability ($\mathcal{R}(P, A) = \texttt{delay}$ means A is not selectable). The definition of selection rules must then consider selectable atoms only.

Definition 3.5 [Selection rules] Let $INIT_P$ stand for the set of initial fragments of $\mathcal{R}$-SLD derivations of a program P in which the last state is a non-empty query Q such that $\mathcal{R}(P, A) \neq \texttt{delay}$ for some A in Q.

A *selection rule* is a function which, when applied to an element in $INIT_P$ yields an occurrence of an atom A in its last query such that $\mathcal{R}(P, A) \neq \texttt{delay}$.

An $\mathcal{R}$-SLD derivation of P and Q via a selection rule s is an $\mathcal{R}$-SLD derivation where the atoms selected are chosen accordingly to s.

The *leftmost selection rule* is the one that always selects in a query the leftmost atom A such that $\mathcal{R}(P, A) \neq \texttt{delay}$. □

Starting from this definition, the notion of $\mathcal{R}$-SLD tree is derivable in the usual way. Again, we observe that for $\mathcal{R}^{sld}$, the notions of selection rule and leftmost selection rule boil down to standard definitions.

3.2 Modeling extensions of SLD-resolution

We are now in the position to model several extensions of SLD-resolution in a uniform framework.

3.2.1 Delay declarations

Let us define:

$$\mathcal{R}^g(P, A) = \begin{cases} P & \text{if } A \in \mathit{delay\_decl}(P) \\ \texttt{delay} & \text{otherwise,} \end{cases}$$

where $\mathit{delay\_decl}(P)$ is a set of atoms closed under renaming and specified by means of some program annotation. We say that an atom A *respects its delay declarations* if $A \in \mathit{delay\_decl}(P)$.

$\mathcal{R}^g$-SLD resolution models the operational semantics of the Gödel programming language [17], where $\mathit{delay\_decl}(P)$ is specified by means of program annotations called *delay declarations* (a less expressive form of such annotations is available in Sicstus Prolog [18], where they are called *block declarations*). In this case, only atoms which are in $\mathit{delay\_decl}(P)$ can be selected. The formal semantics of Gödel does not specify any selection rule, but practical implementations usually adopt the leftmost one. The following program `Perm` for computing permutations of a list is often used as an example:

```
(p1)  perm([], []).
(p2)  perm([X|Xs], Ys) ←
          perm(Xs, Zs),
          delete(X, Ys, Zs).

(p3)  delete(X, [X|Y], Y).
(p4)  delete(X, [H|Y], [H|Z]) ←
          delete(X, Y, Z).
```

The following delay declarations:

```
DELAY perm(X, Y) UNTIL nonvar(X)
DELAY delete(X, Y, Z) UNTIL nonvar(Z)
```

specify the necessary conditions for an atom to be selected. In this example, `perm(`Xs`, `Ys`)` may start to compute a permutation if at least one element is present in the list Xs, and analogously for `delete(`X`, `Ys`, `Xs`)`.

Consider the query `perm([a,f(b),C], A)`. An $\mathcal{R}^g$-SLD tree of `Perm` and the query above is shown in Fig. 1. Selected atoms are underlined. For readability reasons, only some of the mgu's are reported. It is worth noting that all derivations are successful. Also, note that the circled query contains two atoms in it satisfying their respective delay declarations, i.e. delay declarations drive the computation but still allow some degree of freedom in the choice of the selection rule.

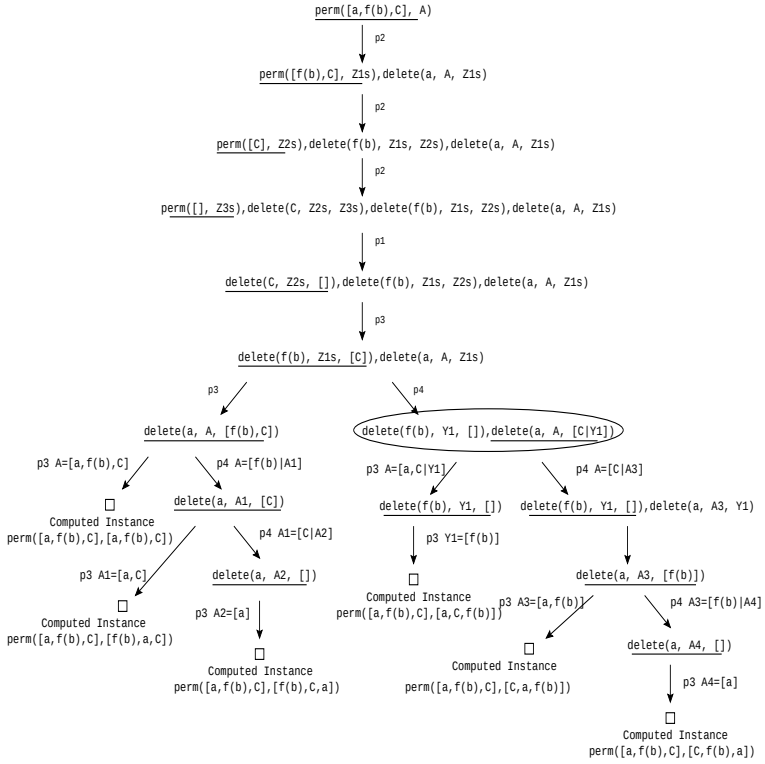

Figure 1: *An $\mathcal{R}^g$-SLD tree*

3.2.2 Input-consuming derivations

A mode for the predicates of program `Perm` is:

$$\texttt{perm}(I,\ O),\ \texttt{delete}(I,\ O,\ I)$$

denoting that the second position of `perm` (resp., `delete`) is *output*, while the first (resp., the first and the third) is *input*.

The notion of input-consuming derivation was introduced and motivated by Smaus [25] in order to model derivations that do not instantiate input positions of selected atoms. We say that $p(\mathbf{s},\ \mathbf{t})$ and a (standardized apart) clause c *ic-unify* if there exists an mgu θ of $p(\mathbf{s},\ \mathbf{t})$ and the head of c such that $dom(\theta) \cap var(\mathbf{s}) = \emptyset$. Also, let us introduce some useful notation:

$$\mathcal{IC}(P, A) = \{H \leftarrow B_1, \ldots, B_n \in P \mid A \text{ and } H \text{ ic-unify}\}$$

$$\mathcal{U}(P, A) = \{H \leftarrow B_1, \ldots, B_n \in P \mid \text{there exists } mgu(A, H)\}.$$

With these notations, input consuming derivations are modelled by $\mathcal{R}^{ic}$-SLD

resolution, where:

$$\mathcal{R}^{ic}(P,A) = \begin{cases} \mathcal{IC}(P,A) & \text{if } \mathcal{IC}(P,A) \neq \emptyset \\ \emptyset & \text{if } \mathcal{U}(P,A) = \emptyset \\ \texttt{delay} & \text{otherwise.} \end{cases}$$

Intuitively, if an atom A ic-unifies with some clause head, then an $\mathcal{R}^{ic}$-SLD resolvent exists. If A does not unify (and then it does not ic-unify) with any clause head, then no $\mathcal{R}^{ic}$-SLD resolvent of P and A can exist, i.e. we have failure. Otherwise, A does not ic-unify with any clause head but still unifies with some: we model[3] such a case by `delay`, since an $\mathcal{R}^{ic}$-SLD resolvent may exists, but it depends on further instantiation of the query where A appears.

A striking difference between $\mathcal{R}^{ic}$ and $\mathcal{R}^{g}$-SLD resolution is that the latter restricts atom selectability only, while the former restricts clause selectability as well. As an example, consider the `Perm` program and the query `delete(a, B, A)`. Since the atom in the query does not respect its delay declaration, it is not selectable, which implies that there is one deadlocked $\mathcal{R}^{g}$-SLD derivation. In contrast, the atom `delete(a, B, A)` ic-unifies with the head of the fact clause, hence there exists an $\mathcal{R}^{ic}$-SLD refutation with computed instance `delete(a, [a|A], A)`. For a more general comparison on the two resolution methods, we refer the reader to [9].

3.2.3 Arithmetic built-in's

We reason here on arithmetic built-in's by giving special meaning to some predefined predicate symbols, including arithmetic predicates `<, =<, =:=, =\=, >, >=` and `is`. We distinguish arithmetic atoms in clauses by writing:

$$A \leftarrow \mathbf{G}, B_1, \ldots, B_n$$

where $\mathbf{G}$ is the collection (not necessarily at the beginning of the body of the clause) of atoms with arithmetic predicates. Following Kunen [19], we assume that arithmetic predicates are declaratively defined by (infinite) sets of *ground* facts, which we implicitly assume to be part of the program. For instance, the built-in `<` is defined by:

$$M_{\texttt{<}} = \{x \texttt{ < } y. \mid x, y \in Gae \wedge val(x) < val(y)\},$$

where Gae is the set of ground arithmetic expressions (gae's for short), and $val(x)$ is the integer represented by the Gae x. The built-in `is` has a slightly different definition:

$$M_{\texttt{is}} = \{x \texttt{ is } y. \mid x \in Gaec, y \in Gae \wedge val(x) = val(y)\},$$

where $Gaec$ is the set of numerals {`0, 1, -1, 2, -2 ...`}. The union of all the sets M_{op}, where op is a built-in is denoted by M_{Ar}. From an operational point

[3]Note that our modeling is the same of [22], but slightly differs from the original definition of [25]. In the latter, it was not specified a name for the situation when for no A atom in a clause $\mathcal{IC}(P,A) \neq \emptyset$, since the main interest was in termination.

of view, atoms with predicates `<`, `=<`, `=:=`, `=\=`, `>` or `>=` can be interpreted only when their arguments are gae's, while atoms with predicate `is` can be interpreted when the right argument is a gae. In these cases, we say that the atoms are *correctly typed*. We model the *correctly typing* precondition by the following assumption.

Assumption 3.6 *We assume that every $\mathcal{R}$ function is always defined on arithmetic atoms as follows, for op in* { `<`, `=<`, `=:=`, `=\=`, `>`, `>=` }*:*

$$\mathcal{R}(P, x \ op \ y) = \begin{cases} M_{Ar} & \textit{if } x, y \textit{ are gae's} \\ \texttt{delay} & \textit{otherwise,} \end{cases}$$

and for op equal to `is`*:*

$$\mathcal{R}(P, x \ op \ y) = \begin{cases} M_{Ar} & \textit{if } y \textit{ is a gae} \\ \texttt{delay} & \textit{otherwise.} \end{cases}$$

□

With these preconditions, there is at most one $\mathcal{R}$-SLD resolvent for a given program and query when the atom selected is arithmetic. Finally, observe that the behavior of Prolog built-in's slightly differs from our semantics. A Prolog derivation ends into a *run-time error* when an arithmetic atom is selected such that it is not correctly typed. In contrast, in our framework, the atom cannot be selected at all (and this may lead to deadlock).

3.2.4 SLDG-resolution

Arithmetic built-in's can be used as *guards*, i.e. tests to control clause selectability. This use is customary in concurrent programming languages and in presence of don't care nondeterminism, where a clause is selected only under some conditions. Let us consider:

$$try(A, H \leftarrow \mathbf{G}, B_1, \ldots, B_n) = \begin{cases} \theta & \text{if } mgu(A, H) = \theta \\ & \wedge\ \mathbf{G}\theta \text{ correctly typed} \\ & \wedge\ M_{Ar} \models \exists \mathbf{G}\theta \\ suspend & \text{if } mgu(A, H) = \theta \\ & \wedge\ \mathbf{G}\theta \text{ not correctly typed} \\ fail & \text{otherwise.} \end{cases}$$

We say that A *guardedly unifies* with c when $try(A, c) = \theta$, i.e. A unifies with the head of c and the guard is correctly typed and satisfiable. If A and the head of c unify but the guard is not correctly typed, guarded unification is still possible after some other derivation steps, so $try(A, c) = suspend$. Finally, if A and the head of c do not unify, or they unify but the guard is correctly typed and unsatisfiable then $try(A, c) = fail$. Called $\mathcal{T}(P, A) = \{c \in P \mid try(A, c) = \theta\}$, we define:

$$\mathcal{R}^{sldg}(P, A) = \begin{cases} \mathcal{T}(P, A) & \text{if } \mathcal{T}(P, A) \neq \emptyset \\ \emptyset & \text{if } \forall\ c \in P.\ try(A, c) = fail \\ \texttt{delay} & \text{otherwise.} \end{cases}$$

In this case, arithmetic atoms in **G** (we call **G** the *guard*) restrict the applicability of a clause in a derivation step. A clause can be selected only if it guardedly unifies with the selected atom. If for every clause $c \in P$, $try(A, c) = fail$ then the selection of A leads to a failed derivation. Finally, if there is no clause that guardedly unifies with A, and at least one clause for which *try* returns *suspend* then the selection of A must be delayed.

The core of such a form of resolution, called SLDG resolution, was introduced by Apt and Luitjes [3]. Another related definition, which admits constraints in guards and does not consider typing, is studied by Maher [20] in the context of *committed-choice* logic programs.

3.2.5 G+SLDG-resolution

Consider now two functions $\mathcal{R}_1$ and $\mathcal{R}_2$. The function:

$$(\mathcal{R}_1 + \mathcal{R}_2)(P, A) = \begin{cases} \texttt{delay} & \text{if } \mathcal{R}_1(P, A) = \texttt{delay} \\ & \quad \text{or } \mathcal{R}_2(P, A) = \texttt{delay} \\ \mathcal{R}_1(P, A) \cap \mathcal{R}_2(P, A) & \text{otherwise,} \end{cases}$$

models the resolution method where an atom and a clause may resolve iff they may resolve both with respect to $\mathcal{R}_1$ and $\mathcal{R}_2$. As an example, the operational semantics of the function:

$$\mathcal{R}^{g+sldg}(P, A) = (\mathcal{R}^g + \mathcal{R}^{sldg})(P, A)$$

restricts clause applicability through guards, and atom selectability through delay declarations. The following program `Partition`, for partitioning a list of gae's, is a simple example:

```
DELAY part(X, Xs, Ls, Bs) UNTIL nonvar(Xs)

part(_,[],[],[]).
part(X,[Y|Xs],[Y|Ls],Bs)  ← X >= Y, part(X,Xs,Ls,Bs).
part(X,[Y|Xs],Ls,[Y|Bs])  ← X <= Y, part(X,Xs,Ls,Bs).
```

In the *concurrent* interpretation of logic programs [24], atoms model processes, shared logical variables model communication by means of multiparty channels, clauses model dynamic process activation, and queries model dynamic networks of asynchronous processes. In this context, $\mathcal{R}^g$ can be interpreted as modeling *synchronization conditions* on process activations (e.g., there is some input in a channel), while $\mathcal{R}^{sldg}$ can be interpreted as modeling *nondeterministic choices* (e.g., if the input is lower or equal than some value then send it to an output channel; otherwise send it to another output channel). Also, in this context deadlocked derivations model *deadlock computations* of the network of processes. As an example, given the query

```
part(4,[1,2,3,4,5,6],Ls,Bs),part(2,Ls,LLs,BBs)
```

`Ls` is a communication channel between the two atoms in the query; the delay declaration allows the right atom/process to read from `Ls` if it is non-empty; and the guards allow an atom/process to make a (non speculative) step that depends on current input.

4 Characterizing successful programs and queries

$\mathcal{R}$-SLD derivations can be infinite, successful, failed or deadlocked. In this section, we provide some sufficient conditions on programs and queries, which allow us to prove absence of infinite derivations, absence of deadlock, and absence of failed derivations. As a result, only successful derivations are admitted.

4.1 Persistent relations

Reasoning on programs and queries is usually done by abstracting the interesting properties (termination, correctness, call-patterns, etc.) by means of some intended relation $\mu(P, Q)$ over programs P and queries Q. Persistency of the intended relation yields a method for conducting inductive proofs of correctness, termination, etc., and has been shown for many analysis frameworks, including well-typing, acceptability and, under some restrictions, simply-moding.

Definition 4.1 [Persistent relations] A relation μ over programs and queries is a *persistent* relation w.r.t. $\mathcal{R}$ if when $\mu(P, Q)$ holds then $\mu(P, Q')$ holds for every $\mathcal{R}$-SLD resolvent Q' of P and Q. □

Since an $\mathcal{R}$-SLD resolvent is an SLD-resolvent, a relation which is persistent w.r.t. $\mathcal{R}^{sld}$ (i.e., along SLD-derivations) is persistent w.r.t. any $\mathcal{R}$.

4.2 Failure free programs and selection rules

We introduce next a notion useful to show absence of deadlock and, under some conditions, of failed derivations.

Definition 4.2 [Failure free programs] Let μ be a persistent relation w.r.t. $\mathcal{R}$, and P a program.

An atom A is *covered* (in P) if there exists $c \in \mathcal{R}(P, A) \neq \texttt{delay}$ such that A unifies with the head of a renaming of c variable disjoint with A.

P is *μ-failure free* if for every non-empty query Q such that $\mu(P, Q)$ holds, there exists a covered atom in Q. □

Intuitively, a program is μ-failure free if for every query Q in the intended relation μ there exists an $\mathcal{R}$-SLD resolvent of P and Q. As an example, P is μ-failure free w.r.t. $\mathcal{R}^{ic}$ if when $\mu(P, Q)$ holds there exists an atom in Q that ic-unifies with the head of some clause of P. As a first property of failure freedom, we observe that it prevents deadlocks.

Lemma 4.3 *Let μ be persistent relation w.r.t. $\mathcal{R}$, and P a μ-failure free program. If $\mu(P, Q)$ holds, then no $\mathcal{R}$-SLD derivation of P and Q is deadlocked.*

Proof. Assume a derivation deadlocks, i.e. $Q \Rightarrow^{\mathcal{R}} \ldots \Rightarrow^{\mathcal{R}} Q' \Rightarrow^{\mathcal{R}} \texttt{deadlock}$. Since μ is a persistent relation, $\mu(P, Q')$ holds. Since P is μ-failure free, there exists a covered atom A in Q', which implies $\mathcal{R}(P, A) \neq \texttt{delay}$, i.e. no deadlock. □

Even if some covered atom in Q exists, it is not necessarily the case a given selection rule selects a covered atom. Failure freedom ensures the existence of a μ-failure free selection rule, which is a rule that selects covered atoms only.

Definition 4.4 [μ-failure free selection rules] Let μ be a persistent relation w.r.t. $\mathcal{R}$. A selection rule s is *μ-failure free for a program P* if for every query Q such that $\mu(P, Q)$ holds, every atom A selected in a derivation of P and Q via s is covered. □

Lemma 4.5 *Let μ be a persistent relation w.r.t. $\mathcal{R}$ and P a μ-failure free program. Then there exists a selection rule which is μ-failure free for P.*

Proof. Consider a rule s that selects only covered atoms, if any. Under the assumption that P is μ-failure free, such a rule is μ-failure free for P. In fact, assume that $\mu(P, Q)$ holds, and let Q' be the last query of an element in $INIT_P$. Since μ is persistent, $\mu(P, Q')$ holds. Since P is μ-failure free, there exists a covered atom in Q'. Therefore, s can always select some covered atom. □

Note that the converse of the lemma does not hold. In fact, assume that $\mathcal{R}(P, A) = \texttt{delay}$ for all P and A; and that $\mu(P, Q)$ holds for all P and Q. Since every derivation of the empty program and any non-empty query is deadlocked at the first derivation step, there is no selected atom by any selection rule. Therefore, any selection rule is μ-failure free. However, the empty program is not μ-failure free.

As the name suggests, μ-failure free selection rules cannot lead to failure.

Lemma 4.6 *Let μ be a persistent relation w.r.t. $\mathcal{R}$, and s be a μ-failure free selection rule for P. If $\mu(P, Q)$ holds, then no $\mathcal{R}$-SLD derivation of P and Q via s is failed.*

Proof. Assume there is a failed derivation $Q \Rightarrow^{\mathcal{R}} \ldots \Rightarrow^{\mathcal{R}} Q' \Rightarrow^{\mathcal{R}} \texttt{fail}$. In particular, for the selected atom A in Q', we have $\mathcal{R}(P, A) = \emptyset$. However, since s is μ-failure free for P and $\mu(P, Q)$ holds, we also have that A is covered, which implies $\mathcal{R}(P, A) \neq \emptyset$. This is a contradiction. □

However, the conclusion does not hold for every selection rule.

Example 4.7 Consider the simple program `LeftRight`:

```
p ← q, p.
```

and the relation such that $\mu(P, Q)$ holds for $P =$ `LeftRight` and $Q =$ `q, ..., q, p`. It is easily checked that μ is persistent and `LeftRight` is μ-failure free. Also, the rightmost selection rule turns out to be μ-failure free. By Lemma 4.6, every $\mathcal{R}^{sld}$-SLD derivation of `LeftRight` and `p` via the rightmost selection rule is not failed. However, there exists a failed $\mathcal{R}^{sld}$-SLD derivation via the leftmost selection rule. □

Obviously, by showing that all $\mathcal{R}$-SLD derivation of P and Q via a μ-failure free selection rule s are finite, we can conclude that they are all successful.

4.3 Failure freedom w.r.t. all selection rules

Lemma 4.5 shows that for a μ-failure free program P there exists some μ-failure free selection rule s. For Q such that $\mu(P, Q)$ holds, Lemma 4.6 states that no $\mathcal{R}$-SLD derivation of P and Q via s can fail.

Under additional hypothesis, we can extend the same conclusion to any selection rule. Our results will consider functions $\mathcal{R}$ that are monotonic.

Definition 4.8 [Monotonicity of $\mathcal{R}$] We assume that for every atom A such that $\mathcal{R}(P, A) \neq \texttt{delay}$, if some instance A' of A is covered then A is covered. □

The $\mathcal{R}^{sld}, \mathcal{R}^{g}$, $\mathcal{R}^{ic}$, $\mathcal{R}^{sldg}$ and $\mathcal{R}^{g+sldg}$ functions are monotonic.

This is obvious for $\mathcal{R}^{sld}$, since A' covered means A' unifies with some clause head and then A unifies with the same head. The same result holds for $\mathcal{R}^{g}$.

Consider $\mathcal{R}^{ic}$: if $\mathcal{R}^{ic}(P, A) \neq \texttt{delay}$ then either A is covered or A unifies with no clause head. In the first case, we have the conclusion. In the second case, we have that neither A' unifies (and then ic-unifies) with any clause head, hence it cannot be covered.

Consider now $\mathcal{R}^{sldg}$. Let A' be covered. If $\mathcal{R}^{sldg}(P, A) \neq \texttt{delay}$ then either A is covered or $try(A, c) = fail$ for every $c \in P$. However, $try(A, c) = fail$ implies $try(A', c') = fail$, and then A' cannot be covered. Therefore, A is covered.

Finally, the results hold for $\mathcal{R}^{g+sldg}$ by observing that an atom is covered w.r.t. $\mathcal{R}^{g+sldg}$ iff it is covered w.r.t. both $\mathcal{R}^{g}$ and $\mathcal{R}^{sldg}$.

Example 4.9 Consider the non-monotonic pathological $\mathcal{R}^{gr}$ function defined as $\mathcal{R}^{gr}(P, A) = P$ if A is ground and $\mathcal{R}^{gr}(P, A) = \emptyset$ otherwise. The intended meaning of $\mathcal{R}^{gr}$ is to admit only ground derivations. However, a non-ground atom in a query leads to failure.

$\mathcal{R}^{gr}$ would become monotonic (and also less pathological) by defining $\mathcal{R}^{gr}(P, A)$ = `delay` for A not ground. In this case, a non-ground atom in a query leads to deadlock. □

Under the assumption of monotonicity, termination via a μ-failure free selection rule implies absence of failures for every derivation.

Definition 4.10 [Termination] Let μ be a persistent relation. A selection rule is *μ-terminating* if $\mu(P, Q)$ implies that every $\mathcal{R}$-SLD derivation of P and Q via s is finite. □

Lemma 4.11 *Let μ be a persistent relation w.r.t. a monotonic function $\mathcal{R}$.*

If there exists a selection rule that is both μ-failure free for a program P and μ-terminating, then for every query Q such that $\mu(P, Q)$ holds, no $\mathcal{R}$-SLD derivation of P and Q is failed.

Proof. Assume, on the contrary, a derivation $\xi = Q \Rightarrow^{\mathcal{R}} \ldots Q' \Rightarrow^{\mathcal{R}} \texttt{fail}$, and let A be the atom selected in Q'. We have that for every $c \in \mathcal{R}(P, A) \neq \texttt{delay}$, A and the head of c do not unify, which implies that A is not covered. We will

show a contradiction. Since μ is persistent, $\mu(P, Q')$ holds. Let s be a selection rule both μ-failure free for P and μ-terminating. By Lemmata 4.3 and 4.6, no derivation of P and Q' via s is deadlocked or failed. Moreover, there exists a finite derivation ξ' of P and Q' via s, which must then be successful. In ξ' a (further instantiated) occurrence of A is eventually selected. Let A' be such an (instantiated) occurrence. Since s is μ-failure free for P, A' is covered. Since $\mathcal{R}$ is monotonic, A is covered as well, which is a contradiction. □

4.4 Success w.r.t. all selection rules

Under the hypotheses of Lemma 4.11, we cannot conclude that all derivations are successful.

Example 4.12 Consider the following program w.r.t. $\mathcal{R}^{sldg}$-SLD resolution:

```
q(0).
p(X).
p(X) ← X < 2, q(Y), p(Y).
```

Let μ be a relation containing the program above with the empty query or with queries of the form: (1) `0 < 2, ..., 0 < 2, q(X), p(X)`; (2) `0 < 2, ..., 0 < 2, p(0)`; (3) `0 < 2, ..., 0 < 2, q(X)`; (4) `0 < 2, ..., 0 < 2`. μ is persistent w.r.t. $\mathcal{R}^{sldg}$: (1) may resolve in (1) or (2) or (3); (2) may resolve in (1), (2) or (4); (3) may resolve in (3) or (4); (4) may resolve in (4) or in the empty query. Also, we observe that the rightmost selection rule is μ-failure free and μ-terminating. However, there exists an infinite derivation via the leftmost selection rule for queries of the form (2). The reason lies in the fact that the rightmost selection rule prevents the selection of the recursive clause, which lead to infinite derivations via the leftmost selection rule. □

Intuitively, by requiring termination of all derivations we have the desired result. Actually, a weaker assumption is sufficient, and also necessary, which is known as bounded nondeterminism [21].

Definition 4.13 [Bounded nondeterminism] Let μ be a persistent relation w.r.t. $\mathcal{R}$. *μ implies bounded nondeterminism* w.r.t. $\mathcal{R}$ if for every $\mu(P, Q)$ there exists $k \in N$ such that every $\mathcal{R}$-SLD refutation of P and Q has length at most k. □

We are now in the position to characterize programs and queries with only successful derivations. Let us introduce the main result of this paper.

Theorem 4.14 *Let μ be a persistent relation w.r.t. a monotonic function $\mathcal{R}$, and P a program. The following are equivalent conditions:*

[A] *(i) P is μ-failure free,*

(ii) there exists a selection rule that is both μ-failure free for P and μ-terminating,

(iii) μ implies bounded nondeterminism.

[B] *(i) for every query Q such that $\mu(P, Q)$ holds, all $\mathcal{R}$-SLD derivation of P and Q are successful.*

[C] *(i) P is μ-failure free,*

(ii) for every query Q such that $\mu(P, Q)$ holds, all $\mathcal{R}$-SLD derivations of P and Q are finite.

Proof. **[A]** $\Rightarrow$ **[B]** Let P and Q such that $\mu(P, Q)$ holds, and consider an arbitrary $\mathcal{R}$-SLD derivation ξ of P and Q.

By Lemma 4.3, *(i)* implies that ξ is not deadlocked.

By Lemma 4.11, *(ii)* implies that ξ is not failed.

By showing that ξ is not infinite, we have that it must be successful, i.e. the conclusion. Assume the contrary, i.e. ξ is such that $Q = Q_0 \Rightarrow^{\mathcal{R}} \ldots \Rightarrow^{\mathcal{R}} Q_n \Rightarrow^{\mathcal{R}} \ldots$. Since μ is persistent, $\mu(P, Q_i)$ holds for $i \geq 0$. Let s be a selection rule such that *(ii)* holds. Since no derivation of P and Q_i via s is deadlocked or failed, and there exists a finite one, we conclude that there exists a successful derivation ξ' of P and Q_i via s. This implies that there exist $\mathcal{R}$-SLD refutations of P and Q of arbitrary length, which contradicts the hypothesis *(iii)* of bounded nondeterminism.

[B] $\Rightarrow$ **[C]**

(i) If $\mu(P, Q)$ holds for Q non-empty then there exists an $\mathcal{R}$-SLD resolvent of P and Q, since no derivation is failed or deadlocked. A fortiori, there exists a covered atom in Q.

(ii) Trivial.

[C] $\Rightarrow$ **[A]**

(i) Trivial.

(ii) By *(i)* and Lemma 4.5, there exists a a selection rule s which is μ-failure free for P. Since for every P and Q such that $\mu(P, Q)$, all derivations of P and Q via s are finite, we conclude that s is also μ-terminating.

(iii) Let P and Q be a program and a query such that $\mu(P, Q)$ holds. Consider the maximal tree where the root is labelled with Q, and such that Q'' is a child of Q' iff $Q' \Rightarrow^{\mathcal{R}} Q''$ holds. Such a tree is finitely branching, since P is a finite set of clauses or, for arithmetical built-in's, the $\mathcal{R}$ function ensures finitely many resolvents. Moreover, since all derivations of P and Q are finite, all paths in the tree must be finite. By Köning's lemma the tree is finite, and hence there exists $k \in N$ such that every $\mathcal{R}$-SLD derivation (and, a fortiori, refutation) of P and Q has length at most k. □

For pure SLD-resolution, i.e. $\mathcal{R}^{sld}$-SLD resolution, condition **[A]** can be simplified. Due to Strong Completeness of SLD-resolution, termination via any selection rule (i.e. **[A]***(ii)*) implies bounded nondeterminism (i.e. **[A]***(iii)*).

4.5 Success w.r.t. some selection rule

We observe that the requirement that μ is a persistent relation is essential in Theorem 4.14.

Example 4.15 Consider the relation $\mu(P, Q)$ requiring that P and Q are simply-moded. Note that μ is not persistent w.r.t. $\mathcal{R}^{sld}$ [3]. Let P be the simple program:

```
q(b).
p(a).
p(X).
```

with modes `q(`O`), p(`I`)`, and let Q be `q(X), p(X)`. Both P and Q are simply moded. Also, P is μ-failure free (note that the mode `q(`O`)` implies that X is a variable in `q(`X`)`) and the leftmost selection rule is μ-terminating.

However, there exists a failed $\mathcal{R}^{sld}$-SLD derivation via the rightmost selection rule `q(X), p(X)` $\Rightarrow$ `q(a)` $\Rightarrow$ `fail`. Therefore, **[B]** does not hold. □

In the example above, the best result we can state is that if $\mu(P, Q')$ holds then all $\mathcal{R}^{sld}$-SLD derivations of P and Q' via the leftmost selection rule are successful. This is possible since simply-moding is persistent in a weak sense, i.e. along derivations via the leftmost selection rule.

Definition 4.16 [Weak persistent relations] A relation μ over programs and queries is *weak persistent via a selection rule s* and w.r.t. $\mathcal{R}$ if when $\mu(P, Q)$ holds for a query Q then $\mu(P, Q')$ holds for every query Q' in an $\mathcal{R}$-SLD derivation of P and Q via s. □

With a small abuse of notation, we say that a selection rule is μ-failure free and a program is μ-terminating via s also when μ is a weak persistent relation.

Theorem 4.17 *Let μ be a relation weak persistent via a selection rule s and a function $\mathcal{R}$.*

If s is both μ-failure free for a program P and μ-terminating, then for every query Q such that $\mu(P, Q)$ holds, all $\mathcal{R}$-SLD derivations of P and Q via s are successful.

Proof. Lemma 4.3 can be restated for weak persistent relations. Also, Lemma 4.6 holds for weak persistent relations. This implies that no $\mathcal{R}$-SLD derivation of P and Q via s can be deadlocked or failed or infinite. Therefore, all of them must be successful. □

5 The leftmost as a μ-failure free selection rule

5.1 Success w.r.t. all selection rules

We refine the notion of persistent relations to left-persistent relations.

Definition 5.1 [Left-persistent relations] A persistent relation μ w.r.t. $\mathcal{R}$ is *left-persistent* w.r.t. $\mathcal{R}$ if for every program P and non-empty query Q such that $\mu(P, Q)$ holds, $\mu(P, A)$ holds for the leftmost atom A in Q. □

The next lemma simplifies the definition of failure freedom for left-persistent relations.

Lemma 5.2 *Let μ be a left-persistent relation w.r.t. $\mathcal{R}$.*

A program P is μ-failure free iff for every atom A such that $\mu(P, A)$ holds, A is covered in P.

Proof. The only-if part is immediate. Consider now the if-part. Let Q be a non-empty query such that $\mu(P, Q)$ holds. Since μ is left-persistent, $\mu(P, A)$ holds for the leftmost atom A in Q. Therefore, there exists a covered atom in Q, namely A. □

For left-persistent relations, a natural candidate as failure free selection rule is the leftmost one. We recall that such a rule selects the leftmost atom A in a query among those such that $\mathcal{R}(P, A) \neq \texttt{delay}$.

Lemma 5.3 *Let μ be a left-persistent relation w.r.t. $\mathcal{R}$ and P a μ-failure free program. Then the leftmost selection rule is μ-failure free for P.*

Proof. Let ξ be a $\mathcal{R}$-SLD derivation of P and a query Q, where $\mu(P, Q)$ holds. Since μ is persistent, $\mu(P, Q')$ holds for the last query Q' in ξ. Since μ is left-persistent, $\mu(P, A)$ holds for the leftmost atom A in Q'. Finally, since P is μ-failure free, A is covered. This implies $\mathcal{R}(P, A) \neq \texttt{delay}$, hence the leftmost selection rule selects a covered atom, namely A. □

From the proof of the lemma, we can conclude that the leftmost selection rule actually selects the leftmost atom in a query. A derivation via such a rule is then a derivation[4] via the leftmost selection rule w.r.t. $\mathcal{R}^{sld}$. A sound termination characterization for such derivations is acceptability. Moreover, acceptability is also a sound method for bounded nondeterminism [22].

Corollary 5.4 *Let μ be a left-persistent relation w.r.t. a monotonic function $\mathcal{R}$, P a μ-failure free program, and Q a query such that $\mu(P, Q)$ holds.*

If P and Q are acceptable by some $|\ |$ and I, then all $\mathcal{R}$-SLD derivations of P and Q are successful.

Proof. Consider the relation:

$$\mu' = \mu \cap \{(P', Q') \mid P' \text{ and } Q' \text{ acceptable by } |\ | \text{ and } I\}.$$

We claim that condition **[A]** of Theorem 4.14 holds for relation μ'. By showing **[A]**, we have **[B]** which implies our conclusion.

First, we observe that μ' is left-persistent since both μ and acceptability are left-persistent. Persistency of acceptability w.r.t. $\mathcal{R}^{sld}$ (and then w.r.t. any $\mathcal{R}$ function) is shown in [23, Lemma 2.3.12]. Let us show conditions **[A]** *(i-iii)*.

[4]The converse is not necessarily true, since the $\mathcal{R}$ function may prevent using some clause of P at some derivation step.

(i). Consider a non-empty query Q such that $\mu'(P,Q)$ holds. Then $\mu(P,Q)$ holds. Since P is μ-failure free, there exists a covered atom in Q. This shows that P is μ'-failure free.

(ii). The leftmost selection rule is μ-failure free for P as shown in Lemma 5.3. Since $\mu' \subseteq \mu$, the leftmost selection rule is also μ'-failure free for P. Moreover, as shown in the proof of Lemma 5.3, the leftmost selection rule always selects the leftmost atom. Since acceptability of P and Q implies termination of derivations where the leftmost atom is always selected [5], we conclude that the leftmost selection rule is also μ'-terminating.

(iii). If $\mu'(P,Q)$ holds then acceptability of P and Q implies termination of P and Q via the leftmost selection rule w.r.t. $\mathcal{R}^{sld}$. By Strong Completeness of SLD-resolution, every $\mathcal{R}^{sld}$-SLD refutation (and then $\mathcal{R}$-SLD refutation) of P and Q is of the same length of some refutation via the leftmost selection rule. Therefore, the length of any $\mathcal{R}$-SLD refutation of P and Q is bounded by some $k \in N$. Summarizing, μ' implies bounded nondeterminism. □

Since, in general, acceptability is a sound but not complete characterization of termination and bounded nondeterminism, the result above is not a complete characterization of programs and queries that have only successful derivations. Even in the case we would have a sound and complete characterization, however, there would be an intrinsic limit with using left-persistent relations.

Definition 5.5 $\mu_0(P,Q)$ holds iff for every non-empty query Q' in an $\mathcal{R}$-SLD derivation of P and Q, the leftmost atom in Q' is covered. □

It is easy to check that μ_0 is the largest left-persistent relation μ such that a program is μ-failure free.

On the positive side, this implies that Corollary 5.4 is suitable to show that *Prolog* programs and queries (for which several left-persistent relations apply, as will be shown later) can be executed via more advanced execution strategies (e.g., don't care nondeterminism, coroutining, parallelism) without incurring into deadlocks, failures, speculative parallelism.

On the negative side, since the method relies on left-persistent relations, μ_0 expresses the fact that programs and queries that can be reasoned about have a natural "left-to-right" or "pipeline" dataflow.

The conclusion of Corollary 5.4 can be slightly generalized to the case that the hypotheses hold for a permutation of the given program.

Definition 5.6 [Permutation] A permutation (for programs) Π_p is a function mapping a program P into a program obtained by permuting atoms in clause bodies of P. A permutation (for queries) Π_q is a function mapping a query Q into a query which is obtained by permuting atoms in it. □

Corollary 5.7 *Let μ be a left-persistent relation w.r.t. a monotonic function $\mathcal{R}$, Π_p a permutation for programs, and Π_q a permutation for queries.*

For a program P and a query Q, assume that $\Pi_p(P)$ is μ-failure free and that $\mu(\Pi_p(P), \Pi_q(Q))$ holds. If $\Pi_p(P)$ and $\Pi_q(Q)$ are acceptable by some $|\ |$ and I, then every $\mathcal{R}$-SLD derivation of P and Q is successful.

Proof. The conclusion follows from Corollary 5.4 by noting that there is a natural one-to-one mapping between $\mathcal{R}$-SLD derivations of P and Q and $\mathcal{R}$-SLD derivations of $\Pi_p(P)$ and $\Pi_q(Q)$. □

5.2 Success w.r.t. the leftmost selection rule

Let us now merge weak-persistent and left-persistent relations.

Definition 5.8 [Weak left-persistent relations] Consider a relation μ which is weak persistent via the leftmost selection rule and w.r.t. $\mathcal{R}$.

μ is *weak left-persistent* w.r.t. $\mathcal{R}$ if for every program P and non-empty query Q such that $\mu(P, Q)$ holds, $\mu(P, A)$ holds for the leftmost atom A in Q. □

As for left-persistent relations, μ-failure freedom can be simplified.

Lemma 5.9 *Let μ be a weak left-persistent relation w.r.t. $\mathcal{R}$.*

A program P is μ-failure free iff for every atom A such that $\mu(P, A)$ holds, A is covered in P. □

Also, the leftmost selection rule turns out to be μ-failure free for weak left-persistent relations and μ-failure free programs.

Lemma 5.10 *Let μ be a weak left-persistent relation w.r.t. $\mathcal{R}$ and P a μ-failure free program. Then the leftmost selection rule is μ-failure free for P.*

Proof. Let ξ be a $\mathcal{R}$-SLD derivation of P and a query Q via the leftmost selection rule, where $\mu(P, Q)$ holds. Since μ is weak persistent via the leftmost selection rule, $\mu(P, Q')$ holds for the last query Q' in ξ. Since μ is weak left-persistent, $\mu(P, A)$ holds for the leftmost atom A in Q'. Finally, since P is μ-failure free, A is covered. This implies $\mathcal{R}(P, A) \neq \texttt{delay}$, hence the leftmost selection rule selects a covered atom, namely A. □

We are now in the position to instantiate Theorem 4.17 for the leftmost selection rule.

Corollary 5.11 *Let μ be a weak left-persistent relation w.r.t. a monotonic function $\mathcal{R}$, P a μ-failure free program, and Q a query such that $\mu(P, Q)$ holds.*

If P and Q are acceptable by some $|\ |$ and I, then all $\mathcal{R}$-SLD derivations of P and Q via the leftmost selection rule are successful.

Proof. Consider the relation:

$$\mu' = \mu \cap \{(P', Q') \mid P' \text{ and } Q' \text{ acceptable by } |\ | \text{ and } I\}.$$

We claim that hypotheses of Theorem 4.17 hold when considering the leftmost selection rule. Weak left-persistency of μ and acceptability imply that μ' is weak persistent via the leftmost selection rule. Also, by Lemma 5.10, the leftmost selection rule is μ-failure free for P, and then μ'-failure free for P. Finally,

if $\mu'(P, Q)$ holds then acceptability of P and Q implies termination via the leftmost selection rule, i.e. such a rule is μ-terminating.

By Theorem 4.17, for every P and Q such that $\mu(P, Q)$ holds and they are acceptable, we can conclude that all $\mathcal{R}$-SLD derivations of P and Q via the leftmost selection rule are successful. □

5.3 A methodology for proving success

We outline below the general strategy for applying Corollaries 5.4 and 5.11 to prove that a program P and a query Q have only successful $\mathcal{R}$-SLD derivations:

step (1) select a relation μ such that $\mu(P, Q)$ holds; μ can be selected from a repertoire of (weak) left-persistent relations w.r.t. $\mathcal{R}$;

step (2) show that P is μ-failure free w.r.t. $\mathcal{R}$;

step (3) show that P and Q are acceptable by a same level mapping and Herbrand interpretation.

The main problem we are now faced is to find out an initial repertoire of (weak) left-persistent relations μ to use at *step (1)*.

We discuss some proposals in the next subsections. The relations presented are obtained by combining well-known left-persistent relations such as those recalled in the Preliminaries. After introducing them, we present some example programs and queries with successful derivations only.

5.3.1 Well-typing is not practical

As a first approximation, consider the following relation.

Definition 5.12 $\mu_1(P, Q)$ holds iff P and Q are well-typed. □

Apt and Luitjes [3] showed that μ_1 is a left-persistent relation w.r.t. $\mathcal{R}^{sld}$, and then w.r.t. every function $\mathcal{R}$. However, μ_1 is too weak to be useful in practice or, in other words, very few programs and queries can be reasoned about using μ_1. Suppose to be in the hypotheses of Corollary 5.4.

Let P be a program that is μ_1-failure free and acceptable by some $|\ |$ and I. Moreover, let A be a well-typed atom which is also acceptable by $|\ |$ and I. By Corollary 5.4, we conclude that all $\mathcal{R}$-SLD derivations of P and A are successful. Since types and acceptability are closed under instantiation, the hypotheses above hold for every instance A' of A. This implies that every ground instance of A admits a refutation. By Soundness of ($\mathcal{R}$-)SLD resolution, the types of A must be accurate enough to describe all atoms that are true in the least Herbrand model of P. This is not the usual meaning of types, which are instead used to circumscribe the set of intended queries.

As an example, the `Perm` program is well-typed with types:

`perm(` *I*: *List*`,` *O*: *List* `)`, `delete(` *I*: *U*`,` *O*: *List*`,` *I*: *List*`).`

In order to check that `Perm` is μ_1-failure free, we have to show that for every `perm(`In`,` Out`)` with In list and Out any term, `perm(`In`,` Out`)` unifies with the head of some clause from `Perm`. Clearly, this is not true for all terms Out.

5.3.2 Well-typing and simply-moding for $\mathcal{R}^{ic}$ and $\mathcal{R}^{g}$

Consider a well-typed atom $p(\mathbf{i{:}I}, \mathbf{o{:}O})$. As already pointed out, we cannot assume that for every input and output values there exists a refutation. A more reasonable assumption is, instead, to require that for every input value there exists *some* output value. This idea is made clear by requiring simply-moding, which ensures that **o** is a linear sequence of variables.

Definition 5.13 $\mu_2(P, Q)$ holds iff P and Q are well-typed and simply-moded, and, among the arithmetic predicates, only `is` may appear in P or Q. □

Observe that we excluded arithmetic atoms (apart from `is`) from appearing in P and Q. Otherwise, consider an arithmetic predicate, e.g. `>`. In order to show failure freedom, a well-typed and simply-moded atom x `>` y must be such that $x, y \in Gae$ and $val(x) > val(y)$. However, such a condition cannot be shown using types, since the type of a predicate argument is not related to the type of another argument (they are *independent* types).

Nevertheless, we observe that the presence of arithmetic atoms (apart from `is`-atoms) is useless if all $\mathcal{R}^{ic}$ or $\mathcal{R}^{g}$-derivations of P and Q are successful. In fact, since those atoms are ground, they can be removed (from the program and the query) without affecting the result (i.e., computed instances, set of derivations, final state of a derivation).

Also, note that that for the predicate `is`: (1) the only meaningful type is $O : Gae$ `is` $I : Gae$; (2) with such a type $M_{\texttt{is}}$ is well-typed; (3) with such a type, if $\mu_2(P, T_1$ `is` $T_2)$ holds then T_1 `is` T_2 is covered w.r.t. any $\mathcal{R}$.

Lemma 5.14 *Assume the type of* `is` *equal to* $O : Gae$ `is` $I : Gae$.
μ_2 *is a left-persistent relation w.r.t* $\mathcal{R}^{ic}$.

Proof. Well-typing has been shown to be (left-)persistent w.r.t. $\mathcal{R}^{sld}$ by Apt and Luitjes [3, Lemma 23]. This holds for predicate `is` as well, since $M_{\texttt{is}}$ is well-typed with the given type of `is`. As a consequence, well-typing is left-persistent w.r.t. $\mathcal{R}^{ic}$. Similarly, Apt and Luitjes [3, Lemma 30] showed that the SLD resolvent of a simply moded program and a simply moded query is simply moded when the selected atom and head of the input clause ic-unify. This is the case for $\mathcal{R}^{ic}$-SLD resolvents when the selected atom is not `is`. In case an atom T_1 `is` T_2 is selected in a simply-moded query, we observe that: (1) the $\mathcal{R}$ functions require $T_2 \in Gae$; (2) the given mode of `is` imply T_1 is a variable. These observations imply that T_1 `is` T_2 and the input clause head ic-unify. Therefore, the $\mathcal{R}^{ic}$-SLD resolvent is simply-moded. □

A similar result holds for $\mathcal{R}^{g}$, under the additional hypothesis that the delay declarations imply matching, a notion borrowed from Apt and Luitjes [3].

Definition 5.15 We say that the delay declarations for a program P *imply matching* if for every non-arithmetic atom $A = p(\mathbf{i}, \mathbf{o})$ such that $A \in delay\_decl(P)$ and for every $B = p(\mathbf{i}', \mathbf{o}')$, head of a renaming of a clause from P which is disjoint with A, if A and B unify, then $\mathbf{i}$ is an instance of $\mathbf{i}'$. □

Using this notion, we introduce a left-persistent relation for $\mathcal{R}^g$.

Definition 5.16 We say that $\mu_3(P, Q)$ holds iff $\mu_2(P, Q)$ holds and the delay declarations of P imply matching.

Lemma 5.17 *Assume the type of* `is` *equal to* $O : Gae$ `is` $I : Gae$.
μ_3 *is a left-persistent relation w.r.t* $\mathcal{R}^g$.

Proof. The proof is similar to the one of Lemma 5.14 for well-typing, and for simply-moding when the select atom has predicate `is`. Consider now a query Q and an $\mathcal{R}^g$-SLD resolvent Q' of P and Q. Let A be the atom selected in Q. We have that $A \in delay\_decl(P)$, which, in turn, implies that the input part of A is an instance of the input part of the head of the input clause. By [3, Lemma 30], the $\mathcal{R}$-SLD resolvent (and then the $\mathcal{R}^g$-SLD resolvent) of P and Q is simply-moded. □

5.3.3 Well-typing and simply-moding for $\mathcal{R}^{g+sldg}$

While arithmetic predicates are used as tests (and may lead to failure) w.r.t. $\mathcal{R}^{ic}$ and $\mathcal{R}^g$-SLD resolution, they are used are "clause selectors" (and cannot lead to failure) w.r.t. $\mathcal{R}^{sldg}$ and $\mathcal{R}^{g+sldg}$-SLD resolution. We next refine relation μ_3 into a relation that is left-persistent w.r.t. $\mathcal{R}^{g+sldg}$.

Definition 5.18 We say that $\mu_4(P, Q)$ holds iff P and Q are well-typed and simply-moded, the delay declarations of P imply matching, and for every arithmetic atom A in Q with $rel(A) \neq$ `is`, we have $A \in M_{Ar}$. □

Compared to μ_3, we now admit arithmetic predicates other than `is`, but only by requiring that they must succeed. The next lemma show persistency of μ_4 w.r.t. $\mathcal{R}^{g+sldg}$-SLD resolution. In particular, it states that for a query with no arithmetic atom, then no $\mathcal{R}^{g+sldg}$-SLD derivation may fail because of an arithmetic atom.

Lemma 5.19 *Assume the type of* `is` *equal to* $O : Gae$ `is` $I : Gae$.
μ_4 *is a left-persistent relation w.r.t* $\mathcal{R}^{g+sldg}$.

Proof. Assume that $\mu_4(P, Q)$ holds. We distinguish two cases.

If an arithmetic atom A is selected with $rel(A) \neq$ `is`, we have $A \in M_{Ar}$. Therefore, the $\mathcal{R}^{g+sldg}$-SLD resolvent is the query Q' equal to Q with A removed. Since A is ground, $\mu_4(P, Q)$ implies then $\mu_4(P, Q')$.

Otherwise, we reason as in the proof of Lemma 5.17 to get that the $\mathcal{R}^{g+sldg}$-SLD resolvent Q' is well-typed and simply-moded. Finally, consider an arithmetic atom A in Q' with $rel(A) \neq$ `is`. If A is also in Q then $\mu_4(P, Q)$ implies

$A \in M_{Ar}$. Otherwise, A is (an instance of an atom) in the guard of the input clause. By definition of $\mathcal{R}^{g+sldg}$-SLD resolution, this implies $A \in M_{Ar}$. Summarizing, $\mu_4(P, Q')$ holds. □

Finally, observe that for any arithmetic atom A such that $\mu_4(P, A)$ holds the requirements of μ_4-failure freedom are trivially satisfied. Therefore, μ_4-failure freedom has to be shown only for non-arithmetic atoms.

5.4 Weak left-persistent

Weak left-persistent relations are required to be persistent along $\mathcal{R}$-SLD derivations via the leftmost selection rule. Obviously, a left-persistent relation is weak left-persistent as well. In this section, we then recall some weak left-persistent relations which are not left-persistent.

5.4.1 Well-typing and simply-moding for $\mathcal{R}^{sld}$ and $\mathcal{R}^{sldg}$

In Section 5.3.2, we have shown that μ_2 is left-persistent w.r.t. $\mathcal{R}^{ic}$ and, under some conditions, w.r.t. $\mathcal{R}^{g}$. However, μ_2 is not left-persistent w.r.t. $\mathcal{R}^{sld}$, since an $\mathcal{R}^{sld}$-SLD resolvent of a simply-moded program and query is not necessarily simply moded [3]. However, Apt and Etalle [2] showed that the $\mathcal{R}^{sld}$-SLD resolvent is simply-moded if the leftmost selection rule is assumed.

Definition 5.20 We say that $\mu_5(P, Q)$ holds iff P and Q are well-typed and simply-moded, and for every arithmetic atom A in Q with $rel(A) \neq$ `is`, we have $A \in M_{Ar}$. □

Lemma 5.21 *Assume the type of* `is` *equal to* $O : Gaec$ `is` $I : Gae$ *and the type of* $op \in \{$ `<, =<, =:=, =\=, >, >=` $\}$ *equal to* $I : Gae\ op\ I : Gae$.
μ_2 *is a weak left-persistent relation w.r.t* $\mathcal{R}^{sld}$.
μ_5 *is a weak left-persistent relation w.r.t* $\mathcal{R}^{sldg}$.

Proof. Consider first μ_2. Assume $\mu_2(P, Q)$, for a non-empty query Q. If the leftmost atom in Q is not an arithmetic one, then the result follows since well-typing is left-persistent and simply-moding is weak left-persistent [2, Lemma 27]. If the leftmost atom A in Q is an arithmetic one, then it is well-typed, which implies $\mathcal{R}^{sld}(P, A) \neq$ `delay`. Therefore, it is selected by the leftmost selection rule. As before, the $\mathcal{R}^{sld}$-SLD resolvent is well-typed and simply moded since the set of clauses defining arithmetic atoms are well-typed and simply-moded with the types assumed by hypothesis.

Consider now μ_5. Assume $\mu_5(P, Q)$, for a non-empty query Q.

If $Q = A, Q'$ for A not an arithmetic atom, then the ($\mathcal{R}^{sldg}$-)SLD resolvent Q'' via the leftmost selection rule is well-typed and simply moded. Consider now an arithmetic atom A in the resolvent, with $rel(A) \neq$ `is`. If A was present in Q then $A \in M_{Ar}$ since $\mu_5(P, Q)$ holds. Otherwise, $A \in M_{Ar}$ by definition of $\mathcal{R}^{sldg}$. Summarizing, $\mu_5(P, Q'')$ holds.

If $Q = A, Q'$ for A arithmetic atom, we distinguish two cases. If $rel(A) \neq$ `is`, then the $\mathcal{R}^{sldg}$-SLD resolvent via the leftmost selection rule is necessarily

Q', which implies $\mu_5(P, Q')$ holds. If $A = T_1$ `is` T_2, then, since Q is well-typed and simply-moded, T_1 is a variable and $T_2 \in Gae$. This implies that A is the leftmost atom in Q such that $\mathcal{R}^{sldg}(P, A) \neq$ `delay` and then, it is selected via the leftmost selection rule. Since the clauses defining `is` are facts, they are simply-moded. As before, the $\mathcal{R}^{sldg}$-SLD resolvent of Q'' of P and Q via the leftmost selection rule is well-typed and simply-moded. Also, since $\mu_5(P, Q)$ holds and an arithmetic atom B in Q'' with $rel(B) \neq$ `is` appears in Q as well, we conclude that $B \in M_{Ar}$. Summarizing, $\mu_5(P, Q'')$ holds. □

5.4.2 Well-assertedness for $\mathcal{R}^{sld}$

We propose a weak left-persistent relation which does not make use of simply-moding.

Definition 5.22 We say that $\mu_6(P, Q)$ holds (w.r.t. $(pre, post)$) iff P and Q are well-asserted (w.r.t. $(pre, post)$), and, among the arithmetic predicates, only `is` may appear in P or Q. □

Lemma 5.23 *Let $(pre, post)$ be a specification such that:*

$$T_1 \texttt{ is } T_2 \in pre \text{ iff } T_1 \text{ is a variable } \wedge T_2 \in Gae$$
$$T_1 \texttt{ is } T_2 \in post \text{ iff } T_1 \texttt{ is } T_2 \in M_{\texttt{is}}$$

μ_6 is a weak left-persistent relation w.r.t $\mathcal{R}^{sld}$.

Proof. Assume $\mu_6(P, Q)$ for a non-empty query Q and let A be the leftmost atom in Q. If $rel(A) \neq$ `is`, then the result follows since the $\mathcal{R}^{sld}$-SLD resolvent of a well-asserted program and query via the leftmost selection rule is well-asserted [4]. If $A = T_1$ `is` T_2, then, since $A \in pre$ by well-assertedness of Q, we have that $T_2 \in Gae$, which implies $\mathcal{R}^{sld}(P, Q) \neq$ `delay`. Therefore, A is selected by the leftmost selection rule. As before, the $\mathcal{R}^{sld}$-SLD resolvent is well-asserted since $M_{\texttt{is}} \subseteq post$ implies that the set of clauses defining arithmetic atoms are well-asserted. □

5.5 Examples

Permutations w.r.t. $\mathcal{R}^{ic}$

Let us show the hypotheses of Corollary 5.4 for the program `Perm` and the query $Q =$ `perm(`Xs`, A)` w.r.t. $\mathcal{R}^{ic}$-SLD resolution, where Xs is any list.

We will use the relation μ_2, which is left-persistent w.r.t. $\mathcal{R}^{ic}$. Below we repeat the program for convenience.

```
(p1)  perm([], []).
(p2)  perm([X|Xs], Ys) ←
          perm(Xs, Zs),
          delete(X, Ys, Zs).

(p3)  delete(X, [X|Y], Y).
(p4)  delete(X, [H|Y], [H|Z]) ←
          delete(X, Y, Z).
```

`Perm` and the query Q are well-typed and simply-moded with the types:

$$\texttt{perm(}\ I\texttt{:}\ List\texttt{,}\ O\texttt{:}\ List\ \texttt{)}\quad \texttt{delete(}\ I\texttt{:}\ U\texttt{,}\ O\texttt{:}\ List\texttt{,}\ I\texttt{:}\ List\texttt{)}.$$

Therefore $\mu_2(\texttt{Perm}, Q)$ holds. Let us prove now that `Perm` is μ_2-failure free. We use Lemma 5.2. Let A be a well-typed and simply-moded atom:

- if $A = \texttt{perm(}Xs\texttt{,}\ Ys\texttt{)}$ then Ys is a variable and Xs is a list. In case $Xs = \texttt{[]}$ then A *ic-unifies* with the head of *(p1)*. Otherwise, A *ic-unifies* with the head of *(p2)*;
- if $A = \texttt{delete(}X\texttt{,}\ Ys\texttt{,}\ Xs\texttt{)}$ then Ys is a variable and Xs is a list. In case $Xs = \texttt{[]}$ then A *ic-unifies* with the head of *(p3)*. Otherwise, A *ic-unifies* both with the head of *(p3)* and with the head of *(p4)*.

This shows that `Perm` is μ_2-failure free. Finally, `Perm` and the query Q are acceptable by $|\ |$ and I, where:

$$|\texttt{perm(}xs\texttt{,}\ ys\texttt{)}| = ll(xs) \qquad |\texttt{delete(}x\texttt{,}\ xs\texttt{,}\ ys\texttt{)}| = ll(ys)$$
$$I = \{\texttt{perm(}xs\texttt{,}\ ys\texttt{)}|\ ll(xs) = ll(ys)\} \cup \{\texttt{delete(}x\texttt{,}\ xs\texttt{,}\ ys\texttt{)}|\ ll(xs) = ll(ys) + 1\}.$$

where the *list-length* function $ll()$ from ground terms to natural numbers is defined as follows:

$$ll(f(t_1, \ldots, t_n)) = \begin{cases} 0 & \text{if } f \neq [.|.] \\ ll(t_2) + 1 & \text{if } f(t_1, \ldots, t_n) = [t_1|t_2]. \end{cases}$$

Therefore, we can apply Corollary 5.4 to conclude that all $\mathcal{R}^{ic}$-SLD derivations of `Perm` and Q are successful.

Permutations w.r.t. $\mathcal{R}^{sld}$

By Lemma 5.21, μ_2 is weak left-persistent w.r.t. $\mathcal{R}^{sld}$. In the previous example, we have shown that: (1) `Perm` is μ_2-failure free w.r.t. $\mathcal{R}^{ic}$; (2) `Perm` and the query Q are acceptable. (1) implies that `Perm` is μ_2-failure free w.r.t. $\mathcal{R}^{sld}$. Therefore, we can apply Corollary 5.11 using the weak left-persistent relation μ_2 to conclude that all $\mathcal{R}^{sld}$-SLD derivations of `Perm` and $Q = \texttt{perm(}Xs\texttt{,}\ \texttt{A)}$ via the leftmost selection rule are successful, where Xs is any list.

Permutations w.r.t. $\mathcal{R}^{g}$

Consider now $\mathcal{R}^{g}$-SLD resolution. We recall the delay declarations of `Perm`:

```
DELAY perm(X, Y) UNTIL nonvar(X)
DELAY delete(X, Y, Z) UNTIL nonvar(Z)
```

We have already observed that `Perm` and a query $Q = \texttt{perm(}Xs\texttt{,}\ \texttt{A)}$ are well-typed and simply-moded, where Xs is any list. Since the delay declarations imply matching, we have that $\mu_3(\texttt{Perm}, Q)$ holds. Also, acceptability of `Perm` and Q has already been observed. Therefore, in order to apply Corollary 5.4 w.r.t. μ_3, we are left with showing μ_3-failure freedom of `Perm` w.r.t. $\mathcal{R}^{g}$. Consider a well-typed and simply-moded atom A:

- if A = `perm(`Xs`, `Ys`)` then Ys is a variable and Xs is a list, hence $A \in$ *delay_decl*(`Perm`) holds. Moreover, if Xs = `[]` then A unifies with the head of *(p1)*. Otherwise, A unifies with the head of *(p2)*. In both cases, A is covered w.r.t. $\mathcal{R}^g$;
- if A = `delete(`X`, `Ys`, `Xs`)` then Ys is a variable and Xs is a list, hence $A \in$ *delay_decl*(`Perm`) holds. In case Xs = `[]` then A unifies with the head of *(p3)*. Otherwise, A unifies both with the head of *(p3)* and with the head of *(p4)*.

In conclusion, the hypotheses of Corollary 5.4 are satisfied. Thus, all $\mathcal{R}^g$-SLD derivations of `Perm` and Q = `perm(`Xs`, A)` are successful, where Xs is any list.

QuickSort w.r.t. $\mathcal{R}^{sldg}$

Consider the `QuickSort` program w.r.t. $\mathcal{R}^{sldg}$-SLD resolution.

```
qs([],[]).
qs([X|Xs],Ys) ←
   part(X,Xs,Littles,Bigs),
   qs(Littles,Ls),
   qs(Bigs,Bs),
   append(Ls,[X|Bs],Ys).

part(_,[],[],[]).
part(X,[Y|Xs],[Y|Ls],Bs) ← X >= Y, part(X,Xs,Ls,Bs).
part(X,[Y|Xs],Ls,[Y|Bs]) ← X <= Y, part(X,Xs,Ls,Bs).

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

Let us now show that the hypotheses of Corollary 5.11 are satisfied for `QuickSort` and a query Q = `qs(`Xs`,Ys)` where Xs is a list of gae's. We will use the weak left-persistent relation μ_5.

`QuickSort` and Q are well-typed and simply-moded with the types:

$$\texttt{qs(}\ I\texttt{:}\ List(Gae)\texttt{,}\ O\texttt{:}\ List(Gae)\texttt{)}$$
$$\texttt{part(}\ I\texttt{:}\ Gae\texttt{,}\ I\texttt{:}\ List(Gae)\texttt{,}\ O\texttt{:}\ List(Gae)\texttt{,}\ O\texttt{:}\ List(Gae)\texttt{)}$$
$$\texttt{append(}I\texttt{:}\ List(Gae)\texttt{,}\ I\texttt{:}\ List(Gae)\texttt{,}\ O\texttt{:}\ List(Gae)\texttt{)}$$

This shows that μ_5(`QuickSort`, Q) holds. Moreover, `QuickSort` and the query Q are acceptable by a same interpretation and level mapping. The detail of the proof can be found in [5]. Finally, `QuickSort` is readily checked to be μ_5-failure free w.r.t. $\mathcal{R}^{sldg}$. By Corollary 5.11, all $\mathcal{R}^{sldg}$-SLD derivations of `QuickSort` and Q via the leftmost selection rule are successful.

QuickSort w.r.t. $\mathcal{R}^{g+sldg}$

Consider now the following delay declarations for `QuickSort`.

```
DELAY qs(X,Y) UNTIL nonvar(X)
DELAY part(X, Xs, Ls, Bs) UNTIL nonvar(Xs)
DELAY append(X,Y,Z) UNTIL nonvar(X)
```

It is immediate to observe that they imply matching. This and μ_5(`QuickSort`, Q) imply μ_4(`QuickSort`, Q), with $Q =$ `qs(`Xs`,Ys)` where Xs is a list of gae's. Also, it is readily checked that `QuickSort` is μ_4-failure free w.r.t. $\mathcal{R}^{g+sldg}$. By Corollary 5.4, all $\mathcal{R}^{g+sldg}$-SLD derivations of `QuickSort` and Q are successful.

Daughter w.r.t. $\mathcal{R}^{sld}$

Consider the simple program `Daughter`:

```
j(X)  ← F = jack, d(F, female(X)).
d(jack, female(ann)).
d(jack, male(paul)).
X = X.
```

and the query `j(X)`. All $\mathcal{R}^{sld}$-SLD derivations for them via the leftmost selection rule are successful. Let us show it using Corollary 5.11.

First of all, we point out that we cannot use relation μ_2 in this case. In fact, if the second argument of predicate `d` is output, then the first clause cannot be simply-moded (the term `female(X)` occurs in output position but it is not a variable). If the second argument of predicate `d` is input, then we cannot show μ_2-failure freedom, since well-typing would require us to show `d(jack, female(`T`))` covered for every term T. Therefore, we cannot use the weak left-persistent relation μ_2. Let us consider then μ_6, i.e. well-assertedness.

Given the specification $(pre, post)$:

$$\begin{aligned} pre &= \{\texttt{j(}T\texttt{)} \mid T \text{ variable}\} \cup \{T_1 \texttt{ = } T_2 \mid T_1 \text{ variable} \wedge T_2 \text{ ground}\} \\ &\quad \cup \{\texttt{d(jack, female(}T\texttt{))} \mid T \text{ variable}\} \\ post &= \{\texttt{j(}T\texttt{)}\} \cup \{T \texttt{ = } T\} \cup \{\texttt{d(}T_1\texttt{, }T_2\texttt{)}\} \end{aligned}$$

`Daughter` and the query `j(X)` are well-asserted. Consider the first clause. The only atom in pre that unifies with the head of the clause is `j(`T`)`, where T is a variable. For every valuation sequence ρ_0, ρ_1, ρ_2 we have $\rho_0 = mgu($`j(`T`)`, `j(X)`$) = \{$`X`$/T\}$. The proof obligation required by well-assertedness are then:

$$\texttt{(F = jack)}\rho_0 \in pre \quad (1)$$

$$\texttt{(F = jack)}\rho_1 \in post \textit{ implies } \texttt{d(F,female(X))}\rho_1 \in pre \quad (2)$$

$$\texttt{(F = jack)}\rho_1 \in post \wedge \texttt{d(F,female(X))}\rho_2 \in post \textit{ implies } \texttt{j(X)}\rho_2 \in post \quad (3)$$

(1) and (3) are trivial by definition of pre and $post$. Consider (2). `(F = jack)`$\rho_1 \in$ $post$ means ρ_1`(F)` = `jack`. Also, since $\rho_1 = \rho_0\sigma_1$ for some σ_1 such that $dom(\sigma_1) \subseteq \{$`F`$\}$, then ρ_1`(X)` $= T$, with T variable. This implies `d(F,female(X))`ρ_1 $\in pre$, i.e. (2). A similar reasoning can be done for the other clauses, which leads to conclude that μ_6(`Daughter`, `j(X)`) holds.

Also, it is readily checked that `Daughter` is μ_6-failure free w.r.t. $\mathcal{R}^{sld}$ (note that it is sufficient to prove that every $A \in pre$ is covered), and that it and `j(X)` are acceptable by some $|\ |$ and I. By Corollary 5.11, we conclude that all $\mathcal{R}^{sld}$-SLD derivations for them via the leftmost selection rule are successful.

6 Related work

In this section, we consider related works on proving absence of failures or termination with success for logic programs. We refer the reader to [4, 13, 22] for surveys on modings, typings, and termination methods for logic programs.

In the context of $\mathcal{R}^{g+sldg}$-SLD resolution, Apt and Luitjes [3] originally reasoned about termination with success for determinate programs and queries, i.e. those admitting only one derivation. In contrast, our approach applies to nondeterministic programs as well. For instance, the `Perm` program and the query used in our examples are not determinate.

A notion of failure-freedom for programs was independently introduced by Bossi and Cocco [7] specifically for the μ_2 relation (well-typing and simply-moding) with the name of *noFD* programs and queries. P and Q (without arithmetic atoms) are *noFD* if P and Q are well-typed and simply-moded and for every A instance of an atom in Q or in clause bodies of P, if A is well-typed and simply moded then it is covered w.r.t. $\mathcal{R}^{sld}$. It is immediate to see that this means requiring: (1) $\mu_2(P, Q)$; and (2) P is μ_2-failure free (actually, *noFD* restricts to require that A is covered only when $\mu_2(P, A)$ holds and A is an instance of an atom in P or in Q). With our definitions, the main result of Bossi and Cocco consists of showing that if P is μ_2-failure free and $\mu_2(P, Q)$ holds, then no $\mathcal{R}^{sld}$-SLD derivation of P and Q via the leftmost selection rule is failed. In this paper, we have generalized the results of Bossi and Cocco by showing absence of failures via any μ-failure free selection rule (Lemma 4.6) for a generic persistent relation μ. In addition, we have proposed a sufficient condition on $\mathcal{R}$ functions (Definition 4.8) and on failure-free selection rules (μ-termination) that allows us to conclude absence of failures via all selection rules.

Bossi and Cocco refined their approach in [8] to the study of logic programs and queries with *at least one successful derivation via the leftmost selection rule.* Their main result can be stated as an immediate consequence of Corollary 5.11.

Corollary 6.1 *Let P be a μ_5-failure free program, and Q a query such that $\mu_5(P, Q)$ holds. If P and Q are acceptable by some $|\ |$ and I, then there exists a successful $\mathcal{R}^{sld}$-SLD derivation of P and Q via the leftmost selection.*

Proof. By Lemma 5.21, μ_5 is weak left-persistent w.r.t $\mathcal{R}^{sldg}$. Since $\mathcal{R}^{sldg}$ is monotonic, we are in the hypotheses of Corollary 5.11 to conclude that all $\mathcal{R}^{sldg}$-SLD derivations of P and Q via the leftmost selection rule are successful. This implies that there exists a successful $\mathcal{R}^{sld}$-SLD derivation of P and Q. By Strong Completeness of SLD-resolution, there exists a successful $\mathcal{R}^{sld}$-SLD derivation of P and Q via the leftmost selection rule. □

In contrast, we have offered a general theoretical framework to reason about programs and queries such that *all* derivations w.r.t. some advanced form of resolution $\mathcal{R}$ are successful. While in pure logic programming (i.e. $\mathcal{R}^{sld}$, possibly considering the leftmost selection rule only) the class of programs and queries above is rather small, such a class becomes much larger when the programmer is allowed to specify restrictions on the selection rule.

In addition, our results still improve over Bossi and Cocco in the case of $\mathcal{R}^{sld}$-SLD resolution. Consider a program P and a query Q without arithmetic atoms. First, with the hypotheses of Corollary 6.1, we can conclude that *all* $\mathcal{R}^{sldg}$-SLD derivations (and hence $\mathcal{R}^{sld}$-SLD derivations) via the leftmost selection rule of P and Q are successful, while [8] can only conclude that *there exists* a successful $\mathcal{R}^{sld}$-SLD derivation. The `Perm` program is an example of this. Second, we have generalized our result to any weak left-persistent relation, while [8] is bound to well-typing and simply-moding. As an example, we reasoned on the `Daughter` program using well-assertedness since the program is not simply-moded.

On the other hand, the approach of Bossi and Cocco applies to *generate & test* clauses, which follow the scheme:

```
gtsolve(Input,Output) ← generate(Input,Output), test(Output).
```

Here, the predicate `test` is naturally supposed to fail in some cases, passing control back to `generate` through backtracking. Therefore, we cannot show that all derivations are successful, but that only some derivation is. However, we observe that we are still in the position of applying the methods of this paper to the `generate` predicate.

Smaus et al. [26] studied error-freedom and termination of logic programs with block declarations [18], namely delay declarations expressing that some arguments of a predicate must be non-variable. They show that if P is a μ_3-failure free program and $\mu_3(P, Q)$ holds, then every $\mathcal{R}^g$-SLD derivation of P and Q is finite if every $\mathcal{R}^g$-SLD derivation of P and Q via the leftmost selection rule is finite. Actually, they do not use the notion of "delay-declarations imply matching", but rather a sufficient syntactical condition (called *input selectability*). As before, this is a special case of our theoretical framework.

Turning the attention on inferring failure-freedom, Debray, López-García and Hermenegildo [14] provided a method that, given mode and type information, can detect whether the clauses defining a predicate *cover* the type of the predicate. We claim that their approach helps us in proving that a program is μ_5-failure free. The non-failure analysis of [14] is based on regular types [12], which are specified by *regular term grammars* in which each type symbol has exactly one defining *type rule*. Consider a predicate $p(\mathbf{i} : \mathbf{I}, \mathbf{o} : \mathbf{O})$ and a clause $p(\mathbf{t}, \mathbf{s}) \leftarrow \mathbf{G}, B_1, \ldots, B_n$. Assume that the input types $\mathbf{I}$ of p are regular types and the guards $\mathbf{G}$ of the clauses defining p are tests over the *Herbrand domain* or over *linear arithmetic over integers* not involving $var(\mathbf{s})$. Debray et al. provide an algorithm that decides whether for every $\mathbf{i} : \mathbf{I}$ there is a clause such that $\mathbf{t} = \mathbf{i}, \mathbf{G}$ holds. In practice, they provide us with a decision procedure for μ_5-failure freedom, under the mentioned hypotheses. In fact, when $\mu_5(P, p(\mathbf{i}, \mathbf{o}))$ holds, then $\mathbf{o}$ is a tuple of distinct variables. Therefore, if $\mathbf{t} = \mathbf{i}, \mathbf{G}$ holds and

$var(\mathbf{s})$ do not occur in $\mathbf{G}$, then also $\mathbf{t} = \mathbf{i}, \mathbf{s} = \mathbf{o}, \mathbf{G}$ holds, i.e. $p(\mathbf{i}, \mathbf{o})$ guardedly unifies with at least one clause head.

7 Conclusions and future work

We have presented a method for reasoning on programs and queries that have only successful derivations. While in pure logic programming such a class is rather small, it becomes much larger when the programmer is allowed to specify restrictions on the selection rule. For this reason, we proposed our results in the context of a generalization of SLD-resolution which takes into account restrictions on atom and/or clause selectability. Such a generalization includes as special instances delay declarations, input-consuming derivations, guarded clause resolution, and can be considered as an original contribution of the paper.

With reference to a persistent relation μ, we have introduced a notion of program μ-failure-freedom (which extends an independently introduced notion [7]). A μ-failure free program P and a query Q in the relation μ cannot have deadlocked or failed derivations via any selection rule s that selects covered atoms, if any. Moreover, at least one such a rule exists. In addition, if we admit termination via s, the result extends to any derivation of P and Q. Moreover, if we admit bounded nondeterminism, we can conclude that all derivations of P and Q are successful. The general framework has been instantiated with reference to the leftmost selection rule, for which many persistent relations are known, and substantiated with some examples. We adopted relations that are persistent w.r.t. $\mathcal{R}^{sld}$, since this implies they are persistent w.r.t. any $\mathcal{R}$. However, we observe that the proof obligations of some of them (e.g. well-typing) could be relaxed for specific $\mathcal{R}$-functions. The boundary of the applicability of the approach lies in those program schemas that naturally must fail, such as the *generate & test* schema.

At the state-of-the-art, it seems complicated to instantiate the framework on selection rules other than the leftmost one. The main problem is the lack of persistent relations to be used with non-leftmost selection rules on specific $\mathcal{R}$ functions. Well-typing, simply-moding and well-assertedness are all tied to a left-to-right execution and to $\mathcal{R}^{sld}$-SLD resolution. For instance, programs where a strict coroutining between two atoms is implied by the $\mathcal{R}$ function cannot be reasoned about using those relations.

Also, as stated by condition **[C]** of Theorem 4.14, a direct approach to show that all derivations are successful is to prove failure-freedom of the program, and termination w.r.t. all selection rules, i.e. *strong termination*, w.r.t. $\mathcal{R}$. The drawback of this approach is that the known methods to show strong termination are not powerful enough for most of the advanced resolution strategies. As an example, the well-known method of recurrency (see Bezem [6]) reason on strong termination for $\mathcal{R}^{sld}$-SLD resolution, but it is too weak to show strong termination for $\mathcal{R}^{g}$-SLD resolution (it is sufficient to observe that recurrency does not take into account delay declarations). As an example, we cannot use

recurrency for proving strong termination of `Perm` w.r.t. $\mathcal{R}^g$. An exception exists for $\mathcal{R}^{ic}$-SLD resolution, where the methods of *quasi-recurrency* [9] and *simply-acceptability* [10] can show strong termination w.r.t. $\mathcal{R}^{ic}$. Again, however, we need persistent relations specifically designed for $\mathcal{R}^{ic}$-SLD resolution, i.e. not tied to a left-to-right execution.

Acknowledgements

We are grateful to the anonymous referees for their valuable comments.

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 E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6A:743–764, 1994.

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

[6] M. A. Bezem. Strong Termination of Logic Programs. *Journal of Logic Programming*, 15(1 & 2):79–98, 1993.

[7] A. Bossi and N. Cocco. Programs without failures. In N. E. Fuchs, editor, *LOPSTR '97*, volume 1463 of *LNCS*, pages 28–48. Springer-Verlag, 1997.

[8] A. Bossi and N. Cocco. Successes in logic programs. In P. Flener, editor, *LOPSTR '98*, volume 1559 of *LNCS*, pages 219–239. Springer-Verlag, 1999.

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

[10] A. Bossi, S. Etalle, S. Rossi, and J.-G. Smaus. On the semantics and termination of logic programs with dynamic scheduling. In D. Sands, editor, *Proc. of the 10th European Symposium on Programming*, volume 2028 of *LNCS*, pages 402–416. Springer-Verlag, 2001.

[11] F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In K. R. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 321–335. MIT Press, 1992.

[12] P. W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, Logic Programming Series, pages 157–187. The MIT Press, 1992.

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

[14] S. Debray, P. López-García, and M. Hermenegildo. Non-failure analysis for logic programs. In *Proceedings of the 1997 International Conference on Logic Programming*, pages 48–62. The MIT Press, 1997.

[15] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[16] W. Drabent and J. Małuszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

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

[18] Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 2003. `http://www.sics.se/isl/sicstuswww/site/documentation.html`.

[19] K. Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7:978–992, 1989.

[20] M. J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Proceedings of the 1987 International Conference on Logic Programming*, pages 858–876. MIT Press, 1987.

[21] D. Pedreschi and S. Ruggieri. Bounded nondeterminism of logic programs. In D. De Schreye, editor, *Proc. of the International Conference on Logic Programming*, pages 350–364. The MIT Press, 1999.

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

[23] S. Ruggieri. *Verification and Validation of Logic Programs*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999.

[24] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[25] 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.

[26] 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.