ON TERMINATION OF GENERAL LOGIC PROGRAMS
W.R.T. CONSTRUCTIVE NEGATION

ELENA MARCHIORI

The notions of acyclicity and acceptability fail to characterize termination of general logic programs adequately under sldnf-resolution, as termination due to floundering is not captured. In this paper we establish the appropriate correspondence by considering sld-resolution with Chan's constructive negation. In particular, the resulting characterization provides a class of programs for which Chan's constructive negation is complete. Moreover, it can be used to formalize and implement problems in non-monotonic reasoning.

1. INTRODUCTION

The aim of this paper is to give an exact description of general logic programs that terminate for all ground queries. This issue was studied by Apt, Bezem and Pedreschi, where sldnf-resolution is considered and the notion of acyclic [1] and acceptable [4] programs are introduced, to deal with an arbitrary and with the leftmost (Prolog) selection rule, respectively. However, they fail to give a complete characterization, because termination due to floundering is not captured. For instance, the program p:

\[ p(X) \leftarrow \neg p(Y) \]

is terminating (floundering) but it is not acyclic (and not acceptable).

In this paper we show that exact descriptions of general logic programs that terminate for all ground queries with an arbitrary and with the leftmost selection rule, respectively, can be obtained when Chan's constructive negation [6] is used, here called sldcnf-resolution. These results are not very surprising. However,
they are significant for the following reasons. They provide a characterization of classes of programs for which Chan's procedure [6] is complete; hence more involved procedures, like [7, 11, 14] are not needed. Moreover, we shall show how they can be used to formalize, and implement by means of sldcnf-resolution, interesting problems in nonmonotonic reasoning.

Let us explain Chan's constructive negation. Informally, for an atom $A$ with finite derivation tree the answers for the query $\neg A$ are produced by negating the answers for $A$. However, the procedure is not applicable if $A$ has an infinite derivation tree, as for instance in the program toy:

\[
\begin{align*}
p(X) & \leftarrow X = a. \\
p(X) & \leftarrow X \neq a. \\
p(X) & \leftarrow p(X).
\end{align*}
\]

Here $p(X)$ has an infinite derivation obtained by selecting always the last clause, hence we cannot apply the above described procedure to $\neg p(X)$. That is, the concept of sldcnf-derivation for $\neg p(X)$ is undefined. This phenomenon renders problematic to reason about termination, where a neat formalization of derivation is required. Therefore, we introduce an alternative top-down definition of sldcnf-resolution. We follow the approach of [2], where to resolve negative literals subsidiary trees are built by constructing their branches in parallel. If this subsidiary construction diverges, then the main derivation is considered to be infinite. As opposed to the procedure by Chan, the formalization of sldcnf-resolution we obtain is always applicable. In particular, as in Chan's procedure, the main derivation fails as soon as in the construction of the subsidiary tree the constraint true is produced as disjunction of some leaves, even if the subsidiary tree is infinite. For instance, in the program toy, the query $\neg p(X)$ has one derivation that is infinite. Moreover, $\neg p(X)$ fails, because the derivation tree for $p(X)$ has the two leaves $X = a$ and $X \neq a$, whose disjunction is equivalent to true.

We consider programs whose sldcnf-derivations of ground queries are finite and refer to them as terminating. We give a syntactic characterization of terminating programs, and show that for these programs queries that have only finite derivations can also be characterized syntactically. Observe that w.r.t. constructive negation the program $p$, given at the beginning of this section, is not terminating. Analogous results are proven when the Prolog selection rule is considered, where this time also a suitable model of the program is used to provide a quasi-syntactic characterization of terminating programs.

Thus, sldcnf-resolution allows us to give an exact description of general programs that terminate for all ground queries. In particular, for these programs and for a bigger class of queries called bounded, sldcnf-resolution is complete, and it is enough powerful to be used to formalize and implement interesting problems in non-monotonic reasoning. For instance, consider the following program YSP, which formalizes the so-called Yale Shooting Problem ([12]).

\[
\begin{align*}
(a) & \quad \text{holds(alive, [ ])} \leftarrow. \\
(b) & \quad \text{holds(loadeed, [load|Xs])} \leftarrow. \\
(c) & \quad \text{holds(dead, [shoot|Xs])} \leftarrow. \\
& \quad \text{holds(loadeed, Xs).}
\end{align*}
\]
(d) \text{ab(alive, shoot, Xs)} \leftarrow \\
\text{holds\{loaded, Xs\}.}

(e) \text{holds\{Xf, [Xe|Xs]\}} \leftarrow \\
\neg \text{ab\{Xf, Xe, Xs\},}

\text{holds\{Xf, Xs\}.}

Here Xf, Xs, and Xe denote variables, representing a generic fluent, situation, and event, respectively. All other terms occurring in the program denote constants. In [1] it is proven that YSP is acyclic and that the query \( Q = \text{holds\{alive, [X, Y]\} } \) is bounded. However, \( Q \) flounders, so no answer can be obtained by means of sld-cnff-resolution. Instead, using constructive negation the answers \( X \neq \text{shoot} \) and \( Y \neq \text{load} \) are obtained.

The paper is organized as follows. After some preliminaries on notation and terminology, in Section 3 a top-down definition of sld-cnff-resolution is given, and the classes of terminating and left-terminating programs are introduced. In Section 4 a syntactic characterization of terminating programs is given, and in Section 5 analogous results are proven for left-terminating programs. In Section 6 we show the relevance of these programs for formalizing and implementing problems in nonmonotonic reasoning. Finally, Section 7 contains some conclusions.

2. NOTATION AND TERMINOLOGY

We shall adopt Prolog syntax and assume that a string starting with a capital letter denotes a variable, while other strings denote constants, terms and relations. A sequence \( X_1, \ldots, X_n \) of distinct variables is abbreviated by \( X \), while \( t \) indicates a sequence of terms. The formula \( X_1 = t_1 \land \cdots \land X_n = t_n \) is denoted by \( X = t \). An equality formula, indicated by \( E \), is an assertion that does not contain any relation symbols other than the equality symbol \( = \). The formula \( \exists (c_1 \land \cdots \land c_n) \) is called simple equality formula, where \( n \geq 0 \), the \( c_i \)'s are equalities or inequalities and \( \exists \) quantifies over some (perhaps none) of the variables occurring in the \( c_i \)'s. The empty conjunction of assertions and the empty disjunction of assertions are denoted by true and false, respectively.

Substitutions are indicated by lowercase greek letters \( \alpha, \beta, \theta, \ldots \). The domain \( \text{dom}(\theta) \) of a substitution \( \theta \) consists of those variables \( X \) s.t. \( X \theta \neq X \). For a set \( V \) of variables the notation \( \theta\upharpoonright_V \) is used to denote the substitution \( \theta' \) whose domain is equal to \( V \cap \text{dom}(\theta) \) and s.t. \( X \theta' = X \theta \) for every \( X \in V \). For an idempotent substitution \( \theta = \{X_1/t_1, \ldots, X_n/t_n\} \), we define \( E_\theta \) to be the equality formula \( X_1 = t_1 \land \cdots \land X_n = t_n \). A substitution \( \rho \) is called renaming, if there exists \( \rho' \) such that \( (\rho\rho')_{\text{dom}(\rho)} = \epsilon \), where \( \epsilon \) denotes the empty substitution. For a syntactic object \( O \) and a renaming \( \rho \), we call \( O\rho \) a variant of \( O \). Moreover, \( O \) is said to be ground if it does not contain any variable. Given two terms/atoms \( s \) and \( t \), \( \text{ngu}(s, t) \) denotes a fixed idempotent most general unifier of \( s \) and \( t \).

Relation symbols are often denoted by \( p, q, r \). The syntax of a general program is extended as follows to contain equality formulas. An (extended) literal, denoted by \( L \), is either an atom \( p(s) \), or a negative literal \( \neg p(s) \), or an equality \( s = t \), or an inequality \( s \neq t \), where \( p \) is not an equality relation and \( \forall \) quantifies over some (perhaps none) of the variables occurring in the inequality. Equalities and inequalities are also called constraints, denoted by \( c \). An (extended) general program, called for brevity program and denoted by \( P \), is a finite set of (universally quantified)
clauses of the form \( H \leftarrow L_1, \ldots, L_m \), where \( m \geq 0 \) and \( H \) is an atom. In the following the letters \( A, B \) are used to indicate atoms, \( C \) and \( Q \) denote a clause and a query, respectively. Moreover \( \text{comp}(P) \) denotes the Clark's completion of a program \( P \). An inequality \( \forall (s \neq t) \) is said to be primitive if it is satisfiable but not valid. For instance, \( X \neq a \) is primitive. A query \( Q = L_1, \ldots, L_n \) is called reduced if \( n = 0 \) or \( L_i \) is a primitive inequality for all \( i \) in \([1, n]\). If \( Q \) is reduced then \( E_Q \) denotes the equality formula \( L_1 \land \cdots \land L_n \). We assume that the Herbrand universe has an infinite number of function symbols, so that reduced queries are satisfiable. The query obtained by removing \( L \) from \( Q \) is denoted by \( Q - \{L\} \). Finally, c.a.s. is used as shorthand for computed answer substitution.

3. \texttt{slcnf-RESOLUTION}

In this section we give an alternative top-down definition of Chan's constructive negation, which will be used to study termination. First, we introduce informally Chan's method, and show a drawback of the original formulation for studying termination. Then, we introduce an alternative definition of Chan's method that overcomes such drawback.

In \texttt{sl-cn-resolution}, for a program \( P \) and a query \( Q \), if \( \theta \) is a c.a.s. for \( Q \) then it can be written in equational form as \( \exists (X_1 = X_1 \theta \land \cdots \land X_n = X_n \theta) \), where \( X_1, \ldots, X_n \) are the variables of \( Q \) and \( \exists \) quantifies over all the other variables. Suppose that all \texttt{sl-cn}-derivations of \( Q \) are finite and do not involve the selection of any negative literals. Then there are only finitely many successful derivations. Let \( \theta_1, \ldots, \theta_k, k \geq 0 \), be the c.a.s.'s of these successful derivations and let \( F_Q \) be the equality formula \( \exists (E_{\theta_1} \lor \cdots \lor E_{\theta_k}) \), where \( \exists \) quantifies over the variables that do not occur in \( Q \). Then the completion \( \text{comp}(P) \) of \( P \) logically implies \( \forall (Q \leftrightarrow F_Q) \), i.e.,

\[
\text{comp}(P) \models \forall (Q \leftrightarrow F_Q).
\]

To resolve negative nonground literals Chan in [6] introduced a procedure here called \texttt{sl-cn-resolution}, where the answers for \( \neg Q \) are obtained from the negation of \( F_Q \). However, this procedure is not defined when \( Q \) has an infinite derivation, and hence the concept of derivation is not defined for \( \neg Q \). This is a serious drawback for the study of termination, where the notion of derivation is of primary importance. Therefore, we propose an alternative definition of \texttt{sl-cn-resolution}, where the subsidiary trees used to resolve negative literals are built in a top-down way, constructing their branches in parallel. If this subsidiary construction diverges, then the main derivation is considered to be infinite.

Let \textit{Tree} be the set containing those trees whose nodes are (possibly marked) queries of (possibly marked) literals, and having substitutions and possibly (variants of) clauses associated to edges. We consider \textit{selected} as marker for literals, and \textit{successful} or \textit{failed} as markers for nodes. A marked literal is called \textit{selected}. As in Chan [6], we assume that a primitive inequality cannot be selected.

\textit{Assumption 3.1.} Primitive inequalities cannot be selected;

An element of \textit{Tree} is called:

- \textit{successful} if at least one leaf is marked as \textit{successful};
• finitely successful if it is finite, all its leaves are marked and there is at least one leaf marked as successful;
• finitely failed if it is finite and all its leaves are marked as failed.

We introduce now the notion of answer and full answer for a query \( Q \), which will be used in the definition of pre-sldcnf-tree.

**Definition 3.1.** (Answer and Full Answer) Let \( Q \) be a query and let \( T \) be a successful tree with root \( Q \). Let \( \xi \) be a branch of \( T \) whose last node is a reduced query, say \( Q' \). Let \( \alpha_1, \ldots, \alpha_n \) be the consecutive substitutions along \( \xi \), and let \( \theta = (\alpha_1 \cdots \alpha_n)|_{\text{vars}(Q')} \). Then the equality formula \( \exists (E_{\theta} \land E_{Q'}) \) is called an answer for \( Q \) in \( T \), where \( \exists \) quantifies over all the variables that do not occur in \( Q \). If \( T \) is finitely successful, then we call full answer of \( Q \) in \( T \), denoted by \( F_Q \), the disjunction of all the answers for \( Q \) in \( T \).

We shall assume that answers are normalized according with the procedure given in [6]. Now we can define the notion of pre-sldcnf-tree. Call subsidiary function a partial function which maps a query with selected literal of the form \( \neg A \) in a tree of Tree with root \( A \).

**Definition 3.2.** (pre-sldcnf-tree) Let \( P \) be a program. A pre-sldcnf-tree in \( P \) is a triple (\( \mathcal{I}, T, \text{subs} \)) s.t. \( \mathcal{I} \) is a set of trees in Tree, \( T \) is an element of \( \mathcal{I} \) called main tree of \( \mathcal{I} \), and \( \text{subs} \) is a subsidiary function. It is inductively defined as follows:

1. \( ((T), T, \text{subs}) \) is a pre-sldcnf-tree, called initial pre-sldcnf-tree, for every \( T \) consisting of one node \( Q \), which is either reduced or it has a selected literal. \( \text{subs} \) is everywhere undefined.
2. If \( \Gamma \) is a pre-sldcnf-tree, then any extension of \( \Gamma \) is a pre-sldcnf-tree.

An extension of a pre-sldcnf-tree \( \Gamma \) (in \( P \)) is obtained from \( \Gamma \) by applying the following steps. Let \( \Gamma = (\mathcal{I}, T_{\text{main}}, \text{subs}) \):

1. Mark all leaves consisting of reduced queries as successful.
2. For every unmarked leaf \( Q \) in some tree \( T \) in \( \Gamma \), let \( L \) be its selected literal. Then
   A. If \( L = A \) is an atom then
      i. if there is no resolvent of \( Q \) in \( P \) then mark \( Q \) as failed;
      ii. otherwise add all the resolvents of \( Q \) as sons of \( Q \) in \( T \), associate to every edge the input clause and the mgu used to compute the corresponding resolvent, and mark a literal in every nonreduced resolvent.
   B. If \( L = \neg A \) is a negative literal then
      i. if \( \text{subs}(Q) \) is undefined then add the tree \( T' \) with the single node \( A \) to \( \mathcal{I} \) and set \( \text{subs}(Q) \) to \( T' \);
      ii. if \( \text{subs}(Q) \) is defined then
         a. if \( \text{subs}(Q) \) is finitely failed then add \( Q \setminus \{L\} \) as son of \( Q \) in \( T \), with one marked literal, if not reduced;
         b. if \( \text{subs}(Q) \) is successful and the disjunction of its answers is equivalent to \( \text{true} \) then mark \( Q \) as failed;
c. if \( \text{subs}(Q) \) is \text{finitely successful} then let \( NA_1 \lor \cdots \lor NA_n \) be the disjunction of simple equality formulas obtained by \text{negating} \( F_A \): for every \( j \in [1, n] \) add the query obtained from \( Q \) by replacing \( L \) with \( NA_j \), with one marked literal if nonreduced, as son of \( Q \) in \( T \).

C. If \( L \) is an equality, say \( s = t \) then
i. if \( s \) and \( t \) are not unifiable then mark \( Q \) as \text{failed};
ii. otherwise add \((Q - \{L\})\theta\) with one marked literal, if non-reduced, as son of \( Q \) in \( T \), where \( \theta = \text{mgu}(s, t) \).

D. If \( L \) is an inequality, say \( \forall(s \neq t) \), then
i. if it is valid then add \( Q - \{L\} \) with one marked literal, if non-reduced, as son of \( Q \) in \( T \);
ii. if it is unsatisfiable then mark \( Q \) as \text{failed}.

In the definition of extension of a pre-sldcnf-tree, we assume that full answers are \text{negated} as described in [6]. As a consequence, the disjuncts \( NA_j \)'s remain within the syntax of a query (see e.g., [6]). Let \( \text{ext}(\Gamma^\prime) \) denote the set of extensions of \( \Gamma^\prime \).

Let \( \mathcal{PT} \) denote the set of pre-sldcnf-trees in \( P \). Consider the partial ordered set \( (\mathcal{PT}, \leq) \), where \( \leq \) is the reflexive and transitive closure of the relation \( \text{Rel} \), which is the minimal relation on pre-sldcnf-trees s.t. \( (\Gamma, \Gamma^\prime) \) is in \( \text{Rel} \), for every \( \Gamma^\prime \in \text{ext}(\Gamma) \).

It is well known that any partial order can be completed into a complete partial order, where the limits of ascending chains are incorporated (see e.g., [8]). Then, let \( C(\mathcal{PT}, \leq) \) be the completion of \( (\mathcal{PT}, \leq) \).

**Definition 3.3.** (sldcnf-tree) An sldcnf-tree for \( Q \) is the limit (in \( C(\mathcal{PT}, \leq) \)) of an ascending chain \( \Gamma_0 \leq \cdots \leq \Gamma_n \leq \cdots \), where for every \( n \geq 1 \), \( \Gamma_n \) is in \( \text{ext}(\Gamma_{n-1}) \), and \( \Gamma_0 = (\{Q\}, Q, \text{subs}) \); moreover, \( \text{subs} \) is the subsidiary function everywhere undefined.

An answer for \( Q \) in the main tree of an sldcnf-tree \( \Gamma \) for \( Q \) is simply called an answer for \( Q \) (in \( \Gamma \)).

To define sldcnf-derivations and finite sldcnf-trees, we use the notion of path. A \text{path} in \( \Gamma \) is a sequence of nodes \( N_0, \ldots, N_i, \ldots \), s.t. for all \( i \), \( N_{i+1} \) is either an immediate descendent of \( N_i \) in some tree in \( \Gamma \), or \( N_{i+1} \) is the root of the tree \( \text{subs}(N_i) \).

**Definition 3.4.** (sldcnf-derivation) Let \( \Gamma \) be a sldcnf-tree for \( Q \). A sldcnf-derivation for \( Q \), denoted by \( \xi \), is a branch in the main tree of \( \Gamma \) starting at the root, together with the set of all trees in \( \Gamma \) whose roots are reachable from some node of \( \text{subs}(Q) \), with \( Q \) in \( \xi \). \( \xi \) is said to be \text{finite} if all paths in \( \Gamma \) fully contained in this branch and these trees are finite.

**Definition 3.5.** (finite sldcnf-tree) An sldcnf-tree is \text{finite} if it does not contain any infinite path.

Now we introduce the notions of terminating and left-terminating program. Intuitively, for a terminating program every ground query has only finite sldcnf-trees, while for a left-terminating program only the sldcnf-trees of ground queries that are obtained by using a leftmost selection rule are required to be finite.
An sldcnf-tree $\Gamma$ is via a selection rule $R$ if in the sequence of pre-sldcnf-trees whose limit is $\Gamma$ the selection rule $R$ specifies every marking of literals.

**Definition 3.6.** (Terminating Program) We say that the program $P$ is terminating if all sldcnf-trees for ground queries (in $P$) are finite. A query is terminating if all sldcnf-trees for $Q$ (in $P$) are finite.

The leftmost selection rule, also called Prolog selection rule, used to define left-terminating programs, marks as selected in every nonreduced node of a pre-sldcnf-tree the leftmost possible literal, where a literal is called possible if it is not a primitive inequality. We call ldcnf-tree an sldcnf-tree via a leftmost selection rule.

**Definition 3.7.** (Left-Terminating Program) A program $P$ is left-terminating if all ldcnf-trees for ground queries are finite. A query is left-terminating if all ldcnf-trees for $Q$ (in $P$) are finite.

In the following two sections we shall provide a syntactic characterization of terminating programs, and a quasi-syntactic characterization of left-terminating programs. We conclude this section with a simple example to illustrate sldcnf-resolution. Here and in the other examples of the paper, a selected literal is underlined, the empty query is denoted by $\square$, and $f$ and $s$ are used as shorthands for the markers failed and successful, respectively.

**Example 3.1.** Consider the program $\text{toy}$ given in the Introduction. The query $\neg p(X)$ fails since its associated tree $\text{subs}(\neg p(X))$ is successful and the disjunction of its answers is true. $\neg p(X)$ has only one derivation, and this derivation is not finite because $\text{subs}(\neg p(X))$ is not finite. The main tree of the sldcnf-tree for $\neg p(X)$ and the subtree $\text{subs}(\neg p(X))$ are represented below.

The main tree:

$$\neg p(X) f$$

The tree $\text{subs}(\neg p(X))$:

\[ \quad \]

4. A CHARACTERIZATION OF TERMINATING PROGRAMS

The formalization of constructive negation given in the previous section allows us to reason in a rigorous way about termination. In this section we give a syntactic characterization of terminating programs.
The standard way to prove termination of a program amounts of finding a suitable function on a well-founded set, and a method that guarantees that for a terminating program it is possible to associate with every computation a descending chain of values of that function. For logic programs, functions called level mappings have been used [1], which map ground atoms to natural numbers. Their extension to negated atoms was given in [4], where the level mapping of \( \neg A \) is simply defined to be equal to the level mapping of \( A \). Here, we have to consider also constraints. Constraints are not themselves a problem for termination, because they are atomic actions whose execution always terminates. Therefore, we shall assume that the notion of level mapping is only defined for literals that are not constraints. However, note that the presence of constraints in a query influences its termination behavior, because for instance a derivation fails finitely if a constraint which is not satisfiable is selected.

**Definition 4.1. (Level Mapping)** A level mapping is a function, denoted by \( \vdash \), from ground literals which are not constraints to natural numbers s.t. \( \vdash \neg A = \vdash A \).

The notion of acyclic program was introduced in [1], and it amounts to a simple condition on the literals of program clauses, namely that the level mapping decreases from the head to each body atom.

**Definition 4.2. (Acyclic Program)** A program \( P \) is acyclic w.r.t a level mapping \( \vdash \) if for all ground instances \( H \leftarrow L_1, \ldots, L_m \) of clauses of \( P \) we have that

\[ |H| > |L_i| \]

holds for all \( i \in [1, m] \) s.t. \( |L_i| \) is defined. \( P \) is acyclic if there exists a level mapping \( \vdash \) s.t. \( P \) is acyclic w.r.t. \( \vdash \).

In [1], it was proven that an acyclic program is terminating when \texttt{sldnf-resolution} is used. We prove here that an analogous result holds when \texttt{sldcnf-resolution} is used. The proof of this result does not present substantial differences with the original proof of Apt and Bezem, and is given for making the paper self-contained.

The concept of bounded query is used, which allows to prove the result for a bigger class of queries that contains all ground queries.

**Definition 4.3. (Bounded Query)** A literal \( L \), which is not a constraint, is called bounded w.r.t. a level mapping \( \vdash \) if the set \( |L| = \{|L'| | L' \text{ ground instance of } L\} \) is finite. A query \( Q = L_1, \ldots, L_n \) is bounded w.r.t. \( \vdash \) if every \( L_i \) is bounded w.r.t. \( \vdash \), for \( i \in [1, n] \) s.t. \( L_i \) is not a constraint.

We shall say that \( L \) is bounded by \( l \) if \( l \) is an upper bound for \( |L| \). If \( L \) is bounded then let \( [L] \) denote the maximum if \( |L| \). Moreover, if \( Q \) is bounded then let \( [Q] \) denote the (finite) multiset (see [10]) consisting of the natural numbers \([L_{i,1}], \ldots, [L_{i,n}]\)

where for \( i \in [1, n] \) we have that \( i \in \{i_1, \ldots, i_n\} \) if \( L_i \) is not a constraint. These quantities will be used in the sequel.

Recall that a multiset is an unordered collection in which the number of occurrences of each element is significant. We shall consider here the multiset ordering on multisets of natural numbers. Formally, a multiset of natural numbers
is a function from the set \((\mathbb{N}, <)\) of natural numbers to itself, giving the multiplicity of each natural number. Then the ordering \(<_{mull}\) on multisets is defined as the transitive closure of the replacement of a natural number with any finite number (possibly zero) of natural numbers that are smaller under \(<\). Since \(<\) is well-founded, the induced ordering \(<_{mull}\) is also well-founded, as a consequence of the König Lemma for infinite terms. For simplicity we shall omit in the sequel the subscript \(mull\) from \(<_{mull}\).

The following two lemmas are simple to prove. They were originally introduced by Apt and Bezem in [1].

**Lemma 4.1.** Let \(\|\|\) be a level mapping and \(L\) a bounded literal. Then, for every substitution \(\theta\), \(L\theta\) is bounded and \(\|L\theta\| \leq \|L\|\).

**Lemma 4.2.** Let \(P\) be acyclic w.r.t. \(\|\|\). Then, for every clause \(H \leftarrow L_1, \ldots, L_n\) of \(P\) and for every substitution \(\theta\) we have: if \(H\theta\) is bounded then \(L_i\theta\) is bounded and \(\|L_i\theta\| < \|H\theta\|\), for \(i \in [1, n]\) s.t. \(L_i\) is not a constraint.

Now we can prove the announced result on acyclic programs.

**Theorem 4.1.** Let \(P\) be an acyclic program. Then every \(sldcnf\)-tree for a bounded query in \(P\) contains only bounded queries and is finite.

**Proof.** Let \(Q\) be a bounded query in a \(sldcnf\)-tree, let \(L\) be its selected literal, and let \(Q'\) be a resolvent of \(Q\) in \(P\). We distinguish the following cases.

\(L\) is an atom. Let \(H \leftarrow L_1, \ldots, L_n\) be the input clause and \(\theta\) the computed mgu to derive \(Q'\). By Lemma 4.1, we have that \(H\theta\) is bounded and \(\|H\theta\| \leq \|L\|\). Then by Lemma 4.2 \(L_i\theta\) is bounded and \(\|L_i\theta\| < \|H\theta\|\). Hence \(Q'\) is bounded and \(\|Q'\|\) is smaller than \(\|Q\|\) in the multiset ordering.

\(L\) is a negative literal, say \(\neg A\). Then \(subs(Q)\) has root \(A\) that is obviously bounded, and \(\|A\|\) is smaller or equal than \(\|Q\|\) in the multiset ordering (since \(|A| = |\neg A|\)). Moreover, every resolvent of \(Q\) (if any) is obtained from \(Q\) by replacing the selected literal with a (possibly empty) conjunction of constraints. Then \(\|Q'\|\) is smaller than \(\|Q\|\) in the multiset ordering.

\(L\) is a constraint. Then the resolvent \(Q'\) of \(Q\) is obtained by removing the selected literal and applying the computed (if any) substitution. Then \(Q'\) is bounded and \(\|Q'\|\) is smaller or equal than \(\|Q\|\) in the multiset ordering.

Note that there can be only finitely many consecutive selections of negative literals and of constraints. Then, the result follows from the fact that the multiset ordering is well founded.

In [1], Apt and Bezem state that terminating programs that do not flounder can be proven to be acyclic. The authors say that this result is rather weak, because simple terminating programs having some floundering derivations cannot be captured. Also, they do not give a proof of this result, because they say it would be too involved. Here we show that an exact characterization of terminating programs can be obtained by considering Chan's constructive negation. To this aim, one has to find a suitable level mapping \(\|\|\) s.t. every ground instance of a clause of \(P\) satisfies the condition of Definition 2 and s.t. every terminating query is bounded.

We first need some preliminary results. The following property of mgu's is useful.
Proposition 4.1. Let \( s, t \) be two terms (atoms) and let \( \theta \) be a substitution. Suppose that \( \alpha = \text{mgu}(s\theta, t\theta) \) exists. Then \( \mu = \text{mgu}(s, t) \) exists and is s.t. \( \theta\alpha = \mu\sigma \), for a suitable \( \sigma \).

**Proof.** Observe that \( \theta\alpha \) is a unifier of \( s \) and \( t \).

The following lemma was originally introduced by Bezem in [5], and is here extended to deal also with equality constraints.

**Lemma 4.3.** Let \( Q \) be a query and \( \theta \) a substitution. Let \( L \) be a literal of \( Q \) which is either an atom or an equality. If \( Q\theta \), with \( L\theta \) as selected literal, has an \text{sldcnf}-resolvent \( Q' \), then \( Q \), with \( L \) as selected literal, has an \text{sldcnf}-resolvent \( Q'' \) s.t. \( Q' = Q''\theta' \) for some substitution \( \theta' \).

**Proof.** If \( L\theta \) is an equality, say \( s\theta = t\theta \), then let \( Q' = (Q - (L))\theta\alpha \) be the resolvent of \( Q\theta \), where \( \alpha = \text{mgu}(s\theta, t\theta) \). Then by Proposition 4.1 \( \mu = \text{mgu}(s, t) \) exists and \( \theta\alpha = \mu\sigma \), for a suitable \( \sigma \). Hence \( Q' = (Q - (L))\mu \) is a resolvent of \( Q \) and \( Q' = Q''\sigma \).

If \( L\theta \) is an atom, then let \( C = H \leftarrow R \) be the input clause and \( Q' = (L_1, \ldots, L_m)\alpha \) be the resolvent obtained by replacing \( L\theta \) with \( R\alpha \), where \( \alpha = \text{mgu}(H, L\theta) \). It is not restricted to assume that \( C \) is also variable disjoint with \( Q \) and with \( \text{vars}(\theta) \). Then by Proposition 4.1 \( \mu = \text{mgu}(H, L) \) exists, and \( \theta\alpha = \mu\sigma \), for a suitable \( \sigma \). Let \( Q'' \) be the resolvent of \( Q \) and \( C \) with selected literal \( L \). Then \( Q''\sigma = Q' \).

To simplify the proofs of the following results, we introduce the notion of specific path at \( k \).

**Definition 4.4.** (Specific Path at \( k \)) Let \( \Gamma \) be an \text{sldcnf}-tree, let \( \pi = Q_0, \ldots, Q_k, \ldots \) be a path of \( \Gamma \), and let \( k \geq 0 \). Then \( \pi \) is a specific path at \( k \) if the following conditions hold:

- the selected literal in \( Q_k \) is not an inequality;
- if the selected literal in \( Q_k \) is a negative literal then \( Q_{k+1} \) is the root of \( \text{subs}(Q_k) \).

Let \( Q \) be a terminating query, and let \( \pi \) be a path in a \text{sldcnf}-tree for \( Q \). Define \( \pi_{pre} = Q_0, \ldots, Q_n \), called specific prefix of \( \pi \), to be a maximal prefix of \( \pi \) s.t. \( \pi \) is a specific path at \( k \), for every \( k < n \). Then let \( \pi_Q \) be the specific prefix of \( \pi \) containing maximal number of nodes, for all paths \( \pi \) in all \text{sldcnf}-trees for \( Q \). Let \( \text{nodes}(\pi_Q) \) denote the number of nodes of \( \pi_Q \). Then a candidate level mapping is the function that maps a ground atom \( A \) to \( \text{nodes}(\pi_A) \).

We show that this is a correct choice.

**Theorem 4.2.** Let \( Q \) be a terminating query and let \( Q' \) be an instance of \( Q \). Then \( \text{nodes}(\pi_Q) \geq \text{nodes}(\pi_{Q'}) \).

We shall prove this theorem by absurd. To this aim we shall need some preliminary results.

**Lemma 4.4.** Let \( P \) be a program and let \( Q \) be a terminating query. Then for all substitutions \( \theta \), \( \pi_{Q\theta} \) is finite.

**Proof.** By contraposition suppose that \( \pi_{Q\theta} \) is infinite. Observe that in \( \pi_{Q\theta} \) every node is either a resolvent obtained via the selection of an atom or an equality, or
the root of a subtree obtained applying \textit{subs} to its predecessor. Then by Lemma 4.3 we can lift \( \pi_{Q_\theta} \) to a prefix of a path in a \texttt{sldcnf}-tree for \( Q \). Hence \( Q \) is not terminating.

Now we can prove Theorem 4.2.

\textbf{Proof of Theorem 4.2.} By Lemma 4.4 we have that \( \text{nodes}(\pi_{Q'}) \) is defined. By absurd, suppose that \( \text{nodes}(\pi_{Q'}) > \text{nodes}(\pi_Q) \). Then by Lemma 4.3 we can lift \( \pi_Q \) to a specific prefix of a path in a \texttt{sldcnf}-tree for \( Q \). Hence \( \text{nodes}(\pi_Q) \geq \text{nodes}(\pi_{Q'}) \). Absurd.

We are now ready to prove the converse of Theorem 4.1, thus obtaining that terminating and acyclic programs coincide.

\textbf{Theorem 4.3.} Let \( P \) be a terminating program. Then for some level mapping \( \mid \mid \)

(i) \( P \) is acyclic w.r.t. \( \mid \mid \),

(ii) for every query \( Q \), \( Q \) is bounded w.r.t. \( \mid \mid \) iff it is terminating.

\textbf{Proof.} Since \( P \) is terminating, then by the König's Lemma it follows that for every ground atom \( A \), the function defined by \( \mid A \mid = \text{nodes}(\pi_A), \mid \neg A \mid = \mid A \mid \) is a level mapping. From \( \text{nodes}(\pi_{\neg A}) > \text{nodes}(\pi_A) \) it follows that \( \text{nodes}(\pi_{\neg A}) > \mid \neg A \mid \).

(ii \( \leftarrow \)) Consider a terminating query \( Q \). We prove that \( Q \) is bounded by \( \text{nodes}(\pi_Q) \). The case where \( [Q] \) is the empty set is immediate. So, let \( l \in [Q] \). Then \( l = \mid L_i \mid \), for some ground instance \( L_1, \ldots, L_n \) of \( Q \) and for some \( i \in [1, n] \).

Then

\( \text{nodes}(\pi_Q) \geq \{ \text{by Theorem 4.2} \} \text{nodes}(\pi_{(L_1, \ldots, L_n)}) \).

Observe that \( \pi_{L_i} \) can be embedded into a prefix of a path for \( L_1, \ldots, L_n \), obtained by replacing every element \( R \) of \( \pi_{L_i} \) by \( L_1, \ldots, L_{i-1}, R, L_i+1, \ldots, L_n \). Then

\( \text{nodes}(\pi_{(L_1, \ldots, L_n)}) \geq \text{nodes}(\pi_{L_i}) \)

\( \geq \{ \text{by the definition of } \mid \mid \} \)

\( \mid L_i \mid \)

\( = l \).

(i) Let \( H\theta \leftarrow L_1 \theta, \ldots, L_n \theta \) be a ground instance of a clause in \( P \). Then we have to show that \( \mid H\theta \mid > \mid L_i \theta \mid \) for \( i \in [1, n] \) s.t. \( \mid L_i \theta \mid \) is defined. Since \( H\theta \theta = H\theta \), then \( \theta \) is a unifier of \( H\theta \) and \( H \). Then there exists \( \mu = \text{mgu}(H\theta, H) \) s.t. \( \theta = \mu \theta' \) and \( (L_1 \mu, \ldots, L_n \mu) \) is a resolvent of \( H\theta \). Then

\( \mid H\theta \mid = \{ \text{definition of } \mid \mid \} \)

\( \text{nodes}(\pi_{H\theta}) > \{ H\theta \text{ is not an inequality and } \pi_{(L_1 \mu, \ldots, L_n \mu)} \text{ is a proper suffix of a path for } H\theta \} \)

\( \text{nodes}(\pi_{(L_1 \mu, \ldots, L_n \mu)}) \)

\( \geq \{ \text{part } (ii \leftarrow \text{, since } L_i \theta \in [L_1 \mu, \ldots, L_n \mu]\} \)

\( \mid L_i \theta \mid \).
Consider a query $Q$ which is bounded w.r.t. $|\cdot|$. Then by (i) and Theorem 4.1 it follows that $Q$ is terminating. □

From Theorem 4.1 and Theorem 4.3 it follows that terminating programs coincide with acyclic programs and that for acyclic programs a query is terminating if and only if it is bounded.

5. LEFT-TERMINATING PROGRAMS

In this section we consider a fixed selection rule, corresponding to the natural extension of the Prolog selection rule to programs containing constraints. We show that results analogous to those of the previous section hold, where the concept of acyclicity is replaced by that of acceptability. The notion of acceptable general program was introduced by Apt and Pedreschi [4]. It is based on the same condition used to define acyclic programs, only that for a ground instance $H \leftarrow L_1, \ldots, L_n$ of a clause, the test $|H| > |L_i|$ is performed only until the first literal $L_n$ that fails. This is sufficient since, due to the Prolog selection rule, literals after $L_n$ will not be executed. To compute $\bar{n}$, the class of models of $\text{comp}(P)$ is considered.

**Definition 5.1.** (Acceptable Program) Let $|\cdot|$ be a level mapping for $P$ and let $I$ be a model of $\text{comp}(P)$. $P$ is acceptable w.r.t. $|\cdot|$ and $I$ if for all ground instances $H \leftarrow L_1, \ldots, L_n$ of clauses of $P$ we have that

$$|H| > |L_i|$$

holds for $i \in [1, \bar{n}]$ s.t. $L_i$ is not a constraint, where

$$\bar{n} = \min(\{n\} \cup \{i \in [1, n] | I \not= L_i\}).$$

$P$ is called acceptable if it is acceptable w.r.t. some level mapping and a model of $\text{comp}(P)$.

We show that a program is left-terminating if and only if it is acceptable. As in the previous section, to extend the result to nonground queries, the notion of boundedness is considered. However, due to the fixed selection rule, the order of the literals in a query is now relevant, and yields the following definition of boundedness. Let $Q = L_1, \ldots, L_n$ be a query, let $|\cdot|$ be a level mapping and let $I$ be a model of $\text{comp}(P)$. For every $i \in [1, n]$ s.t. $L_i$ is not a constraint, consider the set

$$|Q|_i = \{|L'_i| | I \models L'_1, \ldots, L'_{i-1}, \text{ for some ground instance } L'_1, \ldots, L'_i \text{ of } L_1, \ldots, L_i\}$$

**Definition 5.2.** (Bounded Query) Let $|\cdot|$ be a level mapping and let $I$ be a model of $\text{comp}(P)$. A query $Q = L_1, \ldots, L_n$ is bounded (w.r.t. $|\cdot|$ and $I$) if $|Q|_i$ is finite, for every $L_i$ which is not a constraint.

If $Q$ is bounded then we denote by $\|Q\|_i$ the multiset containing the maximum of $|Q|_i$, for every $L_i$ that is not a constraint. Then $Q$ is bounded by $k$ if $k \geq \|Q\|_i$.

**Theorem 5.1.** Let $P$ be an acceptable program and let $Q$ be a bounded query. Then every \text{ldcnf}-tree for $Q$ in $P$ contains only bounded queries and is finite.
On Termination of General Logic Programs

Proof. Let \(|\) and \(I\) be a level mapping and an interpretation, respectively, s.t. \(P\) is acceptable w.r.t. \(|\) and \(I\). Let \(Q = L_1, \ldots, L_n\) and let \(L_i\) be its selected literal. The proof is similar to that of Theorem 4.1 in the cases where \(L_i\) is an atom or a constraint, while in the case where \(L_i\) is a negative literal we have to add an observation about \(I\). So, suppose \(L_i\) is equal to \(\neg A\). Then \(\text{subs}(Q)\) has root \(A\), which is obviously bounded and \(|[A]|_I\) is smaller or equal than \(|[Q]|_I\) in the multiset ordering (since \(|A| = \neg A|\)); moreover every resolvent \(Q'\) of \(Q\) (if any) is bounded and \(|[Q']|_I\) is smaller than \(|[Q]|_I\) in the multiset ordering, since it is obtained from \(Q\) by replacing \(L_i\) with a (possibly empty) conjunction of constraints \(c_1, \ldots, c_k\) s.t. \(I \models \neg L_i \iff (c_1 \land \cdots \land c_k)\).

To show that also the converse of the above result holds, we proceed in a similar way as we did for terminating programs.

Formally, let \(Q\) be a left-terminating query, and let \(\pi\) be a path in a \(ldcnf\)-tree for \(Q\). Define \(\pi_Q\) to be the specific prefix of \(\pi\) containing the maximal number of nodes, for all paths \(\pi\) of a \(ldcnf\)-tree for \(Q\). Let \(\text{nodes}(\pi_Q)\) be the number of nodes of \(\pi_Q\).

Theorem 5.2. Let \(Q\) be a left-terminating query and let \(Q'\) be an instance of \(Q\). Then \(\text{nodes}(\pi_Q) \geq \text{nodes}(\pi_{Q'})\).

We shall prove this theorem by absurd. To this aim we shall use the following persistence lemma.

Lemma 5.1. Let \(P\) be a program and let \(Q\) be a left-terminating query. Then for all substitutions \(\theta\), \(\pi_{Q\theta}\) is finite.

Proof. By contraposition suppose that \(\pi_{Q\theta}\) is infinite. Observe that in \(\pi_{Q\theta}\) every node is either a resolvent obtained via the selection of an atom or an equality, or the root of a subtree obtained applying \(\text{subs}\) to its predecessor. Then by Lemma 4.3 we can lift \(\pi_{Q\theta}\) to a prefix of a path in a \(ldcnf\)-tree for \(Q\). Hence \(Q\) is not terminating. Contradiction.

Now we can prove Theorem 5.2.

Proof of Theorem 5.2. By Lemma 5.1 we have that \(\text{nodes}(\pi_{Q'})\) is defined. By absurd, suppose that \(\text{nodes}(\pi_{Q'}) > \text{nodes}(\pi_Q)\). Then by Lemma 4.3 we can lift \(\pi_{Q'}\) to a specific prefix \(\pi\) of a path in a \(ldcnf\)-tree for \(Q\). Hence we have that \(\text{nodes}(\pi_{Q'}) \geq \text{nodes}(\pi_{Q'}).\) Absurd.

Theorem 5.3. Let \(P\) be a left-terminating program. Then for some level mapping \(|\) and for a model \(I\) of \(\text{comp}(P)\)

(i) \(P\) is acceptable w.r.t. \(|\) and \(I\),
(ii) for every query \(Q, Q\) is bounded w.r.t. \(|\) and \(I\) iff \(Q\) is left-terminating.

Proof. Since \(P\) is left-terminating, then the function that assigns to every ground atom \(A\) the number \(\text{nodes}(\pi_A)\) is a level mapping. From \(\text{nodes}(\pi_{\neg A}) > \text{nodes}(\pi_A)\) it follows that \(\text{nodes}(\pi_{\neg A}) > |\neg A| = |A|\). Choose \(I = \{A \in B_p\mid \text{there is an \(ldcnf\)-refutation of } A \text{ in } P\}\). Then \(I\) is a model of \(\text{comp}(P)\).
(ii $\leftrightarrow$) Consider a left-terminating query $Q$. We show that $Q$ is bounded by $\text{nodes}(\pi_Q)$. The case where $[Q]_l$ is the empty set is immediate. So, let $l \in [Q]_l$. Then for some ground instance $L_1, \ldots, L_n$ of $Q$ and $i \in [1, n]$ with $i = \min([n] \cup \{i \in [1, n] \mid l \neq L_i\})$, we have $l = |L_i|$. Then

\[
\begin{align*}
\text{nodes}(\pi_Q) \\
\geq \{\text{Theorem (5.2)}\} \\
\text{nodes}(\pi_{(L_1, \ldots, L_n)}) \\
\geq \{\text{by construction of } \pi_{(L_1, \ldots, L_n)}\} \\
\text{nodes}(\pi_{L_1, \ldots, L_n}) \\
\geq \{\text{because } l \equiv L_1, \ldots, L_{i-1}\} \\
\text{nodes}(\pi_{L_1, \ldots, L_n}).
\end{align*}
\]

Observe that $\pi_{L_i}$ can be embedded into a prefix of a path for $L_1, \ldots, L_n$, obtained by replacing every element $R$ of $\pi_{L_i}$ by $R$, $L_{i+1}, \ldots, L_n$. Then

\[
\begin{align*}
\text{nodes}(\pi_{L_1, \ldots, L_n}) \geq \\
\text{nodes}(\pi_{L_i}) \\
\geq \{\text{by definition of } | \cdot |\} \\
|L_i| \\
= l.
\end{align*}
\]

(i) The proof is similar to the one of case (i) of Theorem 4.3.

(ii $\rightarrow$) Consider a query $Q$, which is bounded w.r.t. $| |$. Then by (i) and Theorem 5.1 $Q$ is left-terminating.

6. APPLICATION

In this section we give two examples to illustrate how to formalize and implement problems in nonmonotonic reasoning by means of terminating and left-terminating programs, respectively.

6.1. Temporal Reasoning

Various forms of temporal reasoning can be described using acyclic programs. In particular, the program YSP given in the Introduction is a formalization of the so-called Yale Shooting Problem in terms of an acyclic program. We recall the problem following [12]. Consider a person that is alive. The event load implies the fact that the gun becomes loaded. The event shoot in the situation loaded implies the fact that the person becomes dead. Moreover, the property of being alive is abnormal (i.e., it can change its truth value) with respect to a shoot event, given that the gun is loaded. Finally, facts persist under the occurrence of events that are not abnormal. The interest on this problem is due to the fact that its formalization by means of theories about nonmonotonic reasoning yields weak
conclusions. In [1] it is proven that YSP is acyclic w.r.t. the level mapping which assigns to a ground atom of the form \( \text{holds}(t, t') \) the natural number \( 2l(t') \), and to a ground atom of the form \( \text{ab}(t, t', t'') \) the natural number \( 2l(t'') + 1 \), where for a ground term \( t \) of the universe of YSP, if \( t \) is a list then \( l(t) \) denotes its length, otherwise it denotes 0. Consider the query \( \text{holds}(\text{alive}, [X, Y]) \). This query is bounded (by 4), hence it is terminating. The following is an slcnf-tree for \( \text{holds}(\text{alive}, [X, Y]) \).

\[
\text{holds}(\text{alive}, [X, Y])
\]

\[
\sim \text{ab}(\text{alive}, X, [Y]), \text{holds}(\text{alive}, [Y])
\]

\[
X \neq \text{shoot}, \text{holds}(\text{alive}, [Y])
\]

\[
\{Y/X_e\}
\]

\[
X \neq \text{shoot}, \sim \text{ab}(\text{alive}, X_e, [\ ]), \text{holds}(\text{alive}, [\ ])
\]

\[
X \neq \text{shoot}, \text{holds}(\text{alive}, [\ ])
\]

\[
\sim X \neq \text{shoot}s
\]

\[
X \neq \text{shoot}, Y \neq \text{load}, \text{holds}(\text{alive}, [Y])
\]

\[
\{X/\text{shoot}\}
\]

\[
Y \neq \text{load}, \text{holds}(\text{alive}, [Y])
\]

\[
\{Y/X_e\}
\]

\[
Y \neq \text{load}, \sim \text{ab}(\text{alive}, X_e, [\ ]), \text{holds}(\text{alive}, [\ ])
\]

\[
Y \neq \text{load}, \text{holds}(\text{alive}, [\ ])
\]

\[
\sim Y \neq \text{load}s
\]

Where \( \text{subs}(\sim \text{ab}(\text{alive}, X, [Y]), \text{holds}(\text{alive}, [Y])) \) is the following tree.

\[
\text{ab}(\text{alive}, X, [Y])
\]

\[
\{X/\text{shoot}\}
\]

\[
\text{holds}(\text{loaded}, [Y])
\]

\[
\{Y/\text{load}\}
\]

\[
\sim \text{ab}(\text{loaded}, [\ ]), \text{holds}(\text{loaded}, [\ ])
\]

\[
\sim \text{ab}(\text{loaded}, X_e, [\ ]), \text{holds}(\text{loaded}, [\ ])
\]

The two trees \( \text{subs}(Y \neq \text{load}, \sim \text{ab}(\text{alive}, X_2, [\ ]), \text{holds}(\text{alive}, [\ ])) \) and \( \text{subs}(X \neq \text{shoot}, \sim \text{ab}(\text{alive}, X_e, [\ ]), \text{holds}(\text{alive}, [\ ])) \) coincide and are represented below.

\[
\text{ab}(\text{alive}, X_e, [\ ])
\]

\[
\{X_e/\text{shoot}\}
\]

\[
\text{holds}(\text{loaded}, [\ ])
\]

\[
\sim \text{ab}(\text{loaded}, X_e, [\ ]), \text{holds}(\text{loaded}, [\ ])
\]

Notice that by using slcfnf-resolution \( \text{holds}(\text{alive}, [X, Y]) \) flounders.
6.2. Search in Graph Structures

To render the notion of acceptability practical, in the original definition of acceptability, \( I \) is required to be a model of \( P \) that is also a model of \( \text{comp}(P^-) \), where \( P^- \) is defined as follows. Let \( \text{Neg}_P \) denote the set of relations in \( P \) that occur in a negative literal in a body of a clause from \( P \). Say that \( p \) refers to \( q \) if there is a clause in \( P \) that uses the relation \( p \) in its head and \( q \) in its body and say that \( p \) depends on \( q \) if \((p, q)\) is in the reflexive, transitive closure of the relation refers to. Define \( \text{Neg}^*_P \) to be the set of relations in \( P \) on which the relations in \( \text{Neg}_P \) depend on. Then \( P^- \) is the set of clauses in \( P \) in whose head a relation from \( \text{Neg}^*_P \) occurs. We call good model of \( P \) a model of \( P \) which is also a model of \( \text{comp}(P^-) \), and will use it in the following example.

Graph structures are used in many applications, such as representing relations, situations or problems. Two typical operations performed on graphs are find a path between two given nodes and find a subgraph, with some specified properties, of a graph. The following program specialize is an example of the combination of these two operations.

A relation \( \text{spec} \) is defined by the clause (a), s.t. \( \text{spec}(n_1, n_2, n, g) \) is true if \( n_1, n_2 \) are two nodes of a given graph \( g \), and \( n \) is a node that does not occur in any acyclic path of \( g \) connecting \( n_1 \) with \( n_2 \). The relation \( \text{spec} \) is specified as the negation of another relation, called \( \text{unspec} \), where \( \text{unspec}(n_1, n_2, n, g) \) is true if there is an acyclic path of \( g \) connecting \( n_1 \) and \( n_2 \) that contains \( n \).

Acyclic paths of a graph are described by the relation \( \text{path} \), defined by the clause (c), where \( \text{path}(n_1, n_2, g, p) \) calls the query \( \text{path1}(n_1, [n_2], g, p) \). Here the second argument of \( \text{path1} \) is used to construct incrementally a path connecting \( n_1 \) with \( n_2 \): using clause (e), the partial path \([x|p_1]\) is transformed in \([y, x|p_1]\) if there is an edge \([y, x]\) in the graph \( g \) such that \( y \) is not already present in \([x|p_1]\). The construction terminates if \( y \) is equal to \( n_1 \), thanks to clause (d).

So the relation \( \text{path1} \) is defined inductively by the clauses (d) and (e), using the familiar relation \( \text{mem} \), defined by the clauses (f) and (g).

Notice that, from fact (d) it follows that if \( n_1 \) and \( n_2 \) are equal, then \([n_1]\) is assumed to be an acyclic path from \( n_1 \) and \( n_2 \), for any term \( g \).

\[\begin{align*}
(a) \quad \text{spec}(N_1, N_2, N, G) & \leftarrow \\
& \neg \text{unspec}(N_1, N_2, N, G). \\
(b) \quad \text{unspec}(N_1, N_2, N, G) & \leftarrow \\
& \text{path}(N_1, N_2, G, P), \\
& \text{mem}(N, P). \\
(c) \quad \text{path}(N_1, N_2, G, P) & \leftarrow \\
& \text{path1}(N_1, [N_2], G, P). \\
(d) \quad \text{path1}(N_1, [X_1|P_1], G, [N_1|P_1]) & \leftarrow. \\
(e) \quad \text{path1}(N_1, [X_1|P_1], G, P) & \leftarrow \\
& \text{mem}([Y_1, X_1], G), \\
& \neg \text{mem}(Y_1, [X_1|P_1]), \\
& \text{path1}(N_1, [Y_1, X_1|P_1], G, P). \\
(f) \quad \text{mem}(X, [X|Y]) & \leftarrow. \\
(g) \quad \text{mem}(X, [Y|Z]) & \leftarrow \\
& \text{mem}(X, Z). 
\end{align*}\]
Here a graph is represented by means of a list of edges. For instance \( spec(a, b, c, [[a, b], [b, c], [a, a]]) \) holds, where \( a, b, c \) are constants and the graph \( [[a, b], [b, c], [a, a]] \) is represented below.

\[
\begin{array}{c}
\text{a} \\
\xrightarrow{\text{b}} \\
\xrightarrow{\text{c}}
\end{array}
\]

Notice that \texttt{specialize} is not terminating: for instance, the query \( \text{path1}(a, [b, c], d, e) \) has an infinite derivation obtained by selecting at every resolution step the rightmost literal of the query and by choosing as input clause (a variant of) the clause (e).

However, we will show that \texttt{specialize} is acceptable and that the query \( Q = spec(a, b, X, [[a, b], [b, c], [a, a]]) \) is bounded. Then one obtains the following finite \texttt{ldcnf}-tree for \( Q \), where \( \text{edges} \) denotes the list \([a, b],[b, c],[a, a]\).

\[
spec(a, b, X, \text{edges})
\]

\[
\neg \text{unspec}(a, A, X, \text{edges})
\]

\[
X \not\equiv a, X \not\equiv b
\]

The tree \( \text{subs}(\neg \text{unspec}(a, b, X, \text{edges})) \) is given below.

\[
\text{unspec}(a, b, X, \text{edges})
\]

\[
\text{path}(a, b, \text{edges}, P), \text{mem}(X, P)
\]

\[
\text{path1}(a, [b], \text{edges}, P), \text{mem}(X, P)
\]

\[
\text{mem}([Y1, b], \text{edges}), \neg \text{mem}(Y1, [b]), \text{path1}(a, [Y1, b], \text{edges}, P), \text{mem}(X, P)
\]

\[
\{Y1/s\}
\]

\[
\neg \text{mem}(a, [b]), \text{path1}(a, [a, b], \text{edges}, P), \text{mem}(X, P)
\]

\[
\text{path1}(a, [a, b], \text{edges}, P), \text{mem}(X, P)
\]

\[
\{P/a, b\}
\]

\[
\text{mem}(X, [a, b])
\]

\[
\{X/s\}
\]

\[
\square s
\]

\[
\text{mem}(X, [b])
\]

\[
\{X/s\}
\]

\[
\square s
\]
Note that for simplicity we omitted to draw the derivations whose leaves are marked as failed, and the tree \((\text{subs}(\neg \text{mem}(a,[b])), \text{path}(a,[a,b],P), \text{mem}(X,P))\) is the finite failed tree
\[
\begin{array}{c}
\text{mem}(a,[b]) \\
\downarrow_{(s)} \\
\text{mem}(a,[]) f
\end{array}
\]

Notice that by using \text{ldnf}-resolution \(Q\) does flounder.

We prove now that \text{specialize} is acceptable. To this end, one has to find a proper level mapping and a model of \text{specialize} that is a model of the completion \(\text{comp}(\text{specialize})\). Notice that \(\text{specialize}\) consists of six clauses \((b)\)–\((g)\). One can argue that such an expressive model is not needed. Indeed, since clause \((a)\) introduces the new relation \text{spec} using the relations defined in \((b)\)–\((g)\), then to prove that \(\text{spec}(n1,n2,n,g)\) is left-terminating it is sufficient to show that the program \(\text{spec1}\) consisting of the clauses \((b)\)–\((g)\) is acceptable and that \(\text{unspec}(n1,n2,n,g)\) is bounded. In this way one has just to consider a model of \(\text{spec1}\) that is a model of \(\text{comp}(\text{spec1})\), i.e., of the two clauses \((f)\) and \((g)\). Alternative definitions of acceptability that employ less semantic information are investigated in [13].

We introduce the function \(|.|\) defined on ground terms as follows: \(|t_1 t_2| = |t_1| + 1\) and \(|f(t)| = 0\) if \(f \neq [-] [-]\).

For a list \(l\), let \(\text{set}(l)\) denote the set of its elements, i.e., \(\text{set}(l) = \{ \} \) if \(l = [\] \) and \(\text{set}(l) = \{x\} \cup \text{set}(y)\) if \(l = [x|y]\). Moreover, for a list \(p\) and a graph \(g\), let \(p \cap g\) be the list containing as elements those \(x\) that are elements of \(p\) and such that there exists a \(y\) s.t. \([x,y]\) is an element of \(g\).

Consider the interpretation \(I = I_{\text{unspec}} \cup I_{\text{path}} \cup I_{\text{path1}} \cup I_{\text{mem}}\), with
\[
I_{\text{unspec}} = [\text{unspec}(N1,N2,N,G)],
\]
where \([A]\) denotes the set of all ground instances of \(A\);
\[
I_{\text{path}} = \{ \text{path}(n1,n2,g,p) | |g| + 1 \geq |p| \};
\]
\[
I_{\text{path1}} = \{ \text{path1}(n1,p1,g,p) | |p1| - |p1 \cap g| \geq |p| - |p \cap g| \};
\]
\[
I_{\text{mem}} = \{ \text{mem}(s,t) | \text{list s.t. } s \in \text{set}(t) \}.
\]

**Lemma 6.1.** \(I\) is a model of \(\text{spec1}\).

**Proof.**

- It follows immediate that \(I\) is a model of clause \((b)\).
- Consider clause \((c)\). Suppose that \(I \models \text{path}(n1,[n2],g,p)\). Note that \([n2] \cap [n] \leq 1\). Then \(|p| - |p \cap g| \leq 1\). But \(|p \cap g| \leq |g|\). Then \(|p| \leq |g| + 1\), hence \(I \models \text{path}(n1,n2,g,p)\).
- We have that \(I\) models \(\text{path1}(n1,[n1|p1],g,[n1|p1])\), hence it models clause \((d)\).
- Consider clause \((e)\). Suppose that
\[
I \models \text{mem}([y1,x1],g), \neg \text{mem}(y1,[x1|p1]), \text{path}(n1,[y1,x1|p1],g,p).
\]
Then \([y_1, x_1 | p_1] - [y_1, x_1 | p_1] \cap g \geq |p| - |p \cap g|\), where \(y_1 \notin [x_1 | p_1]\) and \([y_1, x_1] \subseteq g\). Then \([y_1, x_1 | p_1] \cap g = 1 + [x_1 | p_1] \cap g\). So \([y_1, x_1 | p_1] - [y_1, x_1 | p_1] \cap g = [x_1 | p_1] - [x_1 | p_1] \cap g\). Then \([x_1 | p_1] - [x_1 | p_1] \cap g \geq |p| - |p \cap g|\). Hence \(I = \text{path}1(n_1, [x_1 | p_1], g, p)\).

- Finally it is easy to check that \(I\) models clauses (f) and (g).

Consider \(\text{spec}1^- = \{(f),(g)\}\): it is easy to check that \(I_{\text{mem}}\) is a model of \(\text{comp}(\text{spec}1^-)\).

Finally, we define the level mapping \(||\) as follows:

\[
|\text{mem}(s, t)| = |t|;
\]

\[
|\text{path}1(n_1, p_1, g, p)| = |p_1| + |g| + 2(|g| - |p_1 \cap g|) + 1;
\]

\[
|\text{path}(n_1, n_2, g, p)| = 3|g| + 3,
\]

\[
|\text{unspec}(n_1, n_2, n, g)| = 3|g| + 4.
\]

Observe that from \(|g| \geq |p_1 \cap g|\) it follows that \(|\text{path}(n_1, p_1, g, p)|\) is well defined.

It is not difficult to check that \(\text{spec}1\) is acceptable w.r.t. \(||\) and \(I\). We present the proofs for clauses (b) and (e). The proofs for the other clauses are similar.

For clause (b) we obtain the following inequalities:

- \(|\text{unspec}(n_1, n_2, n, g)| > 3|g| + 3,
- \(|\text{unspec}(n_1, n_2, n, g)| > |p|\), under the hypothesis \(I = \text{path}(n_1, n_2, n, p)\).

The first condition is easy to check. For the second one, observe that from \(I = \text{path}(n_1, n_2, n, p)\) it follows that \(|g| + 1 \geq |p|\).

For clause (e) we obtain the following inequalities:

- \(|\text{path}(n_1, [x_1 | p_1], g, p)| > |g|\);
- \(|\text{path}1(n_1, [x_1 | p_1], g, p)| \geq [x_1 | p_1]\), under the hypothesis \(I = \text{mem}([y_1, x_1 | p_1])\);
- \(|\text{path}1(n_1, [x_1 | p_1], g, p)| > |\text{path}1(n_1, [y_1, x_1 | p_1], g, p)|, under the hypothesis \(I = \text{mem}([y_1, x_1 | p_1]), \neg \text{mem}([y_1, x_1 | p_1])\).

The first two inequalities are easy to check. For the third one, observe that from \(I = \text{mem}([y_1, x_1 | g]), \neg \text{mem}([y_1, x_1 | p_1])\) it follows that \([y_1, x_1 | p_1] \cap g = 1 + [x_1 | p_1] \cap g\), hence \(|g| - [y_1, x_1 | p_1] \cap g| = (|g| - [x_1 | p_1] \cap g|) - 1\).

7. CONCLUSION

In this paper we studied termination of general logic programs, when \(\text{sld-resolution with constructive negation is considered as execution model. We introduced a top-down definition of the Chan's procedure \([6]\), and use this definition to give a syntactic characterization of programs that terminate for all ground queries, for an arbitrary selection rule. We proved that for these programs queries that have only finite derivations can be described syntactically. We proved analogous results for programs that terminate for all ground queries when the Prolog selection rule is assumed, by means of a quasi-syntactic criterion obtained by taking into account also a model of the considered program.\)
These results are not surprising, and the concepts used to prove them are extensions to constructive negation of already existing concepts. However, such extensions are not immediate; moreover, they provide a neat formalization of Chan’s procedure, and a characterization of two classes of general programs for which there is no need to resort to more sophisticated approaches for constructive negation and the 1988 procedure by Chan [6] is sufficient.

Various approaches to constructive negation were proposed: among them the procedure by Chan [7] based on corouting, the sldfa-resolution by Drabent [11], and the constructive negation for constraint logic programming by Stuckey [14]. These procedures are more general than sldcnf-resolution, because they aim at completeness (w.r.t. three-valued completion) for all programs. To this end they have a mechanism to use (partial) information from infinite derivations which is far more general than the one described above. As a consequence, the termination behavior of programs executed with these procedures, seems to be rather difficult to capture, because of its irregularity.

This research was partly supported by the Esprit Basic Research Action 6810 (Compulog 2). The author would like to thank Krzysztof Apt for proposing the topic of this paper, and the members of the Compulog 2 project for interesting discussions. Moreover, the author would like to thank Jan Ruten for his help and support, and Frank Teusink for useful discussions. Finally, the author would like to thank the referees: their constructive comments and suggestions helped to improve both the content and the presentation of this paper.

REFERENCES

