NORTH-HOLLAND

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) ← ¬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,

Address correspondence to Elena Marchiori, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands. E-mail: elena@cwi.nl.
Received June 1994; accepted May 1995.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010
0743-1066/96/$15.00
SSDI 0743-1066(95)00090-7

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:

```
p(X) ← X=a.
p(X) ← X≠a.
p(X) ← p(X).
```

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]).

```
(a)  holds(alive, [ ]) ← .
(b)  holds(loaded, [load|Xs]) ← .
(c)  holds(dead, [shoot|Xs]) ←
     holds(loaded, Xs).
```

(*d*) `ab(alive, shoot, Xs) ←`
`holds(loaded, Xs).`

(*e*) `holds(Xf, [Xe|Xs]) ←`
`¬ab(Xf, Xe, Xs),`
`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 $Q =$ `holds(alive, [X, Y])` is bounded. However, Q flounders, so no answer can be obtained by means of `sldnf`-resolution. Instead, using constructive negation the answers $X \neq shoot$ and $Y \neq load$ are obtained.

The paper is organized as follows. After some preliminaries on notation and terminology, in Section 3 a top-down definition of `sldcnf`-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 $\mathbf{X}$, while $\mathbf{t}$ indicates a sequence of terms. The formula $X_1 = t_1 \wedge \cdots \wedge X_n = t_n$ is denoted by $\mathbf{X} = \mathbf{t}$. An *equality formula*, indicated by E, is an assertion that does not contain any relation symbols other than the equality symbol $=$. The formula $\exists(c_1 \wedge \cdots \wedge c_n)$ 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 $dom(\theta)$ of a substitution θ consists of those variables X s.t. $X\theta \neq X$. For a set V of variables the notation $\theta_{|V}$ is used to denote the substitution θ' whose domain is equal to $V \cap dom(\theta)$ 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_θ to be the equality formula $X_1 = t_1 \wedge \cdots \wedge X_n = t_n$. A substitution ρ is called *renaming*, if there exists ρ' such that $(\rho\rho')_{|dom(\rho)} = \epsilon$, where ϵ denotes the empty substitution. For a syntactic object O and a renaming ρ, 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, $mgu(s,t)$ 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 $p(\mathbf{s})$, or a negative literal $\neg\, p(\mathbf{s})$, or an equality $s = t$, or an inequality $\forall(s \neq t)$, 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 *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 $L_1 \wedge \cdots \wedge L_n$. 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. sldcnf-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 sld-resolution, for a program P and a query Q, if θ is a c.a.s. for Q then it can be written in equational form as $\exists(X_1 = X_1\theta \wedge \cdots \wedge X_n = X_n\theta)$, where $X_1, \ldots, X_n$ are the variables of Q and $\exists$ quantifies over all the other variables. Suppose that all sld-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 $\exists(E_{\theta_1} \vee \cdots \vee E_{\theta_k})$, where $\exists$ quantifies over the variables that do not occur in Q. Then the completion *comp*(P) of P logically implies $\forall(Q \leftrightarrow F_Q)$, i.e.,

$$comp(P) \models \forall(Q \leftrightarrow F_Q).$$

To resolve negative nonground literals Chan in [6] introduced a procedure here called sldcnf-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 sldcnf-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 *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 *selected* as marker for literals, and *successful* or *failed* as markers for nodes. A marked literal is called *selected*. As in Chan [6], we assume that a primitive inequality cannot be selected.

Assumption 3.1. Primitive inequalities cannot be selected;

An element of *Tree* is called:

- *successful* if at least one leaf is marked as *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 ξ 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 ξ, and let $\theta = (\alpha_1 \cdots \alpha_n)_{|vars(Q)}$. Then the equality formula $\exists(E_\theta \wedge E_{Q'})$ 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 $(\mathcal{T}, T, subs)$ s.t. $\mathcal{T}$ is a set of trees in *Tree*, T is an element of $\mathcal{T}$ called *main tree of $\mathcal{T}$*, and *subs* is a subsidiary function. It is inductively defined as follows:

1. $(\{T\}, T, subs)$ 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. *subs* is everywhere undefined.
2. If Γ is a pre-`sldcnf`-tree, then any *extension* of Γ is a pre-`sldcnf`-tree.

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

1. Mark all leaves consisting of reduced queries as *successful*.
2. For every unmarked leaf Q in some tree T in Γ, 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 *subs*(Q) is *undefined* then add the tree T' with the single node A to $\mathcal{T}$ and set *subs*(Q) to T';
      ii. if *subs*(Q) is *defined* then
         a. if *subs*(Q) is *finitely failed* then add $Q - \{L\}$ as son of Q in T, with one marked literal, if non-reduced;
         b. if *subs*(Q) is *successful* and the disjunction of its *answers* is equivalent to *true* then mark Q as *failed*;

c. if $subs(Q)$ is *finitely successful* then let $NA_1 \vee \cdots \vee NA_n$ be the disjunction of simple equality formulas obtained by *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 *failed*;
   ii. otherwise add $(Q - \{L\})\theta$ with one marked literal, if non-reduced, as son of Q in T, where $\theta = mgu(s, t)$.

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 *failed*.

In the definition of extension of a pre-`sldcnf`-tree, we assume that full answers are *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 $ext(\Gamma)$ denote the set of extensions of Γ.

Let $\mathscr{PT}$ denote the set of pre-sldcnf-trees in P. Consider the partial ordered set $(\mathscr{PT}, \leq)$, where $\leq$ is the reflexive and transitive closure of the relation *Rel*, which is the minimal relation on pre-sldcnf-trees s.t. (Γ, Γ') is in *Rel*, for every $\Gamma' \in ext(\Gamma)$. 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 $C(\mathscr{PT}, \leq)$ be the completion of $(\mathscr{PT}, \leq)$.

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

An answer for Q in the main tree of an `sldcnf`-tree Γ for Q is simply called an *answer for Q (in Γ)*.

To define `sldcnf`-derivations and finite `sldcnf`-trees, we use the notion of path. A *path* in Γ 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 Γ, or N_{i+1} is the root of the tree $subs(N_i)$.

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

Definition 3.5. (finite `sldcnf`-tree) An `sldcnf`-tree is *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 Γ is *via a selection rule R* if in the sequence of pre-`sldcnf`-trees whose limit is Γ 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.

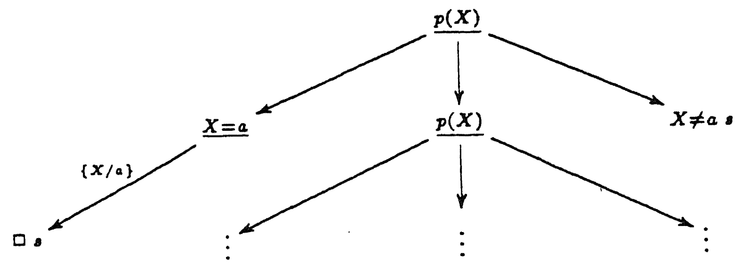
In the following two sections we shall provide a syntactic characterization of terminating programs, and a quasi-syntactic characterization of left-terminating programs. We conclude this section with a simple example to illustrate `sldcnf`-resolution. Here and in the other examples of the paper, a selected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

Example 3.1. Consider the program `toy` given in the Introduction. The query $\neg\, p(X)$ fails since its associated tree *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 *subs*($\neg\, p(X)$) is not finite. The main tree of the `sldcnf`-tree for $\neg\, p(X)$ and the subtree *subs*($\neg\, p(X)$) are represented below.

The main tree:

$$\underline{\neg\, p(X)}\, f$$

The tree *subs*($\neg\, p(X)$):



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 $|\ |$, from ground literals which are not constraints to natural numbers s.t. $|\neg A| = |A|$.

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* $|\ |$ 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 $|\ |$ s.t. P is acyclic w.r.t. $|\ |$.

In [1], it was proven that an acyclic program is terminating when `sldnf`-resolution is used. We prove here that an analogous result holds when `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 $|\ |$ if the set $\|L\| = \{|L'| \mid L' \text{ ground instance of } L\}$ is finite. A query $Q = L_1, \ldots, L_n$ is bounded w.r.t. $|\ |$ if every L_i is bounded w.r.t. $|\ |$, 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 $[\![L_{i1}]\!], \ldots, [\![L_{in}]\!]$, where for $i \in [1, n]$ we have that $i \in \{i1, \ldots, in\}$ iff L_i is *not* a constraint. These quantities will be used in the sequel.

Recall that a *multiset* is a 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 (**N**, $<$) of natural numbers to itself, giving the multiplicity of each natural number. Then the ordering $<_{mul}$ 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 $<_{mul}$ is also well-founded, as a consequence of the König Lemma for infinite terms. For simplicity we shall omit in the sequel the subscript *mult* from $<_{mul}$.

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* θ, $L\theta$ *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* θ *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 θ 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 θ be a substitution. Suppose that $\alpha = mgu(s\theta, t\theta)$ exists. Then $\mu = mgu(s, t)$ exists and is s.t. $\theta\alpha = \mu\sigma$, for a suitable σ.

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 θ 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 `sldcnf`*-resolvent Q', then Q, with L as selected literal, has an* `sldcnf`*-resolvent Q'' s.t. $Q' = Q''\theta'$ for some substitution θ'.*

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 $\alpha = mgu(s\theta, t\theta)$. Then by Proposition 4.1 $\mu = mgu(s, t)$ exists and $\theta\alpha = \mu\sigma$, for a suitable σ. 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 $L\theta\alpha$ with $R\alpha$, where $\alpha = mgu(H, L\theta)$. It is not restricted to assume that C is also variable disjoint with Q and with $vars(\theta)$. Then by Proposition 4.1 $\mu = mgu(H, L)$ exists, and $\theta\alpha = \mu\sigma$, for a suitable σ. 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 Γ be an `sdlcnf`-tree, let $\pi = Q_0, \ldots, Q_k, \ldots$ be a path of Γ, and let $k \geq 0$. Then π 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 $subs(Q_k)$.

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

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 nodes $(\pi_Q) \geq nodes(\pi_{Q'})$.

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 θ, $\pi_{Q\theta}$ 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 *subs* to its predecessor. Then by Lemma 4.3 we can lift $\pi_{Q\theta}$ to a prefix of a path in a `sldcnf`-tree for Q. Hence Q is not terminating. □

Now we can prove Theorem 4.2.

PROOF OF THEOREM 4.2. By Lemma 4.4 we have that $nodes(\pi_{Q'})$ is defined. By absurd, suppose that $nodes(\pi_{Q'}) > nodes(\pi_Q)$. Then by Lemma 4.3 we can lift $\pi_{Q'}$ to a specific prefix of a path in a `sldcnf`-tree for Q. Hence $nodes(\pi_Q) \geq nodes(\pi_{Q'})$. Absurd. □

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

Theorem 4.3. Let P be a terminating program. Then for some level mapping $|\ |$

(i) *P is acyclic w.r.t.* $|\ |$,
(ii) *for every query Q, Q is bounded w.r.t.* $|\ |$ *iff it is terminating.*

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

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

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

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

$$\begin{aligned} & nodes(\pi_{(L_1,\ldots,L_n)}) \geq \\ & \qquad nodes(\pi_{L_i}) \\ & \geq \{\text{by the definition of } |\ |\} \\ & \qquad |L_i| \\ & = l. \end{aligned}$$

(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 $|H\theta| > |L_i\theta|$ for $i \in [1, n]$ s.t. $|L_i\theta|$ is defined. Since $H\theta\theta = H\theta$, then θ is a unifier of $H\theta$ and H. Then there exists $\mu = mgu(H\theta, H)$ s.t. $\theta = \mu\theta'$ and $(L_1\mu, \ldots, L_n\mu)$ is a resolvent of $H\theta$. Then

$$\begin{aligned} & |H\theta| = \{\text{definition of } |\ |\} \\ & \qquad nodes(\pi_{H\theta}) > \\ & \{H\theta \text{ is not an inequality and } \pi_{(L_1\mu,\ldots,L_n\mu)} \text{ is a proper subfix of a path for } H\theta\} \\ & \qquad \text{nodes}(\pi_{(L_1\mu,\ldots,L_n\mu)}) \\ & \geq \{\text{part (ii} \leftarrow\text{), since } L_i\theta \in [\![L_1\mu, \ldots, L_n\mu]\!]\} \\ & \qquad |L_i\theta|. \end{aligned}$$

(ii $\rightarrow$) Consider a query Q which is bounded w.r.t. $| \ |$. 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 $L_{\bar{n}}$ that fails. This is sufficient since, due to the Prolog selection rule, literals after $L_{\bar{n}}$ will not be executed. To compute $\bar{n}$, the class of models of *comp*(P) is considered.

Definition 5.1. (Acceptable Program) Let $| \ |$ be a level mapping for P and let I be a model of *comp*(P). P is *acceptable w.r.t.* $| \ |$ *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

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

P is called *acceptable* if it is acceptable w.r.t. some level mapping and a model of *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 $| \ |$ be a level mapping and let I be a model of *comp*(P). For every $i \in [1, n]$ s.t. L_i is not a constraint, consider the set

$$|Q|_I^i = \{|L_i'| \ | I \models L_1', \ldots, L_{i-1}', \text{ for some ground instance } L_1', \ldots, L_i' \text{ of } L_1, \ldots, L_i\}$$

Definition 5.2. (Bounded Query) Let $| \ |$ be a level mapping and let I be a model of *comp*(P). A query $Q = L_1, \ldots, L_n$ is bounded (w.r.t. $| \ |$ and I) if $|Q|_I^i$ 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 $|Q|_I^i$, 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 `ldcnf`*-tree for Q in P contains only bounded queries and is finite.*

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 *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. $I \models L_i \leftarrow (c_1 \wedge \cdots \wedge c_k)$. □

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 π be a path in a `ldcnf`-tree for Q. Define π_Q to be the specific prefix of π containing the maximal number of nodes, for all paths π of a `ldcnf`-tree for Q. Let *nodes*(π_Q) be the number of nodes of π_Q.

Theorem 5.2. Let Q be a left-terminating query and let Q' be an instance of Q. Then nodes$(\pi_Q) \geq$ *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 θ, $\pi_{Q\theta}$ 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 *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 *nodes*($\pi_{Q'}$) is defined. By absurd, suppose that *nodes*$(\pi_{Q'}) >$ *nodes*(π_Q). Then by Lemma 4.3 we can lift $\pi_{Q'}$ to a specific prefix π of a path in a `ldcnf`-tree for Q. Hence we have that *nodes*$(\pi_Q) \geq$ *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 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 *nodes*(π_A) is a level mapping. From *nodes*$(\pi_{\neg A}) >$ *nodes*(π_A) it follows that *nodes*$(\pi_{\neg A}) > |\neg A| = |A|$. Choose $I = \{A \in B_P |$ there is an `ldcnf`-refutation of A in $P\}$. Then I is a model of *comp*(P).

(ii $\leftarrow$) Consider a left-terminating query Q. We show that Q is bounded by $nodes(\pi_Q)$. The case where $[\![Q]\!]_I$ is the empty set is immediate. So, let $l \in [\![Q]\!]_I$. Then for some ground instance $L_1, \ldots, L_n$ of Q and $i \in [1, \bar{n}]$ with $\bar{n} = min(\{n\} \cup \{i \in [1, n] | I \not\models L_i\})$, we have $l = |L_i|$. Then

$$
\begin{aligned}
& nodes(\pi_Q) \\
\geq\ & \{\text{Theorem } (5.2)\} \\
& \quad nodes(\pi_{(L_1, \ldots, L_n)}) \\
\geq\ & \{\text{by construction of } \pi_{(L_1, \ldots, L_n)}\} \\
& \quad nodes(\pi_{L_i, \ldots, L_{\bar{n}}}) \\
\geq\ & \{\text{because } I \models L_1, \ldots, L_{i-1}\} \\
& \quad nodes(\pi_{L_i, \ldots, L_{\bar{n}}}).
\end{aligned}
$$

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

$$
\begin{aligned}
& nodes(\pi_{L_i, \ldots, L_{\bar{n}}}) \geq \\
& \quad nodes(\pi_{L_i}) \\
\geq\ & \{\text{by definition of } |\ |\} \\
& \quad |L_i| \\
=\ & l.
\end{aligned}
$$

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

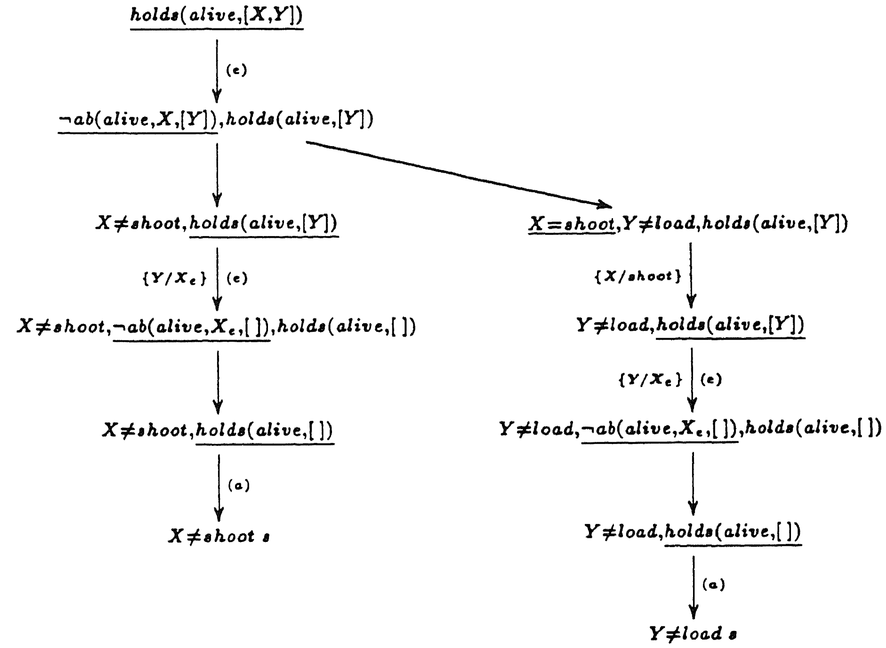
(ii $\rightarrow$) Consider a query Q, which is bounded w.r.t. $|\ |$. Then by (i) and Theorem 5.1 Q is left-terminating. □

6. APPLICATION

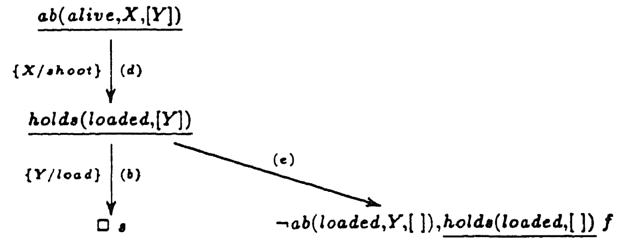
In this section we give two examples to illustrate how to formalize and implement problems in nonmonotonic reasoning by means of terminating and left-terminating programs, respectively.

6.1. Temporal Reasoning

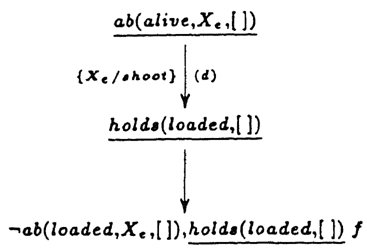
Various forms of temporal reasoning can be described using acyclic programs. In particular, the program *YSP* given in the Introduction is a formalization of the so-called Yale Shooting Problem in terms of an acyclic program. We recall the problem following [12]. Consider a person that is `alive`. The 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 $holds(t, t')$ the natural number $2l(t')$, and to a ground atom of the form $ab(t, t', t'')$ 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 `holds(alive, [X, Y])`. This query is bounded (by 4), hence it is terminating. The following is an `sldcnf`-tree for `holds(alive, [X, Y])`.



where $subs(\neg\, ab(alive, X, [Y]),\ holds(alive, [Y]))$ is the following tree.



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



Notice that by using `sldnf`-resolution `holds(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 *comp*(P^-), where P^- is defined as follows. Let 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 Neg_P^* to be the set of relations in P on which the relations in Neg_P depend on. Then P^- is the set of clauses in P in whose head a relation from Neg_P^* occurs. We call good *model of P* a model of P which is also a model of *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 *spec* is defined by the clause (a), s.t. $spec(n1, n2, n, g)$ is true if $n1, n2$ are two nodes of a given graph g, and n is a node that does not occur in any acyclic path of g connecting $n1$ with $n2$. The relation *spec* is specified as the negation of another relation, called *unspec*, where $unspec(n1, n2, n, g)$ is true if there is an acyclic path of g connecting $n1$ and $n2$ that contains n.

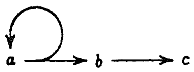
Acyclic paths of a graph are described by the relation *path*, defined by the clause (c), where $path(n1, n2, g, p)$ calls the query $path1(n1, [n2], g, p)$. Here the second argument of *path*1 is used to construct incrementally a path connecting $n1$ with $n2$: using clause (e), the partial path $[x|p1]$ is transformed in $[y, x|p1]$ if there is an edge $[y, x]$ in the graph g such that y is not already present in $[x|p1]$. The construction terminates if y is equal to $n1$, thanks to clause (d).

So the relation *path*1 is defined inductively by the clauses (d) and (e), using the familiar relation *mem*, defined by the clauses (f) and (g).

Notice that, from fact (d) it follows that if $n1$ and $n2$ are equal, then $[n1]$ is assumed to be an acyclic path from $n1$ and $n2$, for any term g.

```
(a)  spec(N1, N2, N, G) ←
     ¬unspec(N1, N2, N, G).
(b)  unspec(N1, N2, N, G) ←
     path(N1, N2, G, P),
     mem(N, P).
(c)  path(N1, N2, G, P) ←
     path1(N1, [N2], G, P).
(d)  path1(N1, [X1|P1], G, [N|1P1]) ←.
(e)  path1(N1, [X1|P1], G, P) ←
     mem([Y1, X1], G),
     ¬mem(Y1, [X1|P1]),
     path1(N1, [Y1, X1|P1], G, P).
(f)  mem(X, [X|Y]) ← .
(g)  mem(X, [Y|Z]) ←
     mem(X, Z).
```

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.



Notice that `specialize` is not terminating: for instance, the query `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 `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 `ldcnf`-tree for Q, where *edges* denotes the list $[[a,b],[b,c],[a,a]]$.

$$
\begin{array}{c}
spec(a,b,X,edges)\\
\big\downarrow \scriptstyle (a)\\
\neg unspec(a,b,X,edges)\\
\big\downarrow\\
X \neq a, X \neq b \;\; s
\end{array}
$$

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

$$
\begin{array}{c}
unspec(a,b,X,edges)\\
\big\downarrow \scriptstyle (b)\\
\underline{path(a,b,edges,P)},mem(X,P)\\
\big\downarrow \scriptstyle (c)\\
\underline{path1(a,[b],edges,P)},mem(X,P)\\
\big\downarrow \scriptstyle (e)\\
\underline{mem([Y1,b],edges)},\neg mem(Y1,[b]),path1(a,[Y1,b],edges,P),mem(X,P)\\
\scriptstyle \{Y1/a\} \big\downarrow \scriptstyle (f)\\
\underline{\neg mem(a,[b])},path1(a,[a,b],edges,P),mem(X,P)\\
\big\downarrow\\
\underline{path1(a,[a,b],edges,P)},mem(X,P)\\
\scriptstyle \{P/[a,b]\} \big\downarrow \scriptstyle (d)\\
\underline{mem(X,[a,b])}
\end{array}
$$

$$
\begin{array}{ccc}
 & \underline{mem(X,[a,b])} & \\
\swarrow \scriptstyle \{X/a\}\,(f) & & \searrow \scriptstyle (g)\\
\square\;\; s & & \underline{mem(X,[b])}\\
 & & \scriptstyle \{X/b\} \big\downarrow \scriptstyle (f)\\
 & & \square\;\; s
\end{array}
$$

Note that for simplicity we omitted to draw the derivations whose leaves are marked as *failed*, and the tree $(subs(\underline{\neg\, mem(a,[b])},\ path1(a,[a,b],P),$ $mem(X,P))$ is the finite failed tree

$$\begin{array}{c} mem(a,[b]) \\ \Big\downarrow{\scriptstyle (g)} \\ mem(a,[\,])\ f \end{array}$$

Notice that by using `ldnf`-resolution Q does flounder.

We prove now that `specialize` is acceptable. To this end, one has to find a proper level mapping and a model of `specialize` that is a model of the completion *comp*(`specialize`$^-$). Notice that `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 *spec* using the relations defined in (b)–(g), then to prove that $spec(n1,n2,n,g)$ is left-terminating it is sufficient to show that the program `spec1` consisting of the clauses (b)–(g) is acceptable and that $unspec(n1,n2,n,g)$ is bounded. In this way one has just to consider a model of `spec1` that is a model of $comp(spec1^-)$, 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: $\|[t|t_s]\| = |t_s| + 1$ and $|f(\mathtt{t})| = 0$ if $f \neq [-|-]$.

For a list l, let $set(l)$ denote the set of its elements, i.e., $set(l) = \{\ \}$ if $l = [\ ]$ and $set(l) = \{x\} \cup set(y)$ 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 $I = I_{unspec} \cup I_{path} \cup I_{path1} \cup I_{mem}$, with

$$I_{unspec} = [unspec(N1, N2, N, G)],$$

where $[A]$ denotes the set of all ground instances of A;

$$I_{path} = \{path(n1,n2,g,p) \mid\ |g| + 1 \geq |p|\};$$

$$I_{path1} = \{path1(n1,p1,g,p) \mid\ |p1| - |p1 \cap g| \geq |p| - |p \cap g|\};$$

$$I_{mem} = \{mem(s,t) \mid t \text{ list s.t. } s \in set(t)\}.$$

*Lemma 6.1. I is a model of spec*1.

PROOF.

- It follows immediate that I is a model of clause (b).
- Consider clause (c). Suppose that $I \models path1(n1,[n2],g,p)$. Note that $\|[n2]\| - \|[n2] \cap g| \leq 1$. Then $|p| - |p \cap g| \leq 1$. But $|p \cap g| \leq |g|$. Then $|p| \leq |g| + 1$, hence $I \models path(n1,n2,g,p)$.
- We have that I models $path1(n1,[n1|p1],g,[n1|p1])$, hence it models clause (d).
- Consider clause (e). Suppose that

$$I \models mem([y1,x1],g),\ \neg\, mem(y1,[x1|p1]),\ path(n1,[y_1,x1|p1],g,p).$$

Then $\|[y1, x1|p1]\| - |[y1, x1|p1] \cap g| \geq |p| - |p \cap g|$, where $y1 \notin [x1|p1]$ and $[y1, x1] \in g$. Then $|[y1, x1|p1] \cap g| = 1 + |[x1|p1] \cap g|$. So $\|[y1, x1|p1]\| - |[y1, x1|p1] \cap g| = \|[x1|p1]\| - |[x1|p1] \cap g|$. Then $\|[x1|p1]\| - |[x1|p1] \cap g| \geq |p| - |p \cap g|$. Hence $I \models path1(n1, [x1|p1], g, p)$.

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

Consider $spec1^- = \{(f), (g)\}$: it is easy to check that I_{mem} is a model of $comp(spec1^-)$.

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

$$|mem(s, t)| = |t|;$$

$$|path1(n1, p1, g, p)| = |p1| + |g| + 2(|g| - |p1 \cap g|) + 1;$$

$$|path(n1, n2, g, p)| = 3|g| + 3,$$

$$|unspec(n1, n2, n, g)| = 3|g| + 4.$$

Observe that from $|g| \geq |p1 \cap g|$ it follows that $|path(n1, p1, g, p)|$ is well defined.

It is not difficult to check that `spec1` 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:

- $|unspec(n1, n2, n, g)| > 3|g| + 3$,
- $|unspec(n1, n2, n, g)| > |p|$, under the hypothesis $I \models path(n1, n2, n, p)$.

The first condition is easy to check. For the second one, observe that from $I \models path(n1, n2, n, p)$ it follows that $|g| + 1 \geq |p|$.

For clause (e) we obtain the following inequalities:

- $|path(n1, [x1|p1], g, p)| > |g|$;
- $|path1(n1, [x1|p1], g, p)| \geq \|[x1, p1]\|$, under the hypothesis $I \models mem([y1, x1]\ g)$;
- $|path1(n1, [x1|p1], g, p)| > |path1(n1, [y1, x1|p1], g, p)|$, under the hypothesis $I \models mem([y1, x1], g), \neg\, mem(y1, [x1|p1])$.

The first two inequalities are easy to check. For the third one, observe that from $I \models mem([y1, x1], g), \neg\, mem(y1, [x1|p1])$ it follows that $|[y1, x1|p1] \cap g| = 1 + |[x1|p1] \cap g|$, hence $(|g| - |[y1, x1|p1] \cap g|) = (|g| - |[x1|p1] \cap g|) - 1$.

7. CONCLUSION

In this paper we studied termination of general logic programs, when `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 coroutining, 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 Rutten 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

1. Apt, K. R., and Bezem, M., Acyclic programs, *New Generation Computing* 9:335–369 (1991).
2. Apt, K. R., and Doets, H. C., A new definition of SLDNF-resolution, *The Journal of Logic Programming* 18:177–190 (1994).
3. Apt, K. R., Marchiori, E., and Palamidessi, C., A declarative approach for first-order built-in's of Prolog, *Applicable Algebra in Engineering, Communication and Computing* 5(2, 4):151–191 (1994).
4. Apt, K. R., and Pedreschi, D., Proving termination of general Prolog programs, in: *Proceedings of the Int. Conf. on Theoretical Aspects of Computer Software*, no. 526 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1991, pp. 265–289.
5. Bezem, M., Strong termination of logic programs, *The Journal of Logic Programming* 15:79–97 (1993).
6. Chan, D., Constructive negation based on the completed database in: *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, 1988, pp. 111–125.
7. Chan, D., An extension of constructive negation and its application in coroutining; in: *Proceedings of the North American Conference on Logic Programming*, 1989, pp. 477–493.
8. Davey, B. A., and Priestley, H. A., *Introduction to Lattices and Order*, Cambridge University Press, 1990.
9. De Schreye, D., and Decorte, S., Termination of logic programs: The never-ending story, *The Journal of Logic Programming* 19, 20:199–260 (1994).
10. Dershowitz, N., Termination of rewriting, *Journal of Symbolic Computation* 3:69–115 (1987).
11. Drabent, W., What is failure? An approach to constructive negation, *Acta Informatica* 32:27–59 (1995).

12. Hanks, S., and McDermott, D., Nonmonotonic logic and temporal reasoning, *Artificial Intelligence* 33:379–412 (1987).
13. Marchiori, E., A methodology for proving termination of general logic programs, in: *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, (IJCAI-95), Montreal, 1995, pp. 356–361.
14. Stuckey, P. J., Constructive negation for constraint logic programming, in: *Proceedings of the 6th Annual Symposium on Logic in Computer Science* (LICS), Amsterdam, 1991, pp. 328–339.