A Methodology for Proving Termination of General Logic Programs

Elena Marchiori
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
e-mail: elena@cwi.nl

Abstract
This paper introduces a methodology for proving termination of general logic programs, when the Prolog selection rule is considered. This methodology combines the approaches by Apt and Bezem [1] and Apt and Pedreschi [2], and provides a simple and flexible tool for proving termination.

1 Introduction
General logic programs (glps for short) provide formalizations and implementations for special forms of non-monotonic reasoning. For example, the Prolog negation as finite failure operator has been used to implement a formulation as logic program of the temporal persistence problem in AI (see [9; 8; 1]). Termination of glp’s is a relevant topic (see [7]), also because the implementation of the operators for the negation, like Clark’s negation as failure [5] and Chan’s constructive negation [4], are based on termination conditions. Two typical examples of glp’s which behave well w.r.t. termination are the so-called acyclic and acceptable programs ([1], [2]). In fact, it was proven in [1] that when negation as finite failure is incorporated into the proof theory, a program is acyclic iff all sld-derivations with arbitrary selection rule of non-floundering ground queries are finite. Floundering is an abnormal form of termination which arises as soon as a non-ground negative literal is selected. A similar result was proven in [2] for acceptable programs, this time with the selection rule restricted to be the Prolog one, which selects always the leftmost literal of a query. In [10] it was shown how one can obtain a complete characterization (i.e. to overcome the drawback of floundering) by considering Chan’s constructive negation procedure instead of negation as finite failure.

The notion of acceptability combines the definition of acyclicity with a semantic condition, that uses a model of the program which is not acyclic. More specifically, a program \( P \) is split into two parts, say \( P_1 \) and \( P_2 \); then one part is proven to be acyclic, the other one to be acceptable, and these results are combined to conclude that the original program is terminating w.r.t. the Prolog selection rule. The decomposition of \( P \) is done in such a way that no relations defined in \( P_1 \) occur in \( P_2 \). We introduce the notion of up-acceptability, where \( P_1 \) is proven to be acceptable and \( P_2 \) to be acyclic, and the one of low-acceptability which treats the converse case (\( P_1 \) acyclic and \( P_2 \) acceptable). We illustrate the usefulness of this approach by means of examples of programs which formalize problems in non-monotonic reasoning.

Even though our main results deal with Chan’s constructive negation only, a simple inspection of the proofs shows that they hold equally well for the case of negation as finite failure.

Our approach provides a simple methodology for proving termination of glp’s, which combines the results of Bezem, Apt and Pedreschi on acyclic and acceptable programs, results widely considered as a main theoretical foundation for the study of termination of logic programs ([7]). We believe that this methodology is relevant for at least two reasons: it overcomes the drawback of [2] for proving termination due to the use of too much semantic information, and it allows to identify for which part of the program termination does or does not depend on the fixed Prolog selection rule.

The remaining of this paper is organized as follows. The next section contains some preliminaries; in Section 3 we explain the notions of acyclicity and acceptability. In Section 4, the notions of up-/low-acceptability are introduced. In Section 5, we introduce a methodology for proving termination of glp’s, based on these notions. Finally, in Section 6 we give some examples. For lack of space, proofs of the results have been omitted. They can be found in the full version of the paper.

2 Preliminaries
We follow Prolog syntax and assume that a string starting with a capital letter denotes a variable, while other strings denote constants, terms and relations. A (extended) general logic 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 ≥ 0, H is an atom, and the L_i's, called literals, are either atoms p(s), or negative literals ¬p(s), or equalities s = t, or inequalities ∀(s ≠ t), where ∀ quantifies over some (perhaps none) of the variables occurring in the inequality. Equalities and inequalities are also called constraints, denoted by c. An inequality ∀(s ≠ t) is said to be primitive if it is satisfiable but not valid. For instance, X ≠ a is primitive. In the following, the letters A, B indicate atoms, while C and Q denote a clause and a query, respectively.

Suppose that all s1d-derivations of Q are finite and do not involve the selection of any negative literals. Then there is a finite number of computed answer substitutions, say θ_1, ..., θ_k, k ≥ 0; let F_Q be the equality formula ∃(E_{θ_1} ∨ ... ∨ E_{θ_k}), where E_{θ_i} is the substitution θ_i written in equational form, and ∃ quantifies over the variables that do not occur in Q. Then the Clark's completion of P logically implies ∀(Q → F_Q), i.e., comp(P) ⊢ ∀(Q ↔ F_Q). To resolve negative non-ground literals, Chan in [4] introduced a procedure, here called s1dcnf-resolution, which maps from [10] that if P is acyclic and Q is bounded then every s1dcnf-tree for Q in P is finite; and that also the converse of this result holds. For a terminating program P, there exists a level mapping | | s.t.: (i) P is acyclic w.r.t. | |; and (ii) for every query Q, Q is bounded w.r.t. | | if all its s1dcnf-derivations are finite. Notice that when negation as finite failure is assumed, (i) holds only if Q does not flounder ([11]). In fact, simple programs, like

p(X) ← ¬ p(Y),

terminate because floundering, but are not acyclic.

To prove termination of logic programs, functions called level mappings have been used [1], which map ground atoms to natural numbers. Their extension to negative literals is selected. For instance a derivation fails finitely if a constraint which is not satisfiable is selected.

3 Acyclic and Acceptable Programs

In this section, the definitions of acyclic and acceptable program are given, together with some useful results from [10].

Definition 3.1 (Acyclic Program) A program P is acyclic w.r.t. a level mapping | | if for all ground instances H ← L_1, ..., L_m of clauses of P we have that |H| > |L_i| holds for every i ∈ [1, m] s.t. L_i is not a constraint. P is called acyclic if there exists a level mapping | | s.t. P is acyclic w.r.t. | |.

With a query Q = L_1, ..., L_n we associate n sets |Q_i| of natural numbers s.t.

|Q_i| = { |L_i| | L_i is a ground instance of L_i}. Q is called bounded w.r.t. | | if every |Q_i| is finite.

Bounded queries characterize a class of queries s.t. every their s1dcnf-derivation is finite. We have proven in [10] that if P is acyclic and Q is bounded then every s1dcnf-tree for Q in P is finite; and that also the converse of this result holds: call a program P terminating if all s1dcnf-derivations of ground queries are finite. Then, for a terminating program P, there exists a level mapping | | s.t.: (i) P is acyclic w.r.t. | |; (ii) for every query Q, Q is bounded w.r.t. | | if all its s1dcnf-derivations are finite. Notice that when negation as finite failure is assumed, (i) holds only if Q does not flounder ([11]). In fact, simple programs, like

p(X) ← ¬ p(Y),

terminate because floundering, but are not acyclic.

For studying termination of general logic programs with respect to the Prolog selection rule, the notion of acceptable program ([2]) was introduced. Its definition is based on the same condition used to define acyclic programs, except that, for a ground instance H ← L_1, ..., L_n of a clause, the test |H| > |L_i| is performed only till the literal L_i which fails. This is sufficient since, due to the Prolog selection rule, literals after L_i will not be executed. To compute N, a class of models of P, here called good models, is used. A model of P is good if its restriction to the relations from Neg_p is a model of comp(P^-), where P^- is the set of clauses in P whose head contains a relation from Neg_p, and Neg_p is defined as follows. Let Neg_p denote the set of relations in P which occur in a negative literal in the 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. Then Neg_p denotes the set of relations in P on which the relations in Neg_p depend on.

Definition 3.2 (Acceptable Program) Let | | be a level mapping for P and let I be a good model of P. P is acceptable w.r.t. | | and I if for all ground instances H ← L_1, ..., L_n of clauses of P we have that |H| > |L_i| holds for i ∈ [1, n] s.t. L_i is not a constraint, where n = min{m | i ∈ [1, n] | I ≠ L_i}. P is called acceptable if it is acceptable w.r.t. some level mapping and a good model of P.
Let $Q = L_1, \ldots, L_n$ be a query, let $||$ be a level mapping and let $I$ be a good model of $P$. Then, with $Q$ we associate $n$ sets of natural numbers s.t. for $i \in [1, n]$, 

$$Q_i = \{L_i^1, L_i^2, \ldots, L_i^n\}$$

Then $Q$ is called bounded if every $|Q_i|$ is finite.

Bounded queries characterize those queries s.t. all their 1dcnf-derivations are finite. In [10], we have shown that similar results as those for terminating programs hold also for left-terminating programs, where a program is left-terminating if all 1dcnf-derivations of ground queries are finite.

4 Up- and Low-Acceptability

To prove that a program $P$ is acceptable is in general more difficult than to prove that it is acyclic, because one has to find a good model of the program. Therefore in this section we introduce two equivalent definitions of acceptability, called up- and low-acceptability, which are simpler to be used, since one has only to find a good acceptability, called up- and low-acceptability, which are in this section we introduce two equivalent definitions of

To prove that a program $P$ is left-terminating, it is decomposed into two suitable parts: then, one part is shown to be acyclic and the other one acceptable. The following notion is used to specify the relationship between these two parts. Recall that a relation is said to be defined in a program if it occurs in the head of at least one clause of the program.

Definition 4.1 Let $P$ and $R$ be two programs. We say that $P$ extends $R$, written $P > R$, if no relation defined in $P$ occurs in $R$.

Informally, $P$ extends $R$ if $P$ defines new relations possibly using the relations defined already in $R$. Then one can imagine the program $P \cup R$ as formed by an upper part $P$ and a lower part $R$, and investigate the cases when either the lower or the upper part of the program is acyclic. This is done in the following sections, by introducing the notions of up- and low-acceptability. For a level mapping $||$, we shall denote by $||_R$ its restriction to the relations defined in the program $R$.

In the following definition, the upper part of the program is proven to be acceptable and the lower part to be acyclic. For two programs $P$, $R$, let $P \setminus R$ denote the program obtained from $P$ by deleting all clauses of $R$ and all literals defined in $R$.

Definition 4.2 (up-acceptability) Let $||$ be a level mapping for $P$. Let $R$ be a set of clauses s.t. $P = P_1 \cup R$ for some $P_1$, and let $I$ be an interpretation of $P \setminus R$. $P$ is up-acceptable w.r.t. $||$, $R$ and $I$ if the following conditions hold:

1) $P_1$ extends $R$.
2) $P \setminus R$ is acceptable w.r.t. $||_{P \setminus R}$ and $I$.
3) $R$ is acyclic w.r.t. $||_{P \setminus R}$.
4) For every ground instance $H \leftarrow L_1, \ldots, L_n$ of a clause of $P_1$, for $i \in [1, n]$, if $L_i$ is defined in $R$ and is not a constraint, then $|H| \geq |L_i|$. A program is up-acceptable if there exists $||$, $R$ and $I$ s.t. $P$ is up-acceptable w.r.t. $||$, $R$ and $I$.

Observe that for $R$ equal to the empty set of clauses, we obtain the original definition of acceptability. Now, we introduce the notion of up-bounded query. Suppose that $P$ is up-acceptable w.r.t. $||$, $R$ and $I$. Consider a query $Q = L_1, \ldots, L_n$. Then, with $Q$ we associate $n$ sets of natural numbers s.t. for $i \in [1, n]$, 

$$|Q_i|_{up,I} = \{L_i^1, L_i^2, \ldots, L_i^n\}$$

where $L_i^1, \ldots, L_i^n$ are all those literals of $L_1, \ldots, L_n$ (whose relations are) defined in $P_1$. Then $Q$ is called up-bounded if every $|Q_i|_{up,I}$ is finite. The following result holds.

Theorem 4.3 Suppose that $P$ is up-acceptable w.r.t. $||$, $R$ and $I$. Let $Q$ be an up-bounded query. Then every 1dcnf-tree for $Q$ in $P$ contains only up-bounded queries and is finite.

The following corollary establishes the equivalence of the notions of acceptability and up-acceptability.

Corollary 4.4 Let $P$ be a general logic program. Then: (i) If $P$ is up-acceptable then $P$ is acceptable. (ii) If $P$ is acceptable then it is up-acceptable.

Now, we consider the converse case, where the lower part of the program is proven to be acceptable and the upper part to be acyclic.

Definition 4.5 (low-acceptability) Let $||$ be a level mapping for $P$. Let $R$ be a set of clauses s.t. $P = P_1 \cup R$ for some $P_1$, and let $I$ be an interpretation of $R$. $P$ is low-acceptable w.r.t. $||$, $R$ and $I$ if the following conditions hold: 1) $P_1$ extends $R$; 2) $P_1 \setminus R$ is acyclic w.r.t. $||_{P_1 \setminus R}$; 3) $R$ is acceptable w.r.t. $||_{P_1 \setminus R}$ and $I$; 4) for every ground instance $H \leftarrow L_1, \ldots, L_n$ of a clause of $P_1$, for $i \in [1, n]$, if $L_i$ is defined in $R$ and is not a constraint, then $|H| \geq |L_i|$. A program is low-acceptable if there exists $||$, $R$ and $I$ s.t. $P$ is low-acceptable w.r.t. $||$, $R$ and $I$. Suppose that $P$ is low-acceptable w.r.t. $||$, $R$ and $I$. Then the notion of low-boundedness is defined as in the previous section, where $|Q_i|_{low,I}$ is replaced by the set

$$|Q_i|_{low,I} = \{L_i^1, L_i^2, \ldots, L_i^n\}$$

where $L_i^1, \ldots, L_i^n$ are all those literals of $L_1, \ldots, L_n$ (whose relations are) defined in $R$. Then the corresponding of Theorem 4.3 and Corollary 4.4 hold, where $up$ is replaced by low.
2.1) Prove that \( P \setminus R \) is acceptable w.r.t. a level mapping, say \(|1|\), and an interpretation.

2.2) Use \(|1|\) to define a level mapping \(|2|\) for \( P \setminus R \) s.t. \( P \setminus R \) is acyclic w.r.t. \(|2|\), and s.t. for every ground instance \( H \leftarrow L_1, \ldots, L_n \) of a clause of \( P_1 \), if \( L_i \) is defined in \( P_1 \) and it is not a constraint, then \(|H|_2 \geq |L_i|_1\) holds.

3) If \( P_i \) extends \( R \) then:
3.1) Prove that \( R \) is acyclic w.r.t. a level mapping, say \(|1|\).
3.2) Use \(|1|\) to define a level mapping \(|2|\) for \( P \setminus R \) s.t. \( P \setminus R \) is acceptable w.r.t. \(|2|\) and an interpretation, and s.t. for every ground instance \( H \leftarrow L_1, \ldots, L_n \) of a clause of \( P_1 \), if \( L_i \) is defined in \( R \) and it is not a constraint, then \(|H|_2 \geq |L_i|_1\) holds.

This method overcomes a drawback of the original method of Apt and Pedreschi to prove left-termination, where one has to find a good model of all the program.

A drawback of our method one immediately observes is its lack of incrementality. In fact, it would be nice to have an incremental, bottom-up method, where the decomposition step 1. is applied iteratively to the subprograms until possible (i.e., until the partition of a subprogram becomes trivial).

In this section we illustrate by means of some examples how various problems in non-monotonic reasoning can be formalized by means of acyclic or acceptable programs. We consider the blocks-world problem, planning in the blocks-world, and search in graph structures.

### Blocks World

The blocks world is a formulation of a simple problem in AI, where a robot is allowed to perform a number of primitive actions in a simple world (see e.g. [11]). Here we consider a simple version of this problem, where there are three blocks, say \( a, b, c \), and three different places of a table, say \( p, q, r \). A block is allowed to lay either above another block or on one of these places. Blocks can be moved from one to another location. The problem consists of specifying when a configuration in the blocks world is possible, i.e., if it can be obtained from the initial situation by performing a sequence of possible moves. We use McCarthy and Hayes situation calculus to formulate the problem, in terms of facts, events and situations. One can distinguish three types of facts: \( \text{loc}(X, L) \) stands for a block \( X \) is in the location \( L \); \( \text{on}(X,Y) \) for a block \( X \) is on a block \( Y \); and \( \text{clear}(L) \) for there is no block in the location \( L \). It is sufficient to consider only one type of event, namely \( \text{move} (X, L) \) into a location \( L \), denoted by \( \text{move}(X, L) \).

Finally, we represent situations by means of lists: \([\ ]\) stands for the initial situation, and \([X e|X s]\) for the one corresponding to the occurrence of the event \( X e \) in the situation \( X s \). Based on the above representation, one can formalize the blocks world by means of the following program `blocks-world`, where \( \text{top}(X) \) denotes the top of the block \( X \), and \( B = \{a, b, c\} \), \( P = \{p, q, r, \text{top}(a), \text{top}(b), \text{top}(c)\} \), and \( L = \{\text{loc}(a, p), \text{loc}(b, q), \text{loc}(c, r)\} \):
The initial situation is described by clauses (loc). The relation holds is used to describe when a fact is possible in a certain situation, while the relation legal-s specifies when a configuration is possible in a certain situation. It is easy to check that blocks-world is acyclic w.r.t. the following level mapping | |, where we use the function | | from ground terms to natural numbers s.t. if y is a list then |y| is its length, otherwise |y| is 0.

Consider for instance the query holds(on(a, Y), Xs): it is bounded, hence every its sldcfn-derivation is finite. We obtain the answers (Y = .nd sibie moves which yield a particular configuration. This blocks-world which define the relation legal-s, whose plan.ning follows this approach, where the clauses of the program, Note that here the initial configuration is any situation which can be reached from the initial state. To prove that planning is left-terminating using Definition 3,2 is rather difficult, because it requires to find a model of planning, which is a model of the completion of the program consisting of the clauses (m1) and (m2) and of all the clauses of blocks-world, but (h3), (h4), (s).

Planning in the Blocks World
We consider now plan-formations in the blocks world, which amounts to the specification of a sequence of possible moves which yield a particular configuration. This problem can be solved by means of a nondeterministic algorithm ([12]): while the desired state is not reached, find a legal action, update the current state, check that it has not been visited before. The following program planning follows this approach, where the clauses of blocks-world which define the relation legal-s, whose union is denoted by r-blocks-world, are supposed to be included in the program. Note that here the initial configuration is any situation which can be reached from the initial state (which is described by the clauses (loc) of blocks-world). Alternatively, as done in [12], one could let unspecified the initialization, which would be provided every time the program is tested.

(\(h3\)) holds(clear(L),Xs) \rightarrow
- busy(L),Xs.
(\(ah\)) abnormal(loc(X,L), move(X,L'),Xs) \rightarrow
(\(bh\)) busy(L),Xs) \rightarrow holds(loc(X,L),Xs).
(\(st\)) legal-s([(a,L1),(b,L2),(c,L3)],Xs) \rightarrow
holds(loc(a,L1),Xs),
holds(loc(b,L2),Xs),
holds(loc(c,L3),Xs).

To prove that planning is left-terminating using Definition 3,2 is rather difficult, because it requires to find a model of planning, which is a model of the completion of the program consisting of the clauses (m1) and (m2) and of all the clauses of blocks-world, but (h3), (h4), (s).

Planning in the Blocks World
We consider now plan-formations in the blocks world, which amounts to the specification of a sequence of possible moves which yield a particular configuration. This problem can be solved by means of a nondeterministic algorithm ([12]): while the desired state is not reached, find a legal action, update the current state, check that it has not been visited before. The following program planning follows this approach, where the clauses of blocks-world which define the relation legal-s, whose plan.ning follows this approach, where the clauses of the program, Note that here the initial configuration is any situation which can be reached from the initial state. To prove that planning is left-terminating using Definition 3,2 is rather difficult, because it requires to find a model of planning, which is a model of the completion of the program consisting of the clauses (m1) and (m2) and of all the clauses of blocks-world, but (h3), (h4), (s).

Planning in the Blocks World
We consider now plan-formations in the blocks world, which amounts to the specification of a sequence of possible moves which yield a particular configuration. This problem can be solved by means of a nondeterministic algorithm ([12]): while the desired state is not reached, find a legal action, update the current state, check that it has not been visited before. The following program planning follows this approach, where the clauses of blocks-world which define the relation legal-s, whose plan.ning follows this approach, where the clauses of the program, Note that here the initial configuration is any situation which can be reached from the initial state. To prove that planning is left-terminating using Definition 3,2 is rather difficult, because it requires to find a model of planning, which is a model of the completion of the program consisting of the clauses (m1) and (m2) and of all the clauses of blocks-world, but (h3), (h4), (s).
unspec(n1, n2, n, g) is true if there is an acyclic path of the graph g connecting the nodes n1 and n2 and containing 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 path1 is used to construct incrementally a path connecting n1 with n2: using clause (e), the partial path [x; p] is transformed in [y; x; p] if there is an edge [y; x] in the graph g such that y is not already present in [x; p]. The construction terminates if y is equal to n1, thanks to clause (d). The relation path1 is defined inductively by the clauses (d) and (e), using the familiar relation member, specified by the clauses (f) and (g).

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

1. spec(n1, n2, N, G) ← 
   ~ unspec(n1, n2, N, G).
2. unspec(n1, n2, N, G) ← 
   path(n1, n2, G, P), member(P).
3. path(n1, n2, G, P) ← 
   path(n1, [n2], G, P).
4. path1(n1, [n1; P1], G, [n1; P1]) ←.
5. member(Y, [X; P1]), 
   path1(n1, [Y; X1; P1], G, P).
6. member(X, [X1; Y]), 
   member(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.

\[ a \rightarrow b \rightarrow c \]

Observe that specialize is not terminating: for instance, the query path1(a, [b, c], d, e) has an infinite derivation obtained by choosing as input clause (a variant of) the clause (e) and by selecting always its rightmost literal.

However specialize is left-terminating. Note that to prove this result using Definition 3.2 requires to find a suitable model of the completion of the program, which is rather difficult. Therefore we prove left-termination by means of low-acceptability.

We prove that specialize is low-acceptable w.r.t. |, spec1 and I, defined as follows. spec1 is the program consisting of all the clauses of specialize but (a). Let spec2 be the program consisting of the clause (a) of specialize. Define the level mapping | | as follows:

\[ |spec(n1, n2, n, g)| = |unspec(n1, n2, n, g)| + 1, \]
\[ |unspec(n1, n2, n, g)| = 0, \]
\[ |member(s, t)| = |t|; \]
\[ |path1(n1, p1, g)| = |p1| + |g| + 2(|g| - |p1 ∩ g|) + 1, \]
\[ |path(n1, n2, g, p)| = 3|g| + 3, \]
\[ |unspec(n1, n2, n, g)| = 3|g| + 4, \]

where for two lists p and g p ∩ g denotes the list containing all elements that both are elements of p and g and such that there exists a y s.t. [x; y] is an element of g.

Let I = unspec ∪ path1 ∪ path ∪ member, where:

\[ I_{unspec} = \{ unspec(N1, N2, N, G) \}, \]
\[ I_{path} = \{ path(n1, n2, g, p) | |g| + 1 ≥ |p| \}, \]
\[ I_{path1} = \{ path1(n1, p1, g, p) | \]
\[ |p1| - |p1 ∩ g| ≥ |p| - |p ∩ g| \}, \]
\[ I_{member} = \{ member(s, t) | s list s.t. s ∈ set(t) \}. \]

It is easy to prove that I is a model of spec1. Moreover N_{spec}^\star = \{ member \} and spec1 = \{ (f), (g) \}. Then I restricted to member is a model of comp(spec1).

Conditions 1-4 of the definition of low-acceptability are easy to check. Consider now the query Q = spec(a, b, X, [[a, b], [b, c], [a, a]]). It is low-bounded. Then one obtains a finite 1dncf-tree for Q, with answer \( X \neq a \land X \neq b \). Notice that by using negation as failure Q does flounder.

Acknowledgments

I would like to thank Krzysztof Apt, Jan Rutten, and Frank Teusink, and the referees. This research was partly supported by the Esprit Basic Research Action 6810 (Compulog 2).

References