Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

A methodology for proving termination of general logic programs

E. Marchiori

Computer Science/Department of Software Technology

CS-R9540 1995

Report CS-R9540
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

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

Termination of logic programs with negated body atoms, here called general logic programs, is an important topic. This is also due to the fact that the computational mechanisms used to process negated atoms, like Clark's negation as failure and Chan's constructive negation, are based on termination conditions. This paper introduces a methodology for proving termination of general logic programs, when the Prolog selection rule is considered. This methodology is based on the notions of low-/up-acceptable program. We prove that low-/up-acceptable programs characterize a class of general logic programs which terminate for a large class of queries, which contains the set of all ground queries. We consider as operational model SLD-resolution augmented with a procedure to deal with negative literals, known as Chan's constructive negation. General logic programs can be used to express concepts and problems in non-monotonic reasoning. We show here, that interesting problems in non-monotonic reasoning can be formalized and implemented by means of up-/low-general logic programs.

AMS Subject Classification (1991): 68Q40, 68Q60, 68T15.
CR Subject Classification (1991): F.3.1., F.4.1, I.2.3.
Keywords: general logic programs, termination, non-monotonic reasoning.
Note: A preliminary version of this paper will appear in: Proceedings of the 1995 International Joint Conference on Artificial Intelligence (IJCAI'95).

1 Introduction

General logic programs (glp's for short) provide formalizations and implementations for special forms of non-monotonic reasoning (see e.g. [2, 7]). 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 Artificial Intelligence (see [16, 13, 1]). Termination of glp's is a relevant topic (see [12] for a general discussion on termination in Logic Programming), also because the implementation of the operators for the negation, like Clark's negation as failure [10] and Chan's constructive negation [9], 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], [5]). 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 (see e.g. [2]). A similar result was proven in [5] for acceptable programs, this time with the selection rule fixed to be the Prolog one, which selects always the leftmost literal of a query. In [17] 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 sldnf-resolution.

The notion of acceptability combines the definition of acyclicity with a semantic condition, that uses a model of the program which has also to be a model of the completion of its "negative part" (see Definition 3.4). Because of this semantic condition, the proof of acceptability may become rather cumbersome. Moreover, finding a model which satisfies the above requirement may be rather difficult.

In this paper we refine the notion of acceptability, by using a semantic condition which refers only to that part 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 prove that the notions of up- and low-acceptability are equivalent to the original definition of acceptability. This result is important because it allows to integrate the two notions in a bottom-up, incremental methodology for proving termination. We illustrate the usefulness of this approach by means of examples of programs which formalize problems in non-monotonic reasoning. In particular, we show that the *planning in the block world* problem can be formalized and implemented by means of an up-acceptable program. This provides a class of queries (up-bounded queries) which can be completely answered in this program, thus yielding answers to the corresponding questions for the planning in the block world problem.

Even though our main theorems (Theorem 4.4 and 4.10) 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 ([12]). We believe that this methodology is relevant for at least two reasons: it overcomes the drawback of [5] 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. Moreover, the examples we give to illustrate the application of our approach, enphazise the fact that systems based on the logic programming paradigm provide a suitable formalization and implementation for non-monotonic reasoning.

The remaining of this paper is organized as follows. Section 2 contains some terminology and notation. In Section 3, the notions of acyclicity and acceptability are explained together with some useful results. In Section 4, the notions of up-/low-acceptability are introduced and discussed. In Section 5 a methodology for proving termination is introduced. Sections 6 and 7 contain examples in non-monotonic reasoning. Finally, in Section 8 we give some conclusions.

2 Notation and Terminology

The following notation will be used. 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. Relation symbols are often denoted by p, q, r. 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 $m \geq 0$, H is an atom, and L_i is a literal. Here we call literals, denoted by L, not only an atom $p(\mathbf{s})$, or a negative literal $\neg p(\mathbf{s})$, but also 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 inequality $\forall(s \neq t)$ is said to be *primitive* if it is satisfiable but not valid. For instance, $X \neq a$ is primitive. In the following, the letters A, B indicate atoms, while C and Q denote a clause and a query, respectively. Moreover, for a substitution $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$, we denote by E_θ the equality formula $(X_1 = t_1 \wedge \ldots \wedge X_n = t_n)$.

In `sld`-resolution, for a program P and a query Q, if θ is a computed answer substitution for Q then it can be written in equational form as $\exists(X_1 = X_1\theta \wedge \ldots \wedge X_n = X_n\theta)$, 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 is a finite number of computed answer substitutions for Q in P, say $\theta_1, \ldots, \theta_k$, with $k \geq 0$. Let F_Q be the equality formula $\exists(E_{\theta_1} \vee \ldots \vee E_{\theta_k})$, where $\exists$ quantifies over the variables that do not occur in Q. Then the Clark's completion of P logically implies $\forall(Q \leftrightarrow F_Q)$, i.e.,

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

To resolve negative non-ground literals, Chan in [9] 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 undefined when Q has an infinite derivation. Then, the notion of (infinite) derivation in this setting is not always defined. Therefore in this paper we refer to an alternative definition of the Chan procedure introduced in [17], 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.

In Section 3, we shall consider a fixed selection rule, where at every resolution step, the leftmost *possible literal* is selected, where a literal is called possible if it is not a primitive inequality. Intuitively, the selection of primitive inequalities is delayed until their free variables become enough instantiated to render the inequalities valid or unsatisfiable. We refer to this selection rule as *Prolog selection rule*, because it coincides with the Prolog one for programs without constraints. The `sldcnf`-trees with Prolog selection rule are here called `ldcnf`-trees.

To prove termination of 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 [5], 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 which are not constraints. However, note that the presence of constraints in a query influences termination, because for instance a derivation fails finitely if a constraint which is not satisfiable is selected.

Definition 2.1 (Level Mapping) A *level mapping* is a function $|\ |$ from ground literals which are not constraints to natural numbers s.t. $|\neg A| = |A|$.

In the following sections we introduce the notions of acyclic and acceptable program.

3 Acyclic and Acceptable Programs

In this section, the definitions of acyclic and acceptable program are given, together with some useful results from [17].

To study termination of general logic programs w.r.t. an arbitrary selection rule, Apt and Bezem introduced the notion of acyclic program.

Definition 3.1 (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 $i \in [1, m]$ 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 $Q = L_1, \ldots, L_n$ we associate n sets $|Q|_i$ of natural numbers s.t.

$$|Q|_i = \{|L_i'| \mid L_i' \text{ is a ground instance of } L_i\}.$$

Q is called bounded w.r.t. $|\ |$ if every $|Q|_i$ is finite.

Bounded queries characterize a class of queries s.t. every their `sldcnf`-derivation is finite.

Theorem 3.2 *Let P be an acyclic program and let Q be a bounded query. Then every* `sldcnf`*-tree for Q in P contains only bounded queries and is finite.*

The converse of Theorem 3.2 also holds. A query is *terminating* (w.r.t. P) if all its `sldcnf`-derivations (in P) are finite. A program P is said to be *terminating* if all ground queries are terminating w.r.t. P.

Theorem 3.3 *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 Q is terminating.*

From Theorems 3.2 and 3.3 it follows that terminating programs coincide with acyclic programs and that for acyclic programs a query has a finite `sldcnf`-tree if and only if it is bounded. Notice that when negation as finite failure is assumed, Theorem 3.3 holds only if Q does not flounder ([1]). For instance, the program:

```
p(X) ← ¬ p(Y).
```

is terminating (floundering) but it is not acyclic.

For studying termination of general logic programs with respect to the Prolog selection rule, Apt and Pedreschi in [5] introduced the notion of acceptable program. This notion is based on the same condition used to define acyclic programs, except that, for a ground instance $H \leftarrow L_1, \ldots, L_n$ of a clause, the test $|H| > |L_i|$ is performed only till the first literal $L_{\overline{n}}$ which fails. This is sufficient since, due to the Prolog selection rule, literals after $L_{\overline{n}}$ will not be selected. To compute $\overline{n}$, 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 3.4 (Acceptable Program) Let $|\ |$ be a level mapping for P and let I be an interpretation of P. P is called *acceptable w.r.t.* $|\ |$ *and* I if I is a good model of P and for all ground instances $H \leftarrow L_1, \ldots, L_n$ of clauses of P we have that $|H| > |L_i|$ holds for $i \in [1, \overline{n}]$ s.t. L_i is not a constraint, where $\overline{n} = min(\{n\} \cup \{i \in [1, n] \mid I \not\models L_i\})$.

P is called *acceptable* if it is acceptable w.r.t. some level mapping and interpretation.

Let $Q = L_1, \ldots, L_n$ be a query, let $|\ |$ be a level mapping and let I be a good model of P. For every $i \in [1, n]$ s.t. L_i is not a constraint, consider the set

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

Definition 3.5 (Bounded Query) Let $|\ |$ be a level mapping and let I be a good model of 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 set containing the maximum of $|Q|_I^i$, for every L_i which is not a constraint. Then Q is bounded by k if $k \geq l$, for every $l \in |[Q]|_I$.

Bounded queries characterize those queries s.t. all their `ldcnf`-derivations are finite.

Theorem 3.6 *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.*

A query is called *left-terminating* (w.r.t. P) if all its `ldcnf`-derivations are finite. A program P is called *left-terminating* if every ground query is left-terminating w.r.t. P.

Theorem 3.7 *Let P be a left-terminating program. Then for some level mapping $|\ |$, and for a good model I of 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.*

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 model of a subprogram, which is obtained discarding those clauses forming an acyclic program. Informally, to prove that a program is left-terminating, it is decomposed into two suitable parts: then, one part is shown to be acyclic and the other one to be acceptable. The following notion, also used e.g. in [4] to prove in a modular way the termination of pure (i.e. without negation) Prolog programs with built-in's, is used to specify the relationship between these two parts. A relation is said to be defined in a program if it occurs in the head of at least one clause of the program. Moreover, a literal is defined in a program P if its relation (symbol) is defined in P.

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. For instance, the program

```
p ← q,r.
```

extends the program

```
q ← s.
s ←.
```

Then one can imagine the program $P \cup R$ as formed by an *up*per part P and a *low*er 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.

4.1 Up-Acceptability

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, say P and R, let $P \setminus R$ denote the program obtained from P by deleting all clauses of R and all literals defined in R. For instance, if P consists of the clause $p \leftarrow q, r$ and R is the clause $r \leftarrow s$, then $P \setminus R$ is the program $p \leftarrow q$.

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 called *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. $|\ |_{|R}$;

4. for every ground instance $H \leftarrow L_1, \ldots, L_n$ of a clause of P_1, for $i \in [1, n]$, let $L_{i1}, \ldots, L_{ik}$ be those literals among $L_1, \ldots, L_i$ which are defined in P_1. Then, if $I \models L_{i1}, \ldots, L_{ik}$, and if L_i is defined in R and is not a constraint, then $|H| \geq |L_i|$.

A program is called *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*.

Definition 4.3 (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 every L_i which is not a constraint, we associate the following set of natural numbers:

$$|Q|_i^{up,I} = \{|L'_i| \mid L'_1, \ldots, L'_n \text{ is a ground instance of } Q \text{ and } I \models L'_{k_1} \wedge \ldots \wedge L'_{k_l}\},$$

where $L'_{k_1}, \ldots, L'_{k_l}$ are all those literals of $L'_1, \ldots, L'_{i-1}$ whose relations are defined in P_1. Then Q is called up-bounded, if every $|Q|_i^{up,I}$ is finite.

We now prove that all the `ldcnf`-derivations of an up-bounded query are finite. We show that every query in an `ldcnf`-derivation of an up-bounded query is up-bounded. Moreover, we associate with every up-bounded query a value in a well-founded set, and show that, if Q is a query of the `ldcnf`-derivation and Q' is its resolvent, then the value associated with Q' is smaller than the one associated with Q. We choose as well-founded set the set of pairs of multisets of natural numbers, ordered by means of the lexicographic order. Recall that a *multiset* (see e.g. [11]) is a unordered collection in which the number of occurrences of each element is significant. 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. For simplicity we shall omit in the sequel the subscript *mult* from $<_{mul}$.

With an up-bounded query Q, is associated a pair $\pi(Q)_{up,I} = (|[Q]|_{up,I,P_1}, |[Q]|_{up,I,R})$ of multisets, where for a program P, the set $|[Q]|_{up,I,P}$ is defined as

$$|[Q]|_{up,I,P} = bag(max|Q|_{k_1}^{up,I}, \ldots, max|Q|_{k_m}^{up,I}),$$

where $L_{k_1}, \ldots, L_{k_m}$ are those literals of Q which are not constraints and which are defined in P; and $max|Q|_i^{up,I}$ denotes the maximum of $|Q|_i^{up,I}$, which is assumed to be equal to 0 if $|Q|_i^{up,I}$ is the empty set.

Recall that the lexicographic order on pairs of finite multisets, denoted by $\prec$, is s.t. $(X, Y) \prec (Z, W)$ iff either $X < Z$, or $X = Z$ and $Y < W$. Here X, Y, Z, W denote (finite) multisets and $<$ denotes the multiset order on multisets of natural numbers. Then we can prove the following result.

Theorem 4.4 *Suppose that P is up-acceptable w.r.t. $|\,|$, R and I. Let Q be an up-bounded query. Then every* `ldcnf`*-derivation for Q in P contains only up-bounded queries and is finite.*

Proof. Let ξ be a `ldcnf`-derivation for Q in P. We prove by induction on the number n of elements of ξ that every query of ξ is up-bounded, and that for every two consecutive queries of ξ, say Q_1 and Q_2, if the selected literal of Q_1 is not a constraint, then $\pi(Q_1)_{up,I} > \pi(Q_2)_{up,I}$. The base case ($n = 1$) is immediate. Now suppose that $n > 1$, and that we have proven the result for all $i \leq k$, for some $k < n$. Let $Q_1 = L_1, \ldots, L_n$ be the k-th query of ξ, and let L_i be its selected literal. Then, Q_1 is up-bounded by the induction hypothesis. Let Q_2 be the resolvent of Q_1. That Q_2 is up-bounded follows by Q_1 up-bounded and by the definition of up-acceptability (here also condition 4 is used). Now, suppose that L_i is not a constraint. Then, we show that $\pi(Q_2)_{up,I}$ is smaller than $\pi(Q_1)_{up,I}$ in the lexicographic order. If L_i is defined in P_1 then the first component of $\pi(Q_2)_{up,I}$ becomes smaller because of condition 2. Otherwise, if L_i is defined in R then the first component of $\pi(Q_2)_{up,I}$ does not increase because of condition 1, while the second one becomes smaller because of condition 3.

Then, the conclusion follows from the fact that the lexicographic ordering is well-founded, and from the fact that, in a derivation a constraint can be consecutive selected only for a finite number of times. □

The following corollary establishes the equivalence of the notions of acceptability and up-acceptability. It follows directly from Theorem 4.4 and Theorem 3.7.

Corollary 4.5 *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.*

In some cases, as for instance for the program `hamilton` given in Section 7, the notion of up-acceptability does not help to simplify the proof of termination. However, we can define a slight generalization of this notion, where the condition that P_1 extends R is weakened as follows. For a set S of relations, denote by $P_{|S}$ the part of P containing only those clauses which define the relations from S.

Definition 4.6 Let P and R be two programs. We say that P *weakly extends* R, written $P >_w R$, if for some set S of relations we have that:

- $P = P_1 \cup P_{|S}$, and P_1 extends $P_{|S}$;
- R extends $P_{|S}$; and
- $P \setminus P_{|S}$ extends $R \setminus P_{|S}$.

For instance, the program

```
p(X) ← q(X), r(X).
r(f(X)) ← r(X).
```

weakly extends the program

```
q(X) ← s(X), r(X).
s(X) ←.
```

Note that only the relations of S which are defined in P play a role in the above definition. Moreover, observe that Definition 4.1 is a particular case of the above definition, obtained by considering $P_{|S}$ to be equal to $\emptyset$ (which includes the case where $S = \emptyset$). Then, we can define the notion of weakly up-acceptability, which is obtained from Definition 4.2 by replacing in condition 1 the word *extends* by the phrase *weakly extends*.

Definition 4.7 (weakly 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 called *weak up-acceptable w.r.t.* $|\ |$*,* R *and* I if the following conditions hold:

1. P_1 weakly extends R;

2. $P \setminus R$ is acceptable w.r.t. $|\ |_{|P \setminus R}$ and I;

3. R is acyclic w.r.t. $|\ |_{|R}$;

4. for every ground instance $H \leftarrow L_1, \ldots, L_n$ of a clause of P_1, for $i \in [1, n]$, let $L_{i1}, \ldots, L_{ik}$ be those literals among $L_1, \ldots, L_i$ which are defined in P_1. Then, if $I \models L_{i1}, \ldots, L_{ik}$, and if L_i is defined in R and is not a constraint, then $|H| \geq |L_i|$.

Using this notion, we can prove the analogous of Theorem 4.4. To this aim, we need to use triples of multisets, instead of pairs, with the lexicographic ordering. Recall that the lexicographic ordering on triples of finite multisets, denoted by $\prec$, is s.t. $(X_1, X_2, X_3) \prec (Y_1, Y_2, Y_3)$ iff either $X_1 < Y_1$, or $X_1 = Y_1$ and $X_2 < Y_2$, or $X_1 = Y_1$ and and $X_2 = Y_2$ and $X_3 < Y_3$. We consider the triple:

$$\tau(Q)_{up,I} = (|[Q]|_{up,I,P'}, |[Q]|_{up,I,R'}, |[Q]|_{up,I,P_{|S}}),$$

where P' is $P \setminus P_{|S}$, and R' is $R \setminus P_{|S}$. So, we can prove the following result.

Theorem 4.8 *Suppose that P is weakly up-acceptable w.r.t. $|\,|$, R and I. Let Q be an up-bounded query. Then every* `ldcnf`*-derivation for Q in P contains only up-bounded queries and is finite.*

Proof. Let S be the set of relations used to prove that P is weakly up-acceptable w.r.t. $|\,|$, R and I. Let ξ be a `ldcnf`-derivation for Q in P. We prove by induction on the number n of elements of ξ that every query of ξ is up-bounded, and that for every two consecutive queries of ξ, say Q_1 and Q_2, if in Q_1 a literal which is not a constraint is selected, then $\tau(Q_1)_{up,I} > \tau(Q_2)_{up,I}$. The base case ($n = 1$) is immediate. Now suppose that $n > 1$, and that we have proven the result for all $i \leq k$, for some $k < n$. Let $Q_1 = L_1, \ldots, L_n$ be the k-th query of an up-bounded query of ξ, and let L_i be its selected literal. Then Q_1 is up-bounded by the induction hypothesis. Let Q_2 be the resolvent of Q_1. That Q_2 is up-bounded follows by Q_1 up-bounded, and by the definition of weakly up-acceptability (here also condition 4 is used). Now, suppose that L_i is not a constraint. Then, we show that $\tau(Q_2)_{up,I}$ is smaller than $\tau(Q_1)_{up,I}$ in the lexicographic ordering. If L_i is defined in P_1 and not in $P_{|S}$, then the first component of $\tau(Q_2)_{up,I}$ becomes smaller because of condition 2. If L_i is defined in R then the first component of $\tau(Q_2)_{up,I}$ does not increase because of condition 1, while the second one becomes smaller because of condition 3. Finally, if L_i is defined in $P_{|S}$, then the first and second components of $\tau(Q_2)_{up,I}$ do not increase, because of condition 1, while the third one becomes smaller because of condition 2.

Then, the conclusion follows from the fact that the lexicographic ordering is well-founded, and that in a derivation, constraints can be consecutively selected only for a finite number of times. □

4.2 Low-Acceptability

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 4.9 (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 called *low-acceptable w.r.t. $|\,|$, R and I* if the following conditions hold:

1. P_1 extends R;

2. $P \setminus R$ is acyclic w.r.t. $|\,|_{|P \setminus R}$;

3. R is acceptable w.r.t. $|\,|_{|R}$ 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 $|Q|_i^{up,I}$ is replaced by the set

$$|Q|_i^{low,I} = \{|L'_i| \mid L'_1, \ldots, L'_n \text{ is a ground instance of } Q \text{ and } I \models L'_{k_1} \wedge \ldots \wedge L'_{k_l}\},$$

where $L'_{k_1}, \ldots, L'_{k_l}$ are all those literals of $L'_1, \ldots, L'_{i-1}$ (whose relations are) defined in R, and which are not constraints.

To prove the analogous of Theorem 4.4 for low-bounded queries, we associate with a low-bounded query Q a pair $\pi(Q)_{low,I} = (|[Q]|_{low,I,P_1}, |[Q]|_{low,I,R})$ of multisets, where for a program P,

$$|[Q]|_{low,I,P} = bag(max|Q|_{k_1}^{low,I}, \ldots, max|Q|_{k_m}^{low,I}),$$

where $L_{k_1}, \ldots, L_{k_m}$ are those literals defined in P which are not constraints.

Then the following result holds.

Theorem 4.10 *Suppose that P is low-acceptable w.r.t. $|\ |$, R and I. Let Q be a low-bounded query. Then every* `ldcnf`*-derivation for Q in P contains only low-bounded queries and is finite.*

Proof. The proof is similar to that of Theorem 4.4, where one replaces $\pi(Q)_{up,I}$ with $\pi(Q)_{low,I}$. □

The following result is a direct consequence of Theorems 4.10 and 3.7.

Corollary 4.11 *Let P be a general logic program. Then:* **(i)** *If P is low-acceptable then P is acceptable.* **(ii)** *If P is acceptable then it is low-acceptable.*

5 A Methodology

Definitions 4.2 and 4.9 provide us with a method for proving left-termination of general logic programs. For a program P, the method can be illustrated as follows:

1. Find a maximal set R of clauses of P s.t. R forms an acyclic program and $P = P_1 \cup R$ is s.t. either P_1 extends R or vice versa.

2. If R extends P_1 then:

   (a) Prove that $P \backslash R$ is acceptable w.r.t. a level mapping, say $|\ |_1$, and an interpretation.

   (b) Use $|\ |_1$ to define a level mapping $|\ |_2$ for R s.t. 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 R, if L_i is defined in P_1 and is not a constraint, then $|H|_2 \geq |L_i|_1$ holds.

3. If P_1 extends R then:

(a) Prove that R is acyclic w.r.t. a level mapping, say $| \, |_1$.

(b) 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 I, and s.t. 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, and if those literals among $L_1, \ldots, L_i$ which are defined in P_1, say $L_{i1}, \ldots, L_{ik}$, are s.t. $I \models L_{i1}, \ldots, L_{ik}$, 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. 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). This is possible because by Corollaries 4.5 and 4.11, a program is up-/low-acceptable iff it is acceptable. Then, in the conditions 2 of Definition 4.2 and 3 of Definition 4.9 we can prove up-/low-acceptability instead of acceptability. The resulting method is illustrated as follows.

- Find a partition of P, say $P_1, \ldots, P_n$ s.t. for every $i \in [1, n-1]$:
  - $P_{i+1} > P_i$ (P_{i+1} extends P_i);
  - either P_i or P_{i+1} is acyclic; and
  - if P_{i+1} is acyclic then it is a maximal set of clauses from $P_1 \cup \ldots \cup P_{i+1}$ which forms an acyclic program.
- Prove that for every $i \in [1, n]$, the program $P_1 \cup \ldots \cup P_i$ is up- or low-acceptable.

We can prove that $P_1 \cup \ldots \cup P_i$ is up- or low-acceptable in an incremental way, as follows. Suppose that for an $i < n$, $P_1 \cup \ldots \cup P_i$ has been proven up- or low-acceptable w.r.t. $| \, |_1$ and some interpretation. Then:

1. If P_{i+1} is acyclic then use $| \, |_1$ to define a level mapping $| \, |_2$ for $P_{i+1} \setminus P_i$ s.t. $P_{i+1} \setminus P_i$ 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_{i+1}, if L_j is defined in P_i and it is not a constraint, then $|H|_2 \geq |L_j|_1$ holds.

2. If P_i is acyclic then use $| \, |_1$ to define a level mapping $| \, |_2$ for $P_{i+1} \setminus P_i$ s.t. $P_{i+1} \setminus P_i$ 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_{i+1}, for $j \in [1, n]$, let $L_{j1}, \ldots, L_{jk}$ be those literals among $L_1, \ldots, L_j$ which are defined in P_{i+1}. Then, if $I \models L_{j1}, \ldots, L_{jk}$, and if L_j is defined in P_i and is not a constraint, then $|H|_2 \geq |L_j|_1$ holds.

It is easy to check that this methodology is correct, i.e., that if $P_1 \cup \ldots \cup P_i$ is up-/low-acceptable then using the above algorithm we obtain that also $P_1 \cup \ldots \cup P_{i+1}$ is up-/low-acceptable.

Observe that by using this incremental bottom-up approach, one obtains the subprogram R to be used to prove up-/low-acceptability (either P_1 or P_n), together with a potential level mapping $| \, |$ (the union of the level mappings of the P_i's). However, the interpretation I is not obtained constructively. Thus, this method is less powerful than the non-incremental one, because it does not allow to deal with non-ground queries (by means of the notion of boundedness) except for those consisting of just one literal.

In the following two sections, we illustrate how various problems in non-monotonic reasoning can be formalized by means of up-/low-acceptable programs. We consider the blocks-world problem, and search in graph structures.

6 On The 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 for instance [19]). Here we consider a simple version of this problem by [20], where there are three blocks, say a, b, c, and three different places of a table, say p, q and 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. A possible initial situation is illustrated in Figure 1.

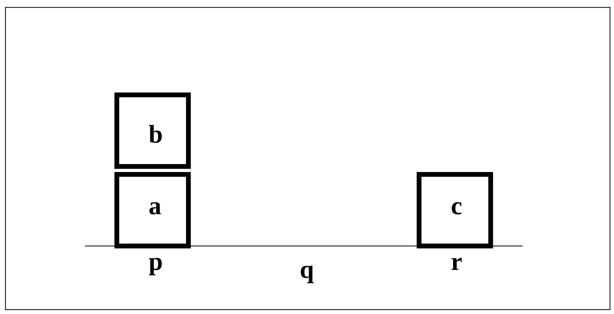


Figure 1: The Blocks-World

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. A clausal representation of this problem is given for instance in [15], where it is described in terms of pre- and post-conditions. Here we prefer to use McCarthy and Hayes situation calculus [18] to formulate the problem, in terms of facts, events and situations. One can distinguish three types of *facts*: $loc(X, L)$ stands for *a block X is in the location L*; $on(X, Y)$ for *a block X is on a block Y*; and $clear(L)$ for *there is no block in the location L*. It is sufficient to consider only one type of *event*, namely *move a block X into a location L*, denoted by $move(X, L)$. Finally, we represent *situations* by means of lists: [ ] stands for the initial situation, and $[Xe|Xs]$ for the one corresponding to the occurrence of the event Xe in the situation Xs.

Based on the above representation, one can formalize the blocks world by means of the following program `blocks-world`, where $top(X)$ denotes the top of the block X, $\mathcal{B} = \{a, b, c\}$, $\mathcal{P} = \{p, q, r, top(a), top(b), top(c)\}$, and $\mathcal{L} = \{loc(a, p), loc(b, q), loc(c, r)\}$. Notice that 1), 2) and 3) represent sets of clauses.

1) `holds(l,[])` $\leftarrow$. `l`$\in \mathcal{L}$

```
2)  block(bl) ← .  bl∈ 𝓑
3)  place(pl) ← .  pl∈ 𝓟
4)  holds(loc(X,L),[move(X,L)|Xs]) ←
   block(X),
   place(L),
   holds(clear(top(X)),Xs),
   holds(clear(L),Xs),
   L ≠ top(X).
5)  holds(loc(X,L),[Xe|Xs]) ←
   block(X),
   place(L),
   ¬ abnormal(loc(X,L),Xe,Xs),
   holds(loc(X,L),Xs).
6)  holds(on(X,Y),Xs) ←
   holds(loc(X,top(Y)),Xs).
7)  holds(on(X,Y),Xs) ←
   holds(loc(X,top(Z)),Xs),
   holds(loc(Z,top(Y)),Xs).
8)  holds(clear(L),Xs) ←
   ¬ busy(L,Xs).
9)  abnormal(loc(X,L), move(X,L'),Xs) ←.
10)  busy(L,Xs)←
   holds(loc(X,L),Xs).
11)  legal-s([(a,L1),(b,L2),(c,L3)],Xs) ←
    holds(loc(a,L1),Xs),
   holds(loc(b,L2),Xs),
   holds(loc(c,L3),Xs).
```

The initial situation is described by 1). 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.

$$|holds(x,y)| = \begin{cases} 3*|y|+1 & \text{if } x \text{ is of the form } loc(r,s), \\ 3*|y|+3 & \text{if } x \text{ is of the form } clear(r,s), \\ 3*|y|+4 & \text{if } x \text{ is of the form } on(r,s), \\ 0 & \text{otherwise.} \end{cases}$$

$|busy(x,y)| = 3*|y|+2$,
$|block(x)| = 0$,
$|place(x)| = 0$,
$|abnormal(x,y,z)| = 0$,
$|legal-s(x,y)| = 3*|y|+2$.

Consider for instance the query `holds(on(a,Y),[Xs])`: it is bounded, hence every its `sldcnf`-derivation is finite. We obtain the answers $(Y = b \wedge Xs = move(a,top(b)))$ and $(Y = c \wedge Xs = move(a,top(c)))$. Below is pictured a derivation yielding the first answer.

$$\underline{holds(on(a,Y),[Xs])}$$

$$\Big\downarrow\ 6)$$

$$\underline{holds(loc(a,top(Y)),[Xs])}$$

$$\{Xs/move(a,top(Y))\}\ \Big\downarrow\ 4)$$

$$\underline{block(a)},place(top(Y)),holds(clear(top(a)),[\,]),holds(clear(top(Y)),[\,]),top(Y)\neq top(a)$$

$$\Big\downarrow\ 2)$$

$$\underline{place(top(Y))},holds(clear(top(a)),[\,]),holds(clear(top(Y)),[\,]),top(Y)\neq top(a)$$

$$\{Y/b\}\ \Big\downarrow\ 3)$$

$$\underline{holds(clear(top(a)),[\,])},holds(clear(top(b)),[\,]),top(b)\neq top(a)$$

$$\Big\downarrow\ 8)$$

$$\underline{\neg busy(top(a),[\,])},holds(clear(top(b)),[\,]),top(b)\neq top(a)$$

$$\Big\downarrow$$

$$\underline{holds(clear(top(b)),[\,])},top(b)\neq top(a)$$

$$\Big\downarrow\ 8)$$

$$\underline{\neg busy(top(b),[\,])},top(b)\neq top(a)$$

$$\Big\downarrow$$

$$\underline{top(b)\neq top(a)}$$

$$\Big\downarrow$$

$$\square\ s$$

Here both the `sldcnf`-trees $subs(\underline{\neg busy(top(a),[\,])},holds(clear(top(b)),[\,]),top(b)\neq top(a))$ and $subs(\underline{\neg busy(top(b),[\,])},top(b)\neq top(a))$ are of finite failure. The latter is illustrated below:

$$\underline{busy(top(b),[\,])}$$

$$\Big\downarrow\ 10)$$

$$\underline{holds(loc(X,top(b)),[\,])}\ f$$

Suppose now that we would like to know when the block a remains in its initial position p after the occurrence of an action. This can be expressed by means of the query `holds(loc(a,p),[A])`. This query is bounded, hence every its `sldcnf`-derivation is finite. The following is an `sldcnf`-tree for `holds(loc(a,p),[A])`, where all the derivations yielding a failure have been omitted.

$$\underline{holds(loc(a,p),[A])}$$
$$\Big\downarrow {\scriptstyle 5)}$$
$$\underline{\neg abnormal(loc(a,p),A,[\,])}, holds(loc(a,p),[\,])$$
$$\Big\downarrow$$
$$\forall L(A \neq move(a,L)), \underline{holds(loc(a,p),[\,])}$$
$$\Big\downarrow {\scriptstyle 1)}$$
$$\forall L(A \neq move(a,L))\ s$$

The `sldcnf`-tree $subs(\underline{\neg abnormal(loc(a,p),A,[\,])}, holds(loc(a,p),[\,]))$ is given below:

$$\underline{abnormal(loc(a,p),A,[\,])}$$
$$\{A/move(a,L)\}\ \Big\downarrow {\scriptstyle 9)}$$
$$\square\ s$$

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 transforms the initial configuration in a particular final configuration, as illustrated for instance by Figure 2.

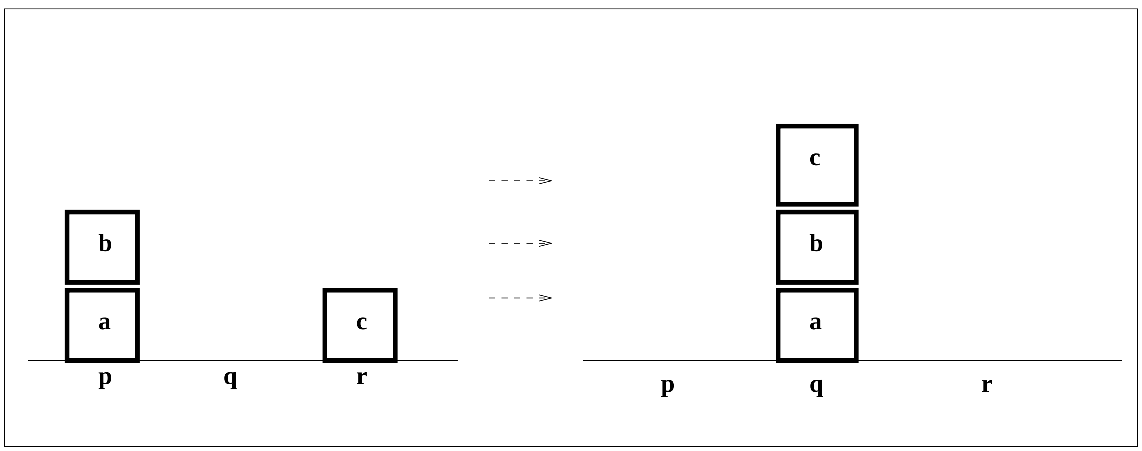


Figure 2: Planning in the Blocks-World

This problem can be solved by means of a nondeterministic algorithm (see e.g. [21]): *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 initialization (which is described by 1) of `blocks-world`). Alternatively, as done in [21], one could let unspecified the initialization, which would be provided every time the program is tested.

```
1p)  transform(Xs,St,Plan) ←
   state(St0),
   legal-s(St0,Xs),
   trans(Xs,St,[St0],Plan).
2p)  trans(Xs,St,Vis,[ ]) ←
   legal-s(St,Xs).
3p)  trans(Xs,St,Vis,[Act|Acts]) ←
   state(St1),
   ¬ member(St1,Vis),
   legal-s(St1,[Act|Xs]),
   trans([Act|Xs],St,[St1|Vis],Acts).
4p)  state([(a,L1),(b,L2),(c,L3)]) ←
   P=[p,q,r,top(a),top(b),top(c)],
   member(L1,P),
   member(L2,P),
   member(L3,P).
5p) member(X,[X|Y]) ←.
6p) member(X,[Y|Z]) ←
   member(X,Z).
```

To prove that `planning` is left-terminating using the original definition of acceptability (see Definition 3.4) 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 5p) and 6p) and of all the clauses of `blocks-world`, but 6), 7), 11).

We show that the proof is simpler when using the notion of up-acceptability. We prove that `planning` is up-acceptable w.r.t. $|\ |$, `r-blocks-world`, and I defined as follows. The level mapping $|\ |$ for `planning` is the one of the previous example when restricted to `r-blocks-world`, and is defined as follows for the other relations.

$|transform(x, y, z)| = N + 3 * (|x| + 1) + 2 + 3 + 1;$
$|trans(x, y, z, w)| = N - card(el(z) \cap S) + 3 * (|x| + 1) + 2 + 3 + |z|;$
$|state(x)| = 7;$
$|member(x, y)| = |y|.$

Here $el(z)$ denotes $set(z)$ if z is a list, the empty set otherwise; $card(el(z)\cap S)$ is the cardinality of the set $el(z) \cap S$; $|x|$ is defined as in the previous example; and N denotes the cardinality of S. Note that $(N - card(el(z) \cap S))$ is greater or equal than 0. Then $|\,|$ is well defined. Let `tras` denote the program `planning\r-blocks-world`, given below:

```
1'p)  transform(Xs,St,Plan) ←
   state(St0),
   trans(Xs,St,[St0],Plan).
2p)  trans(Xs,St,Vis,[ ]) ←.
3'p)  trans(Xs,St,Vis,[Act|Acts]) ←
```

```
   state(St1),
   ¬ member(St1,Vis),
   trans([Act|Xs],St,[St1|Vis],Acts).
4p)  state([(a,L1),(b,L2),(c,L3)]) ←
   P=[p,q,r,top(a),top(b),top(c)],
   member(L1,P),
   member(L2,P),
   member(L3,P).
5p) member(X,[X|Y]) ←.
6p) member(X,[Y|Z]) ←
   member(X,Z).
```

It is easy to check that condition 1 of the definition of up-acceptability is satisfied. Moreover, we have already proven in the previous example, that condition 3 is satisfied, i.e. that `r-blocks-world` is acyclic. One can immediately check that condition 4 is satisfied by construction. So, it remains to prove condition 2. To this end, consider the following interpretation I of `tras`: let $set(y)$ be the set of elements of the list y, and $S = \{[(a, p1), (b, p2), (c, p3)] \mid \text{ for } i \in [1, 3], pi \in \{p, q, r, top(a), top(b), top(c)\}\}$. Let:

$I_{transform} = [transform(X, Y, Z)]$,
$I_{trans} = [trans(X, Y, Z, W)]$,
$I_{member} = \{member(x, y) \mid y \text{ list s.t. } x \in set(y)\}$,
$I_{state} = \{state(x) \mid x \in S\}$.

Then $I = I_{transform} \cup I_{trans} \cup I_{member} \cup I_{state}$. It is easy to prove that I is a model of `tras`. Moreover, $Neg^*_{tras} = \{member\}$, and `tras`$^-$ is equal to $\{5p), 6p)\}$. Then it is easy to check that I restricted to $\{member\}$ is a model of $comp($`tras`$^-)$. To show that `tras` is acceptable w.r.t. I and $|\,|$, we use the following properties of $|\,|$, which are easy to be checked:

$$|transform(x, y, z)|_1 \geq 8, \tag{1}$$

$$|trans(x, y, z, w)|_1 \geq 8, \tag{2}$$

and

$$|trans(x, y, z, w)|_1 > |z|. \tag{3}$$

Consider a ground instance:
$transform(xs, xt, plan) \leftarrow state(st0), trans(xs, st, [st0], plan).$
of $1p)$. Then from (1) we have that:

$$|transform(xs, xt, plan)| > |state(st0)|.$$

Now, suppose that $I \models state(st0)$. Then $st0 \in S$, so $card(el(S \cap el([st0])) = 1$; hence:

$$|transform(xs, xt, plan)| > |trans(xs, st, [st0], plan)|.$$

Consider a ground instance:
$trans(xs, st, vis, [act|acts]) \leftarrow$
$\quad state(st1), \neg member(st1, vis), trans([act|xs], st, [st1|vis], acts).$
of $2'p)$. Then from (2) we have that:

$$|trans(xs, st, vis, [act|acts])| > |state(st1)|,$$

and from (3) we have that

$$|trans(xs, st, vis, [act|acts])| > |\neg member(st1, vis)|.$$

Now, suppose that $I \models state(st1), \neg member(st1, vis)$. Then $st1 \in S$, but $st1 \notin set(vis)$; so $card(S \cap el([st1|vis])) = card(S \cap el(vis)) + 1$; hence $N - card(S \cap el([st1|vis])) < N - card(S \cap el(vis))$. So,

$$|trans(xs, st, vis, [act|acts])| > |trans([act|xs], st, [st1|vis], acts)|.$$

The proof for the remaining clauses of `tras` is similar.

Consider the query `transform([ ],st,Plan)`, where `st` is a given state. This query is up-bounded, hence by Theorem 4.4 all its `ldcnf`-derivations are finite, and produce a plan of actions which transforms the initial state [ ] into the final one `st`. Notice that this query has an infinite `sldcnf`-derivation, which is obtained by selecting always the rightmost literal of the clause $4p$).

7 Search in Graph Structures

Graph structures are used in AI for many applications, such as representing relations, situations or problems (see e.g. [8]). Two typical operations performed on graphs are *find a path between two given nodes*, and *find a subgraph with some specified properties*. We consider two programs based on these operations. The first program is called `specialize`. It resolves the following problem. Given a graph g, and two nodes n_1, n_2, find a node n which does not belong to any acyclic path in g from n_1 to n_2. The second program is called `hamiltonian`. It resolves a classical problem on graphs, namely to find a Hamiltonian path. Recall that an Hamiltonian path is an acyclic path which contains all the nodes of the graph. Both these programs incorporate the following set of clauses, denoted by `acy-path`, which specify the notion of acyclic path:

```
p1) path(N1,N2,G,P) ←
      path1(N1,[N2],G,P).
p2) path1(N1,[N1|P1],G,[N1|P1]) ←.
p3) path1(N1,[X1|P1],G,P) ←
      member([Y1,X1],G),
      ¬ member(Y1,[X1|P1]),
      path1(N1,[Y1,X1|P1],G,P).
p4) member(X,[X|Y]) ←.
p5) member(X,[Y|Z]) ←
      member(X,Z).
```

Here, acyclic paths of a graph are described by the relation *path*, defined by the clause p1), where $path(n1, n2, g, p)$ calls the query $path1(n1, [n2], g, p)$. The second argument of $path1$ is used to construct incrementally an acyclic path connecting $n1$ with $n2$: using clause p3), 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 p2). Thus the relation *path*1 is defined inductively by the clauses p2) and p3), using the familiar relation *member*, specified by the clauses p4) and p5). Notice that, from p2) 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.

Observe also that here, a graph is represented by means of a list of edges. For instance, the graph $[[a,b],[b,c],[a,a]]$ is represented in Figure 3. For graphs consisting only of one node, we adopt the convention that they are represented by the list $[[a,\perp]]$, where $\perp$ is a special new symbol.

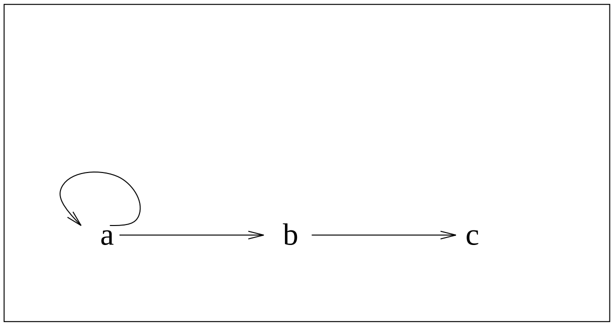


Figure 3: The graph $[[a,b],[b,c],[a,a]]$

7.1 Specialize

The program `specialize` consists of the clauses:

```
1) spec(N1,N2,N,G) ←
   ¬ unspec(N1,N2,N,G).
2) unspec(N1,N2,N,G) ←
   path(N1,N2,G,P),
   member(N,P).
```

augmented with the program `acy-path`. The relation *spec* is specified as the negation of *unspec*, where $unspec(n1,n2,n,g)$ is true if there is an acyclic path of the graph g connecting the nodes $n1$ and $n2$ and containing n. For instance, `spec(a,b,c,[[a,b],[b,c],[a,a]])` holds.

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 $p3$) and by selecting always its rightmost literal. However `specialize` is left-terminating. Note that to prove this result using Definition 3.4 requires to find a suitable model of the completion of the program, which is rather difficult. Therefore we prove left-termination by means of the notion 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 1). Let `spec2` be the program consisting of the clause 1) of `specialize`. Define the level mapping $|\ |$ as follows:

$|spec(n1, n2, n, g)| = 3|g| + 5,$
$|unspec(n1, n2, n, g)| = 3|g| + 4,$
$|member(s, t)| = |t|;$
$|path1(n1, p1, g, p)| = |p1| + |g| + 2(|g| - |p1 \cap g|) + 1,$
$|path(n1, n2, g, p)| = 3|g| + 3,$

where for two lists p and g, $p \cap g$ denotes the list containing as elements those x which are elements of p and such that there exists a y s.t. $[x, y]$ is an element of g.

Let $I = I_{unspec} \cup I_{path} \cup I_{path1} \cup I_{member}$, where:

$I_{unspec} = [unspec(N1, N2, N, G)],$
$I_{path} = \{path(n1, n2, g, p) \mid |g| + 1 \geq |p|\},$
$I_{path1} = \{path1(n1, p1, g, p) \mid |p1| - |p1 \cap g| \geq |p| - |p \cap g|\},$
$I_{member} = \{member(s, t) \mid t \text{ list s.t. } s \in set(t)\}.$

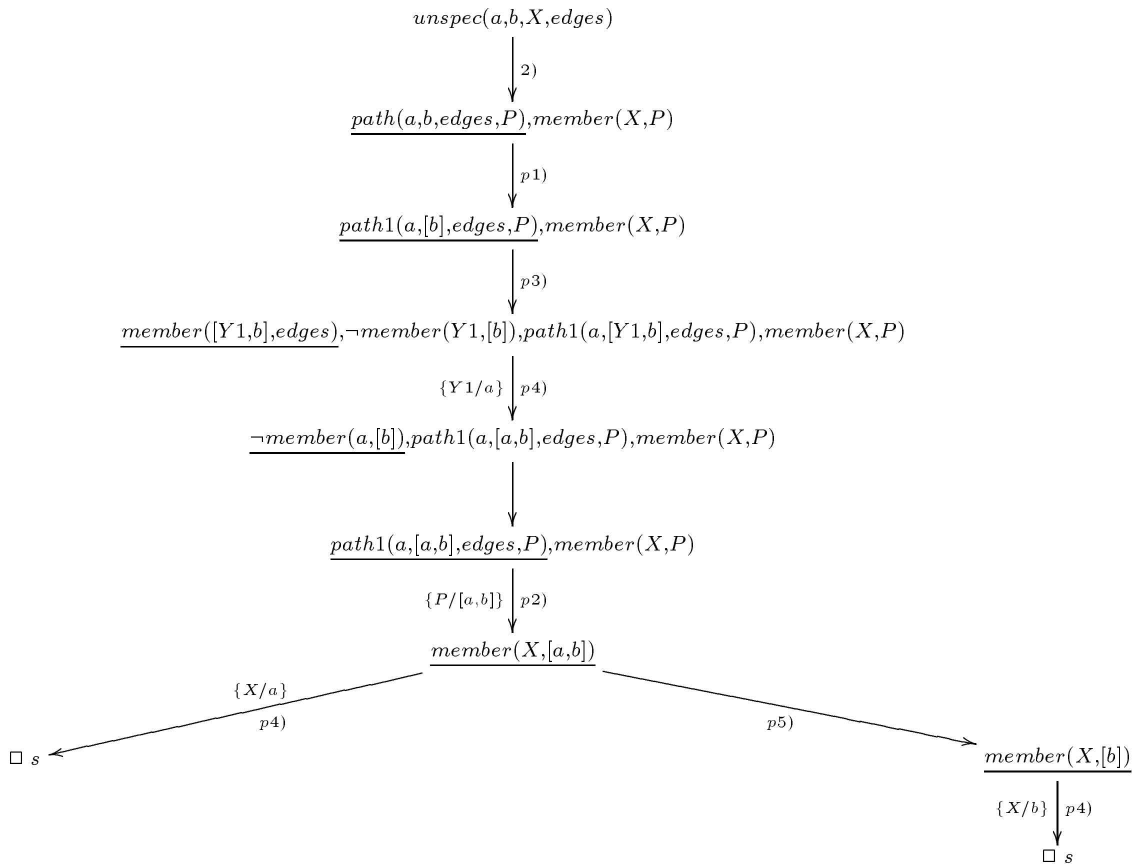
It is easy to prove that I is a model of $spec1$. For instance, consider clause p1). Suppose that $I \models path1(n1, [n2], g, p)$. Note that $|[n2]| - |[n2] \cap g| \leq 1$. Then $|p| - |p \cap g| \leq 1$. But $|p \cap g| \leq |g|$. Then $|p| \leq |g| + 1$, hence $I \models path(n1, n2, g, p)$. Consider clause p3). Suppose that $I \models member([y1, x1], g), \neg member(y1, [x1|p1]), path1(n1, [y1, x1|p1], g, p)$. 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 $I \models path1(n1, [x1|p1], g, p)$.

The proof for the other clauses is analogous. We have that, $Neg^*_{spec1} = \{member\}$ and $spec1^- = \{(f), (g)\}$. Then it is routine to check that I, restricted to $member$, is a model of $comp(spec1^-)$. Finally, one can easily check that the conditions 1-4 of the definition of low-acceptability are satisfied.

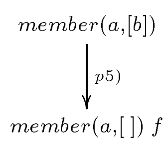
Consider now the query $Q =$ `spec(a,b,X,[[a,b],[b,c],[a,a]])`. It is low-bounded. Then, one obtains the following finite `ldcnf`-tree for Q, where $edges$ denotes the list $[[a, b], [b, c], [a, a]]$,

$$\begin{array}{c} spec(a,b,X,edges) \\ \Big\downarrow{\scriptstyle 1)} \\ \neg unspec(a,b,X,edges) \\ \Big\downarrow \\ X \neq a, X \neq b\ \ s \end{array}$$

with answer $(X \neq a \wedge X \neq b)$. The tree $subs(\neg unspec(a, b, X, edges))$ is given below, where for simplicity we omitted to draw the derivations whose leaves are marked as *failed*.



and the tree $subs(\underline{\neg member(a,[b])}, path1(a,[a,b],P), member(X,P))$ is the finitely failed tree



Notice that by using negation as failure Q does flounder.

Suppose now that we want to determine which sequence of actions can lead to the state represented by the node b, starting from a state represented by the node a. This problem can be expressed by means of the query $Q =$ `path(a,b,[[a,b],[b,c],[a,a]],P)`.

Then one obtains the following finite `ldcnf`-tree for Q, where *edges* denotes the list $[[a,b],[b,c],[a,a]]$.

$$
\begin{array}{c}
\underline{path(a,b,edges,P)} \\
\Big\downarrow{\scriptstyle p1)} \\
\underline{path1(a,[b],edges,P)} \\
\Big\downarrow{\scriptstyle p3)} \\
\underline{member([Y1,b],edges)},\neg member(Y1,[b]),path1(a,[Y1,b],edges,P) \\
{\scriptstyle \{Y1/a\}}\Big\downarrow{\scriptstyle p4)} \\
\underline{\neg member(a,[b])},path1(a,[a,b],edges,P) \\
\Big\downarrow \\
\underline{path1(a,[a,b],edges,P)} \\
{\scriptstyle \{P/[a,b]\}}\Big\downarrow{\scriptstyle p2)} \\
\square\ s
\end{array}
$$

and the tree $subs(\underline{\neg member(a,[b])}, path1(a,[a,b],P))$ is the finitely failed tree equal to $subs(\underline{\neg member(a,[b])}, path1(a,[a,b],P), member(X,P))$.

7.2 Hamiltonian

In this section we illustrate the application of our methodology and of the notion of weakly up-acceptability by means of a program which defines an hamiltonian path of a graph.

The program `hamiltonian` consists of the clauses:

```
1) ham(G,P) ←
     path(N1,N2,G,P),
     cov(P,G).
2) cov(P,G) ←
     ¬ notcov(P,G).
3) notcov(P,G) ←
     node(X,G), ¬ member(X,P).
4) node(X,G) ←
     member([X,Y],G).
5) node(X,G) ←
     member([Y,X],G).
```

augmented with the program `acy-path`. The relation $ham(g,p)$ is specified in terms of *path* and *cov*, i.e. it is true if p is an acyclic path of g which covers all its nodes. The relation *cov* is specified as the negation of another relation, called *notcov*, where $notcov(p,g)$ is true if there is a node of g which does not occur in p. Finally, the relation *node* is defined in terms of *member* in the expected way. Then, we have for instance that `ham([[a,b],[b,c],[a,a],[c,b]], [a,b,c])` holds, corresponding to the path drawn in bold in the graph pictured in Figure 4.

Observe, that `hamiltonian` is not terminating, because `acy-path` is not.

However, `hamiltonian` is left-terminating. Note that to prove this result using Definition 3.4 requires to find a suitable model of the completion of the program consisting of the clauses

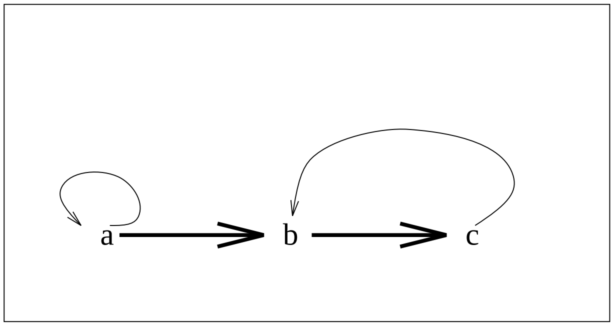


Figure 4: The Hamiltonian path of $[[a,b],[b,c],[a,a],[c,b]]$

3), 4), $p4$) and $p5$). Therefore, we use the notion of weakly up-acceptability (see Definition 4.7), where the program is split into two parts such that one part extends weakly the other one. We choose as upper part the program `acy-path` augmented with clause 1), and call it `up-ham`; and as lower part the remaining set of clauses, indicated by `low-ham`. Moreover, we choose as set S of relations the set $\{member\}$. It is easy to check that `up-ham` weakly extends `low-ham`.

Call `up-low-ham` the program `up-ham\low-ham`:

```
1) ham(G,P) ←
     path(N1,N2,G,P).
p1) path(N1,N2,G,P) ←
     path1(N1,[N2],G,P).
p2) path1(N1,[N1|P1],G,[N1|P1]) ←.
p3) path1(N1,[X1|P1],G,P) ←
     member([Y1,X1],G),
     ¬ member(Y1,[X1|P1]),
     path1(N1,[Y1,X1|P1],G,P).
p4) member(X,[X|Y]) ←.
p5) member(X,[Y|Z]) ←
     member(X,Z).
```

We show that `up-low-ham` is acceptable. To this end, it is sufficient to prove that `up-low-ham` is low-acceptable. Indeed, we split `up-low-ham` in an upper part, that we call `up1-low-ham`, consisting of the clause 1), and a lower part, which is the program `acy-path`, consisting of the remaining clauses. Clearly `up1-low-ham\acy-path` is acyclic, for instance with respect to the following level mapping:

$|ham(g,p)| = 3|g| + 4$.

Moreover, `acy-path` is acceptable with respect to the level mapping and model defined in the previous example. Then, conditions of Definition 4.9 of low-acceptability are satisfied.

To conclude the proof, it remains to show that `low-ham` is acyclic, and that condition 4 of Definition 4.7 is satisfied. The program `low-ham` is given below:

```
2) cov(P,G) ←
     ¬ notcov(P,G).
3) notcov(P,G) ←
     node(X,G), ¬ member(X,P).
4) node(X,G) ←
     member([X,Y],G).
5) node(X,G) ←
     member([Y,X],G).
```

Consider the level mapping:
$|cov(p,g)| = |p| + |g)| + 3$;
$|notcov(p,g)| = |p| + |g| + 2$;
$|node(s,t)| = |t| + 1$;
$|member(s,t)| = |t|$.

Then, it is easy to check that `low-ham` is acyclic w.r.t. $|\ |$. Moreover, condition 4 of Definition 4.7 is satisfied: In fact, consider a ground instance

$$ham(g,p) \leftarrow path(n1,n2,g,p), cov(p,g).$$

of 1) and suppose that $I \models path(n1,n2,g,p)$, where I is the model that we used to prove that `acy-path` is acceptable. Then we have that $|g|+1 \geq |p|$. Then, $|ham(g,p)| = 3|g|+4 > |p|+|g|+3 = |cov(p,g)|$.

So, we have proven that **`hamilton`** is left-terminating. Consider the query $ham(P,[[a,b],[b,c],[a,a],[c,b]])$. This query is up-bounded, hence left-terminating. Its answer is $P = [a,b,c]$.

Observe that if we replace the clauses 4), 5) describing *node* by the clause

```
node(X,G) ←
   member(Y,G),
   member(X,Y).
```

then we could not apply our technique to prove left-termination, because this program is not terminating (it is in fact only left-terminating).

8 Conclusion

In this paper we proposed a simple method for proving termination of a general logic program, with respect to SLD-resolution with constructive negation and the Prolog selection rule. This method is based on alternative, yet equivalent, definitions of the notion of acceptability, where the original notion of acceptability is combined with the one of acyclicity. These alternative definitions provide a more practical method, where the semantic information used to prove acceptability is minimalized. We illustrated the relevance of this methodology by means of some examples. These examples show that SLD-resolution augmented with Chan's constructive negation allows to express and implement interesting problems in non-monotonic reasoning.

We would like to conclude with an observation on related work. In [6], Apt and Pedreschi introduced a modular approach for proving acceptability of *pure* Prolog programs, i.e. without negation. The extension of this approach to programs containing negated atoms is not treated. To prove termination of general Prolog programs in a modular way, using the notion of acceptability, is rather difficult, because one has to provide a way to combine models of the completion of the parts of the program, to build a model of the completion of the program. Apt and Pedreschi do not tackle this problem. Also this paper does not solve this problem: instead, it provides an alternative way to prove acceptability, where one tries to simplify the proof by using as minimal semantic information as possible, possibly in an incremental way using the methodology illustrated in Section 5.

Acknowledgements

This research was partially supported by the Esprit Basic Research Action 6810 (Compulog 2). I would like to thank Krzysztof Apt for proposing the study of acyclic and acceptable programs, Jan Rutten for his help and support, and Frank Teusink for useful discussions.

References

[1] K. R. Apt, M. Bezem. Acyclic Programs. New Generation Computing, Vol. 9, 335–363, 1991.

[2] K. R. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[3] K. R. Apt., H. C. Doets. A new definition of SLDNF-resolution. The Journal of Logic Programming, vol.18, pag.177-190, 1994.

[4] K. R. Apt, E. Marchiori, C. Palamidessi. A Declarative Approach for First-order Built-in's of Prolog. Applicable Algebra in Engineering, Communication and Computing, vol.5, No. 2/4, pp. 159-191, 1994.

[5] K. R. Apt, D. Pedreschi. Proving Termination of General Prolog Programs. In T. Ito and A. Meyer, editors, *Proceedings of the Int. Conf. on Theoretical Aspects of Computer Software,* LNCS 526, pp.265-289, Berlin, 1991, Springer Verlag.

[6] K. R. Apt, D. Pedreschi. Modular Termination Proofs for Logic and Pure Prolog Programs. In G. Levi, editor, *Advances in logic programming theory.* Oxford University Press, 1994.

[7] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19,20:73–148, 1994.

[8] I. Bratko. PROLOG Programming for Artificial Intelligence. Addison-Wesley, 1986.

[9] D.Chan. Constructive Negation Based on the Completed Database. In *Proceedings of the 5th Int. Conf. and Symp. on Logic Programming*, pp. 111-125, 1988.

[10] K.L. Clark. Negation as Failure. In H. Gallaire and J. Minker eds., *Logic and Databases*, pp. 293-322. Plenum Press, NY, 1978.

[11] N. Dershowitz. Termination of Rewriting. Journal of Symbolic Computation, 3, pp. 69-116, 1987.

[12] D. De Schreye, S. Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19,20, 1994.

[13] C. Evans. Negation as Failure as an Approach to the Hanks and McDermott Problem. Proc. of the 2nd International Symposium om AI, pp. 23-27, 1990.

[14] S. Hanks, D. McDermott. Nonmonotonic Logic and Temporal Reasoning. Artificial Intelligence, 33, pp. 379-412, 1987.

[15] R. Kowalski. Logic for Problem Solving. Elsevier North-Holland, New York, 1979.

[16] R. Kowalski, M. Sergot. A Logic Based Calculus of Events. New Generation Computing, 4, pp. 67-95, 1986.

[17] E. Marchiori. On Termination of General Logic Programs w.r.t. Constructive Negation. Submitted, 1994.

[18] J. McCarthy, P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence 4, eds. B. Meltzer, D. Michie, pp. 463-502, 1969.

[19] N.J. Nilsson. Principles of Artificial Intelligence. Springer-Verlag, 1982.

[20] E.D. Sacerdoti. A Structure for Plans and Behaviour. Elsevier North-Holland, New York, 1977.

[21] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1994. *2nd edition.*