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 (glp's 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 has also to be a model of the completion of its "negative part" (see Definition 3.2). 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 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

$m \geq 0$, H is an atom, and the L_i's, called literals, are either atoms $p(\mathbf{s})$, or negative literals $\neg p(\mathbf{s})$, or equalities $s = t$, or inequalities $\forall(s \neq t)$, where $\forall$ 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.

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, say $\theta_1, \ldots, \theta_k$, $k \geq 0$; let F_Q be the equality formula $\exists(E_{\theta_1} \vee \ldots \vee E_{\theta_k})$, where E_{θ_1} is the substitution θ_1 written in equational form, and $\exists$ 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 [4] 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's procedure introduced in [10], where the subsidiary trees used to resolve negative literals are built in a top-down way, constructing their branches in parallel. We shall also 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 call with slight abuse *Prolog selection rule* this selection rule. Then sldcnf-trees with Prolog selection rule are 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 [2], 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 [10].

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 every $i \in [1, m]$ 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. We have proven in [10] that if P is acyclic and Q is bounded then every sldcnf-tree for Q in P is finite; and that also the converse of this result holds: call a program P *terminating* if all sldcnf-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. $|\ |$ iff all its sldcnf-derivations are finite. Notice that when negation as finite failure is assumed, (i) holds only if Q does not flounder ([1]). 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 $H \leftarrow L_1, \ldots, L_n$ 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 executed. 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 3.2 (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* $H \leftarrow L_1, \ldots, L_n$ *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 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 $i \in [1, n]$,

$$|Q|_i^I = \{|L'_i| \mid L'_1, \ldots, L'_n \text{ is a ground instance of } Q \text{ and } I \models L'_1 \wedge \ldots \wedge L'_{i-1}\}.$$

Then Q is called bounded if every $|Q|_i^I$ is finite.

Bounded queries characterize those queries s.t. all their ldcnf-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 ldcnf-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 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 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 *upp*er 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.

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. $| \ |_{|R}$; **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 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*. 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 $i \in [1, n]$,

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

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. 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* ldcnf*-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)** $P \setminus R$ 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. Then the corresponding of Theorem 4.3 and Corollary 4.4 hold, where *up* is replaced by *low*.

5 A Methodology

Definitions 4.2 and 4.5 provide us with a method for proving left-termination of general logic programs. For a program P, the method can be informally 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:

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 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 it is not a constraint, then $|H|_2 \geq |L_i|_1$ holds.
3) If P_1 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). This is possible, because a program is up-/low-acceptable iff it is acceptable. Then in the conditions 2 of Definition 4.2 and 3 of Definition 4.5 we can prove up-/low-acceptability instead of acceptability. The resulting method is informally 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 $P_0 \cup \ldots \cup P_i$ is up- or low-acceptable.

We can prove that $P_0 \cup \ldots \cup P_i$ 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 I. 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}, if L_j is defined in P_i and it is not a constraint, then $|H|_2 \geq |L_j|_1$ holds.

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. 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 only one literal.

Apt and Pedreschi in [3] 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, and also our method does not solve this problem. Instead, our approach provides an alternative methodology for proving acceptability, where one tries to simplify the proof by using as minimal semantic information as possible.

6 Application

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 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. 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*: $loc(X, L)$ 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, and $\mathcal{B} = \{a, b, c\}$, $\mathcal{P} = \{p, q, r, top(a), top(b), top(c)\}$, and $\mathcal{L} = \{loc(a,p), loc(b,q), loc(c,r)\}$:

```
(loc)  holds(l,[]) ←.  l ∈ ℒ
(blo)  block(bl) ←.  bl ∈ ℬ
(pla)  place(pl) ←.  pl ∈ 𝒫
(h1)  holds(loc(X,L),[move(X,L)|Xs]) ←
   block(X),
   place(L),
   holds(clear(top(X)),Xs),
   holds(clear(L),Xs),
   L ≠ top(X).
(h2)  holds(loc(X,L),[Xe|Xs]) ←
   block(X),
   place(L),
   ¬ abnormal(loc(X,L),Xe,Xs),
   holds(loc(X,L),Xs).
(h3)  holds(on(X,Y),Xs) ←
   holds(loc(X,top(Y)),Xs).

(h4)  holds(on(X,Y),Xs) ←
   holds(loc(X,top(Z)),Xs),
   holds(loc(Z,top(Y)),Xs).
```

```
(h5)  holds(clear(L),Xs) ←
        ¬ busy(L,Xs).
(ab)  abnormal(loc(X,L), move(X,L'),Xs) ←.
(bu)  busy(L,Xs)←  holds(loc(X,L),Xs).
(st)  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 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.

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

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

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 ($Y = c \wedge Xs = move(a, top(c))$).

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 initialization (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.

```
(t)  transform(Xs,St,Plan) ←
       state(St0), legal-s(St0,Xs),
       trans(Xs,St,[St0],Plan).
(t1) trans(Xs,St,Vis,[ ]) ←
       legal-s(St,Xs).
(t2) trans(Xs,St,Vis,[Act|Acts]) ←
       state(St1),
       ¬ member(St1,Vis),
       legal-s(St1,[Act|Xs]),
       trans([Act|Xs],St,[St1|Vis],Acts).
(s)  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).
(m1) member(X,[X|Y]) ←.
(m2) member(X,[Y|Z]) ←
       member(X,Z).
```

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 ($h3$), ($h4$), (st).

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 be the program planning\r-blocks-world. We consider the following interpretation I of tras: let $set(y)$ be the set of elements of the list y, and $S = \{[(a,p1),(b,p2),(c,p3)] \mid \text{for } i \in [1,3], pi \in \{p,q,r,top(a),top(b),top(c)\}\}$.

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

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, $Neg^{*}_{P \setminus R} = \{member\}$, and tras$^-$ is equal to $\{(m1),(m2)\}$. Then it is easy to check that I restricted to $\{member\}$ is a model of $comp$(tras$^-$). Moreover, conditions 1-4 of up-acceptability are satisfied: for instance, the proof of condition 2 (tras is acceptable w.r.t. I and $|\ |$) is based on the following properties of $|\ |$: $|transform(x,y,z)| \geq 8$, $|trans(x,y,z,w)| \geq 8$, and $|trans(x,y,z,w)| > |z|$). Consider the query transform([ ],st,Plan), where st is a given state. This query is up-bounded, hence by Theorem 4.3 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 (s).

Search in Graph Structures

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.* The following program specialize is an example of the combination of these two operations.

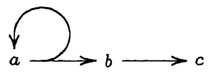
The relation *spec* is specified by clause (a), as the negation of another relation, called *unspec*, where

$unspec(n1, n2, n, g)$ is true if there is an acyclic path of the graph g connecting the nodes $n1$ and $n2$ and containing n. 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 $[x|p1]$ 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). 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.

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



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 the 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:

$$
\begin{aligned}
&|spec(n1, n2, n, g)| = |unspec(n1, n2, n, g)| + 1,\\
&|unspec(n1, n2, n, g)| = 0,\\
&|member(s, t)| = |t|;\\
&|path1(n1, p1, g, p)| = |p1| + |g| + 2(|g| - |p1 \cap g|) + 1,\\
&|path(n1, n2, g, p)| = 3|g| + 3,\\
&|unspec(n1, n2, n, g)| = 3|g| + 4,
\end{aligned}
$$

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:

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

It is easy to prove that I is a model of $spec1$. Moreover $Neg^*_{spec1} = \{member\}$ 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 `ldcnf`-tree for Q, with answer $(X \neq a \wedge X \neq b)$. 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

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

[2] K. R. Apt, D. Pedreschi. Proving Termination of General Prolog Programs. In *Proc. TACS'91*, LNCS 526, pp.265-289, 1991, Springer Verlag.

[3] 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.

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

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

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

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

[8] C. Evans. Negation as Failure as an Approach to the Hanks and McDermott Problem. Proc. of the 2nd Int. Symp. on AI, pp. 23-27, 1990.

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

[10] E. Marchiori. On Termination of General Logic Programs w.r.t. Constructive Negation. Journal of Logic Programming, 1995, to appear.

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

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