Some uses of constructive negation for classical problems in non-monotonic reasoning

E. Marchiori

Computer Science/Department of Software Technology

Report CS-R9466 December 1994

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Some Uses of Constructive Negation for Classical Problems in Non-monotonic Reasoning

Elena Marchiori

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

Abstract

We use general logic programs to formalize some classical problems in non-monotonic reasoning. More specifically, this paper shows how some forms of temporal reasoning and of planning problems can be easily formalized by using two restricted classes of programs, called acyclic and acceptable programs. Moreover, we show how for these programs more queries can be answered when a form of constructive negation is incorporated into the proof theory.

AMS Subject Classification (1991): 68Q40, 68Q60, 68T15.
CR Subject Classification (1991): F.3.1., F.4.1, I.2.3.
Keywords & Phrases: general logic programs, constructive negation, non-monotonic reasoning.
Note: This research was partly supported by the Esprit Basic Research Action 6810 (Compulog 2).

1. Introduction

Pure classical logic is inadequate to represent the common sense human reasoning, since the latter is in general non-monotonic. To this end various proposals have been introduced to provide formal foundations of non-monotonic reasoning, like circumscription ([17]), default theory ([22]), autoepsitemic logic ([19]) and the closed world assumption ([21]). Non-monotonic reasoning and logic programming are closely related. The closed world assumption has provided a formal justification to a procedural form of negation, known as *negation as failure*, which has been introduced in logic programming. This procedural form of negation allows the efficient implementation of non-monotonic formalisms in Prolog or in other declarative languages. Logic programming can also be used to provide formalizations for special forms of non-monotonic reasoning ([2, 8]). For example, the Prolog negation as failure operator has been used to formalize the temporal persistence problem in AI (see [15], [1]). However, in negation as failure a negative literal containing some variables cannot be resolved. This situation is called *floudering*. As a consequence a number of queries have no answers. This limits the applicability of the method: for instance, the general logic program given in [1] which formalizes the well-known Yale Shooting Problem, gives no answer to interesting questions like "which conditions on a generic event occurring in a generic situation guarantee that the fact *alive* holds ?". To overcome this problem of the negation as failure a new kind of procedural form of negation called *constructive negation* has been proposed by Chan in [10]: informally, the answers to a negative query $\neg Q$ are obtained by negating the answers of Q. However, this procedure is not defined when Q has an infinite derivation. This drawback has been solved by Marchiori in [16] where a top-down definition of this constructive negation procedure has been introduced.

In this paper, we study how various problems in non-monotonic reasoning can be formalized by using logic programs augmented with this form of negation. More specifically, we show that it is sufficient to consider a restricted class of programs, called acyclic and acceptable, to describe interesting problems dealing with some forms of temporal reasoning, taxonomy, and planning. Acyclic programs were introduced by Apt and Bezem in [1]: they are defined by means of a syntactic condition, which guarantees that for a class of queries, which includes the ground queries, all sldnf-derivations (i.e., sld-derivation with negation as failure) are finite. Acceptable programs were introduced by Apt and Pedreschi in [6] to provide analogous results when ldnf-derivations are considered, where the Prolog computation rule is

Report CS-R9466
ISSN 0169-118X
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

always applied. Note that the acceptability condition is not anymore syntactic, because it employes also a model of the program. The converse of these results, i.e. that if all ground queries of a program have only finite sldnf-derivations (ldnf-derivations) then the program is acyclic (acceptable) do not hold due to the problem of the *floundering*. However, this problem can be solved when the constructive negation is used, where the definition of acyclic and acceptable programs is naturally extended according with this new form of negation. This expected result was proven in [16]. The relevance of using constructive negation and acyclic/acceptable programs relies on the possibility of formalizing in a neat and simple way a number of problems in non-monotonic reasoning.

The aim of this paper is to substantiate this result by using a number of classical examples in non-monotonic reasoning. These examples include some forms of temporal reasoning, problems dealing with taxonomy, and the planning problem in the blocks world. First, a problem is described in the framework of the situation calculus ([18]). Then it is translated into a general logic program, which is either shown to be acyclic or to be acceptable. To prove that a program is acceptable is in general more difficult than to prove that it is acyclic, because also a suitable model of the program has to be used. However, it is often the case that a part of the program satisfies a stronger property, namely acyclicity, which can be established without using any model. For this reason, we introduce here an equivalent new concept of acceptability, called *e-acceptability*. Informally, to prove e-acceptability, the program is split into two parts; then one part is shown to be acyclic and the other to be acceptable.

The remaining of this paper is organized as follows. Some preliminaries conclude this section. Section 2 contains the definition of sldcnf-resolution. In Section 3, the notion of acyclicity is given together with some useful results. Section 4 contains analogous material, but this time for acceptable programs. Moreover, the notion of e-acceptability is introduced. Finally, Section 5 contains four examples formalizing various problems in non-monotonic reasoning, together with examples of interesting queries.

1.1 Preliminaries

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. An *equality formula*, indicated by E, is an assertion that does not contain any relation symbols other than the equality symbol $=$. The empty conjunction of assertions and the empty disjunction of assertions are denoted by *true* and $false$, respectively. The formula $\exists(c_1 \wedge \ldots \wedge c_n)$ is called *simple equality formula*, where $n \geq 0$, the c_i's are equalities or inequalities and $\exists$ quantifies over some (perhaps none) of the variables occurring in the c_i's.

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

2. sldcnf-RESOLUTION

In sld-resolution, for a program P and a query Q, if θ is a c.a.s. for Q then it can be written in equational form as $\exists(X_1 = X_1\theta \wedge \ldots \wedge X_n = X_n\theta)$, 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. Let $\theta_1, \ldots, \theta_k$, $k \geq 0$, be all the c.a.s.'s for Q in P and 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 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 [10] introduced a procedure called sldcnf-resolution, where the answers for $\neg Q$ are obtained from the negation of F_Q. However, this procedure is not defined when Q has an infinite derivation.

In [16] a definition of sldcnf-resolution is given, which overcomes the drawback of the original one, by defining the subsidiary trees used to resolve negative literals 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 this section we describe the corresponding formalization.

Let $Tree$ be the class of trees whose nodes are (possibly marked) queries of (possibly marked) literals, with substitutions and input clauses associated to edges. We consider *selected* as marker for literals, *successful* and *failed* as markers for nodes. A literal is called *selected* if it is marked as selected. Moreover it is assumed that *primitive literals cannot be selected.*

A *pre-*sldcnf*-tree* $\mathcal{T}$ is a subset of $Tree$ together with a partial function *subs* assigning to nodes containing a selected negative literal $\neg A$ a tree in $\mathcal{T}$ with root A. $\mathcal{T}$ contains one special element called *main tree.*

Consider a tree T of a *pre-*sldcnf*-tree.* T is called *successful* if some of its leaves are marked as *successful.* T is called *finitely successful* if it is finite, all its leaves are marked and there is at least one leaf marked as *successful.* T is called *finitely failed* if it is finite and all its leaves are marked as *failed.*

We introduce now the concept of sldcnf-answer and full answer for a query Q, which are used to give an inductive definition of pre-sldcnf-tree.

Consider a branch in a successful tree T with root Q which ends with a reduced query, say Q'. Let $\alpha_1, \ldots, \alpha_n$ be the consecutive mgu's along this branch. Let $\theta = (\alpha_1 \ldots \alpha_n)_{|vars(Q)}$. Then the equality formula $\exists(E_\theta \wedge E_{Q'})$ is called sldcnf*-answer for* Q, where $\exists$ quantifies over all the variables that do not occur in Q. If T is finitely successful, then we call *full answer*, denoted by F_Q, the disjunction of all the answers for Q.

Definition 2.1 **(Inductive Definition of pre-sldcnf-tree)**

(i) $\{T\}$ is a pre-sldcnf-tree, called *initial pre-*sldcnf*-tree*, where T contains only one node which has a selected literal if it is not reduced; *subs* is undefined.

(ii) If $\mathcal{T}$ is a pre-sldcnf-tree, then any *extension* of $\mathcal{T}$ is a pre-sldcnf-tree.

The *extension of a pre-*sldcnf*-tree* $\mathcal{T}$ is defined by the following sequence of steps.

1. Mark all reduced queries as *successful.*
2. For every unmarked leaf Q in some tree $T \in \mathcal{T}$, let L be its selected literal. Then:
   - if $L = A$ is an atom then:
     (a) if there is no resolvent of Q in P then mark Q as *failed*;
     (b) otherwise, add all the resolvents of Q as sons of Q in T, associate to every edge the input clause and the mgu used to compute the corresponding resolvent, and mark a literal in every non-empty resolvent.
   - if $L = \neg A$ is a negative literal then:
     (a) if *subs*(Q) is *undefined* then add the tree T' with the single node A to $\mathcal{T}$ and set *subs*(Q) to be equal to T';
     (b) if *subs*(Q) is *defined* then:
       – if *subs*(Q) is *finitely failed* then add $Q - \{L\}$ as son of Q in T, with one marked literal;
       – if *subs*(Q) is *successful* and the disjunction of its *answers* is equivalent to *true* then mark Q as *failed*;

- if $subs(Q)$ is *finitely successful* then let $NA_1 \vee \ldots \vee NA_n$ be the disjunction of the simple equality formulae obtained by negating F_A: for every $j \in [1,n]$ add the query obtained from Q by replacing L with the formula NA_j, with one marked literal, as son of Q in T.

- if L is an equality, say $s = t$ then:
  (a) if s and t are not unifiable then mark Q as *failed*;
  (b) otherwise, add $(Q - \{L\})\theta$ with one marked literal, as son of Q in T, where $\theta = mgu(s,t)$.
- if L is an inequality, say $\forall(s \neq t)$, then
  (a) if it is valid then add $Q - \{L\}$ with one marked literal as son of Q in T;
  (b) if it is unsatisfiable then mark Q as *failed.*

□

A pre-sldcnf-tree can be regarded as a special directed graph with two types of edges, those from the tree structure and those connecting a node N with the root of $subs(N)$. Therefore, the concepts of *inclusion* between such trees and of *limit* of a growing sequence of trees are defined.

Definition 2.2 **(sldcnf-tree)** An sldcnf-*tree* is the limit of the sequence $\mathcal{T}_0, \ldots, \mathcal{T}_n, \ldots$, such that $\mathcal{T}_0$ is an *initial pre-*sldcnf*-tree*, and $\mathcal{T}_{i+1}$ is an *extension* of $\mathcal{T}_i$. □

We say that $\mathcal{T}$ is an sldcnf*-tree for* Q if Q is the root of the main tree of $\mathcal{T}$. Then an *answer for* Q is an answer for the root of the main tree.

A *path* in $\mathcal{T}$ is a sequence of nodes $N_0, \ldots, N_i, \ldots$, s.t. for all i, N_{i+1} is either an immediate descendent of N_i in some tree in $\mathcal{T}$, or is the root of the tree $subs(N_i)$. We say that an sldcnf-tree is *finite* if it does not contain any infinite path.

Definition 2.3 **(sldcnf-derivation)** A *(pre-)*sldcnf*-derivation for* Q, denoted by ξ, is a branch in the main tree, whose root is Q, of a (pre-)sldcnf-tree $\mathcal{T}$ together with the set of all trees in $\mathcal{T}$ whose roots can be reached from the nodes of this branch. ξ is said to be *finite* if all paths of $\mathcal{T}$ fully contained in this branch, and these trees, are finite. □

In the following two sections, generalizations to extended general programs of the concepts of acyclicity ([1]) and of acceptability ([6]) are given. These concepts are based on the notion of level mapping. A *level mapping* is a function $|\ |$ from ground literals to natural numbers s.t.:

- $|\neg A| = |A|$,
- $|A| = 0$ if A is a constraint,
- $|A| > 0$ otherwise.

3. Acyclic Programs

In this section we recall the definition of acyclic program, and some useful results from [16].

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 $i \in [1,m]$.

P is *acyclic* if there exists a level mapping $|\ |$ s.t. P is acyclic w.r.t. $|\ |$. □

If P is acyclic then all sldcnf-derivations of ground queries are finite. The following definition of bounded query allows to extend this result to a bigger class of queries that contains all ground queries.

Definition 3.2 **(Bounded Query)** A literal L is called *bounded* w.r.t. a level mapping $|\ |$ if $|\ |$ is bounded on the set $[L]$ of ground instances of L. If L is bounded then $|[L]|$ denotes the maximum that $|\ |$ takes on $[L]$. Then we say that L is bounded by l if $l \geq |[L]|$. A query $Q = L_1, \ldots, L_n$ is called bounded w.r.t. $|\ |$ if every L_i is bounded w.r.t. $|\ |$, for $i \in [1, n]$. If Q is bounded then $|[Q]|$ denotes the (finite) multiset (see [12]) consisting of the natural numbers $|[L_1]|, \ldots, |[L_n]|$. □

The following theorem shows that acyclic programs terminate for bounded queries.

Theorem 3.3 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.

We say that the program P is *terminating* if all `sldcnf`-derivations of ground queries are finite. The converse of Theorem 3.3 also holds.

Theorem 3.4 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 all its `sldcnf`-derivations are finite.

From Theorem 3.3 and Theorem 3.4 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 these results do not hold when negation as failure is assumed, because of the abnormal form of termination caused by the selection of non-ground negative literals.

4. Acceptable Programs

In this section we consider a fixed selection rule, corresponding to the natural extension of the Prolog selection rule to programs containing constraints. The notion of acceptable program ([6]) 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 $\overline{n}$, the class of models of P, whose restriction to the relations from Neg_P^* are models of $comp(P^-)$, is considered, 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 4.1 **(Acceptable Program)** Let $|\ |$ be a level mapping for P and let I be a model of P whose restriction to the relations from Neg_P^* is a model of $comp(P^-)$. P is called *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 $i \in [1, \overline{n}]$, 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 model of P whose restriction to the relations from Neg_P^* is a model of $comp(P^-)$. □

The following definition of `ldcnf`-resolution formalizes the fixed selection rule we consider.

Definition 4.2 (`ldcnf`**-tree**) A *(pre-)*`ldcnf`*-tree* is a (pre-)`sldcnf`-tree where the selection rule is such that in every node the leftmost possible literal is marked, where a literal is called *possible* if it is not a primitive inequality. □

We refer to this selection rule as *Prolog selection rule*, due to its strong similarity with the one used in Prolog. Intuitively, the selection of primitive inequalities is delayed until their free variables become enough instantiated to render the inequality valid. `ldcnf`-resolution is used to define the concept of left-terminating program. A program P is *left-terminating* if all `ldcnf`-derivations of ground queries are finite.

If P is acceptable then all `ldcnf`-derivations of ground queries are finite. The following definition of bounded query allows to prove this result for a bigger class of queries that contains all ground queries. Let $Q' = L_1, \ldots, L_n$ be a ground query, let $|\ |$ be a level mapping and let I be a model of P whose restriction to the relations from Neg_P^* is a model of $comp(P^-)$. Then we associate to Q' the multiset

$$|Q'|_I = bag(|L_1|, \ldots, |L_{\overline{n}}|),$$

where $\overline{n} = min(\{n\} \cup \{i \in [1,n] \mid I \not\models L_i\})$. Then to a query Q is associated the set of multisets

$$|[Q]|_I = \{|Q'|_I \mid Q' \text{ is a ground instance of } Q\}.$$

Definition 4.3 **(Bounded Query)** The query Q is *bounded* by k w.r.t. $|\ |$ and I if $k \geq l$ for $l \in \cup|[Q]|_I$, where $\cup|[Q]|_I$ denotes the set-theoretic union of the elements of $|[Q]|_I$. Q is called *bounded* w.r.t. $|\ |$ and I if for some k it is *bounded* by k w.r.t. $|\ |$ and I. □

Theorem 4.4 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.

Theorem 4.5 Let P be a left-terminating program. Then for some level mapping $|\ |$ and for a model I of $comp(P)$

(i) P is acceptable w.r.t. $|\ |$ and I,

(ii) for every query Q, Q is bounded w.r.t. $|\ |$ and I iff all its `ldcnf`-derivations are finite.

So, to prove that a program is acceptable is in general more difficult than to prove that it is acyclic, because of the use of a suitable model of the program. In the following, we introduce an equivalent definition of acceptability, called e-acceptability, which is in general simpler to apply. The program is split in two suitable parts: one part is shown to be an acyclic program, while the other one is shown to satisfy the acceptability test. Formally, the following notion, originally introduced in [4], is used.

Definition 4.6 Let P and R be two programs. We say that P *extends* R, written $P > R$, if:

(i) P and R define different relations;

(ii) 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 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 4.7 **(e-acceptability)** A program P is called *e-acceptable wrt* R if the following conditions hold:

1. $P = P_1 \cup R$, for some P_1 which extends R;
2. R is acyclic wrt a level mapping, say $|\ |_2$;
3. $P \setminus R$ is acceptable, wrt a level mapping, say $|\ |_1$, and a model, say 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 then $|H|_1 \geq |L_i|_2$.

A program is *e-acceptable* if there exists R s.t. P is e-acceptable wrt R. □

Observe that for R equal to the empty set of clauses, we obtain the original definition of acceptability. Now, the notion of *e-bounded query* can be given as follows. Suppose that P is e-acceptable w.r.t. R; let $|\ |_1$ and I be the corresponding level mapping and model for $P \setminus R$, and let $|\ |_2$ be the corresponding level mapping for R.

Consider a ground query $Q' = L_1, \ldots, L_n$. We associate with Q' the multiset

$$|Q'|_I = bag(|L_1|, \ldots, |L_{\overline{n}}|),$$

where $\overline{n} = min(\{n\} \cup \{i \in [1,n] |\ L_i \text{ defined in } P_1 \text{ and } I \not\models L_i\})$, and where

$$|L_i| = \begin{cases} |L_i|_1 & \text{if } L_i \text{ is defined in } P \setminus R \\ |L_i|_2 & \text{if } L_i \text{ is defined in } R \\ 0 & \text{if } L_i \text{ is a constraint.} \end{cases}$$

Then, we associate with a query Q the following set of multisets:

$$|[Q]|_I = \{|Q'|_I \mid Q' \text{ is a ground instance of } Q\}.$$

We say that a query Q is *e-bounded* by k w.r.t. $|\ |$ and I if $k \geq l$ for $l \in \cup|[Q]|_I$, where $\cup|[Q]|_I$ denotes the set-theoretic union of the elements of $|[Q]|_I$. Q is called *e-bounded* w.r.t. $|\ |$ and I if for some k it is *e-bounded* by k w.r.t. $|\ |$ and I.

The following result holds.

Theorem 4.8 Suppose that P is e-acceptable wrt R. Let Q be an e-bounded query. Then every `ldcnf`-tree for Q in P contains only e-bounded queries and is finite.

Proof. Let $|\ |_1$ and I, and $|\ |_2$ be respectively the level mapping and model for $P \setminus R$ and the level mapping for R s.t. P is e-acceptable w.r.t. R. Let $|\ |$ be the corresponding level mapping used to prove that Q is e-bounded. Let $Q = L_1, \ldots, L_n$ and suppose that L_i is its selected literal. We distinguish the following two cases.

- L_i is defined in $P \setminus R$. Then the conclusion follows by condition 3 of Definition 4.7, by Theorem 4.4 and by the condition 4 of Definition 4.7.

- L_i is defined in R. Then the conclusion follows by conditions 1 and 2 of Definition 4.7, and by Theorem 3.3. □

The following corollary establishes the equivalence of the notions of acceptability and e-acceptability. It follows directly from by Theorem 4.8 and Theorem 4.5.

Corollary 4.9 Let P and R be extended general logic programs.

- If P is e-acceptable w.r.t. R then P is acceptable.
- If P is acceptable then it is e-acceptable w.r.t. (the empty program) $\emptyset$.

5. Some Classical Problems in Non-monotonic Reasoning

In this section we show how various problems in non-monotonic reasoning can be formalized by means of acyclic or acceptable programs.

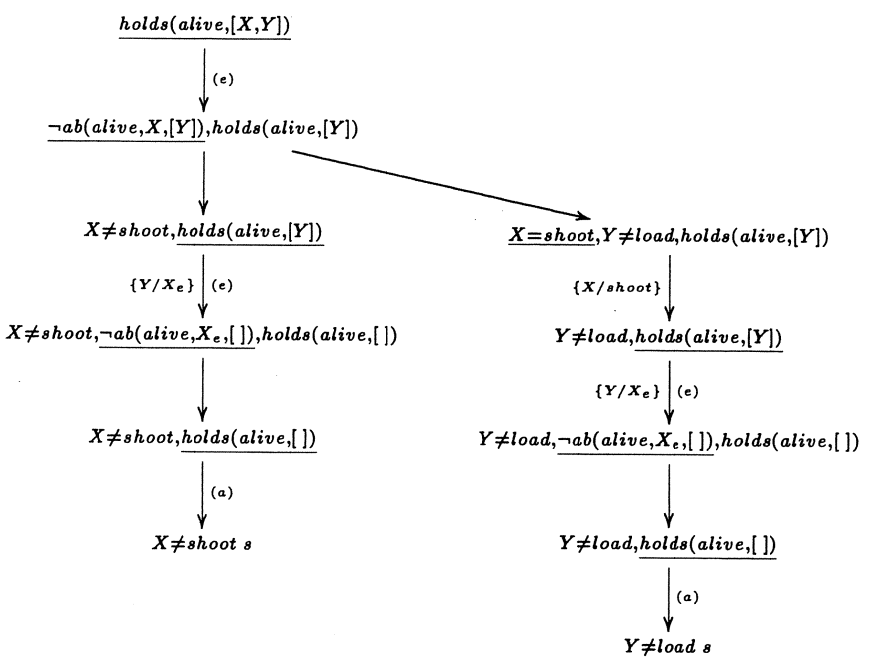
5.1 Temporal Reasoning

It has been shown in [1] how various forms of temporal reasoning can be described using acyclic programs. In particular, the following program YSP is a formalization of the so-called Yale Shooting Problem in terms of an acyclic program.

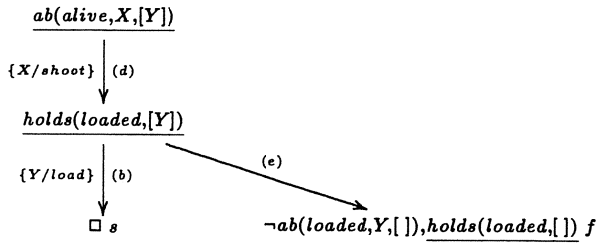
(*a*) `holds(alive, []) ←.`
(*b*) `holds(loaded,[load|Xs]) ←.`
(*c*) `holds(dead,[shoot|Xs]) ← holds(loaded,Xs).`
(*d*) `ab(alive,shoot,Xs) ← holds(loaded,Xs).`
(*e*) `holds(Xf,[Xe|Xs]) ← ¬ ab(Xf,Xe,Xs), holds(Xf,Xs).`

Here Xf, Xs and Xe denote variables, representing a generic *fact*, *situation* and *event*, respectively. We recall the problem following [13]. Consider a person which is `alive`. The occurrence of the event `load` implies the fact that the gun becomes `loaded`. The event `shoot` in the situation `loaded` implies the fact that the person becomes `dead`. Moreover it is abnormal for a person to be `alive` when the event `shoot` happens in the situation `loaded`. Finally facts persist under the occurrence of events which are not abnormal. The interest on this problem is due to the fact that its formalization by means of theories about non-monotonic reasoning yields weak conclusions.

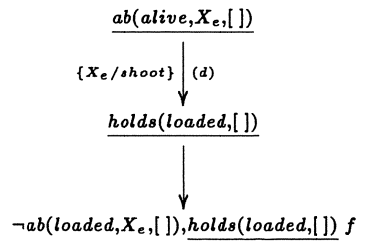
Suppose we want to know which assumptions on X and Y are needed to explain the current state where the person is `alive` in the situation resulting from the occurrence of a generic event in a generic situation. This problem can be expressed by means of the query `holds(alive,[X,Y])`. This query is bounded, hence every `sldcnf`-derivation is finite. The following is an `sldcnf`-tree for `holds(alive,[X,Y])`.



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



Moreover, the two trees $subs(Y \neq load, \underline{\neg ab(alive, X_e, [\,])}, holds(alive, [\,]))$ and $subs(X \neq shoot, \underline{\neg ab(alive, X_e, [\,])}, holds(alive, [\,]))$ coincide and are represented below.



So, an explanation of the current state is that either X is not equal to *shoot* or Y is not equal to *load*. Notice that by using `sldnf`-resolution `holds(alive,[X,Y])` flounders.

5.2 Taxonomies

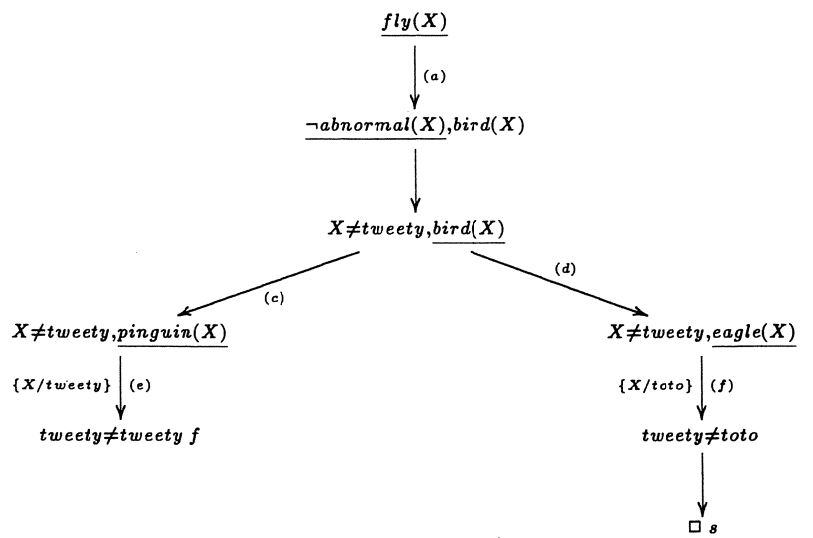
In this example we show how to represent taxonomies with general logic programs. The classical `tweety` problem in non-monotonic reasoning can be formalized by the following program *Tweety*.

(*a*) `fly(X)` ← ¬ `abnormal(X), bird(X).`
(*b*) `abnormal(X)` ← `pinguin(X).`
(*c*) `bird(X)` ← `pinguin(X).`
(*d*) `bird(X)` ← `eagle(X).`
(*e*) `pinguin(tweety)` ←.
(*f*) `eagle(toto)` ←.

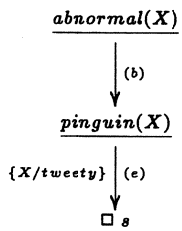
It is easy to check that *Tweety* is acyclic, by choosing the following level mapping.

$|fly(x)| = 3,\ |abnormal(x)| = 2,\ |bird(x)| = 2,\ |pinguin(x)| = 1,\ |eagle(x)| = 1.$

Suppose we want to know which assumption on a bird is needed to explain the current state in which it flies. This can be expressed by the query `fly(X)`. This query is bounded and $X \neq tweety$ is its answer, as illustrated below by the `sldcnf`-tree for `fly(X)`,



where $subs(abnormal(X))$ is the tree below:



So, the additional assumption on X is that it has not to be equal to *tweety*. Notice that by using `sldnf`-resolution `fly(X)` flounders.

5.3 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. In [20] this problem is extensively studied, and alternative formalizations are presented.

In the following example a formalization of the simple "blocks world problem" ([23]) is given, by means of an acyclic program. There are three blocks a, b, c and three different places p, q and r of a table. A block can 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 a blocks world is possible, i.e., if it can be obtained from the initial situation by performing a sequence of possible moves. For instance, it is not possible to move a block if there is another block on its top. A clausal representation of this problem is given in [14], where it is described in detail in terms of pre- and postconditions which specify, respectively, when a move is possible, and the result of performing a move. Here we prefer to use McCarthy and Hayes situation calculus ([18]) to present the problem, to keep the exposition uniform w.r.t. the previous examples. Thus we distinguish facts, events and situations.

We consider three types of *facts*: $loc(X, L)$ denotes the fact *a block X is in the location L*; $on(X, Y)$ denotes the fact *a block X is on a block Y*;and $clear(L)$ denotes the fact *there is no block in the location L*.

Next, only one type of *event* is given: $move(X, L)$ denotes the event *a block X is in a location L*.

Finally, situations are denoted by lists: the initial *situation* is denoted by [ ], while $[Xe|Xs]$ denotes the situation corresponding to the occurrence of the event Xe in the situation Xs.

So the blocks world can be described as follows. Let $top(X)$ denote the top of the block X. In the initial situation [ ] the blocks are assumed to be on specific locations.

- In the situation $[move(X, L)|Xs]$, obtained from the situation Xs performing the event $move(X, L)$, the fact $loc(X, L)$ holds if L is not equal to the top of X and if in Xs there was no block neither above X nor in the location L.

- In the situation $[Xe|Xs]$ the fact $loc(X, L)$ holds if in the situation Xs, X was in the location L and the event Xe was not abnormal w.r.t. X, i.e., if Xe was not of the form $move(X, -)$.

- In the situation Xs the fact $on(X, Y)$ holds if:
  - either the fact $loc(X, top(Y))$ holds,
  - or there exists a block Z such that both the facts $loc(X, top(Z))$ and $on(Z, Y)$ hold.

- In the situation Xs the fact $clear(L)$ holds if for every block X the fact $loc(X, L)$ does not hold.

Moreover, we define the legal state for a situation Xs to be the list $[(a, p1), (b, p2), (c, p3)]$ s.t. $loc(a, p1)$, $loc(b, p2)$, and $loc(c, p3)$ hold in Xs. Legal states will be used in the following section, where we will deal with plans in the blocks world.

The above description can be easily translated into a general logic program, yielding the following program `blocks-world`:

```
(i1)  holds(loc(a,p),[]) ←.
(i2)  holds(loc(b,q),[]) ←.
(i3)  holds(loc(c,r),[]) ←.

(b1)  block(a) ←.
(b2)  block(b) ←.
(b3)  block(c) ←.

(p1)  place(p) ←.
(p2)  place(q) ←.
(p3)  place(r) ←.
(p4)  place(top(a)) ←.
(p5)  place(top(b)) ←.
(p6)  place(top(c)) ←.

(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).
```

It is easy to check that `blocks-world` is acyclic wrt the following level mapping $|\ |$, where if y is a list then $|y|$ is set to be its length, otherwise $|y|$ is set to be equal to 0.

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

$|busy(x,y)| = 3*|y|+2,$

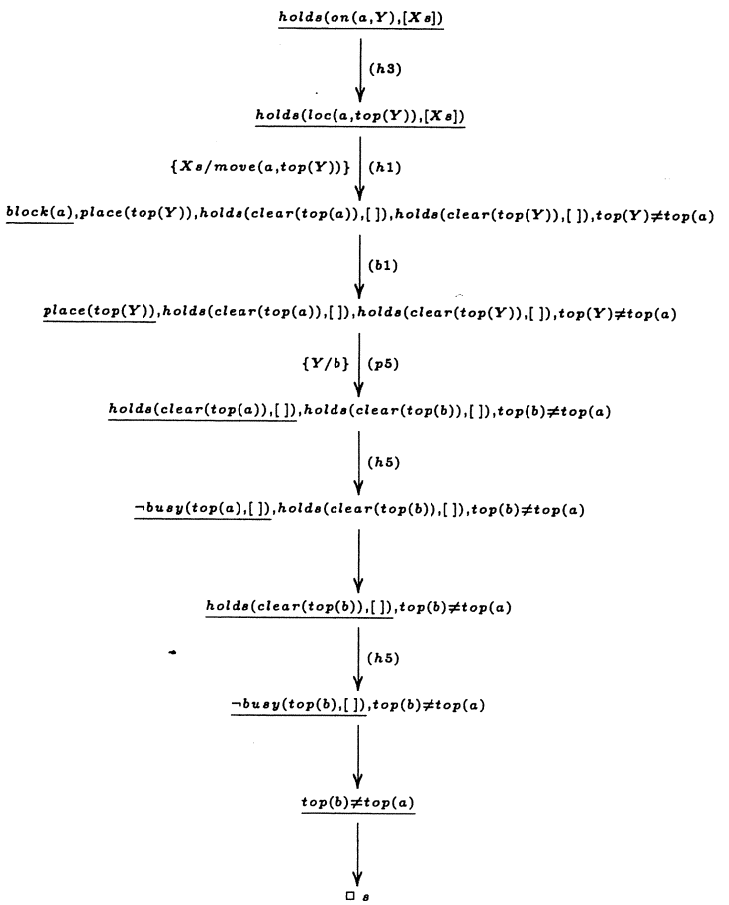
$|block(x)| = 0,$

$|place(x)| = 0,$

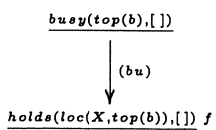
$|abnormal(x,y,z)| = 0,$

$|legal-s(x,y)| = 3*|y|+2.$

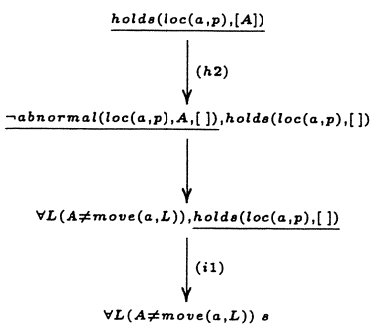
Suppose that we would like to know when the block a is above some other block in a situation obtained by performing an event starting from the initial situation. This can be expressed by means of the query `holds(on(a,Y),[Xs])`. This query is bounded, hence every its `sldcnf`-derivation is finite. The two answers $Y = b$ and $Y = c$ are obtained: below, a derivation yielding the first answer is drawn.



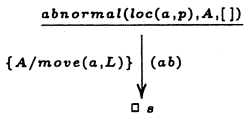
Here both the `sldcnf`-trees $subs(\underline{\neg busy(top(a),[\,]),holds(clear(top(b)),[\,]),top(b) \neq top(a)})$ and $subs(\underline{\neg busy(top(b),[\,])},top(b) \neq top(a))$ are of finite failure. The latter is illustrated below:



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.



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



Notice that using sldnf-resolution this query does flounder.

5.4 Planning in the Blocks World

The following example illustrates how acceptable programs can be used to formalize problems in non-monotonic reasoning. More specifically, we consider here plan-formations in the blocks world. This problem consists of finding a plan in the blocks world, i.e., to specify a sequence of possible moves to obtain a particular configuration by restacking blocks. The initial configuration is here specified by a situation which can be reached from the initialization described by the three clauses (i1), (i2), (i3) of the program `blocks-world` given previously. Alternatively, one can let unspecified the initialization, which has then to be provided every time the program is tested. This latter choice is taken in Sterling and Shapiro [24]. The problem of finding a plan in the blocks world can be solved by means of a nondeterministic algorithm, informally explained as follows ([24]): *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 predicate `legal-s` are supposed to be included in the program.

```
(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).
```

We prove that `planning` is e-acceptable w.r.t. the program `r-blocks-world`, which is obtained from `blocks-world` by deleting the clauses ($h3$), ($h4$). Let T be the program `planning\r-blocks-world`. Then T is the following program:

```
(t')  transform(Xs,St,Plan) ← state(St0), trans(Xs,St,[St0],Plan).

(t1') trans(Xs,St,Vis,[ ]) ←.
(t2') trans(Xs,St,Vis,[Act|Acts]) ←
      state(St1), ¬ member(St1,Vis), 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).
```

It is easy to check that condition 1 of the definition of e-acceptability is satisfied. Concerning the other three requirements, we proceed as follows.

First we define a suitable model of T. Consider the following interpretations, where $[L]$ denotes the set of ground instances of L, $set(y)$ denotes the set of elements of the list y, $S \stackrel{\text{def}}{=} \{[(a,p1),(b,p2),(c,p3)] \mid \text{for } i \in [1,3], pi \in \{p,q,r,top(a),top(b),top(c)\}\}$ and N denotes the cardinality of S:

$I_{transform} = [transform(X,Y,Z)]$,

$I_{trans} = [trans(X,Y,Z,W)]$,

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

$I_{state} = \{state(x) \mid x \in S\}$.

Let $I = I_{transform} \cup I_{trans} \cup I_{member} \cup I_{state}$. Then it is easy to prove that I is a model of T. Moreover, T^- is equal to the program$\{(m1),(m2)\}$, and I_{member} is a model of $comp(T^-)$.

Next, we define a level mapping $|\ |_1$ for T as follows. Let $el(z)$ denote $set(z)$ if z is a list, the empty set otherwise. Let $card(el(z) \cap S)$ be the cardinality of the set $el(z) \cap S$. Then $|\ |_1$ is defined as follows, where $|x|$ is defined in the previous example:

$|transform(x,y,z)|_1 = N + 3 * (|x| + 1) + 2 + 3 + 1$;

$|trans(x,y,z,w)|_1 = N - card(el(z) \cap S) + 3 * (|x| + 1) + 2 + 3 + |z|$;

$|state(x)|_1 = 7$;

$|member(x,y)|_1 = |y|$.

Note that $(N - card(el(z) \cap S)$ is greater or equal than 0. Then $|\ |_1$ is well defined. Moreover,

$$|transform(x,y,z)|_1 \geq 8, \tag{5.1}$$

$$|trans(x,y,z,w)|_1 \geq 8, \tag{5.2}$$

and

$$|trans(x,y,z,w)|_1 > |z|. \tag{5.3}$$

We prove that T is acceptable w.r.t. $|\ |_1$ and I.

Consider a ground instance:

$transform(xs,xt,plan) \leftarrow state(st0), trans(xs,st,[st0],plan)$

of (t'). Then from (5.1) we have that:

$$|transform(xs,xt,plan)|_1 > |state(st0)|_1.$$

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

$$|transform(xs,xt,plan)|_1 > |trans(xs,st,[st0],plan)|_1.$$

Consider a ground instance:

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

of $(t2')$. Then from (5.2) we have that:

$$|trans(xs,st,vis,[act|acts])|_1 > |state(st1)|_1,$$

and from (5.3) we have that

$$|trans(xs, st, vis, [act|acts])|_1 > |\neg member(st1, vis)|_1.$$

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,

$$|trans(xs, st, vis, [act|acts])|_1 > |trans([act|xs], st, [st1|vis], acts)|_1.$$

It is easy to check that also the remaining clauses of T satisfy the test of acceptability.

Now, clearly `r-blocks-world` is acyclic w.r.t. the level mapping $|\ |_2$ defined as in the previous example. Moreover, condition 4 of the definition of e-acceptability follows by construction of $|\ |_1$. This concludes the proof that `planning` is e-acceptable w.r.t. `r-blocks-world`.

Suppose that we would like to find a plan of actions which lead from the initial situation to a given state, say `st`. This can be expressed by means of the query `transform([ ],st,Plan)`. This query is e-bounded, hence by Theorem 4.8 every its `ldcnf`-derivation is finite. Notice that using `sldcnf`-resolution this query has an infinite derivation.

6. Conclusion

In this paper we showed how to formalize some classical problems in non-monotonic reasoning by means of acyclic and acceptable programs. Moreover, we showed that more queries can be answered when a form of constructive negation, based on the constructive negation procedure by Chan, is used. Thus, we demonstrated that a simple class of general logic programs augmented with a simple form of constructive negation is sufficient for the treatment of interesting problems in non-monotonic reasoning.

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, Special Issue, 1994.
5. K. R. Apt, D. Pedreschi. Studies in pure Prolog: Termination. Symposium on Computational Logic, J. W. Lloyd, editor, 150–176, Berlin, Springer-Verlag, 1990.
6. K. R. Apt, D. Pedreschi. Proving Termination of General Prolog Programs. Report CS-R911, Centre for Mathematics and Computer Science,1991.
7. K. R. Apt, D. Pedreschi. Modular Termination Proofs for Logic and Pure Prolog Programs. Report CS-R9316, Centre for Mathematics and Computer Science,1993.
8. C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19,20:73–148, 1994.
9. M. Bezem. Characterizing Termination of Logic Programs with Level Mappings. Proceedings of the 1990 North American Conference on Logic Programming , 1990.
10. D.Chan. Constructive Negation Based on the Completed Database. Proceedings of the 9th International Conference and Symposium on Logic Programming, 1988.
11. D.Chan. An extension of Constructive Negation and its Application in Coroutining. Proceedings of the North American Conference on Logic Programming, 1989.
12. N. Dershowitz. Termination of Rewriting. Journal of Symbolic Computation, 3, pp. 69-116, 1987.
13. S. Hanks, D. McDermott. Nonmonotonic Logic and Temporal Reasoning. Artificial Intelligence, 33, pp. 379-412, 1987.
14. R. Kowalski. Logic for Problem Solving. Elsevier North-Holland, New York, 1979.

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

16. E. Marchiori. On Termination of General Logic Programs wrt Constructive Negation. Submitted, 1994.

17. J. McCarthy. Circumscription - A Form of Non-monotonic Reasoning. J. of Artificial Intelligence 13, pp. 27-39, 1980.

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. R.C. Moore. Semantic Considerations of Non-monotonic Logic. J. of Artificial Intelligence 25, pp.75-94, 1985.

20. N.J. Nilsson. Principles of Artificial Intelligence. Springer-Verlag, 1982.

21. R. Reiter. On Closed-World Data Bases. Logic and Data Bases, Eds. H. Gallaire, J. Minker, Plenum Press, New York, pp. 55-76, 1978.

22. R. Reiter. A Logic for Default Theory. J. of Artificial Intelligence 13, pp.81-132, 1980.

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

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