

SEMANTIC TREES FOR DISJUNCTIVE LOGIC PROGRAMS

JOHN R. FISHER *Computer Science Department,
California State Polytechnic University, Pomona, CA 91768
jrfisher@csupomona.edu*

Abstract: In order to design software that is intended to compute answers to queries that are in accordance with some logic programming semantics, one would like to offer up a formal specification of the software design which could be used profitably to construct the software, and one would want to be able to prove that the specification is in fact faithful to the semantics. This paper presents a constructive formal specification of semantic trees and truth-value determinations using semantic trees for disjunctive logic programs with negation as failure. This specification methodology directly supports the design of top-down interpreters for well-founded semantics.

Keywords: logic programming, disjunctive logic programs, indefinite and definite programs, negation as failure, formal negation, program trees, full trees, semantic trees, well-founded semantics, bounded trail property.

1. Motivation and Background

Consider the familiar Prolog meta-interpreter program M for Prolog:

```
prove(true) :- !.  
prove((G,Goals)) :- !, prove(G), prove(Goals).  
prove(G) :- clause(G,B), prove(B).
```

Which tree is searched by M?

For example, consider the abstract logic program P (or its Prolog equivalent):

```
p <- q(1), q(2)  
q(x) <-  
r(x) <-  
q(2) <- q(1)  
r(1) <-
```

For the goal $\leftarrow p$, an SLD-derivation (or the Prolog search) searches (or grows) the following *derivation tree* **D**, pictured below on the left, using the clauses of **P** (adapted from Bol, 1991, p. 117):



However, **M** actually searches (or grows) the tree **T**, pictured above on the right.

So **M** does not (exactly) simulate SLD deduction. To emphasize this point further, suppose that **M** were to be augmented with a loop-check, such as in

```

prove(true, Trail) :- !.
prove((G,Goals), Trail) :- !, prove(G, Trail), prove(Goals, Trail).
prove(G, Trail) :- loop_check(G,Trail), !, fail.
prove(G, Trail) :- clause(G,B), prove(B,[G|Trail]).

loop_check(G,[F|R]) :- G == F, !.
loop_check(G,[F|R]) :- loop_check(R).
  
```

Then the new **M** appropriately finds **NO** loops in **T**. But, an augmented SLD procedure with a loop check on lead (or selected) goals could prune **D** at the second occurrence of 'q(1)', marked with the asterisk (*), and fail to answer correctly for the goal $\leftarrow p$.

We call a tree like **T** a *semantic program tree*. For positive logic programs, the general definition of a semantic program tree, or a *P-tree* for short, requires the trees themselves to be finite (a finite data structure), to have unordered branchings determined by ground instances of clauses of the program **P**, and to allow repeated (but separately identifiable) nodes (because the clauses of **P** could sometimes lead to repeated occurrences).

We say that a ground positive literal of the program is a *tree-consequence* of the program **P** provided that there is some (finite) **P**-tree rooted at the literal having all 'true' leaves. Then

Proposition. *A ground literal L is a tree-consequence of the positive logic program P if, and only if, L belongs to the least model of P .*

Thus, we see that M directly implements the tree-based "semantics" defined above (which is equivalent to the standard least model semantics). Or, to exaggerate a little, we could say that M is an executable version of the tree-based semantics for positive programs. It is interesting that the tree-based specification is both a *requirements* specification (because it is equivalent to least-model semantics) and a *design* specification (because of its direct relationship to the meta-interpreter M).

(The reader is invited to write the Prolog meta-interpreter that *does* simulate SLD-deduction. It is not M !)

Now, if we turn our attention to logic programs with negation as failure, we will see that the distinction between derivation trees and semantic trees is more important. Both SLDNF-resolution and SLS-resolution modify the SL-resolution engine in attempts to compute reasonable answers for goals for (non disjunctive) logic programs with negation as failure. SLDNF is based upon Clarks completion semantics (Clark, 1978) and is appropriately suited to programs without loops (and finite failure). SLS (Przymusinski, 1989) attempts to accommodate programs that have some loops for positive literals only, but is restricted to stratified programs (so no "loops through negation"). Also, SLS resolution, being based on the SL-derivation trees, is prone to unsound loop checks like in the previous example (the tree D).

The problem here is that SLDNF and SLS are searching, or growing, the wrong trees!

We take a considerably different approach in this paper. The class of disjunctive programs with negation as failure considered here contain disjunctive clauses of the form

$$A_1, A_2, \dots, A_k \leftarrow B_1, \dots, B_m, \text{not}(C_1), \dots, \text{not}(C_n)$$

where each A_i , each B_j , and each C_l is a positive literal (possibly containing variables), $k \geq 1$, $m, n \geq 0$. The sequence of literals A_1, A_2, \dots, A_k constitutes the *head* of the clause and this sequence is a *disjunction*. The sequence of literals $B_1, \dots, B_m, \text{not}(C_1), \dots, \text{not}(C_n)$ is a *conjunction* and is the *body* of the clause. In particular, there is no stratification assumed. We design a constructive "specification of semantics" based upon semantics program trees (**NOT** SL-type procedural trees), in a fashion similar to that discussed above for positive programs. This specification uses a (finite) tree data structure to determine (or support) meanings. The clauses of the program, together with well specified contrapositive clause forms associated with the program, are used to specify the semantic trees. The tree-based specification must accommodate looping along trails in

the trees. The trails can stay in a tree or leave at a negative leaf. Recursion, or looping, may be positive (within a single tree), or through negation (involving nodes in more than one tree). The tree-based specification must specify generally when a P-tree is "full" enough: For non disjunctive logic programs, this happens when all leaves are 'true', or a repeat node (from the current or some previous tree), or negative of the form not(L), in which case P-trees rooted at L need to be considered. (For disjunctive logic programs, the characterization is somewhat more complicated.) The specification uses three truth values, with positive looping counting as failure, and looping through negation counting as indeterminacy (undefined truth value). Lastly, but closely related to the previous requirements, we require that the tree-based semantics should correspond -- as much as we can guarantee -- to the well-founded semantics for non disjunctive logic programs with negation as failure (VanGelder, Ross, and Schlipf, 1991).

The resulting constructive tree-based specification of semantics for disjunctive logic programs then serves as the design (and requirements) specification for a meta-interpreter that computes well-founded semantics, just as for positive programs the tree-consequence semantics introduced earlier could serve as a design (and requirements) specification for the meta-interpreter M. For "reasonable" programs, the tree-based specification properly and significantly subsumes what SLDNF and SLS can compute, and, in addition, the tree-based specification gives an extension to disjunctive logic programs with negation as failure.

The paper provides the formal definitions for the tree-based specifications and characterizes the basic propositions regarding its properties. In particular, for non-disjunctive logic programs with negation as failure, the tree-based semantics is provably equivalent to the well-founded semantics if Bp is finite. We believe that the relationship to well-founded semantics holds much more generally.

The literature has references to similar trees for non-disjunctive logic programs, referred to as "clause trees" (Pereira, Alferes, and Aparacio, 1990), or sometimes as "proof trees" (Bruffaerts and Henin, 1988). We believe the formal specification approach taken by this paper is unique. There is the possibility that other formal software specification methodologies would be applicable.

2. Disjunctive Logic Programs

Let us assume that P is a disjunctive logic program whose clauses may have negation-as-failure literals in the bodies of its clauses. Thus, the clauses of P can be described as having the form

$$A_1, A_2, \dots, A_k \leftarrow B_1, \dots, B_m, \text{not}(C_1), \dots, \text{not}(C_n)$$

where A_i , each B_j , and each C_k is a positive literal, $k \geq 1$, $m, n \geq 0$. The sequence of literals A_1, A_2, \dots, A_k constitutes the *head* of the clause and this sequence is a *disjunction*. The sequence of literals $B_1, \dots, B_m, \text{not}(C_1), \dots, \text{not}(C_n)$ is a *conjunction* and is the *body* of the clause. The sequence B_1, \dots, B_m is the *positive part* of the body and the sequence $\text{not}(C_1), \dots, \text{not}(C_n)$ is the *negative part* of the body. If $k=1$ then the clause is said to be *definite*, otherwise it is *indefinite*. An *indefinite* program must have at least one indefinite clause, otherwise the program is *definite*.

In what follows, we will need to refer to contrapositive forms of a clause. A *primary alternative* of the clause

$$A_1, A_2, \dots, A_k \leftarrow B_1, \dots, B_m, \text{not}(C_1), \dots, \text{not}(C_n)$$

has the form

$$A_j \leftarrow \text{alt}(\sim A_1), \dots, \text{alt}(\sim A_{j-1}), \text{alt}(\sim A_{j+1}), \dots, \text{alt}(\sim A_k), B_1, \dots, B_m, \text{not}(C_1), \dots, \text{not}(C_n)$$

where $1 \leq j \leq k$. There are k primary alternatives if $k \geq 2$. If $k=1$ then the clause is definite and does not have any primary alternatives. The ' \sim ' denotes *formal negation*. The 'alt' forms are special markers for the alternatives. Note that there are now two kinds of negation that could be referred to: 'not' is negation as failure, and ' \sim ' is formal negation. We will need to maintain a careful distinction between these two negations. For a primary alternative the sequence $\text{alt}(\sim A_1), \dots, \text{alt}(\sim A_{j-1}), \text{alt}(\sim A_{j+1}), \dots, \text{alt}(\sim A_k)$ is called the *alternative part* of the body.

There are other contrapositive forms of clauses of an indefinite program that could be useful. These are called *backlinks* (for reasons that will be apparent later). They are formed as follows. Suppose that

$$A \leftarrow \alpha, B, \beta$$

is either a definite clause of P or a primary alternative whose head is the positive literal A and B is a literal in the positive part of the body; α, β are (possibly empty) sequences of the other literals of the body. Then

$$\sim B \leftarrow \alpha, \sim A, \beta$$

is a backlink clause, where ' \sim ' is formal negation. Later in this paper, we will characterize which backlinks are *potentially useful*.

Working Example. Consider the indefinite program P (X is a variable)

$p(X), q(X) \leftarrow r(X), \text{not}(s(X))$
 $s(a) \leftarrow p(a)$
 $r(a) \leftarrow \quad r(b) \leftarrow$
 $d(X) \leftarrow p(X), w(X)$
 $d(X) \leftarrow q(X), v(X)$
 $w(a) \leftarrow \quad w(b) \leftarrow \quad v(a) \leftarrow \quad v(b) \leftarrow \quad v(c) \leftarrow$
 $k(X) \leftarrow \text{not}(d(X))$

The primary alternatives of the indefinite clause are

$p(X) \leftarrow \text{alt}(\sim q(X)), r(X), \text{not}(s(X))$
 $q(X) \leftarrow \text{alt}(\sim p(X)), r(X), \text{not}(s(X))$

Here are all of the possible backlinks:

$\sim r(X) \leftarrow \text{alt}(\sim q(X)), \sim p(X), \text{not}(s(X))$
 $\sim r(X) \leftarrow \text{alt}(\sim q(X)), \sim q(X), \text{not}(s(X))$
 $\sim p(X) \leftarrow \sim d(X), w(X)$
 $\sim w(X) \leftarrow p(X), \sim d(X)$
 $\sim q(X) \leftarrow \sim d(X), v(X)$
 $\sim v(X) \leftarrow q(X), \sim d(X)$

Given a disjunctive logic program P , we say that the *usable* clauses of P are the definite clauses of P together with the primary alternatives of P and the backlink clauses of P . Note that the only usable clauses of P that actually belong to P are the definite clauses of P . The other usable clauses are contrapositive forms of clauses of P .

3. Program Trees

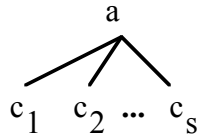
P -trees are constructed using the usable clauses of P . Let B_P be the Herbrand base of P (set of ground positive literals of P), and let

$$\sim B_P = \{ \sim b \mid b \in B_P \}.$$

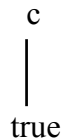
The *branchings* for P -trees are formed using the usable clauses of P . If

$$a \leftarrow c_1, \dots, c_s$$

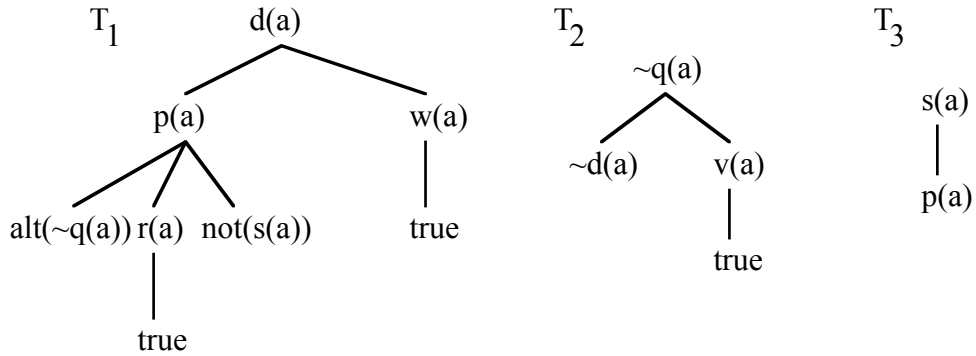
is a ground instance of a usable clause of P , then the corresponding branching node is

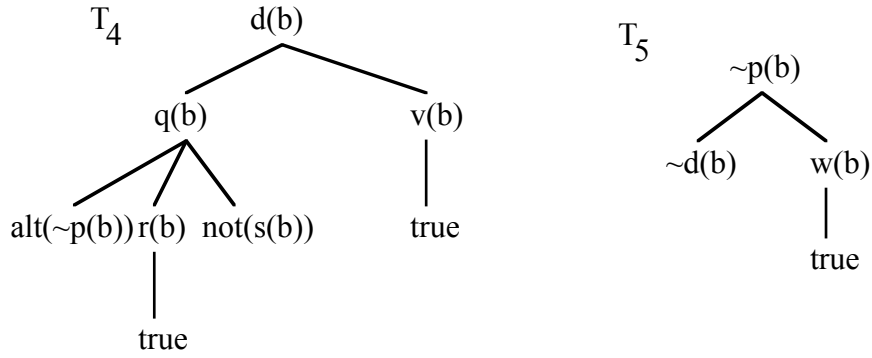


P-trees of height 0 are just elements of $B_P \cup \sim B_P$. P-trees of height 1 are those just described using a single branching node, rooted at some $a \in B_P \cup \sim B_P$. If T is a P-tree and c is a leaf not of the form 'alt(-)' or 'not(-)' then T may be extended using another branching at that leaf, as described above for P-trees of height 1. Negation-as-failure nodes 'not(-)' and alternatives 'alt(-)' *must be* leaves in the P-trees. Inductively, A P-tree is any *finite* tree that can be constructed in this fashion. The height of such a tree is, in general, the length of the longest branch from the root of the P-tree to its deepest leaf. If $c \leftarrow$ is a ground instance of a unit clause of P , then we write the corresponding branching P-tree node as



For the Working Example. Here are some P-trees that will be referred to later.



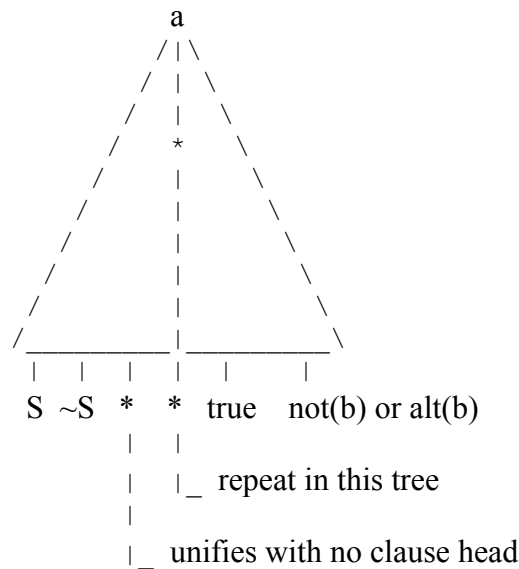


Definition 1. Suppose that $a \in B_P \cup \sim B_P$ and that S is a subset of $B_P \cup \sim B_P$. Then T is an *S-full P-tree rooted at a* if T is a P -tree rooted at a each of whose leaf nodes is either

- 1) an element of S , or
- 2) of the form $\sim b$ where $b \in S$
- 3) a literal in $B_P \cup \sim B_P$ which does not unify with the head of any clause of P , or
- 4) a literal in $B_P \cup \sim B_P$ which has itself as an ancestor in T , or
- 5) the true leaf, true, or
- 6) a negation-as-failure node of the form $\text{not}(b)$, or
- 7) an alternative form $\text{alt}(b)$.

If P -tree T is $\{\}$ -full then we simply say that T is *full*.

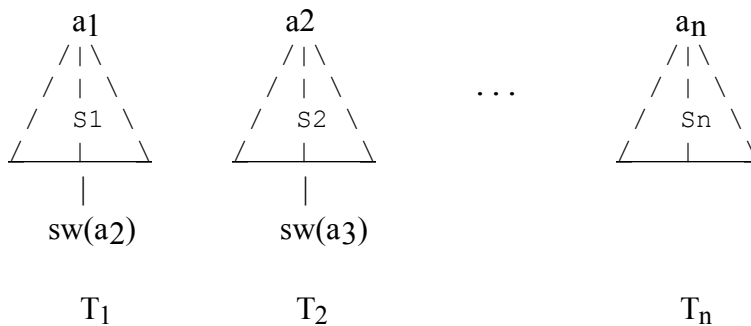
Pictorially, we can portray such a tree as follows (all leaves having one of these forms)



An *ancestor trail* is a sequence a_0, a_1, \dots, a_n of nodes in P-trees such that a_{i+1} is either a positive node which is a child of a_i or else $a_{i+1} = b$ where 'not(b)' or 'alt(b)' is a child of a_i . Note that ancestor trails can wind through several trees. Trails can leave a particular tree at a negative leaf.

For the Working Example. Consider the ancestor trail $\{d(a), p(a), \sim q(a), \sim d(a)\}$ which winds from T_1 through T_2 . T_1 is full, and T_2 is $\{d(a), p(a)\}$ -full. Similarly, the trail $\{d(a), p(a), s(a), p(a)\}$ leads through T_1 and into T_3 . T_3 is $\{d(a), p(a)\}$ -full. One *could*, of course, extend T_3 some more, to produce a full P-tree. This is a general phenomenon, characterized in the following proposition.

Proposition 1. *Suppose that 'sw' refers to either of the predicates 'not' or 'alt'. Suppose that T_1, \dots, T_n is a sequence of P-trees rooted at a_1, \dots, a_n , respectively, that $sw(a_{i+1})$ is a leaf of T_i , $i = 1, \dots, n$, that S_i is the ancestor trail in T_i of $sw(a_{i+1})$, and that T_{i+1} is an $(S_1 \cup \dots \cup S_i)$ -full P-tree rooted at a_{i+1} , $i = 1, \dots, n-1$. (In particular, T_1 is a full P-tree.) Pictorially, we would have*



Then, some extension of T_n is a full P-tree rooted at a_n .

We have used 'sw' to represent either of 'not' or 'alt', suggesting that 'sw' means "switch to another tree". In the conclusion of this proposition the term *extension* refers to a P-tree formed using clauses of P in the natural way to create a bigger tree containing the given tree as a top portion (the extended branches being below some of the leaves of the given tree).

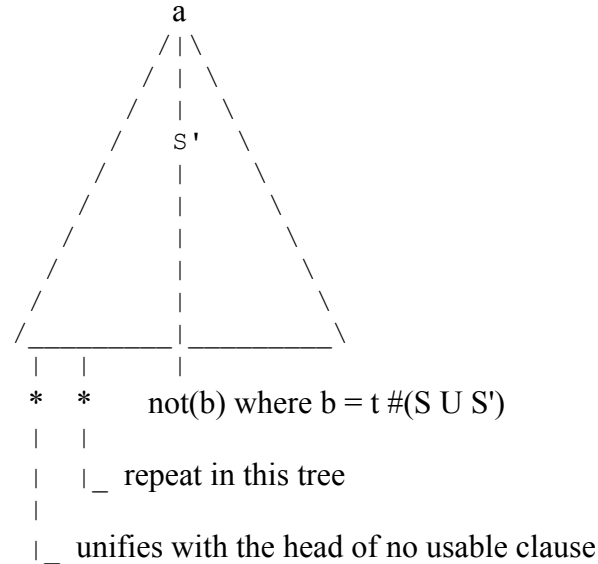
S-full P-trees, where S is a trail of ancestors, are important because of their use in forcing a termination to the computations of truth values. Here are the technical definitions.

4. Tree-based Semantics

Define a mathematical relation R on the set

- (i) a literal $b \in B_P \cup \sim B_P$ which does not unify with the head of any usable clause of P,
- (ii) a literal $c \in B_P \cup \sim B_P$ which has itself as an ancestor in T
- (iii) $\text{not}(b)$ where $b = t \# (S \cup S')$, and S' is the set of positive literals which are ancestors of the leaf $\text{not}(b)$ in T.

Pictorially, every S-full P-tree rooted at a must have at least one leaf of the following forms:



Note that a leaf of the form 'alt(-)' never can contribute to failure (f truth value) of the root of the tree. Alt-leaves can contribute to "truth" by allowing resolution with an ancestor, but otherwise their appearance contributes to "indeterminacy".

For the Working Example. We have

- (a) $\sim q(a) = t \# \{d(a), p(a)\}$ is established using T_2 ,
- (b) $\sim p(b) = t \# \{d(a), p(a)\}$ is established using T_5 ,
- (c) $s(a)$ is neither t nor $f \# \{d(a), p(a)\}$ (neither definition part 1 nor 2 applies to $s(a)$),
- (d) $s(b) = f \# \{\}$ since $s(b)$ unifies with the head of no usable clause,
- (e) $d(a)$ is neither t nor $f \# \{\}$, because of (c),
- (f) $d(b) = t \# \{\}$, using (b) and (d) in T_4 .

Note that we did not explicitly examine all P-trees rooted at $d(a)$ in order to confirm (e). We leave this to the reader.

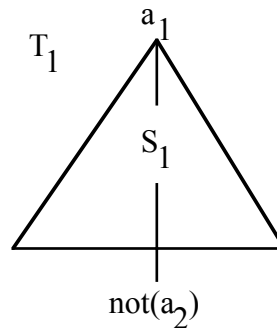
The *bounded trail property* (BTP) states that every ancestor trail (if sufficiently extended) through a forest of P-trees eventually stops at a node having no descendants or else the trail eventually repeats an element previously encountered on the trail. Note that the working example satisfies the BTP.

Proposition 2. *Disjunctive programs without function symbols satisfy the bounded trail property.*

Two example programs which do not have the BTP are $P_1 = \{p(x) \leftarrow p(f(x))\}$ and $P_2 = \{p(x) \leftarrow \text{not}(p(f(x)))\}$. Many programs with lots of function symbols do have the BTP. We believe that the BTP is interesting (and convenient for characterizations) but make no claims here regarding its *general* decidability, since it requires a suspicious sounding "halting" requirement.

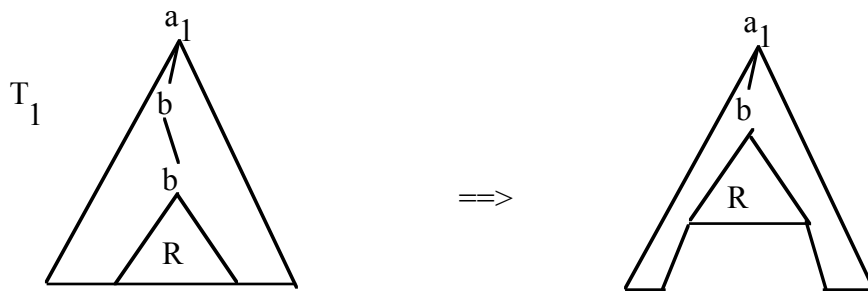
Proposition 3. *If P has the bounded trail property, then for every literal a , at least one of $a = t \# \{\}$ or $a = f \# \{\}$ does not hold.*

Proof. Suppose, to the contrary, that $a_1 = t \# \{\}$ and $a_1 = f \# \{\}$. Consider a full P -tree T_1 corresponding to $a_1 = t \# \{\}$. T_1 must have at least one negation-as-failure (NAF) node $\text{not}(a_2)$, since otherwise $a_1 = f \# \{\}$ could not hold.



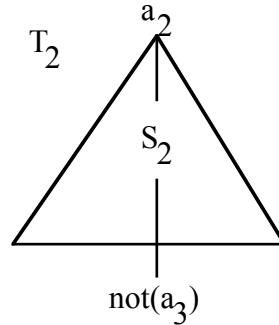
It must be the case that $a_2 = t \# S_1$ and $a_2 = f \# S_1$, where S_1 is the ancestor trail leading from root a_1 down to the leaf $\text{not}(a_2)$.

If trail S_1 had a repeated element b , then T_1 could be consolidated by moving the subtree under the repeated node b up and attach it to the previous occurrence of b .

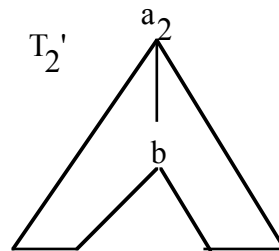


All such repeats along trail S_1 could be eliminated in this way. (The "set" S_1 remains the same.) So we can assume that trail S_1 has no repeated elements.

Since $a_2 = t \# \{\}$, there must be an S_1 -full P-tree T_2 rooted at a_2 which also has at least one NAF node $\text{not}(a_3)$ such that $a_3 = t \# (S_1 \cup S_2)$ and $a_3 = f \# (S_1 \cup S_2)$.

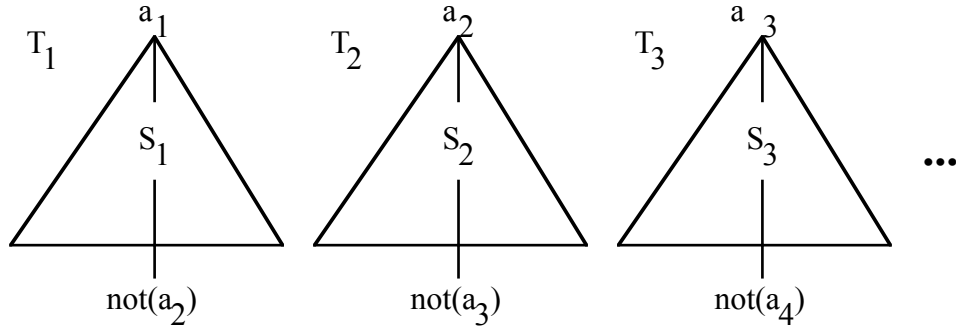


As for S_1 , we can assume that the trail S_2 has no repeated nodes. Moreover, if b belongs to both S_2 and S_1 , then consider the tree T_2' obtained by truncating T_2 below b .



T_2' is S_1 -full, because $b \in S_1$. Since $a_2 = f \# S_1$, T_2' must still have a "false" leaf of the form $\text{not}(a_3')$. This leaf is also a leaf of T_2 so $a_3 = f \# S_2'$ where S_2' is the ancestor trail in T_2 (and in T_2') leading from root a_2 to the leaf $\text{not}(a_3')$. Since $\text{not}(a_3')$ is a "false" leaf of T_2' , $a_3' = t \# S_2'$ also. If S_2' does not repeat any element of S_1 , then use a_3' and S_2' instead of a_3 and S_2 , respectively. Otherwise T_2' could be truncated further, forcing consideration of another NAF leaf $\text{not}(a_3'')$. This could not continue indefinitely since there are only finitely many NAF leaves in T_2 . Thus, one could assume that S_2 and S_1 are disjoint.

Continuing this process generates successive trees



such that $a_k = t \# S_{k-1}$ and $a_k = f \# S_{k-1}$ (with $S_0 = \phi$), trail S_k contains distinct elements, and no element of S_k belongs to $S_1 \cup \dots \cup S_{k-1}$. Thus $S_1 \cup S_2 \cup S_3 \cup \dots$ is generating an unbounded trail, contrary to the bounded trail property. Thus $a_1 = t \# \{\}$ and $a_1 = f \# \{\}$ is not possible. q_e

Thus, for programs with the bounded trail property, we may finish the truth-value definition, as follows.

Definition 2(part 3). $a = u \# \{\}$ means that neither $a = t \# \{\}$ nor $a = f \# \{\}$ holds.

For the Working Example. We can now say that $d(a) = p(a) = u \# \{\}$; i.e., that $d(a)$ and $p(a)$ are undetermined by the program. Of particular interest are the truth values determined for $k(a)$, $k(b)$ and $k(c)$. Recall that the definition of k in P was

$$k(x) \leftarrow \text{not}(d(x))$$

We have

$$\begin{aligned} k(a) &= f \# \{\} \text{ since } d(a) = t \# \{\} \\ k(b) &= u \# \{\} \text{ since } d(b) = u \# \{\} \\ k(c) &= t \# \{\} \text{ since } d(c) = f \# \{\} \end{aligned}$$

Now, of course, the definitions of truth value based on trees must be used with care. For example, in the program

$$\begin{aligned} a &\leftarrow \text{not}(b) \\ a &\leftarrow \text{not}(a) \\ a &\leftarrow c \end{aligned}$$

we have that $a = t \# \{\}$ based upon the first clause (or corresponding tree), whereas if the first clause were ignored, then we would have had $a = u \# \{\}$, and we would have had $a = f \# \{\}$ if only the last clause were available. As for well founded semantics, t supersedes

u, which in turn supersedes f; that is, $t > u > f$. For the tree-based semantics, this is a consequence of the three parts of the definition for truth values. A rough characterization of this would be: a literal is true if at least one tree supports with all "truthful" leaves, or the literal is false if all trees trying to support have at least one "failing" leaf, otherwise the literal is indeterminate.

An example can be used to motivate the use of 'alt' literals in the alternative clauses. Consider the program

```
a, b <-
c <- not(a).
```

Now we have $a = u \# \{\}$. To emphasize why this is the case, consider that

```
  a
  |
alt(~b)
```

is the only full P-tree rooted at a, and clearly $\sim b = f \# \{\}$, but this last fact does not "falsify" the alt($\sim b$) leaf, as previously noted. Thus $c = u \# \{\}$.

A bottom-up characterization for the semantics corresponding to the semantic trees specification can be given as follows. Let us assume that the program P itself is already grounded, and let us also assume that for any ground positive literal L, L only occurs (as one of the disjuncts) in the head of finitely many clauses of P. A sequence of programs P_i and sequences of truth sets T_i , false sets F_i , and undetermined sets U_i are define by induction.

```
P0 = P
T0 = the set of heads of body-less clauses of P0. These can be disjuncts.
F0 = the set of literals occurring in the head of no clause of P0.
U0 = BP \ (T0 U F0).
```

Now, assumming that P_i , T_i , F_i , and U_i have been defined for $i < k$,

```
Pk is obtained from Pk-1 by modifying or deleting clauses of Pk-1:
Erase body literals L from clauses of Pk-1 when L ∈ Tk-1.
Erase body literals not(L) from clauses of Pk-1 when L ∈ Fk-1.
Erase a clause of Pk-1 when the clause has a body literal not(L) and L ∈ Tk-1.
```

(Erase clauses $D \leftarrow \dots$ where $D \in T_{k-1}$.)

$$\begin{aligned}
 T_k &= T_{k-1} \cup \{\text{heads of body-less clauses of } P_k\} \\
 &\quad \cup \{\text{stretch and factor disjuncts in } T_{k-1} \text{ using clauses from } P_k\}. \\
 F_k &= F_{k-1} \cup \{\text{positive literals of } U_{k-1} \text{ now occurring in the head of no clause of } P_k\}. \\
 U_k &= B_P \setminus (T_k \cup F_k).
 \end{aligned}$$

The definition of the T_k is suggested by the approach of Rajasekar, et. al. (1989), where disjuncts or states are used. *Stretching* and *factoring* can be understood using an example. Suppose that disjunct 'a v b' is in T_{k-1} and that clauses 'c v d \leftarrow b' and 'c \leftarrow a' are in P_k . Then a v b can be stretched using the two clauses, obtaining 'c v c v d', and then factoring produces 'c v d' in T_k . These operations correspond to clausal resolution on the body literals of the clauses ('a' and 'b' in the example), followed by the elimination of repeated factors produced in the resolvent; this is a traditional theorem-proving technique. Stretching can only be performed using clauses with a single positive body literal (but the corresponding head may be disjunctive).

Finally, let the *net* truth, false, and undefined sets be given as follows:

$$\begin{aligned}
 T &= \bigcup T_k \quad k = 1.. \infty \\
 F &= \bigcup F_k \quad k = 1 .. \infty \\
 U &= B_P \setminus (T \cup F)
 \end{aligned}$$

For the Working Example. Consider ground instances of the program clauses.

$$\begin{aligned}
 p(a), q(a) &\leftarrow r(a), \text{not}(s(a)) \\
 p(b), q(b) &\leftarrow r(b), \text{not}(s(b)) \\
 p(c), q(c) &\leftarrow r(c), \text{not}(s(c))
 \end{aligned}$$

$$s(a) \leftarrow p(a)$$

$$r(a) \leftarrow \quad r(b) \leftarrow$$

$$\begin{aligned}
 d(a) &\leftarrow p(a), w(a) \\
 d(b) &\leftarrow p(b), w(b) \\
 d(c) &\leftarrow p(c), w(c)
 \end{aligned}$$

$$\begin{aligned}
 d(a) &\leftarrow q(a), w(a) \\
 d(b) &\leftarrow q(b), w(b) \\
 d(c) &\leftarrow q(c), w(c)
 \end{aligned}$$

$$w(a) \leftarrow \quad w(b) \leftarrow \quad v(a) \leftarrow \quad v(b) \leftarrow \quad v(c) \leftarrow$$

k(a) <- not(d(a))
k(b) <- not(d(b))
k(c) <- not(d(c))

Then

$T_0 = \{r(a), r(b), w(a), w(b), v(a), v(b), v(c)\}$
 $F_0 = \{s(b), s(c), r(c), w(c)\}$

P₁:

p(a), q(a) <- not(s(a))
p(b), q(b) <-

s(a) <- p(a)

d(a) <- p(a)
d(b) <- p(b)

d(a) <- q(a)
d(b) <- q(b)

k(a) <- not(d(a))
k(b) <- not(d(b))
k(c) <- not(d(c))

$T_1 = T_0 \cup \{p(b) \vee q(b)\}$
 $F_1 = F_0 \cup \{p(c), q(c), d(c)\}$

P₂:

p(a), q(a) <- not(s(a))

s(a) <- p(a)

d(a) <- p(a)
d(b) <- p(b)

d(a) <- q(a)
d(b) <- q(b)

k(a) <- not(d(a))
k(b) <- not(d(b))
k(c) <-

$T_2 = T_1 \cup \{d(b), k(c)\}$ Stretch and factor $p(b) \vee q(b)$
 $F_2 = F_1$

P₃:

$p(a), q(a) \leftarrow \text{not}(s(a))$

$s(a) \leftarrow p(a)$

$d(a) \leftarrow p(a)$

$d(a) \leftarrow q(a)$

$k(a) \leftarrow \text{not}(d(a))$

$T = T_2 = \{r(a), r(b), w(a), w(b), v(a), v(b), v(c), p(b) \vee q(b), d(b), k(c)\}$

$F = F_2 = \{s(b), s(c), r(c), w(c), p(c), q(c), d(c)\}$

$U = \{p(a), q(a), s(a), d(a), k(a)\} \quad q$

For the working example, the bottom-up characterization of semantics and the tree-based specification give the same truth values to positive literals. We conjecture that this is true more generally:

Conjecture. *If B_P is finite then the bottom-up characterization of semantics gives the same truth values as the tree-based specification. That is (restricting T to positive literals rather than disjuncts),*

$T = \text{true positive literals} = \{a \in B_P \mid a = t \# \{\}\}$

$U = \text{undefined literals} = \{a \in B_P \mid a = u \# \{\}\}$

$F = \text{false positive literals} = \{a \in B_P \mid a = f \# \{\}\}$

For nondisjunctive programs we have the following proposition, proved in Fisher (1993a).

Proposition 4. *For nondisjunctive logic programs with negation as failure, if B_P is finite, then the tree semantics is the same as well-founded semantics characterized using the bottom-up definition.*

A stronger version of this bottom-up characterization would interpret disjunction exclusively (if it could): If positive literal A has been added to T_k , if each of A and $B_1, B_2, \dots, B_n \in U_{k-1}$ and disjunct $D = A \vee B_1 \vee B_2 \vee \dots \vee B_n \in T_{k-1}$ then remove D from T_k

and add each of B_1, B_2, \dots, B_n to F_k . In addition, one must insist that F_k be purged of positive literals that appear in T_k (literals which used to be false because of exclusive disjunction, but now have become true because they are separately supported). The stronger approach would be in adherence to the *generalized closed world assumption* (GCWA). The version presented above corresponds more to the *weak generalized closed world assumption* (WGCA) suggested for positive programs in Rajasekar, et. al. (1989). The GCWA approach forces more "disjunctive literals" to be *false* because disjunction is being interpreted exclusively.

For example, consider the logic program

```
a, b <-
b <-
c <- not(a).
```

Both the tree-based semantics and the bottom-up characterization conclude that $a = u \# \{\}$, whereas a semantics using the GCWA would insist that "a is false".

In (Fisher, 1989), the ancestor resolution rule was given a procedural characterization and used as an enhancement for SLD resolution, resulting in what was called SLD/AER deduction (SLD + *ancestor erasure rule*). In (Fisher, 1992), formal negation was called *explicit negation*, following the ideas presented in (Fisher, 1988). The set-theoretical semantics considered in (Fisher, 1989) used sets of disjuncts, or states, as presented and characterized in (Rajasekar, et. al., 1989). The characterizations in (Fisher, 1989) of soundness and completeness for procedural SLD/AER were for disjunctive programs *without* negation as failure (so-called positive programs). The tree-based semantics presented in this paper provides a generalization of the previous concepts to disjunctive programs *with negation as failure*, using an extension of well founded semantics.

We are not here claiming to have *the correct approach* to semantics for disjunctive programs. Rather, our purpose is to explain the top-down, semantic tree specification approach. We cannot offer a tree-based specification that would correspond to the stronger version imposing the GWCA. An excellent discussion of semantics issues for disjunctive logic programs is in the paper by Apt and Bol (1994).

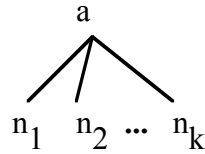
In the truth value definition, negation-as-failure nodes and alternative nodes were not allowed to be the roots of P-trees, and no truth value was independently ascribed to 'not(...)' nor to 'alt(...)' literals. Informally, we do so as follows:

```
not(b) = t # S if b = f # S
not(b) = f # S if b = t # S
not(b) = u # S if b = u # S
```

$$\begin{aligned} \text{alt}(\sim b) &= t \# S \text{ if } \sim b = t \# S \\ \text{alt}(\sim b) &= u \# S \text{ if } \sim b = f \text{ or } u \# S \end{aligned}$$

Using this informal notation, we have

Proposition 5 (Tabulation). *Suppose that*



is a P-tree branching based upon a usable clause of P. Suppose that $a = v \# \{\}$, and that $n_i = v_i \# \{a\}$ for $i = 1, \dots, k$, where $v, v_i \in \{t, u, f\}$. If this is the only P-tree rooted at a then

$$a = \min\{v_i \mid i = 1, \dots, k\} \# \{\}$$

where the ordering is the usual $t > u > f$. On the other hand, if T_1, \dots, T_m are all of the P-trees rooted at a, and if $a = v_j \# \{\}$ when only the subprogram growing T_j is considered, then, for a net result

$$a = \max\{v_j \mid j = 1, \dots, m\} \# \{\}.$$

This shows that tree semantics (and well founded semantics, insofar as the two coincide) is a sort of "maxi-min computation". Using a metaphor of deliberation, one seeks the strongest overall argument, where each individual argument is only supported by (or is as strong as) its weakest evidence.

5. Useful Clauses

Usable clauses were for an indefinite logic program were characterized in the first section. It is probably apparent that not all of the usable clauses would actually be needed to grow P-trees in order to determine truth values. The following proposition shows that formally negative literals can never sustain a 't' truth value on their own.

Proposition 6. *Suppose that P is a disjunctive logic program with no formally negative literals in any clause. Then, for any formally negative ground literal $\sim a \in \sim B_P$ we have*

$$\sim a = f \# \{\}$$

The proposition may seem surprising at first, but recall that $\sim a = t \# S$ has only occurred in the examples only when S contained sufficient ancestors for ancestor resolution.

Definition 3. Suppose that P is a disjunctive logic program and that H is a positive literal (which can contain variables). H is said to be an *indefinite literal* (with respect to P) provided either that h unifies with some literal in the head of some indefinite clause of P , or else there is some definite clause $A \leftarrow B_1, \dots, B_r, \text{not}(C_1), \dots, \text{not}(C_s)$ of P such that H and A have most general unifier σ and for some $j=1, \dots, r$, $\sigma(B_j)$ is an indefinite literal.

In the Working Example of section 1, the literals $p(x)$, $q(x)$, $s(a)$, $d(x)$ are all indefinite literals (as would be any variants or instances of any of these literals).

Definition 4. Suppose that, as before,

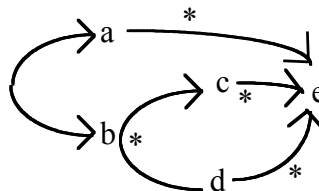
$$\sim B \leftarrow \alpha, \sim A, \beta$$

is a backlink clause of P . This backlink is said to be *potentially useful* provided that the positive literal B is an indefinite literal.

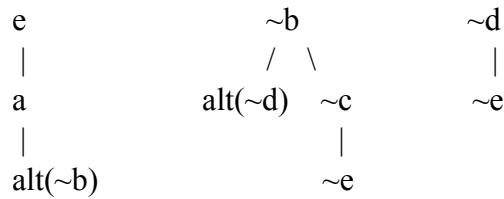
Here is another small example, showing how indefinite information depending upon other indefinite information can be accessed through usable clauses.

$a, b \leftarrow$
 $c, d \leftarrow b$
 $e \leftarrow a$
 $e \leftarrow c$
 $e \leftarrow d$

The following diagram shows the program's logical dependencies graphically



We have marked the clauses that generate potentially useful backlinks with an asterisk (*); for example, for the clause $e \leftarrow a$, the actual backlink would be $\sim a \leftarrow \sim e$. Note that these potentially useful backlinks "link back from indefinite literals". Let the reader write down all of the usable clauses for this indefinite program. Here are some P-trees that establish $e = t \# \{\}$:



Note that the middle tree shows a use of a backlink of a primary alternative clause. The last tree shows $\sim d = t \# \{e, a, \sim b\}$, the next to last shows that $\sim b = t \# \{e, a\}$, and the first that $e = t \# \{\}$. Note how the backlinks allow reasoning "around the arrow diagram".

Proposition 7. *Suppose that P is a disjunctive logic program, that $a \in B_P$, that a is an indefinite literal of P , and that $a = t \# \{\}$, using a supporting forest of program trees F . Then any backlink clause actually used to grow a branch of some tree in F must be a potentially useful backlink.*

Let the reader provide an example where a potentially useful backlink cannot actually be used to support a truth value of t for any literal. This explains the "potential" in the terminology "potentially useful".

For an indefinite logic program, the ratio of potentially useful to usable backlinks is called the *backlink utility ratio*

$$\frac{\text{Number of potentially useful backlinks}}{\text{Number of usable backlinks}}$$

For the Working Example of section 1, this ratio is $2/6=1/3$. It should be possible to establish some mathematical relationships for this ratio in terms of parameters which measure the number of disjuncts in heads of clauses, the occurrence of indefinite literals in the bodies of clauses, etc.

6. Procedural Specification

A meta-interpreter corresponding to the tree-based specification of semantics should have the following features (many of which are like the meta-interpreter M in Section 1):

- 1) grow P -trees from roots, using usable clauses, keeping track of ancestors, and partial truth-value computations (see 7),
- 2) detect 'true' termination and treat it as success (t) on the trail currently being computed,

- 3) detect positive recursion (repeat within current tree), and treat it as failure (f) on the current trail,
- 4) detect recursion through negation (repeat from previous tree), treating it as indeterminacy (u) on the current trail,
- 5) follow the semantic rules for determining truth values whenever primary alternative clauses are used,
- 6) examine all trees that can grow from a selected root, and
- 7) stay *deep* in P-trees so that net truth values are computed for negations, and alternatives. This requires the *tabulation* of partial results until a net truth value is determined, as characterized in Proposition 5 above.

A procedural specification in the form of a Prolog meta-interpreter is provided for definite logic programs in (Fisher, 1993a), and the author has extended this program for disjunctive logic programs in order to compute truth values as specified in this paper. The Prolog meta-interpreter computes answers for goals containing variables.

7. Formally Negative Programs

Note that in the previous sections, all disjunctive logic programs were assumed to be free of formal negation when specified. Formally negative literals arose only from the primary alternative clauses and the backlink clauses. In this section we consider logic programs having formally negative literals in their original specification. Such programs will be said to be *formally negative programs*. For example,

```

~a <- b
c <- ~a
c <- ~b

```

One would like tree-based semantics to support $c = t \# \{\}$. Using the previous (three part) definitions for truth values based upon trees, *but extending the definition of a backlink*, we could make the following program tree supporting truth for c:

```

  c
  |
  ~a
  |
  b
  |   This is a "new" backlink generated by c <- ~b
  ~c

```

Note that the leaf provides the opportunity for ancestor resolution (as before). The backlink actually used above is formed according to the previous conventions provided that we use a typical rule for double (formal) negation of literals:

$$\sim \sim a = a$$

We propose to add this double-negation reduction rule to the process that generates backlinks. So, for example, $c \leftarrow \sim b$ would generate $\sim \sim b \leftarrow \sim c$, which would become $b \leftarrow \sim c$. Note that the concept of potentially useful backlink, previously formulated only for disjunctive programs specified with positive literals, would need modification for formally negative programs!

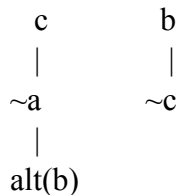
Consider a modification of the previous formally negative program:

```

~a, ~b <-
c <- ~a
c <- ~b

```

For this version, one would need to consider the following program trees, to establish $c = t \# \{ \}$.



Formal negation affords a representation for exclusive disjunction. For example, in

```

a, b <-
~a <- b
~b <- a
c <- ~a
c <- ~b

```

the first three clauses are intended to represent an exclusive disjunction (forced exclusion). The reader is invited to draw semantic trees that would justify the conclusion $c = t \# \{ \}$.

An important consideration for formally negative programs is the crucial issue of *formal consistency*. For example, in the formally negative program

```
a ← b, not(~c)
b ←
~a ←
```

we would have $a = t \# \{\}$, and $\sim a = t \# \{\}$. There are local arguments, or support, for a and for its formal negation $\sim a$, so there is conflict in this knowledge base (formally negative program).

Formally negative programs call for the specification of some kind of *conflict semantics*. Possible approaches include the following:

*) Allow only consistent programs. For example, only allow formally negative literals in the heads of clauses if the corresponding positive occurrences in the heads of clauses have guards in the bodies of those clauses, as in the following program.

```
a ← b, not(~a)
~a ← c
b ←
c ←
```

*) Allow inconsistencies, but find a way to "localize" them. This could be (understandably) difficult to characterize in a general way. For example, in the program

```
c ← a
c ← b
a, b ←
~a ←
~b ←
```

$c = t \# \{\}$, but the apparent support of c , which is the clause $a, b \leftarrow$ is formally inconsistent with the last two clauses of the program. It will be a very demanding task to characterize a usable concept of a literal being *consistently supported*.

We leave these issues to a later paper. See (Fisher, 1988) and (Fisher, 1989, Appendix D) for some related work.

References

- Apt, K.R., and R. Bol (1994). Logic programming and negation: a survey, Preprint.
- Bol, R.N. (1991). *Loop Checking in Logic Programming*, Thesis, Centre for Mathematics and Computer Science, Amsterdam, 1991.
- Bruffaerts, A., and Henin, E. (1988). Proof trees for negation-as-failure: yet another Prolog meta-interpreter. *Proceedings of the Workshop on Meta-Programming in Logic Programming*, University of Bristol, Bristol, England, June, 133-146.
- Clark, K. (1978). Negation as Failure. In H. Gallaire and J. Minker, eds., *Logic and Databases*, Plenum Press, 293-322.
- M.R. Genesereth, and N.J. Nilsson (1987), *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann.
- Fisher, J.R. (1988). Logic programs with explicit negation, *Tech. Report #1988-01*, Computer Science Department, California State Polytechnic University.
- Fisher, J.R. (1992). Tree specification of semantics for logic programs with negation as failure, *Proc. Third Annual California State University Artificial Intelligence Symposium*, California State University, San Luis Obispo, June 1992, 158-66.
- Fisher, J.R. (1989), *GPL Notes, Generalized Prolog*, *Tech. Report #1989-01*, California State Polytechnic University, Pomona.
- Fisher, J.R. (1993a), Top-down tree specification of semantics for logic programs with negation as failure, *Tech. Report #1993-06*, Computer Science Department, California State Polytechnic University, Pomona.
- Fisher, J.R. (1993b), Bounded logic programs with negation as failure, *Tech. Report #1993-16*, Computer Science Department, California State Polytechnic University, Pomona.
- Lloyd, J.W. (1987), *Foundations of Logic Programming*, Springer-Verlag, 2nd ed. 1984.
- Pereira, L.M., J. Alferes, and J.N. Aparacio, Top-down procedures for well-founded semantics, *Technical Report*, AI Centre, Uninova, Oct. 1990.
- Przymusiński, T.C., On the declarative and procedural semantics of logic programs, *J. Automated Reasoning*, 5, 1989, 167-205.

Rajasekar, A., J. Lobo, and J. Minker (1989). Weak generalized closed world assumption, *J. Automated Reasoning*, Vol. 5, No. 3, September 1989, pp. 293-307.

Van Gelder, A., A. Ross, and J.S. Schlipf (1991). The well-founded semantics for general logic programs. *J. ACM*, Vol. 38, No. 3, July 1991, pp. 620-650.