Logic Programs Using Semantic Trees

IONELA MANIU, DANIEL HUNYADI
Computer Science Department,
Lucian Blaga University, Sibiu

ROMANIA

GEORGE MANIU
Mannagement department
Spiru Haret University, Brasov

ROMANIA

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.*

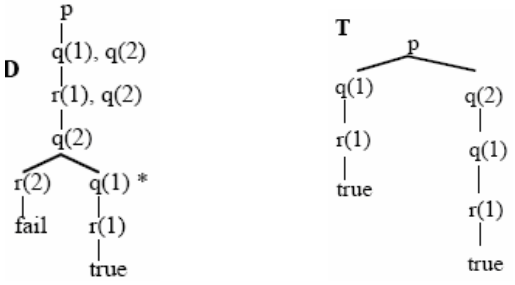
Keywords: *Logic programming, Negation as failure, Formal negation, Program trees, Semantic trees, Bounded trail property.*

1. Motivation and Background

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 <- p, the following *derivation tree* D, pictured below on the left, using the clauses of P:



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

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 branching 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.

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 Ṃ directly implements the tree-based "semantics" defined above (which is equivalent to the standard least model semantics). 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).

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. [2]

The class of disjunctive programs with negation as failure considered here contain disjunctive clauses of the form:

$$A_1,A_2,...,A_k \text{ <- } B_1,...,B_m,\text{not}(C_1),...,\text{not}(C_n)$$

where each A_i, each B_i, and each C_i is a positive literal (possibly containing variables), k >= 1, m,n >= 0. The sequence of literals $A_1,A_2,...,A_k$ constitutes the *head* of the clause and this sequence is a *disjunction*. The sequence of literals $B_1,...,B_m,\text{not}(C_1),...,\text{not}(C_n)$ 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. This specification uses a (finite) tree data structure to determine (or support) meanings. The clauses of the program, together with well specified contra positive 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'.

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.

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.

The paper provides the formal definitions for the tree-based specifications and characterizes the basic propositions regarding its properties. In particular, for nondisjunctive logic programs with negation as failure, the tree-based semantics is provably equivalent to the well-founded semantics if B_P 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" , or sometimes as "proof trees".

2. Disjunctive Logic Programs

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

$$A1,A2,...,Ak <- B_1,...,B_m,not(C_1),...,not(C_n)$$

where Ai, each Bi, and each Ci is a positive literal, k >= 1, m,n >= 0. The sequence of literals A1,A2,...,Ak constitutes the *head* of the clause and this sequence is a *disjunction*. The sequence of literals $B_1,...,B_m,not(C_1),...,not(C_n)$ is a *conjunction* and is the *body* of the clause. The sequence $B_1,...,B_m$ is the *positive* part of the body and the sequence $not(C_1),...,not(C_n)$ 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:

$$A1,A2,...,Ak <- B_1,...,B_m,not(C_1),...,not(C_n)$$

has the form:

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

where 1<= j <= k. There are k primary alternatives if k>=2. If k=1 then the clause is definite and does not have any primary alternatives. The '~' 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 '~' is formal negation.

We will need to maintain a careful distinction between these two negations. For a primary alternative the sequence $alt(\sim A_1)$, ...,$alt(\sim A_{j-1}),alt(\sim A_{j+1}),...,alt(\sim A_k)$ 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.* They are formed as follows. Suppose that:

$$A <- \underline{\alpha}B, \beta$$

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, βare (possibly empty) sequences of the other literals of the body. Then:

$$\sim B <- \underline{\alpha} \sim A, \beta$$

is a backlink clause, where '~' is formal negation.

Working Example.

Consider the indefinite program P (X is a variable):

```
p(X), q(X) <- r(X), not(s(X))
s(a) <- p(a)
r(a) <- r(b) <-
d(X) <- p(X), w(X)
d(X) <- q(X), v(X)
w(a) <- w(b) <- v(a) <- v(b) <- v(c) <-
k(X) <- not(d(X))
```

The primary alternatives of the indefinite clause are:

```
p(X) <- alt(~q(X)), r(X), not(s(X))
```

q(X) <- alt(~p(X)), r(X), not(s(X))

Here are all of the possible backlinks:

~r(X) <- alt(~q(X)), ~p(X), not(s(X))
~r(X) <- alt(~q(X)), ~q(X), not(s(X))
~p(X) <- ~d(X), w(X)
~w(X) <- p(X), ~d(X)
~q(X) <- ~d(X), v(X)
~v(X) <- q(X), ~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

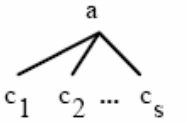
P-trees are constructed using the usable clauses of P. Let B_P be the set of ground positive literals of P, and let

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

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

$$a \leftarrow c_1,...,c_S$$

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 <- is a ground instance of a unit clause of P, then we write the corresponding branching P-tree node as:

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 ~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 not(b),or
7) an alternative form alt(b).

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

An *ancestor trail* is a sequence $a_0,a_1,...,a_n$ 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.

4. Tree-based Semantics

Define a mathematical relation R on the set:

$$(B_P \cup \sim B_P) \times \{t,f,u\} \times 2^{(B_P \cup \sim B_P)}$$

where t, f, u stand for 'true', 'false', 'undetermined', respectively. We will use the notation "...=...#..." to describe this relation. That is, write a = v # S provided that (a,v,S) is in R, where a is a ground atom in $B_P \cup \sim B_P$, v is a truth value in {t,f,u} and S is a subset of $B_P \cup \sim B_P$. The definition is recursive. Define *a = t # S* to mean that there is some S-full P-tree T rooted at a such that every leaf node of T is either:

(i) true,
(ii) of the form ~b where b $\in$ S,
(iii) of the form not(c), and c = f # (S U S') where S' is the set of positive literals
which are ancestors of not(c) in T, or
(iv) of the form alt(~d), where ~d = t # (S U S') where S' is the set of positive
literals which are ancestors of alt(~d) in T.

Condition (ii) is called the *ancestor resolution* rule.

Define *a = f # S* to mean that every S-full P-tree T rooted at a has at least one leaf which has one of the following forms:

(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) not(b) where b = t # (S U S'), and S' is the set of positive literals which are
ancestors of the leaf not(b) in T.

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".

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.

Disjunctive programs without function symbols satisfy the bounded trail property.

Two example programs which do not have the BTP are P1 = {p(x) <- p(f(x))} and P2 = {p(x) <- not(p(f(x))}. Many programs with lots of function symbols do have the BTP.

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

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

Define *a = u # {}* means that neither a = t # {} nor a = f # {} holds.

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

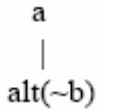
a <- not(b)
a <- not(a)
a <- c

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:



is the only full P-tree rooted at a, and clearly ~b = f # {}, but this last fact does not "falsify" the alt(~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.

$P_0 = P$

T_0 = the set of heads of body-less clauses of P_0. These can be disjuncts.

F_0 = the set of literals occurring in the head of no clause of P_0.

$U_0 = B_P \setminus (T_0 \cup F_0)$.

Now, assumming that P_i, T_i, F_i, and U_i have been defined for $i < k$, P_k is obtained from P_{k-1} by modifying or deleting clauses of P_{k-1}:

Erase body literals L from clauses of P_{k-1} when $L \in T_{k-1}$. Erase body literals not(L) from clauses of P_{k-1} when $L \in F_{k-1}$. Erase a clause of P_{k-1} when the clause has a body literal not(L) and $L \in T_{k-1}$. (Erase clauses D <- ... where $D \in T_{k-1}$.)

$T_k = T_{k-1} \cup$ {heads of body-less clauses of P_k} $\cup$ {stretch and factor disjuncts in T_{k-1} using clauses from P_k}.

$F_k = F_{k-1} \cup$ {positive literals of U_{k-1} now occurring in the head of no clause of P_k}.

$U_k = B_P \setminus (T_k \cup F_k)$.

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 <- b' and 'c <- 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:

$T = \cup T_k$, $F = \cup F_k$ unde $k = 1 .. \infty$

$U = B_P \setminus (T \cup F)$

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

p(a), q(a) <- r(a), not(s(a))
p(b), q(b) <- r(b), not(s(b))
p(c), q(c) <- r(c), not(s(c))
s(a) <- p(a)
r(a) <- r(b) <-
d(a) <- p(a), w(a)
d(b) <- p(b), w(b)
d(c) <- p(c), w(c)
d(a) <- q(a), w(a)
d(b) <- q(b), w(b)
d(c) <- q(c), w(c)
w(a) <- w(b) <- v(a) <- v(b) <- v(c) <-
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)}
P_1:
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) v q(b)}
$F_1 = F_1 \cup$ {p(c), q(c), d(c)}
P_2:
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) v q(b)
$F_2 = F_1$
P_3:
p(a), q(a) <- not(s(a))
s(a) <- p(a)
d(a) <- p(a)
d(a) <- q(a)
k(a) <- not(d(a))
$T = T_2$ = {r(a), r(b), w(a), w(b), v(a), v(b), v(c), p(b) v 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)}

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

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 $B_1, B_2, ..., B_n \in U_{k-1}$ and disjunct $D = A \vee B_1 \vee B_2 \vee ... B_n \in T_{k-1}$ then remove D from T_k and add each of $B_1, B_2,...,B_n$ to F_k. In addition, one must insist that F_k be purged of positive literals that appear in T_k . The stronger approach would be in adherence to the *generalized closed world assumption* (GCWA).

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).

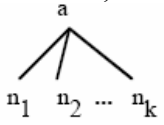
Both the tree-based semantics and the bottom-up characterization conclude that a = u # {}, whereas a semantics using the GCWA would insist that "a is false".

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. Our purpose is to explain the top-down, semantic tree specification approach.

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
alt(~b) = t # S if ~b = t # S
alt(~b) = u # S if ~b = f or u # S

Using this informal notation, we have, supposing 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,...,k, where v, $v_i \in$ {t,u,f}. If this is the only P-tree rooted at a then a = min{v_i | i = 1, ..., k} # {} where the ordering is the usual t > u > f. On the other hand, if T_1,..., 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 | j = 1,..., m}# {}.

This shows that tree semantics 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.

Suppose that P is a disjunctive logic program with no formally negative literals in any clause. Then, for any formally negative ground literal ~a $\in$ ~B_P we have ~a = f # {}

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

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 <-B_1,...,B_r,not(C_1),...,not(C_s) of P such that H and A have most general unifier σ and for some j=1,...,r, σ (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).

Suppose that, as before, ~B <- α, ~A, β is a backlink clause of P. This backlink is said to be *potentially useful* provided that the positive literal B is an indefinite literal.

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 apotentially useful backlink.

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. Conclusion.

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. An excellent discussion of semantics issues for disjunctive logic programs is in the paper by Apt and Bol (1994)[1].

References

[1] Apt, K.R., and R. Bol (1994). Logic programming and negation: a survey, Preprint.

[2] Clark, K. (1978). Negation as Failure. Logic and Databases, Plenum Press.

[3] M.R. Genesereth, and N.J. Nilsson (1999), Logical Foundations of Artificial Intelligence, State Polytechnic University, Pomona.

[4] Van Gelder, A., A. Ross, and J.S. Schlipf (2000). The well-founded semantics for general logic programs.