Top–down query evaluation for well–founded semantics with explicit negation

José Júlio Alferes Carlos Viegas Damásio Luís Moniz Pereira[1]
CRIA, Uninova and DCS, U. Nova de Lisboa
2825 Monte da Caparica, Portugal
{jja|cd|lmp}@fct.unl.pt

Abstract. In this paper we define a sound and complete top–down semantic tree characterization, that includes pruning rules, of the well–founded semantics for programs extended with explicit negation (*WFSX*), and compare it to other related approaches. It is amenable to a simple implementation, and by its nature readily allows pre-processing into Prolog, showing promise as an efficient basis for further development.

1 Introduction

For some time now, programming in logic has been shown to be a viable proposition. Although Horn clause programming augmented with the NOT operator (i.e. Prolog) under the SLDNF derivation procedure does allow negative conclusions, these are only drawn by default (or implicitly), just in case the corresponding positive conclusion is not forthcoming in a finite number of steps – hence the specific form of closed world assumption of the completion semantics given to such programs.

This form of negation is capable of dealing with incomplete information, by assuming false exactly what is not true in a finite manner. However, there remains the issue of non-terminating computations, even for finite programs. To deal with this and other problems of the completion semantics, a spate of semantic proposals were set forth from the late eigthies onwards including the well-founded semantics (WFS) of [12], which deals semantically with non-terminating computations, and thereby giving semantics to every program. For this semantics several query evaluation procedures have been defined [3, 5, 7, 14, 18, 20, 21, 23, 24].

In recent years several authors (e.g. [13, 19, 26]) have shown the importance of extending LP with a second kind of negation $\neg$, for use in deductive databases, knowledge representation, and non-monotonic reasoning (NMR) . Different semantics for extended LPs with $\neg$-negation (ELP) have appeared (e.g. [13, 17, 26]). The generalization for ELP of WFS defined in [17], *WFSX*, to which the present paper refers, provides an added qualitative representational expressivity that can capture a wide variety of logical reasoning forms, and serve as an instrument for programming them, as we've shown in [19].

Because of its properties, which other approaches do not fully enjoy, *WFSX* is a natural candidate to being the semantics of choice for ELPs. Namely, it exhibits the structural properties of: simplicity, cumulativity, rationality, relevance, partial evaluation, and polynomial complexity for Datalog. By simplicity we mean that it can be simply characterized by two iterative fixpoint operators, without recourse to three-valued logic. By cumulativity [9], we refer to the efficiency related ability of using lemmas, i.e. the addition of lemmas does not change the semantics of a program. By rationality [9], we refer to the ability to add the negation of a non-provable conclusion without changing the semantics, which is important for efficient default reasoning. By relevance [9], we mean that the top-down evaluation of a literal's truth-value requires only the call-graph below it. By partial evaluation [9] we mean that the semantics of a partially evaluated program keeps to that of the original.

To the best of our knowledge, the only existing top-down procedure for ELP is the one defined in [25]. These procedures implement a generalization of the Kunen semantics [15] with the notion of conservative derivability of [26]. However, as pointed out by the author, the implemented semantics does not always give the intuitive expected results.

In this paper we define a sound and complete top-down semantic tree characterization of *WFSX*, that includes pruning rules. It is amenable to a simple implementation, and by its nature readily allows pre-processing into Prolog, showing promise as an efficient basis for further development.

Since *WFSX* coincides with WFS on normal programs, our method is applicable to it and, for ground programs, compares favourably with previous approaches (cf. section 5).

As proven in [1], *WFSX* is sound wrt to the answer-sets semantics of [13], and it is a better aproximation to answer-sets than simply using WFS plus the renaming of $\neg$-literals (cf. example 1 where WFS plus the renaming of $\neg$-literals yields $\{not\ \neg b, not\ \neg c\}$ and is properly included in *WFSX*). Thus, our top-down method can be used as a sound one for answer-sets, that provides less incompleteness than others.

For lack of space, proofs of all theorems are omitted.

2 *WFSX* overview

In this section we define the language of extended logic programs and briefly review the *WFSX* semantics [17]. An extended program is a set of rules of the form: $L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$, $(0 \leq m \leq n)$, where each L_i is an

[1] We thank Esprit BR project Compulog 2 (no. 6810), and JNICT - Portugal for their support.

ECAI 94. *11th European Conference on Artificial Intelligence* Edited by A. Cohn
Published in 1994 by John Wiley & Sons, Ltd.

objective literal. An objective literal is either an atom A or its explicit negation $\neg A$. The set of all objective literals of a program P is called the extended Herbrand base of P and denoted by $\mathcal{H}(P)$. If L is an objective literal, *not* L is called a default literal. Literals are either objective or default literals. By *not* $\{a_1, \ldots, a_n, \ldots\}$ we mean $\{not\ a_1, \ldots, not\ a_n, \ldots\}$. An interpretation I of an extended program P is denoted by $T \cup not\ F$, where T and F are disjoint subsets of $\mathcal{H}(P)$. Objective literals in T are said to be *true* in I, objective literals in F *false by default* in I, and in $\mathcal{H}(P) - I$ *undefined* in I.

WFSX follows from WFS for normal programs plus the coherence requirement relating the two forms of negation: *"for any objective literal L, if $\neg L$ is entailed by the semantics then not L must also be entailed"*. This requirement states that whenever some literal is explicitly false then it must be assumed false by default. Here we present *WFSX* in a distinctly different manner with respect to its original definition, based on the application of the Gelfond-Lifschitz Γ operator (cf. [13]) and on a notion of semi-normal programs. The equivalence between both definitions is proven in [1].

Definition 1 *The* semi-normal version of a program P *is the program P_s obtained from P by adding to the (possibly empty) Body of each rule $L \leftarrow Body$ the default literal not $\neg L$, where $\neg L$ is the $\neg$-complement of L.*

We'll use $\Gamma(S)$ (resp $\Gamma_s(S)$) to denote $\Gamma_P(S)$ (resp $\Gamma_{P_s}(S)$).

Definition 2 *An interpretation $T \cup not\ F$ is called a* partial stable model *of P iff $T = \Gamma\Gamma_s T$ and $F = \mathcal{H}(P) - \Gamma_s T$.*

Not all programs have partial stable models (e.g. $P = \{a;\ \ \neg a\}$ has none). Those programs are called contradictory.

Theorem 1 *Every non-contradictory program P has a least (wrt $\subseteq$) partial stable model, the* well-founded model *of P ($WFM(P)$). $WFM(P)$ can be obtained by iterating $\Gamma\Gamma_s$, starting from $\{\}$.*

Example 1 Let $P = \{a \leftarrow not\ b;\ b \leftarrow not\ a;\ c \leftarrow not\ c;\ \neg a\}$. Its WFM is $\{\neg a, b, not\ a, not\ \neg b, not\ \neg c\}$.

Theorem 2 *For normal programs (i.e. without $\neg$negation)* WFSX *coincides with the well-founded semantics of [12].*

3 Semantic tree characterization of *WFSX*

In this section we define a sound and complete top-down semantic tree characterization of *WFSX* for (possibly infinite) ground programs. It is not our aim in this section to address the problems of loops and of termination for programs without function symbols. These are dealt with further down.

The top-down characterization relies on the construction of AND-trees (T-trees), whose nodes are either assigned the status successful or failed. A successful (resp. failed) tree is one whose root is successful (resp. failed). If a literal L has a successful tree rooted in it then it belongs to the WFM; otherwise, i.e. all trees for L are failed, L does not belong to the WFM. Note that, unlike other methods for WFS [3, 5, 6, 7], we deliberately do not assign the status *unknown* to nodes. We do so because in this way the characterization is most similar to SLDNF. Thus *failure does not mean falsity*, but simply failure to prove verity.

We start with the simpler problem of programs without explicit negation. It is well known [5, 6, 7, 18, 24] that the main issues in the definition of top-down procedures for WFS are infinite positive recursion, and infinite recursion through negation by default (hereafter called negative recursion). The former gives rise to the truth value false (so that the query L should fail and the query *not* L succeed, for some L involved in the recusion), and the latter to the truth value undefined (so that both L and *not* L should fail). Apart from these problems we mainly follow the ideas of SLDNF, where atoms with no rules fail, *true* succeeds,[2] atoms resolve with program rules, and the negation as failure rule that *not* L succeeds if L fails and fails if L succeeds.

In order to solve the problem of positive recursion we follow the same approach as in SLS-resolution [21], i.e. we consider a failure rule for not necessarily finite branches.

Example 2 Let $P = \{p \leftarrow p\}$. The only tree for p, $p - p - \ldots$, is infinite. So p fails and consequently *not* p succeeds.

For negative recursion the solution is not so simple, because as noted above in this case we want to fail both L and *not* L, which violates the negation as failure rule. To deal with this problem we introduce a new kind of tree, TU-tree, that rather than proving verity, proves nonfalsity. TU stands for true or undefined, i.e. non-false. Now, for any L, the verity proof of *not* L fails iff there is a nonfalsity proof of L.

TU-trees are contructed similarly to T-trees: atoms with no rules are failed leafs (as they fail to be true or undefined), *true* succeeds, atoms resolve with program rules (since a literal L is true or undefined if there is a rule for L whose body is true or undefined), and *not* L fails if L succeeds in a verity proof, and succeeds otherwise (note that *not* L is true or undefined iff L is true). Having these two kinds of trees, it becomes easy to assign a status to an occurrence of a literal L involved in negative recursion. Indeed, and this is the crux of our method, since in this case according to WFS L is undefined, it must be assigned the status failed if it is in T-tree, and successful if it is in a TU-tree.

The formalization of these solutions yields a correct characterization of WFS for normal programs. Now we show how to generalize the characterization to deal with explicit negation in *WFSX*. In a lot of points, the treatment of extended programs is akin to that of normal ones, where instead of atoms we refer to objective literals. The main difference in the generalization to extended programs resides in the treatment of negation by default. In order to fulfill the coherence requirement there must be an additional way to succeed a verity proof of *not* L. In fact *not* L is true if $\neg L$ is true.

Example 3 Consider P of example 1. The only T-tree for b is $b - not\ a$. According to the methods described for normal programs in order to prove *not* a we have to look at all possible TU-trees for a. The only one is $a -_u not\ b$. Since this is a case of negative recursion, *not* b succeeds and consequently *not* a fails. However, since $\neg a$ is true, by coherence *not* a (and thus b also) must succeed, by the additionally required method of proof for default literals.

Thus, for extended programs, in a T-tree *not* L succeeds iff all TU-trees for L fail *or if there is a successful T-tree for $\neg L$*, and fails otherwise.

[2] In the sequel we assume, without loss of generality, that the only fact of a program is *true*.

Care must also be taken in nonfalsity proofs because the coherence requirement overrides undefinedness (cf. [17]).

Example 4 Let $P = \{a \leftarrow b;\ b \leftarrow not\ c:\ c \leftarrow not\ c;\ \neg b\}$, whose WFM is $\{\neg b, not\ a, not\ b, not\ \neg b, not\ \neg c\}$.

To check verity of *not a*, one has to build TU-trees for a. The only is $a -_u b -_u not\ c$. Now we have to look for T-trees for c. The only possible one is $c - not\ c$ which, the reader can check, is failed. Thus the TU-tree for a is successful, and *not a* fails.

However, *not a* belongs to the WFM of P. Note that this problem occurs because the node labeled b in the TU-tree succeeds and should fail since b is false. Indeed the falsity of b in the WFM is imposed by the coherence principle, since $\neg b$ is true. Note that in the trees above $\neg b$ is not even used.

In TU-trees coherence imposes a modification in the form status are assigned to objective literals. Namely: in a TU-tree an objective literal L is failed if there is a successful T-tree for $\neg L$ or if one of its childs is failed; it is successful otherwise. Clearly the first condition for assigning failed to L can be removed if in TU-trees L is expanded to $L \wedge not\ \neg L$.

Summarizing, and formally:

Definition 3 *A T-tree (resp.TU-tree) for a ground extended program P is an AND-tree with root labeled A, nodes labeled by literals, and constructed top-down starting from the root by expanding successively new nodes using the rules:*

- *For a node n labeled with an objective literal A: if there are no rules for A in P then n is a leaf; otherwise, select a rule $A \leftarrow B_1, \ldots, B_j, not\ C_1, \ldots, not\ C_k$ from P, In a T-tree the successors of n are $B_1, \ldots, B_j, not\ C_1, \ldots, not\ C_k$. In a TU-tree there are additionally the successors $not\ \neg B_1, \ldots, not\ \neg B_j$.*
- *Nodes labeled with default literals are leaves.*

Definition 4 *Each node in a T-tree (resp.TU-tree) has an associated status that can be either failed or successful. All infinite trees are failed. A finite T-tree (resp.TU-tree) is successful if its root is successful and failed if its root is failed. The status of a node in a finite tree is determined according to the following rules:*

- *A leaf node n labeled with an objective literal L is successful if $L = true$, and is failed otherwise;*
- *A leaf node n in a T-tree labeled with the literal not A is successful if all TU-trees with root A (subsidiary trees of n) are failed or if there is a successful T-tree with root $\neg A$ (the only other subsidiary tree of n); n is failed if there is a successful TU-tree with root A and all T-trees with root $\neg A$ are failed;*
- *A leaf node n in a TU-tree labeled with literal not A is successful if all T-trees with root A (subsidiary trees of n) are failed; n is failed if there is a successful T-tree with root A;*
- *An intermediate node n in a T-tree (resp.TU-tree) is successful if all its children are successful, and is failed if one of its children is failed.*

After applying the previous rules some nodes may still have their status undetermined due to infinite negative recursion. To undetermined nodes in T-trees the status failed is assigned, and in TU-trees the status successful is assigned.

Theorem 3 *Let P be a ground (possibly infinite) non-contradictory extended logic program, M its WFM according to* WFSX, *and let L be an arbitrary fixed literal. Then, there is a successful T-tree with root L iff $L \in M$.*

This theorem only guarantees correctness for non-contradictory programs. However, it is possible to determine with the above characterization whether a program is contradictory:

Theorem 4 *An extended program P is contradictory iff there exists some objective literal L of P such that there are successful T-trees for both L and $\neg L$.*

The above definition is directly presented for extended programs. But since normal programs are a special case of extended ones, and in them *WFSX* coincides with WFS (cf. theorem 2), the definitions also apply for characterizing the WFM of normal programs. Moreover, for such programs some simplifications can be made. Namely, all those regarding explicit negation.

4 Pruning rules

Although sound and complete, the method described in the previous section is not effective (even for finite ground programs). In fact, and because it furnishes no mechanism for detecting loops, termination is not guaranteed. Completeness here is only ideal completeness.

To guarantee termination (at least) for finite ground programs, in this section we introduce rules that prune the search space, and eliminate both cyclic positive recursion and cyclic negative recursion.

To detect both kinds of cyclic recursions we use two kinds of ancestors: *local ancestors* are assigned to nodes of a tree, and are used for detecting cyclic positive recursion; *global ancestors* are assigned to trees, and are used to detect cyclic negative recursion.

Definition 5 *The* local ancestors *of a node n in a tree T are the literals appearing in the path from the root of T to n, with the exclusion of n.*[3]

Definition 6 *A non-subsidiary tree has no* global ancestors. *Let T be a subsidiary T-tree (resp. TU-tree) of a leaf n from tree T'. The global ancestors of T are the global ancestors of T' plus the local ancestors of n.*

Global ancestors are divided into two sets: global T- and global TU-ancestors. Global T-ancestors (resp. TU-ancestors) are those that occur in T-trees (resp. TU-trees). By global ancestors of a node n in a tree T we mean those of T.

Given a node n, a pruning rule is able to decide whether or not it is possible to assign a status (successful or failed) to the node simply by inspecting its local and global ancestors. To deal with the non-termination problem of cyclic positive recursion it suffices to test if a given node appears in its local ancestors; i.e. rule 1: *"If a node is labeled with an objective literal belonging to its set of local ancestors then it is marked failed, and its successors are ignored".*

[3] Notice that the elements of this set are objective literals.

To treat cyclic negative recursion, tests in the global ancestors are necessary. It is easily shown that any form of this recursion reduces to one of the four combination cases, depending on the cycle occurring between the two possible tree types. The nodes in T-trees are marked failed, and those in TU-trees successful. Moreover, all these combinations can be reduced to one of them:

Lemma 5 *All cyclic negative recursions can be detected in nodes of T-trees by looking at its global T-ancestors.*

We now show that the same does not hold for any other combination case, i.e. there are cycles that are only detectable with the test of lemma 5:

Example 5 The negative cycle in $\{a \leftarrow not\ b;\ b \leftarrow not\ a\}$ is not detectable by only testing if literals in T-trees belong to its global TU-ancestors, nor by only testing if literals in TU-trees belong to its global T-ancestors.

Only testing if literals in TU-trees belong to its global TU-ancestors does not detect the recursion in $\{\neg a \leftarrow not\ a;\ \neg a\}$ because successive T-trees $\neg a - not\ a$ are generated.

Lemma 5 yields pruning rule 4: *"If literal L in one T-tree occurs in its global T-ancestors then it is assigned the status failed, and its successors are ignored".*

Theorem 6 *The pruning rules 1 and 4 are necessary and sufficient for guaranteeing that all positive and negative cyclic recursions are eliminated.*

Theorem 7 *A T-tree or a TU-tree is successful (resp. failed) wrt a logic program P iff the corresponding tree using pruning rules 1 and 4 is successful (resp. failed).*

The other tests (coming from the other combination cases) give rise to additional sound pruning rules. These can be used to diminish the search space and return an answer sooner.

The material presented in this section enables us to construct SLX,[4] a sound, complete, and terminating top-down query evaluation procedure for all ground finite extended logic programs with *WFSX*. This property still holds for ground versions of logic programs with variables and functor symbols that obey the bounded-term property. Lack of space prevents us giving a formal description of SLX. That will be presented in the forthcoming paper [2].

5 Comparisons

To the best of our knowledge, paper [25] is the only in the literature that addresses the topic of proof procedures for extended logic programs. The author uses the notion of conservative derivability [26] as the proof-theoretic semantics for extended programs. The paper provides a program transformation from such programs to normal ones. Then it is proved that Kunen semantics [15] applied to the transformed program is sound and complete wrt conservative derivability. This approach has several problems mainly motivated by the interpretation of default negation as finite failure as recognized by the author. For instance, in the program $\{a \leftarrow a\}$ the literal $\neg a$ is false but a is undefined, contrary to the results obtained by answer sets and WFSX where *not a* is true. As a final remark, conservative derivability is not defined for programs with functor symbols. Therefore the approach is only applicable to extended programs without functor symbols. As shown in the previous sections, $WFSX$ solves all these problems properly.

From now on we will restrict our comparisons to the scope of normal logic programs. We start by considering SLDNF-resolution [8, 16]. The non termination problems of SLDNF, even for finite ground programs, motivated the development of the semantics that correctly handles both types of infinite recursion (positive and negative). Przymusinski in [22] showed that SLDNF is sound wrt to WFS, i.e. it can be seen as a crude approximation to WFS. Our method can be seen as a generalization of SLDNF, because all recursion problems are reduced to and solved with the notions of failure and success of trees, interpreting *not L* true in T-trees (resp. TU-trees) when all possible "proofs" for L in TU-trees (resp. T-trees) fail. This (re-)understanding of *not* greatly simplifies the construction of meta-interpreters and enables the compilation of logic programs directly into Prolog code.

Przymusinsky in [22] introduced SLS-resolution that extends SLDNF to stratified programs. This is achieved by understanding negation by default as negation as (possibly infinite) failure instead of finite failure. An extension of SLS for normal programs is defined in [21] but assumes the existence of the dynamic stratification. Przymusinski argues that the use of dynamic stratification made is effectively computable in some cases and reduces to avoiding negative recursion in the derivations. But the author doesn't show how this can be implemented in practice. Further extensions of this work [20, 23, 24] implicitly or explicitly assume a positivistic selection rule that selects negative literals in parallel as stated in [3]. For more details the reader is referred to [3]. The definition of T-trees and TU-trees doesn't impose any order on the selection of the literals, i.e. it is independent of the selection rule.

Apt and Bol [3] present a definition of SLS that deals with all general programs and all selection rules. Their approach rests on the definition of oracle SLS-trees. But to build such trees is necessary to know the WFM "a priori". Then they define a SLS-tree as the limit of a sequence of oracle SLS-trees. However, as recognized by the authors, it is not at all clear how these trees could be constructed in a top-down way.

A top-down procedure for finite ground programs was presented in [18]. This procedure suffers from severe efficiency problems. When expanding a default literal an exponencial number of resolvents in the size of the program can be generated, because all combinations of literals are generated for failing all the rule bodies for the literal of a default goal, and so a procedure that has polynomial complexity turns out to be exponential. Also, the procedure is not easily extendable to handle infinite ground programs because infinite resolvents can be generated. Furthermore, simple extensions of this procedure to the non-ground case introduce more floundering problems.

Another way of detecting loops is to use tabulation techniques. Several approaches use them to define query evaluation procedures for WFS. These methods are more general than ours in the case of function free non-ground programs,

[4] Where the X stands for eXtended programs.

but for important classes of normal programs the aforesaid techniques are unnecessary. Our main criticism is that they do not draw a clear border between their use of tabulation and the overall method incorporating it.

Bol and Degerstedt [5] method constructs only one table but the treatment of negation by default literals supposes them undefined at first, only to be forced to a complex filtering stage later.

For further and more extended comparisons, the reader is referred to [3] and [7] which defines SLG resolution. SLG relies on program transformations based on modular partial evaluation rewrite rules for top-down queryevaluation under WFS; it focuses on positive goals, and delays non-ground negative ones. Our approach, in contrast, is akin to a semantic tree refutation method.

Eshgi and Kowalski's abductive procedure [11] corrected in [10] is sound wrt to preferred extensions. In spite of not treating positive recursion and non-cyclic negative recursion, it has several similarities with our method. The "abductive derivations" and "consistency derivations" resemble our T-trees and TU-trees. The big difference is how negative cyclic recursion is treated: they succeed abductive derivations when the former kind of recursion is detected. Accordingly, they don't compute the WFM, but the preferred extensions of [10].

6 Conclusions and future work

We've presented a semantic tree characterization that is amenable for the definition of a purely top–down query evaluation procedure for ground extended programs with *WFSX*. In theory SLX is applicable to infinite ground programs. Thus, a natural required generalization of SLX is one for nonground programs.

A straightforward generalization method for non-ground programs would be to proceed as usual in the expansion of goals with rule variants, and keeping the test for inclusion in the ancestors lists. However it has two problems: first, as shown in [4], this loop detection method does not guarantee termination in the nonground case, even for term–bounded programs; second, the procedure flounders on nonground default literals. Nevertheless, its correctness and termination are guaranteed for call–consistent term–bounded programs. To guarantee termination for nonground term–bounded programs, we intend to introduce tabulation methods into SLX.[5] Another subject of future work is that of introducing in SLX constructive negation techniques for solving the floundering problem.

Given the similarities between SLX and Prolog execution, it has been rather easy to implement both a Prolog interpreter, and a pre–processor that compiles *WFSX* programs into Prolog, without the need for meta–calls.[6] It is our intention to proceed with the study of more efficient implementations of SLX. Indeed, this is the subject of a national project proposal in cooperation with LIACC in Porto, in which we intend to implement SLX using sidetracking, parallelism, and low–level support for tabulation.

[5] Note that by being cumulative, *WFSX* is amenable to such methods.

[6] This implementation is available on request.

REFERENCES

[1] J. J. Alferes, *Semantics of Logic Programs with Explicit Negation*, Ph.D. dissertation, Universidade Nova de Lisboa, 1993.

[2] J. J. Alferes, C. V. Damásio, and L. M. Pereira, 'SLX - A top-down derivation procedure for programs with explicit negation', Technical report, CRIA, Uninova, (1994).

[3] K. Apt and R. Bol, 'Logic programming and negation: a survey', *Journal of Logic Programming*, (1993). To appear.

[4] K. Apt, R. Bol, and J. Klop, 'On the safe termination of Prolog programs', in *Proc. ICLP'89*, eds., Levi and Marteli, pp. 353–368. MIT Press, (1989).

[5] R. Bol and L. Degerstedt, 'Tabulated resolution for well founded semantics', in *Proc. ILPS'93*. MIT Press, (1993).

[6] W. Chen and D. S. Warren, 'A goal-oriented approach to computing well-founded semantics', in *Int. Joint Conf. on L. P.*, ed., K. Apt, pp. 589–603. MIT Press, (1992).

[7] W. Chen and D. S. Warren, 'Query evaluation under the well founded semantics', in *PODS'93*, (1993).

[8] K. Clark, 'Negation as failure', in *Logic and Data Bases*, eds., H. Gallaire and J. Minker, pp. 293–322. Plenum Press, (1978).

[9] J. Dix, 'A framework for representing and characterizing semantics of logic programs', in *KRR'93*, eds., B. Nebel, C. Rich, and W. Swartout. Morgan Kaufmann, (1992).

[10] P. M. Dung, 'Negation as hypotheses: An abductive framework for logic programming', in *Proc. ICLP'91*, ed., K. Furukawa, pp. 3–17. MIT Press, (1991).

[11] K. Eshghi and R. Kowalski, 'Abduction compared with negation by failure', in *Proc. ICLP'89*. MIT Press, (1989).

[12] A. Van Gelder, K. A. Ross, and J. S. Schlipf, 'The well-founded semantics for general logic programs', *Journal of the ACM*, **38**(3), 620–650, (1991).

[13] M. Gelfond and V. Lifschitz, 'Logic programs with classical negation', in *Proc. ICLP'90*, eds., Warren and Szeredi, pp. 579–597. MIT Press, (1990).

[14] D. B. Kemp, P. J. Stuckey, and D. Srivastava, 'Query Restricted Bottom-up Evaluation of Normal Logic Programs', in *Proc. JICSLP'92*, pp. 288–302. MIT Press, (1992).

[15] K. Kunen, 'Negation in logic programming', *Journal of LP*, **4**, 289–308, (1987).

[16] J. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987.

[17] L. M. Pereira and J. J. Alferes, 'Well founded semantics for logic programs with explicit negation', in *Proc. ECAI'92*, ed., B. Neumann, pp. 102–106, (1992).

[18] L. M. Pereira, J. N. Aparício, and J. J. Alferes, 'A derivation procedure for extended stable models', in *Int. Joint Conf. on AI*. Morgan Kaufmann, (1991).

[19] L. M. Pereira, J. N. Aparício, and J. J. Alferes, 'Nonmonotonic reasoning with logic programming', *J. of Logic Programming.*, **17**(2, 3 & 4), 227–263, (1993).

[20] H. Przymusinska, T. C. Przymusinski, and H. Seki, 'Soundness and completeness of partial deductions for well-founded semantics', in *LPAR'92*, ed., A. Voronkov. LNAI 624, (1992).

[21] T. Przymusinski, 'Every logic program has a natural stratification and an iterated fixed point model', in *8th Symp. on Princ. of Database Sys.* ACM SIGACT-SIGMOD, (1989).

[22] T. Przymusinski, 'On the declarative semantics of semantics of logic programs', *J. Automated Reas.*, **5**, 167–205, (1989).

[23] T. Przymusinski and D.S. Warren, 'Well-founded semantics: Theory and implementation'. Draft, 1992.

[24] K. A. Ross, 'A procedural semantics for well-founded negation in logic programs', *J. Logic Programming*, **13**, 1–22, (1992).

[25] F. Teusink, 'A proof procedure for extended logic programs', in *Proc. ILPS'93*. MIT Press, (1993).

[26] G. Wagner, 'Neutralization and preeemption in extended logic programs', Technical report, Freie Universität Berlin, (1993).