TR-337

An Evaluation Method for Stratified Programs under the Extended Closed World Assumption

by
H. Seki & H. Itoh

Fe

An Evaluation Method for Stratified Programs under the Extended Closed World Assumption

Hirohisa SEKI and Hidenori ITOH
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, Japan

Abstract

This paper considers a semantics for stratified databases called the *ECWAs* (extended closed world assumption for stratified databases), and proposes a query evaluation algorithm which is shown to work as an interpreter under the ECWAs for a class of stratified databases. The ECWAs is a natural extension of the closed world assumption (CWA) [Rei78] by introducing the notion of stratification. The semantics under the ECWAs is shown to be equivalent to the minimal model defined by the fixpoint semantics (e.g., [ABW86], [LST86], [VG86]). The proposed evaluation algorithm is based on OLD resolution with tabulation [TS86], and its soundness and completeness is proved for a broader class of stratified programs than the previously proposed one [ABW86].

1 Introduction

Since Clark introduced the negation as failure rule [Cla78], the problem of dealing with negative information in logic programs has been extensively studied by many researchers. Its importance is not confined to logic programming, but has also been stressed in the field of deductive databases. Reiter's closed world assumption (CWA) [Rei78] is one of the most natural and simple approaches from the database's point of view.

When applied to Horn databases, it is known that the CWA is consistent and works well. When a database consists of non-Horn clauses, however, the CWA becomes inconsistent even for simple databases (e.g., [She84]). To solve this problem, Minker proposed an extension of CWA called *Generalized CWA* (GCWA) [Min82], and Gelfond et al. [GP86], [GPP86] gave its further modification called *Extended CWA* (ECWA). These formalizations deal with a set of universal first order formulas as a database. In this paper, we confine ourselves to an important class called *stratified* programs[1] [ABW86]. For a stratified program, we introduce *ECWAs*, which is a natural extension of Reiter's CWA and is also a special case

[1] In this paper, we use the terms *programs* and *databases* interchangeably. From the theoretical point of view, a database is just a logic program [Llo84].

of ECWA, by introducing the notion of stratification. Then, the semantics of a stratified program under the ECWAs is shown to be equivalent to the *minimal model* defined by the fixpoint semantics [ABW86], [LST86], [Lif86], [VG86].

Although the ECWAs (or equivalently, the minimal model M_P in [ABW86]) seems to provide a natural semantics for stratified databases, it does not immediately suggest a practical implementation of its query evaluation algorithm. Several researchers have proposed its interpreter in a restricted case such as function-free formulas [ABW86], [GP86]. This paper proposes a simple query evaluation algorithm which faithfully implements the ECWAs for stratified programs under reasonable assumptions. The evaluation algorithm is based on OLD resolution with tabulation [TS86], augmented with the negation as failure rule. Its correctness is proved for a broader class of stratified databases than than the previously proposed ones.

Section 2 gives the definition of the ECWAs, and the equivalence to the minimal model defined by the fixpoint semantics is shown. The relation to the *tight tree semantics* [VG86] and its problems are also discussed. In section 3, after specifying a class of stratified programs considered in this paper, a query evaluation algorithm is proposed. Its soundness and completeness are shown in section 4.

2 Extended CWA for Stratified Programs

2.1 Negation: From the Viewpoint of Recursive Query Processing

The *closed world assumption* (CWA) proposed by Reiter [Rei78] gives an inference rule to add negative information to logic programs. According to this assumption, if a ground atom A is not a logical consequence of a given program, then we infer $\neg A$. The CWA is often said to be a natural assumption from the database's viewpoint. In this subsection, we reconsider it from the viewpoint of recursive query processing.

Recently, recursive query processing is one of the most active research area in the field of deductive databases (e.g., [BR86a],[BR86b], [SZ87], and the references therein). There have been proposed many optimization methods such as Magic Sets [BR86b], the Alexander Method [RLK86]. In these approaches, when a program and a query are given, they are first transformed into a set of rules whose bottom-up evaluation is devised to simulate its Prolog-like top-down behaviours. The bottom-up evaluation is essentially the computation of the fixpoint of rules using a similar mapping T_P ([Llo84], p. 30) from the set of all Herbrand interpretations of transformed rules to itself. This bottom-up computation proceeds until no new atoms are generated.

Suppose that a Horn database P_0 is given. Suppose further that, in P_0, there is a recursively defined clause of form: $p \leftarrow \cdots p$, which is the only clause of the definition of p. When a query $\leftarrow p$ is given, the Alexander Method, for example, will repeatedly generate atoms corresponding to the call to the subgoal p (i.e., *call_p*), and then will terminate without generating atoms corresponding to its solution (i.e., *sol_p*). Thus, even when the corresponding SLD-resolution of $\leftarrow p$ for P_0 falls into an infinite loop, the bottom-up computation by Magic Sets/Alexander Method *terminates* without generating its solutions. Hence, in the above-mentioned query processing methods, there is no distinction between finite failure and falling into an infinite loop, and they are both considered to be false. This is exactly what the CWA infers.

Keeping this in mind, when we consider an extension of recursive query processing methods into general logic programs, the problem arises what semantics we should assume for such a program. If a database consists of non-Horn clauses, it is well known that the CWA becomes inconsistent even for simple databases (e.g., [She84]). In the next subsection, we consider an extension of CWA for stratified programs such that falling into a infinite loop is also considered to be false. An query evaluation algorithm under its semantics is given in section 3.

2.2 Semantics Based on Extended CWA

A *general logic program* [Llo84] is a set of rules which can contain both positive and negative goals in their bodies. We introduce an important class of general logic programs called *stratified* programs[ABW86]. We follow the definition by [LST86].

DEFINITION 2.1 {stratified program}

A general logic program P is *stratified* if its predicates can be partitioned into levels so that, in every program clause $p \leftarrow L_1, ..., L_n$, the level of every predicate in a positive literal is less than or equal to the level of p and the level of every predicate in a negative literal is less than the level of p. □

It is convenient to suppose that the levels of a stratified program are 0, 1, ..., k for some integer k, where k is the *minimum* number satisfying the above definition. This is assumed throughout this paper. In this case, P is said to have the maximum level k and is denoted by $P = P_0 + ... + P_k$, where P_i is a set of clauses whose head predicates have level i. Note that P_0 is a set of Horn clauses. To make the discussion simple, each predicate occurring in a program is assumed to have its definition.

We consider the following semantics for stratified programs. Gelfond [Gel87] proposed a similar definition for propositional logic programs. The following is an extension to the first order stratified programs.

DEFINITION 2.2 {ECWA for stratified programs (ECWAs)}

Let P be a stratified program and $P = P_0 + ... + P_k$. Suppose that a sequence of the set of axioms E_0,...,E_k, $ECWAs$ is defined as follows:

$$\begin{aligned}
E_0 &= P_0 \cup \{\neg L \mid L \text{ is a positive ground literal of level 0 and } P_0 \not\vdash L\} \\
E_1 &= E_0 \cup P_1 \cup \{\neg L \mid L \text{ is a positive ground literal of level 1 and } E_0 \cup P_1 \not\vdash L\} \\
& \quad \ldots \\
E_k &= E_{k-1} \cup P_k \cup \{\neg L \mid L \text{ is a positive ground literal of level } k \text{ and } E_{k-1} \cup P_k \not\vdash L\} \\
ECWAs &= E_k,
\end{aligned}$$

where $T \vdash L$ means that L is a logical consequence of T. Then, the semantics of a stratified program P under the ECWAs is defined by a pair of sets of ground atoms (SS, FS) defined as follows (SS is intended to be the success set of P while FS means the false set of P):

$$\begin{aligned} SS &= \{A \mid A \text{ is a positive ground literal such that } ECWAs \vdash A\} \\ FS &= \{A \mid \neg A \text{ is a negative ground literal such that } ECWAs \vdash \neg A\} \qquad \square \end{aligned}$$

The above definition gives a natural extension of Reiter's CWA [Rei78] in the following respects : (i) if a program is Horn, then ECWAs coincides with CWA and (ii) function symbols are allowed. ECWAs means ECWA for a *stratified* program. Gelfond et al. [GPP86] proposes ECWA for a general program, in which no notion of stratification is introduced.

The fixpoint semantics of a stratified program is given by several researchers [LST86], [ABW86], [Lif86], [VG86]. We will now review the fixpoint semantics and then show the equivalence of the semantics under the ECWAs and the fixpoint semantics.

DEFINITION 2.3 {mapping T_{P_i}}

Let P be a stratified program and $P = P_0 + ... + P_k$. For every $i = 0, ..., k$, a mapping $T_{P_i}(M)$, where M is a subset of the Herbrand base, is defined as follows:

- $T_{P_i}(M)$ contains M.
- For any rule, $A \leftarrow L_1 \wedge \cdots \wedge L_n$ in P_i, if there exists some substitution θ of ground terms of variables such that $L_1\theta \wedge \cdots \wedge L_n\theta$ are true in M, then ground atom $A\theta$ is in $T_{P_i}(M)$.
- $T_{P_i}(M)$ contains no other atoms.

Let $T^j_{P_i}$ be T_{P_i} applied j times, and we define

$$T^{\omega}_{P_i}(M) = \bigcup_{l<\omega} T^l_{P_i}(M) \qquad \square$$

Using the above mapping, the fixpoint semantics of a stratified program is defined as follows:

DEFINITION 2.4 {fixpoint semantics}

Let P be a stratified program and $P = P_0 + ... + P_k$. Then,

- $SS^{fp}_0 = T^{\omega}_{P_0}(\phi)$. Note that P_0 is a set of Horn clauses and SS^{fp}_0 is the least fixpoint of T_{P_0}.
- For $i > 0$ $(i = 1, ..., k)$,
$$SS^{fp}_i = T^{\omega}_{P_i}(SS^{fp}_{i-1}).$$

The fixpoint semantics of P is defined by a pair of sets of ground atoms (SS^{fp}, FS^{fp}) as follows:

$$\begin{aligned} SS^{fp} &= SS^{fp}_k \\ FS^{fp} &= HB - SS^{fp}, \text{ where } HB \text{ is the Herbrand base of } P. \qquad \square \end{aligned}$$

Following [ABW86], let denote SS^{fp} by M_P. **Proposition 2.1** shows that both the ECWAs and M_P coincide with each other. Before that, we need the following lemma:

Lemma 2.1 *Let P be a stratified program and $P = P_0 + ... + P_k$. For every l $(0 \leq l \leq k)$,* *(1)* SS_l^{fp} *is an Herbrand model of* $E_{l-1} \cup P_l$

(2) $E_{l-1} \cup P_l \vdash L \Leftrightarrow L \in SS_l^{fp}$,

where L *is a positive ground atom of level* l *and* E_l *is defined in DEFINITION 2.2, and let* E_{-1} *be* ϕ.

Proof: See Appendix. □

Proposition 2.1 *Let* P *be a stratified program and* $P = P_0 + \ldots + P_k$. *Then, the semantics of a stratified program* P *under the ECWAs is equivalent to its fixpoint semantics, i.e.,*

$$(SS, FS) = (SS^{fp}, FS^{fp}).$$

Proof: Obvious from Lemma 2.1. □

Example 2.1

$$\begin{array}{lll} p & \leftarrow & q, \neg r \\ q & \leftarrow & \\ r & \leftarrow & r \end{array}$$

In the above program, predicates q, r are of level 0 and predicate p is of level 1. $SS = \{q, p\}$ and $FS = \{r\}$. Note that the meaning of a program is not its completion [Cla78]. □

2.3 Relation to Tight Tree Semantics and Its Problems

Several important results relating to the semantics of stratified programs have been obtained recently (e.g., [ABW86], [GP86], [GPP86], [Lif86], [Prz86a], [LST86], [VG86]). Among them, we examine the *tight tree semantics* proposed by Van Gelder [VG86] in this subsection, since the notion of the *bounded term size property* (defined below) introduced in [VG86] plays an important role also in this paper.

First, we give an informal explanation of the tight tree semantics (as for the strict definition, see [VG86]). In the definition of the tight tree semantics, derivations are considered at ground instance level, namely, each ground goal is resolved using ground instantiated rules. Although negation is defined as finite failure, proof attempts are limited to *tight* derivations, that is, derivations expressed by trees in which no node has an identical ancestor. Those derivation trees are said to be *tight NF-trees.*

When there exist a tight NF-tree whose root is a positive atom A, then A is considered to be in the success set SS^t. Suppose that a positive atom A is in SS^t. Then, any ground derivation tree in which there is a node labelled with $\neg A$ is considered to be failed, and is discarded. For any positive ground atom L, if there is no ground derivation tree whose root matches L, then L is considered to be in the general failure set GF^t. The tight tree semantics is defined by the pair (SS^t, GF^t).

Consider **Example 2.1 again.** Under the tight tree semantics, r is classified into GF^t because of the restriction of tightness, hence, in turn, p, is included in SS^t, which is equivalent to the semantics under the ECWAs in this case. Generally, however, all atoms in the Herbrand base of a given stratified program are not classified into either SS^t or GF^t. For the purpose of completely classifying all the atoms, he introduced a condition called the *bounded term size property* in [VG86]. To state its definition, one more definition is required, which is intended to ensure the *freeness from floundering* [Cla78].

DEFINITION 2.5 {safe for negation}

Let R be a computation rule (for SLDNF-derivation) (e.g., [Llo84]) which always selects only *positive atoms* in goals. A general logic program is said to be *safe for negation* if, for any R, every SLDNF-derivation via R has the property that any variable in a negative atom is also in some positive atom of the same goal or in the top-level clause. □

Let the *size of a term* be the count of its functors, constants and variables.

DEFINITION 2.6 {bounded term size property[2]}

Let R be a computation rule defined in the above definition. Then, a general logic program has the *bounded term size property* if there exists a function $f(n)$ such that, for any R, whenever the top level goal has no argument whose term size exceeds n, then no subgoal in any SLDNF-derivation via R has an argument whose term size exceeds $f(n)$, whether the derivation is successful or not. (Note that, in each derivation, *most general unifiers* are used and only positive atoms are expanded.) □

As stated in [VG86], the bounded term size property seems to be quite a natural condition, especially for database applications, where no function symbols are usually introduced. It seems that programs without this property are usually intended to be non-terminating (such as number generation programs) or they might be buggy. Under the bounded term size property, it is claimed that stratified programs which are safe for negation are completely classified by the tight tree semantics, i.e., every atom in the Herbrand base is either in SS^t or in GF^t, and it is shown that it is equivalent to the semantics under the minimal model defined by the fixpoints. The following simple example, however, shows that these conditions are still insufficient.

Example 2.2

$$\begin{aligned} q(f(a)) &\leftarrow \\ p(X,Y) &\leftarrow p(X,Z) \end{aligned}$$

The above first clause is defined to only for the purpose of introducing a constant and a function symbol into the Herbrand universe. Then, a positive atom $p(a, f(a))$, for example, is *not* classified either into SS^t or GF^t, because there exists an infinite tree of form: $p(a, f(a)) \leftarrow p(a, f(f(a))) \leftarrow p(a, f(f(f(a)))) \leftarrow \cdots$.

The reason why the above atom is unclassified is that the bounded term size property is defined by SLDNF-derivation (using most general unifiers), while the tight tree semantics is defined in terms of tight NF-trees, which are composed of ground instantiated rules. Hence, a tight NF-tree of a stratified program is *not* always finite even if it has the bounded term size property.

[2]Note that the definition of the bounded term size property in this paper is different from that of [VG86] in that (i) we separate the condition of safety for negation from the bounded term size property defined in [VG86], and (ii) it is not assumed that the form of function $f(n)$ is actually known.

On the contrary, the semantics under the ECWAs (and also the minimal model) classifies every ground atom of form: $p(t_1, t_2)$ into FS, since there is no finite proof of $p(t_1, t_2)$. One of the contributions of this paper is to propose a practical algorithm which faithfully implements the semantics under the ECWAs when a stratified program has the bounded term size property.

2.4 From Semantics to a Practical Algorithm

Although the semantics under the ECWAs (or equivalently, the minimal model M_P) is quite natural and simple, the definition does not suggest a practical implementation of its evaluation algorithm. [ABW86] tried to give algorithms under the above mentioned semantics, and [GP86] proposes a query evaluation procedure in a slightly different context. To our knowledge, their methods are confined to restricted cases such as propositional logic formulas. Przymusinski [Prz86b] proposed a query answering algorithm for circumscriptive and closed-world theories. Since his formalism treats a more general case such as arbitrary clauses, the proposed algorithm becomes more complex and seems to have inefficiencies in a general theorem prover such as *subsumption checking.*

It is obvious from **Example 2.1** that the usual SLDNF-refutation is not sufficient. Some loop-trap mechanism seems to be necessary to prevent computation from falling into infinite loops. As for ground atoms, it would be clear that along any path on a proof tree, the same ground literal need not occur more than once and such a path can be failed. The difficulty is that atoms in a goal usually contain variables. When a new goal is a *variant* of its ancestor on some path on a proof tree, then we cannot fail that branch, since it would lead to an unsound evaluation algorithm. Hence, to propose a practical algorithm under the ECWAs (or M_P) is a research area in its own right. Our purpose is to give an algorithm which faithfully implements the ECWAs, hopefully with a *least* modification of the usual SLDNF-refutation. The next section describes such an algorithm and its correctness is proved in section 4.

3 An Evaluation Algorithm under Extended CWAs

3.1 A Class of Stratified Programs

In this subsection, we make clear what class of stratified programs is considered as a target of our evaluation algorithm under the ECWAs. At first, we employ OLDNF-resolution as a basis of our evaluation algorithm. OLDNF-resolution, which is a special case of SLDNF-resolution, uses a selection function which always selects the leftmost atom in goals just as the conventional Prolog interpreter does. Now, we introduce a notion ensuring floundering freeness via this selection rule by modifying DEFINITION 2.5.
DEFINITION 3.1 {OLD-safe for negation}

Consider a modification of SLDNF-resolution such that (i) when the leftmost atom of any goal is a positive atom, then resolve that atom as SLDNF-resolution does, (ii) when the leftmost atom of any goal is a negative atom (let the goal be of form $\leftarrow \neg L, \alpha$ where α is possibly empty sequence), then next try

$\leftarrow \alpha$ as its subgoal[3]. Let this computation rule be R_0. Then, a general logic program is called *OLD-safe for negation* (meaning "safe for negation via ordered linear derivation") if every top-down derivation via R_0 has the property that, whenever the leftmost atom of any goal is negative, then any variable (if any) in that atom also appear in the top-level clause. □

Suppose that a general logic program is OLD-safe for negation and the top-level clause is ground. Then, it is clear from the definition that, whenever the leftmost atom of any subgoal is a negative atom, it is instantiated into a ground term. In general, we introduce the following definition.

DEFINITION 3.2 {OLD-floundering free}

Let P be a general logic program and R_0 be a computation rule defined in the above definition. A query Q is said to be *OLD-floundering free* with respect to P when any derivation of the top-level goal Q via R_0 has the property that, whenever the leftmost atom of any goal is negative, then every argument of the atom is instantiated into a ground term. □

Intuitively, OLD-floundering freeness means that, if a query is given in "appropriately" instantiated form, then, in any goals of SLDNF-resolution via a computation rule R_0, the leftmost negative atom of each goal is instantiated into a ground term when that atom is evaluated, hence, there is no danger of falling into floundering. For the practical point of view, this condition can be considered natural, since Prolog users are expected to use its "not-predicate" in such a manner, otherwise, its intended semantics has changed. Furthermore, it seems to be possible that the recent work on mode inference by abstract interpretation (e.g., [KK87]) is applied (by slight modification) to decide whether a given query is OLD-floundering free or not.

In summary, we consider the following class of stratified programs as a target of our evaluation algorithm.

DEFINITION 3.3 {OLD-canonical program}

A stratified program is said to be *OLD-canonical* if it has the bounded term size property and is OLD-safe for negation. □

3.2 An Evaluation Algorithm: OLDTNF Resolution

The evaluation algorithm that we propose here is called *OLDTNF resolution*. It is based on OLDT resolution [TS86], augmented with negation as failure rule, just as usual SLDNF-resolution is based on SLD-resolution. OLDT resolution (OLD resolution with *tabulation*) was first proposed by Tamaki-Sato [TS86], and similar work have been given by several researchers ([Vie87][4], [Die87], [KK87]).

In the following, we use the terminology and definitions introduced in [TS86] as much as possible. Apart from the treatment of negation, the deference between ours and that of [TS86] is in that (i) every predicate is designated as the table predicate and (ii) term-depth abstraction is not introduced, since it is not necessary in our case.

[3]The purpose of introducing this computation rule is only for defining the freeness from floundering. Hence, any leftmost negative atom in a goal is simply skipped, since negation as failure rule is only a test and cannot make any bindings.

[4]Vieille's framework is more general than OLDT resolution in that a more flexible selection function is allowed.

The basic principles of OLDT-resolution is to prevent the interpreter from repeatedly trying to solve the same goal and thus to cut off any infinite branch, by introducing the tabulation techniques into OLD derivation. An informal explanation of the behaviour of OLDT resolution is as follows. When a goal is the first one to be tried in a computation path, it is said to be a *solution node*. All solutions obtained from the solution node are stored in a table called the *solution table*. On the other hand, when the same goal appears in a computation path, it is said to be a *lookup node*. A lookup node is resolved only with the atoms in the solution table, i.e., atoms stored in the solution table are used as lemmas. When there is no solutions resolvable with a lookup node, then the computation of the lookup node suspends until new resolvable solutions are registered in the solution table.

We first model our evaluation algorithm in terms of OLDNF resolution, and then introduce the tabulation technique into it.

DEFINITION 3.4 {OLDNF resolution, OLDNF tree, OLDNF refutation}

Let P be a stratified program which is OLD-canonical and G be an OLD-floundering free query. Then, *OLDNF resolution* is a special case of SLDNF resolution [Llo84] such that the computation rule always selects the leftmost atom in each goal. The *OLDNF tree* is similarly defined as a special case of the SLDNF-tree. An OLDNF tree of G is an OLDNF tree such that its root is labelled with G, and its nodes are labelled with OLDNF resolvents[5] and its edges are labelled with substitutions.

An *OLDNF refutation* of G is a path in an OLDNF tree of G from the root to a node labelled with the null clause. Let $\theta_1, ..., \theta_n$ be the substitutions labelled to edges in the path. The *substitution* of the refutation is the composition $\theta = \theta_1 \circ ... \circ \theta_n$, and the *solution* of the refutation is $G\theta$. □

DEFINITION 3.5 {subrefutation}

Let $G_0, G_1, ..., G_k$ be a sequence of labels of the nodes, and $\theta_1, ..., \theta_n$ be the labels of the edges on a path in an OLDNF tree. Let Γ and Δ be sequences of atoms. The path is called a *subrefutation* of Γ when i) G_0 is of the form: "Γ, Δ" and ii) each G_i $(1 \le i \le k-1)$ has $\Delta\theta_1 \circ \cdots \circ \theta_i$ as its last sequence and G_k is $\Delta\theta_1 \circ \cdots \circ \theta_n$, and iii) the number of atoms in G_i is greater than that of G_k.

A subrefutation is said to be a *unit subrefutation* if Γ in the above is an atom. The substitution and the solution of a subrefutation is defined similarly as in the case of a refutation. □

Now we give several definitions necessary for the tabulation.

DEFINITION 3.6 {solution table}

A *solution table* is a set of entries. Each entry consists of a pair of a key and a list (called a *solution list*), where the key is a positive atom and the solution list is a list of atoms such that each atom in the list is an instance of its key. □

Every node of the OLDTNF tree is classified as either a *solution node*, a *lookup node*, or a *negative node*.

DEFINITION 3.7 {solution node, lookup node}

Let Tr be an OLDNF tree and Ts a solution table. A node in Tr is said to be a *lookup node* when the leftmost atom of the node is an instance of some key in Ts. When the leftmost atom of the node

[5] We sometimes use a *node* and a *resolvent* of its label interchangeably, when the meaning is clear from the context.

is a negative atom, then the node is said to be a *negative node*. Otherwise, it is called a *solution node*. □

DEFINITION 3.8 {lookup table, associated solution list}

Let Tr and Ts be the same as in DEFINITION 3.7. A *lookup table* Tl of (Tr, Ts) is a set of pointers pointing from each lookup node in Tr into a solution list of some key K in Ts such that the leftmost atom of the lookup node is an instance of K. A tail list of a solution list pointed from a lookup node is called an *associated solution list* of the lookup node. □

Using the above-mentioned definitions, the basic structure of OLDTNF derivation is defined.

DEFINITION 3.9 {OLDTNF structure, initial OLDTNF structure}

An *OLDTNF structure* is a triple (Tr, Ts, Tl), where Tr is an OLDNF tree, Ts is a solution table and Tl is a lookup table of Tr and Ts.

The *initial OLDTNF structure* of a query G is the triple $T_0 = (Tr_0, Ts_0, Tl_0)$, where Tr_0 is a single node v_0 labelled with G, and Ts_0 is the solution table consisting of only one entry whose key is the leftmost atom of G with an empty solution list, and Tl_0 are empty. □

DEFINITION 3.10 {extension of an OLDTNF structure}

Let P be a stratified program which is OLD-canonical, and T be an OLDTNF structure (Tr, Ts, Tl). Suppose further that the root of Tr is labelled with an OLD-floundering free query. An *immediate extension* of T by P is the result of the following operations.

(1) Select a terminal node v, labelled with $\leftarrow A, \Gamma$, where A is a atom and Γ is a (possibly empty) sequence of atoms.

- *(OLD extension)* When v is a solution node, let $C_1, ..., C_k$ be all the clauses (if any) in P such that each C_i is of the form : $B_i \leftarrow M_{i1}, ..., M_{iq_i}$ and A and B_i have the mgu θ_i, respectively. Then add k child nodes labelled with each G_1,...,G_k to v, where each G_i is $\leftarrow (M_{i1}, ..., M_{iq_i}, \Gamma)\theta_i$, respectively. The edge from v to the node G_i is labelled with θ_i.
- *(lookup extension)* When v is a lookup node and its associated solution list is not empty, let $B_1\tau_1, ..., B_k\tau_k$ be all the elements in that list such that A and $B_i\tau_i$ have the mgu θ_i, and let G_i be $\leftarrow \Gamma\theta_i$, respectively. Then add k child nodes labelled with each $G_1, ..., G_k$ to v. The edge from v to the node G_i is labelled with θ_i.
- *(negation as failure)* When A is a negative ground atom of form $\neg A_0$, then, an attempt is made to construct a finitely failed OLDTNF tree with $\leftarrow A_0$ at the root[6]. If $\leftarrow A_0$ has succeeds (i.e., there exists any solution), then the subgoal $\leftarrow A$ fails and so the goal v also fails. If $\leftarrow A_0$ fails finitely under a *fair* search strategy (see DEFINITION 3.12), then the subgoal $\leftarrow A$ succeeds and the node v has a unique child node G of the form: $\leftarrow \Gamma$. The edge from v to G is labelled with the identity substitution.

[6] We create a new initial OLDTNF structure of $\leftarrow A_0$, where its solution table and lookup table are also newly created. For the conceptual simplicity, a new structure is treated separately from the one it's derived from, hence no solution table is shared among different structures. Several optimizations such as sharing solution tables should be introduced when implemented.

(2) If the above lookup extension is performed to a node v, then replace the pointer from v to the one pointing to the last of its associated solution list.

(3) After the above operations, a new node labelled with a non-null clause is classified as a lookup node if the leftmost atom of the new node is an instance of some key in Ts. Otherwise, it is a solution (negative) node if the leftmost atom is positive (negative), respectively.

(4) For a new lookup node (if any), add a pointer from it to the head of the solution list of the corresponding key.

(5) For a new solution node (if any), add a new entry whose key is the leftmost atom of the label of the new node and whose solution list is the empty list. When a new node is a lookup node, add no entry. For each unit subrefutation of atom L (if any) starting from a solution node and ending with some of the new nodes, add its solution $L\lambda$ to the last of the solution list of L in Ts, if $L\lambda$ is not in the solution list.

An OLDTNF structure (Tr', Ts', Tl') is said to be an *extension* of OLDTNF structure (Tr, Ts, Tl) if (Tr', Ts', Tl') is obtained from (Tr, Ts, Tl) through successive application of immediate extensions. □

DEFINITION 3.11 {OLDTNF refutation}

Let P be a stratified program which is OLD-canonical, and G be an OLD-floundering free query. An *OLDTNF refutation* of G by P is a path in some extension of the initial OLDTNF structure of G, from the initial root to a node labelled with the null clause. The *substitution* of the OLDTNF refutation and the *solution* of the OLDTNF refutation are defined similarly as for the OLDNF refutation. □

As is stated in the definition of negation as failure rule in DEFINITION 3.10, we must specify an appropriate class of search strategies.

DEFINITION 3.12 {fair search strategy}

A search strategy in the OLDTNF structure is said to be *fair* if, for every node N in the OLDTNF structure, N is selected within a finite number of steps. □

4 The Correctness of the Evaluation Algorithm

In this section, we give the proof of the soundness and completeness of OLDTNF resolution. As its preliminaries, the following lemma ensures that the OLDTNF structure for any ground query is finite when a stratified program is OLD-canonical.

Lemma 4.1 *Assume that a stratified program P is OLD-canonical. Then, for any ground atom G, the search process for the OLDTNF refutation of $\leftarrow G$ is finite, whether the search is successful or not.*

Proof: See **Appendix**. □

Before proving the correctness of the evaluation algorithm, we need the following lemma.

Lemma 4.2 *Assume that a stratified program P is OLD-canonical. Let $G_1, ..., G_n$ be a (possibly empty) sequence of atoms. Suppose further that there exists a positive atom G_0 of the level r_0 such that the level of each positive atom (if any) among $\{G_1, ..., G_n\}$ is less than or equal to r_0 and the level of every negative atom (if any) among $\{G_1, ..., G_n\}$ is less than r_0.*

(Soundness a) *When $\leftarrow G_1, ..., G_n$ has an OLDTNF-subrefutation with the substitution θ, then any ground instance of $(G_1, ..., G_n)\theta$ is true in (SS, FS).*

(Soundness b) *If a ground atom G_0 is in FS, an OLDTNF-refutation of G_0 is finitely failed.*

(Completeness) *Suppose that ground instance $G'_1, ..., G'_n$ of $G_1, ..., G_n$ is true with respect to (SS, FS). Let T be an OLDTNF structure for a OLD-floundering free query and v is a node in T. Suppose further that v is labeled with $\leftarrow (G_1, ..., G_m)\theta$ $(m \geq n)$ where $G'_1, ..., G'_n$ is an instance of $(G_1, ..., G_n)\theta$. Then, there exists an extension of T under a fair search strategy such that T contains an OLDTNF subrefutation of $\leftarrow (G_1, ..., G_n)\theta$ which starts from v and whose solution subsumes $G'_1, ..., G'_n$.*

Proof: See **Appendix**. □

The correctness of OLDTNF resolution is immediate from the above lemma.

Proposition 4.1 *Assume that a stratified program P is OLD-canonical.*

(Soundness) *When an OLD-floundering free query $\leftarrow G$ has an OLDTNF-refutation with the substitution θ, then any ground instance of $G\theta$ is true in (SS, FS).*

(Completeness) *Suppose that a ground atom G' is in SS. Then, for any OLD-floundering free query $\leftarrow G$ such that G' is an instance of G, there is an OLDTNF-refutation tree of root G under a fair search strategy such that G' is an instance of the solution of the refutation.* □

5 Concluding Remarks

We have introduced a semantics for stratified databases called the ECWAs. The ECWAs is a natural extension of the closed world assumption (CWA) [Rei78] , and is also a modification of both Generalized CWA [Min82] and Extended CWA [GPP86] by introducing the notion of stratification. The semantics under the ECWAs is shown to be equivalent to the minimal model M_P defined by the fixpoint semantics.

A query evaluation algorithm was also proposed, which is shown to work as an interpreter for the ECWAs under reasonable assumptions. The class of stratified databases for which the evaluation algorithm is shown to be correct is broader than the one given by [ABW86]. The proposed interpreter is based on OLDT resolution, augmented with the negation as failure rule, just as SLDNF resolution is based on SLD resolution, augmented with the negation as failure rule. Its characteristic is the simplicity of the implementation.

We have imposed some conditions on stratified programs such as the bounded term size property and OLD-safe for negation. Although we do not suppose that these conditions are too restrictive from the practical point of view, especially for database applications, finding weak syntactical (sufficient) conditions for stratified programs to ensure the correctness of the evaluation algorithm is a research area in it own right.

Finally, for implementation issues, an interpreter based on OLDT-resolution has been already implemented on DEC-20 written in Prolog [KK87]. Hence, it is quite straightforward to implement our evaluation algorithm simply by augmenting the negation as failure rule. Although the proposed evaluation algorithm computes an answer to a given query in almost top-down manner, the bottom-up computation methods like Magic Sets/Alexander Method is shown to be extended also to stratified programs [SI88].

Compared with previous work, the contributions of this paper could be summarized as follows :

1) The semantics under the ECWAs is introduced, and its relation with other formalisms such as the minimal model and the tight tree semantics is examined.

2) An evaluation algorithm under the ECWAs is proposed, which is quite a simple extension of OLDT resolution, augmented only with negation as failure rule.

3) A class of stratified programs is specified, for which OLDTNF-resolution is shown to be sound and complete. The class is broader than the previously proposed ones [ABW86].

Acknowledgement

The authors wish to express their thanks to Kazuhiro Fuchi (Director of ICOT) for providing us with the opportunity to pursue this research, and Koichi Furukawa (Vice Director of ICOT) for his advice and encouragement. We would also like to thank Tadashi Kanamori (Mitsubishi Electric Corp.) and Allen Van Gelder (Univ. of California, Santa Cruz) for valuable comments on our earlier draft.

References

[ABW86] K.R. Apt, H. Blair, and A. Walker. Towards A Theory of Declarative Knowledge. In J. Minker, editor, *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, pages 546–623, 1986. Washington, DC.

[BR86a] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. of the ACM-SIGMOD Conference*, pages 16–52, 1986. Washington, DC.

[BR86b] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proc. Fifth ACM Symposium on Priciples of Database Systems*, pages 269–284, 1986.

[Cla78] K.L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Database*, pages 293–322, Plenum Press, 1978.

[Die87] S.W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Proc. 1987 Symposium on Logic Programming*, pages 264–272, IEEE Computer Society, 1987.

[Gal87] H. Gallaire. Boosting Logic Programming. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 962–988, Melbourne, 1987.

[Gel87] M. Gelfond. On Stratified Autoepistemic Theories. In *Proc. AAAI-87*, pages 207–211, 1987.

[GP86] M. Gelfond and H. Przymusinska. Negation as Failure: Careful Closure Procedure. *J. Artificial Intelligence*, 30:273–287, 1986.

[GPP86] M. Gelfond, H. Przymusinska, and T. Przymusinski. The Extended Closed World Assumption and Its Relationship to Parallel Circumscription. In *Proc. Fifth ACM Symposium on Principles of Database Systems*, pages 133–139, 1986.

[KK87] T. Kanamori and T. Kawamura. *Analyzing Success Patterns of Logic Programs by Abstract Hybrid Interpretation.* ICOT Technical Report TR-279, ICOT, 1987.

[Lif86] V. Lifschitz. On the Declarative Semantics of Logic Programs with Negation. In J. Minker, editor, *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, pages 420–432, 1986. Washington, DC.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming.* Springer, 1984.

[LST86] J.W. Lloyd, E.A. Sonenberg, and R.W. Topor. *Integrity Constraint Checking In Stratified Databases.* Technical Report 86/5, Dept. of Computer Science, Univ. of Melbourne, 1986.

[Min82] J. Minker. On indefinite data bases and the closed world assumption. In *Proc. Sixth Conference on Automated Deduction*, pages 292–308, Lecture Notes in Computer Science 138, Springer, Berlin, 1982.

[Prz86a] T. C. Przymusinski. On the Semantics of Stratified Deductive Databases. In J. Minker, editor, *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, pages 433–443, 1986. Washington, DC.

[Prz86b] T. C. Przymusinski. Query Answering In Circumscriptive And Closed-World Theories. In *Proc. AAAI-86*, pages 186–190, 1986.

[Rei78] R Reiter. On Closed World Data Bases. In H. Gallaire and J. Minker (Eds.), editors, *Logic and Database*, pages 55–76, Plenum Press, 1978.

[RLK86] J. Rohmer, R. Lescouer, and J.M. Kerisit. The Alexander Method — A Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing*, 4(3):273–285, 1986.

[She84] J.C. Shepherdson. Negation as Failure: A Comparison of Clark's Completed Data Base and Reiter's Closed World Assumption. *J. Logic Programming*, 1:51–79, 1984.

[SI88] H. Seki and H. Itoh. *On the Power of Continuation Passing: A Recursive Query Processing Method for Stratified Databases.* ICOT Technical Report, ICOT, 1988. in preparation.

[SZ87] D. Sacca and C. Zaniolo. Implementation of Recursive Queries for a Data Language Based on Pure Horn Logic. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 104–135, Melbourne, 1987.

[TS86] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, London, 1986.

[VG86] A. Van Gelder. Negation as Failure Using Tight Derivations for General Logic Programs. In *Proc. 1986 Symposium on Logic Programming*, pages 127–138, IEEE Computer Society, 1986.

[Vie87] L. Vieille. A Database-complete Proof Procedure Based on SLD-resolution. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 74–103, Melbourne, 1987.

Appendix: Proofs

Lemma 2.1

Proof: The proof is by induction on l. When $l = 0$, it is clear, since P_0 is a set of Horn clauses (see e.g., [Llo84]). Assume that the proposition holds for $l = i$ $(0 \leq i \leq k-1)$. From the definition of E_i and the induction assumption, it is easily seen that SS^{fp}_{i+1} is an Herbrand model of E_i. From Proposition 1 of [LST86], SS^{fp}_{i+1} is also a model of P_{i+1}. Hence, SS^{fp}_{i+1} is an Herbrand model of $E_i \cup P_{i+1}$, which proves the first part of the lemma. As for (ii) of the lemma, the only if-part is obvious, since SS^{fp}_{i+1} is a model of $E_i \cup P_{i+1}$. The if-part can be also easily proved from the definition of SS^{fp}_{i+1}. □

Lemma 4.1

Proof: The proof is by induction on the stratified level of atom G. Suppose that the level of G is 0. From the bounded term size property, the length of a solution list is bounded by a constant. Hence, the branching factor of each lookup node is bounded by a constant. The length of a path is also bounded by a constant, since the number of solution nodes is bounded by a constant and every lookup node in a path decrease the number of atoms in the label by one. Thus, the size of the OLDTNF structure is bounded by a constant. Next, suppose that the lemma holds for level i $(0 \leq i < k)$, where k is the maximum level of stratification. Note that, like a lookup node, every negative node in a path decrease the number of atoms in the label by one, and that each OLDTNF structure created by negation as failure rule of the immediate extension for a negative node is finite from the induction assumption. Then, the proof is similar to that of the base case. □

In order to prove **Lemma 4.2**, the following two definitions are introduced.

DEFINITION A-1 {size of a ground atom, size of a sequence of ground atoms}

Suppose that a positive ground atom A is in SS (the success set). Since A is also in SS^{fp}, there exists at least one tight NF-tree (defined in [VG86]) for A. Then, the *size* of a ground positive atom A is the minimal number of the nodes of such tight NF-trees for A, plus the total number of its leaves (note that, when a ground positive atom A has a unit clause $\leftarrow L$ such that A is an instance of L, the size of

A is 2). If a ground positive atom A is in FS, then the *size* of $\neg A$ is defined to be 2. The *size of a null clause* is defined to be 1.

The *size* of a sequence of ground atoms $(A_1, ..., A_n)(n \geq 0)$ is defined to be the summation of each size of A_i. □

Lemma 4.2

Proof: (Sketch) We use induction on the level of an atom G_0. Suppose that the level of G_0 is 0. In this case, clauses used in OLDTNF-refutation are Horn clauses. The proof of (**Soundness a**) and (**Completeness**) is obvious from the soundness and the completeness of OLDT-refutation [TS86] and from the fact that SS_0 is equivalent to the least Herbrand model of programs of level 0. From **Lemma 4.1**, the proof of (**Soundness b**) is also obvious.

Next, suppose that the lemma holds for any atom G_0 of the level less than r. Let the level of an atom G_0 be $(r+1)$. The proof of (**Soundness a, b**) are omitted, since they are not difficult. As for the proof of (**Completeness**), we use again induction on the triple (S, T, v), where S is the size of a sequence of goals $G'_1, ..., G'_n$ and T is an OLDTNF structure and v is a node in T. The well-founded ordering is defined as follows:

(S, T, v) precedes (S', T', v')

iff $S < S'$, or

$S = S'$ and v' is a lookup node but v is not.

(Induction Basis) When $L = 1$, then the proposition is trivial, since a goal is a null clause.

(Induction Step) When G_1 is a negative atom, $G_1\theta$ in v is ground from the assumption of the OLD-floundering free query. Let $G_1\theta$ be of the form : $\neg A'_1$, where A'_1 is a ground atom. Since A'_1 is in FS and the level of A'_1 is less than r, the OLDTNF-tree for A'_1 is finitely failed from (**Soundness b**). Hence, v has the immediate descendant node v_1 labelled with $\leftarrow G_2\theta, ..., G_m\theta$. Then, The proposition is immediate from the induction assumption.

Next, consider the case where G_1 is a positive atom. We consider two cases depending on whether the node v is a solution node or not.

(case a): Suppose that v is a solution node. Since G'_1 is in SS, there exists a tight NF-tree Tr_1 for G'_1. Let $L'_1, ..., L'_k$ be a sequence of ground atoms of all the children of G'_1 in Tr_1. From the definition of an NF-tree, there exists a clause C in P of the form: $L_0 \leftarrow L_1, ..., L_k$ such that $G'_1 \leftarrow L'_1, ..., L'_k$ is an instance of $L_0 \leftarrow L_1, ..., L_k$. Since G'_1 is also an instance of $G_1\theta$, $G_1\theta$ and L_0 are unifiable. Hence, $G_1\theta$ and C are resolvable. Let τ_0 be the mgu of $G_1\theta$ and L_1, and v_1 be the child of v labelled with $\leftarrow (L_1, ..., L_k)\tau_0, (G_2, ..., G_m)\theta \circ \tau_0$. Since the size of $(L'_1, ..., L'_k, G'_2, ..., G'_n)$ is less than that of $(G'_1, G'_2, ..., G'_n)$, from the induction assumption, there exists an extension T_1 of T such that T_1 contains an OLDTNF subrefutation $\leftarrow (L_1, ..., L_k)\tau_0, (G_2, ..., G_m)\theta \circ \tau_0$ which starts from v and the solution of which subsumes $(L'_1, ...L'_k, G'_2, .., G'_m)$. Hence, the proposition is immediate.

(case b): Suppose that v is a lookup node. Then, there is a corresponding solution node v_s in T which is of the form: $\leftarrow G_1\lambda, \Delta$ where G'_1 is an instance of $G_1\lambda$ and Δ is a (possibly empty) sequence of goals. Since the size of G'_1 is less than or equal to the size of $(G'_1, ..., G'_n)$ and v_s satisfies the condition of the above lemma, it follows from the induction assumption that there exists an extension T' of T such that contains a subrefutation of $\leftarrow G_1\lambda$ which starts from v_s and whose solution subsumes G'_1. When we denote its solution by $G_1\lambda'$, the solution list of $G_1\lambda$ in T' includes $G_1\lambda'$. Since G'_1 is an instance of both $G_1\theta$ and $G_1\lambda'$, hence a goal $\leftarrow G_1\theta, ..., G_m\theta$ and a unit clause $G_1\lambda' \leftarrow$ are resolvable (let its substitution be τ_0), v has a child v_1 labelled with $\leftarrow (G_2, ..., G_m)\theta \circ \tau_0$. Noting that the size of $(G'_2, ..., G'_m)$ is less than $(G'_1, G'_2, ..., G'_m)$, again by induction hypothesis we have an extension T'' which contains an subrefutation s of $\leftarrow (G_2, ..., G_n)\theta \circ \tau_0$ which starts from v_1 and whose solution subsumes $G'_2, ..., G'_n$. The path in T'' starting from v and followed by the subrefutation s constitute the required subrefutation of $\leftarrow (G_1, ..., G_n)\theta$. $\square$