Order Number 9029978

On the completeness of SLDNF-resolution

Shen, Zhizhang, Ph.D.

City University of New York, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

ON THE COMPLETENESS OF SLDNF-RESOLUTION

by

Zhizhang Shen

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1990

This manuscript has been read and accepted for the Graduate faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

4/19/90 — Howard C. Wennerman

Date — Chair of Examining Committee

April 20, 1990 — T C Wendtkamper

Date — Executive Officer

Professor Michael Anshel (CCNY)

Professor Melvin Fitting (Lehman)

Professor Ken McAloon (Brooklyn)

Supervisory Committee

The City University of New York

ABSTRACT

ON THE COMPLETENESS OF SLDNF-RESOLUTION

by

Zhizhang Shen

Adviser: Professor Howard C. Wasserman

By the completeness of logic programming, we mean that all inferences supported by the declarative semantics are also supported by the procedural semantics. In the context of this dissertation, the completed program semantics is the declarative semantics and SLDNF-resolution is the procedural correspondent.

In this dissertation, after reviewing the basic syntactical and semantical issues for first order logic, and in particular, for logic programs, we carry out a systematic study of completeness for logic programming: we present some causes for incompleteness of SLDNF-resolution with respect to

the completed program semantics; we then review a series of completeness results for several related classes of logic programs and present an extension of a known completeness result.

Moreover, we put forward a new approach towards obtaining completeness results: we begin with an interesting example of incompleteness, and go on to develop the idea of the coincidence of two semantics, one defined in terms of the so-called declaratively-relevant part of a logic program with respect to a goal, and the other defined in terms of the so-called procedurally-relevant part. Based on the close relationship between the semantic coincidence and completeness, we provide a new characterization of the completeness of SLDNF-resolution with respect to the completed program semantics. Finally, we give an effectively decidable condition equivalent to completeness, under a set of reasonable restrictions on the programs and goals. As the condition is in no means necessary, it is expected that yet weaker conditions may be obtained following this approach.

DEDICATED TO

MY YOUNG COLLEAGUES AND FRIENDS IN WUHAN UNIVERSITY OF WATER TRANSPORTATION AND ENGINEERING, WUHAN, PEOPLE'S REPUBLIC OF CHINA.

ACKNOWLEDGEMENT

At this moment, I have so many people to thank that I feel it is impossible to list them all.

I want to thank my young colleagues and friends in the Wuhan University of Water Transportation and Engineering, to whom this dissertation is dedicated, for the friendship and encouragement they showed me when I was among them, as a lonely person far away from home. I wish them well.

I want to thank the leaders of the Wuhan University of Water Transportation and Engineering, in particular, those of the Department of Electric Engineering and Computer Science. Without their recommendation, the State Committee on Education of the People's Republic of China would not select me as a visiting scholar and send me to the United States in 1985.

I want to thank the Department of Computer Science of Queens College of the City University of New York. The Department accepted me as a visiting scholar and has been displaying generous hospitality in this 5 year period.

I want to thank Dr. Rootenberg and Dr. Brown, Chairmen of the Department of Computer Science of Queens College; Dr. Beckman and Dr. Wesselkamper, Executive Officers of the Ph.D. Program in Computer Science in the Graduate Center of the City University of New York and Dr. Wasserman, my dissertation advisor. Without their recommendation, I would not possibly receive the financial support from Queens College and the Graduate Center of CUNY in the form of University Fellowship, Graduate Scholarship, Graduate Assistantship and Adjunct Lectureship in the past 4 years. I also want to thank the Chinese Government for the financial support they provided in the period of 1985-1986.

During this 5 year period, I have been helped by many professors in both Queens College and the CUNY Graduate Center. I owe them a lot. In particular, I want to thank Dr. Wasserman. He is not only my dissertation advisor, but also my mentor in the past 5 years. He helps me to overcome various obstacles and always provides useful advice. During the dissertation period, he has been patient, inspiring and rigorous. He let me think more and understand more. To him, I cannot say enough thanks. I also want to thank the other members of my dissertation advisory committee: Dr. Anshel, Dr. Fitting and Dr. McAloon for their help and encouragement in their own ways.

My father is always a tough teacher to me. He is always confident on my success, as well. Thank you, father, for your hard push. It worked.

My mother didn't think that I could make it, until convinced by my father. Well, I didn't think that I could make it either, until convinced by myself. It is difficult, but easier than I once thought. Thank you, mother, for your sincere care. It helped.

Final words go to my wife: not just thanks but love as well.

Zhizhang Shen

Queens College

May 1990

CONTENT

Chapter 3. CURRENTLY KNOWN COMPLETENESS RESULTS.

Chapter 4. AN EXTENSION OF A COMPLETENESS RESULT.

Chapter 5. A NEW APPROACH TOWARDS OBTAINING COMPLETENESS.

Chapter 1. LOGIC PROGRAMS: SYNTAX AND SEMANTICS.

A logic program is a set of special first order sentences. Accordingly, a brief presentation of the syntax and semantics of first order languages and logic will be a very helpful preamble. We provide this in section 1.1 and 1.2.

Among logic programs, the class of definite programs is already well studied. In section 1.3, some basic results for this important but quite restricted class of logic programs will be given. In order to have a more expressive syntax than that of definite programs, we need to support negation. But this cannot easily fit into the frame of classical semantics because of the need for the efficiency of computation. In sections 1.4 and 1.5, we will discuss the problem of implementing negation in logic programming; in particular, we will discuss Negation-as-Failure and Clark's completed program semantics, which supports Negation-as-Failure.

1.1. Syntax of First Order Languages.

A first order language, L, is based on an alphabet of variable symbols, constant symbols, function symbols,

predicate symbols, connectives and quantifies, with the constant, function, and predicate symbols considered the "non-logical" part of the alphabet.

There is a countably infinite number of variables, zero or more n-ary function symbols and one or more n-ary predicate symbols for at least one n (constant symbols being regarded as 0-ary function symbols).

It's well known that the set of "$\neg$"(negation) and "$\wedge$" (conjunction) and the set of "$\neg$" and "$\vee$"(disjunction) are two complete connective sets([22]). As all of the other conventional connectives such as "$\leftarrow$"(implication) and "$\leftrightarrow$"(equivalence) are commonly used in the practice of logic programming, we allow all of them in a program and they follow the standard operational rules. The same thing pertains to the two quantifies "$\forall$"(for all) and "$\exists$" (there exists); i.e. one of those is sufficient, but we assume the use of both.

Although it is sufficient to have just those symbols as mentioned above([32]), it is convenient to also have such punctuation symbols as comma(,) and period(.), as well as parenthesis("(" and ")").

Definition 1.1.1. Let L be a first order language. By an expression in L, we mean a finite concatenation of symbols (from the alphabet) of L, including the null sequence. Variable-free expressions are known as ground expressions.

Most expressions make no sense. Among those which do make sense, are terms and formulas.

A term can be used to specify an individual.

Definition 1.1.2. Let L be a first order language. By a term in L, we mean either a variable symbol, a constant symbol, or an expression of L of the form: $f(t_1,\ldots,t_n)$, where f is an n-ary function symbol of L and $t_1,\ldots, t_n$ are terms of L, $n \geq 1$.

For example, let '+' be a function symbol of L, and let both '1' and '2' be constant symbols; then +(1,2) is a term, more usually written as 1+2.

An atomic formula can be used to express a simple property of individuals.

Definition 1.1.3. Let L be a first order language. By an atomic formula (or atom) of L, we mean an expression in L of the form $p(t_1,\ldots,t_n)$, where p is an n-ary predicate symbol of L and $t_1,\ldots t_n$ are terms of L, $n \geq 1$.

For example, let '>' be a predicate symbol in L, then >(+(1,2), 2) is an atomic formula, usually written as (1+2)>2.

Definition 1.1.4. Let L be a first order language. By a literal of L, we mean an atomic formula of L, or the negation of an atomic formula of L.

Now, we are in the position to define the notion of formula of a first order language, which can be used to express a general property of individuals.

Definition 1.1.5. Let L be a first order language. By a formula of L, we mean an atomic formula of L, or an expression of L in one of the following forms: $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \leftarrow \phi_2$, $\phi_1 \leftrightarrow \phi_2$, $\exists x\phi_2$ or $\forall x\phi_2$, where both ϕ_1 and ϕ_2 are formulas of L, and x is a variable.

For example, $((1+2) \wedge (\neg(2>1)))$ is a formula.

Definition 1.1.6. Let A and C be expressions. By saying that C occurs in A, we mean that A is the concatenation of B, C and D, in that order, for some expressions B and D. In this case, we also call C a sub-expression of A. If C and A are both formulas (resp. terms), then C is called a sub-formula (resp. sub-term) of A.

For example, 2 occurs in 1+2.

Definition 1.1.7. Let x be a variable, and let ϕ be a formula. Every occurrence in ϕ of x within a sub-formula of the form $\exists x\phi_1$ or $\forall x\phi_1$ is called a bound occurrence in ϕ; every other occurrence of x in ϕ is called a free occurrence in ϕ.

For example, in $\forall x\ p(x, y)$, x is bound but y is free. In the formula $(\forall x\ r(x) \leftarrow p(x, y))$, the only occurrence of y is free and the third occurrence of x is free, but the first two occurrences of x are bound.

Definition 1.1.8. Let ϕ be a formula. By saying ϕ is closed, we mean that no variable in ϕ has a free occurrence in ϕ. A closed formula is also called a sentence.

For example, $\forall x \forall y\ p(x, y)$ is a sentence, but $\forall y\ p(x, y)$ is not.

Definition 1.1.9. Let ϕ be a formula. By the universal closure of ϕ, $\forall(\phi)$, we mean the formula obtained from ϕ by prefixing a universal quantifier for every variable with a free occurrence in ϕ.

The existential closure $\exists(\phi)$ is similarly defined.

For example, the universal closure of $r(x,y) \leftarrow (\exists x\ p(x, y))$ is $(\forall x(\forall y(r(x) \leftarrow (\exists x\ p(x, y)))))$.

Henceforth, when no ambiguity is possible, we shall omit many of the parentheses from formulas. For example, the preceding formula will be written $\forall x \forall y(r(x) \leftarrow \exists x\ p(x, y))$.

Now, we are ready to define the syntactical structure of logic programs. We begin by defining the notion of "clause".

Definition 1.1.10. Let $A_1, \ldots, A_n, B_1, \ldots, B_m$ be atomic formulas. We call the universal closure of the following formula a clause: $A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m$.

Usually, a clause is written as $A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$, with the understanding that each comma(,) occurring on the left hand side of "$\leftarrow$" means a disjunction and each comma(,)

occurring on the right hand side means a conjunction.

The above representation of a clause is justified by the fact that the two involved expressions are equivalent to each other.

We also say that the A_i's constitute the head of the clauses, and the B_j's constitute the body.

Definition 1.1.11. By a general logic program, we mean a finite set of clauses.

1.2. Semantics of First Order Languages.

Given a set of formulas in a first order language, L, the focal point of the semantics of L is the idea of "interpretation", by which we specify the meanings of the non-logical symbols of L.

Definition 1.2.1. By an interpretation of a first order language L, we mean an ordered pair <D, I>, where D, called the domain of the interpretation, is a non-empty set of individuals and I is a mapping which assigns meaning to each of the function symbols and predicate symbols in L. More precisely, for any n-ary function symbol, f, I will map f into an n-place operation on D; for any n-ary predicate symbol, p, I will map p into an n-place relation on D.

Definition 1.2.2. By a variable assignment with respect to an interpretation, we mean a mapping from the set of all variables to the domain of that interpretation.

Given an interpretation <D, I> of L and a variable assignment, A, with respect to <D, I>, there is a uniquely determined "valuation", $Val_{I,A}$, which assigns values in D to terms of L and truth values to formulas of L in a standard way. For any two variable assignments A_1 and A_2 and any sentence, ϕ, of L, $Val_{I,A1}(\phi) = Val_{I,A2}(\phi)$, and thus we write $Val_I(\phi)$ to stand for the unique truth value assigned under I, using any variable assignment.

We have the following:

Definition 1.2.3. Let ϕ be a first order sentence, and let <D, I> be an interpretation. By saying that <D, I> is a model of ϕ, we mean that ϕ is mapped to true under <D, I>; i.e. $Val_I(\phi)$ = true.

Let S be a set of sentences. By saying that an interpretation is a model of S, we mean that it's a model of every sentence in S.

Definition 1.2.4. Let ϕ be a sentence, and let S be a set of sentences. By saying that ϕ is a logical consequence of S, denoted by $S \models \phi$, we mean that every model of S is also a model of ϕ.

Definition 1.2.5. Let S be a set of formulas. By saying

that S is satisfiable, we mean that S has a model. We say that S is unsatisfiable, if it is not satisfiable.

Theorem 1.2.1. Let ϕ be a sentence, and let S be a set of sentences. Then ϕ is a logical consequence of S if and only if(iff) $S \cup \{\neg\phi\}$ is unsatisfiable.

The theorem tells us that in order to confirm that ϕ is a logical consequence of a set of closed formulas, we need to show a set of closed formulas is unsatisfiable. But following the definition of unsatisfiability, one would have to check every possible interpretation, an infeasible task, as there is a denumerable number of possible interpretations.

Fortunately, it turns out that in some cases we only need to pay attention to a limited class of interpretations, the so called Herbrand interpretations.

Definition 1.2.7. Let L be a first order language. By the Herbrand Universe U_L for L, we mean the set of all ground terms of L (with one constant symbol adjoined to L in case L has no constant symbols).

Definition 1.2.8. Let L be a first order language. By the Herbrand Base for L, we mean the set of all ground atoms of L.

Definition 1.2.9. Let L be a first order language. By a Herbrand interpretation for L, we mean an interpretation for L which has as its domain, the Herbrand Universe for L, and maps every constant symbol in L to itself, and, if f is any n-

ary function symbol in L, then the interpretation maps f onto the n-place operation, H(f), on U_L such that for all $t_1,\ldots,t_n$ in U_L, $H(f)(t_1,\ldots,t_n)=f(t_1,\ldots,t_n)$.

As the interpretation of constants and functions in a given language is uniquely determined, a Herbrand interpretation is usually represented as a subset of the associated Herbrand base.

Definition 1.2.10. Let S be a set of sentences in a first order language L, and let <D, I> be a Herbrand Interpretation for L. If <D, I> is a model of S, then <D, I> is called a Herbrand model of S.

The importance of Herbrand model is reflected in the following lemma and theorem([21]).

Lemma 1.2.2. Let S be a set of clauses and suppose that S has a model, then s has a Herbrand model.

Theorem 1.2.3. Let S be a set of clauses. Then S is unsatisfiable if and only if S has no Herbrand model.

Now, the question of whether a sentence ϕ is a logical consequence of a given general logic program is answerable by checking whether or not the set of clauses consisting of the program, together with the negation of the original query, ϕ, is unsatisfiable; moreover, if ϕ is of the form $\exists(B_1\wedge\ldots\wedge B_m)$, then $\neg\phi$ is equivalent to $\leftarrow B_1,\ldots,B_m$, and thus from the previous discussion of Herbrand models, it follows that we need only check whether there is a Herbrand model for the previously

mentioned set.

1.3. Semantics for Definite Programs.

In the last section, we discussed general semantic issues for first order language; in particular, for general logic programs. As a logic program is a special set of first order sentences, some relevant issues can be simplified. In this section, we will discuss the simplest of all of logic programs, the class of definite programs.

Definition 1.3.1. Let P be a logic program. By saying that P is definite, we mean each clause of P has exactly one positive literal; i.e. each clause may be written in the form $A \leftarrow B_1,..,B_m$.

For example, the following program implements the concatenation of two lists:

append(nil, x, x)

append(x.y, z, x.u) ← append(y, z, u)

The above program has the following intended semantics: the concatenation of an empty list and any list x is just list x; and the concatenation of list x.y(i.e., x is the first item while y is the rest of the list) and list z is a list with x as the first item and u as the rest if the concatenation of y and z is u.

Definition 1.3.2. By a definite goal we mean a clause consisting entirely of (one or more) negative literal, i.e. a clause which may be written in the form $\leftarrow A_1,..,A_k$, with A_i's being atoms.

For example, with the goal ←append(1.2,3.4, 1.2.3.4), we want to confirm whether the list (1 2 3 4) is the concatenation of list (1 2) and list (3 4), given the logic program for 'append'.

More generally, we can also find out values for some unknown variables in a goal, which is more interesting and important from the programming point of view.

1.3.1. Least Model Semantics.

In speaking of assigning a declarative semantics to a program, we mean to specify which ground atoms in the language of the program will be regarded as true and which ground atoms will be regarded as false. There is no controversy in this matter for a definite program.

As a logic program is a set of first order formulas, it is quite natural to assign a declarative semantics using the already well-developed first order model theory. This approach is particularly appropriate for the class of definite programs. Every definite program has at least one Herbrand model (the associated Herbrand base). Moreover, it must have a unique least Herbrand model. This is the intersection of all

Herbrand models for that definite program. It can be proved that this intersection is still a model([1]) and obviously a subset of every other Herbrand model of this program.

Definition 1.3.3. Let P be a definite program. By M_p we mean the smallest Herbrand model for P.

M_p, being a Herbrand model, consists of a set of ground atoms. It has been shown that it is exactly the set of ground atoms which are logical consequences of program P([21]). Therefore, M_p is associated with P as its intended meaning, or declarative semantics.

M_p can also be characterized by using a fixed point approach. It has been shown to be identical with the least fixed point of an operator, the so-called T-operator, associated with a definite program([1],[9],[21]).

The "Introduction" of Lifschitz([20]) gives a simple example and a clear description of the various approaches to the assignment of a declarative semantics: Logical consequences of the program, least Herbrand model of the program, and the least fixed point of the operator associated with the program and their relationship.

Apt([2]) and Emden([9]) are more detailed and are classical references. Przymusinski([24],[25]) pointed out that

when the involved query is not simply existential, then the least model semantics could lead to some problems, the so-called "the universal query problem".

1.3.2. SLD-Resolution.

After one assigns a declarative semantics to the class of definite programs, one would like to provide an effective computational procedure to discover those ground atoms which are true according to the semantics.

Suppose we have found such a procedure. If we regard the set of ground atoms for which the answer from the procedure is positive as true and the set of ground atoms corresponding to negative answers as false, we would have yet another way to assign semantics to the program. As that kind of semantics is associated with a procedure, it is called a procedural semantics.

The procedural semantics assigned to the class of definite programs is based on SLD-resolution(Linear resolution with Selection procedure for Definite program).

We have the following definitions(from Lloyd([21])):

Definition 1.3.4. By Θ, a substitution, we mean a mapping from the set of variables to the set of terms. A substitution is usually represented as a finite set of bindings, in the form of $\{v_1/t_1,\ldots,v_n/t_n\}$, which means that v_i is mapped to t_i

for all i, $1 \leq i \leq n$; for all the other variables, they are just mapped to themselves.

Definition 1.3.5. Let $\Theta=\{v_1/t_1, \ldots, v_n/t_n\}$ be a substitution and let E be an expression. By $E \cdot \Theta$, the instance of E by Θ, we mean the expression obtained from E by simultaneously replacing each occurrence of the variable v_i in E by the term t_i.

Let S be a set of expressions, i.e. $S=\{E_1, \ldots, E_n\}$; and let Θ be a substitution. By $S \cdot \Theta$, we mean the set obtained by applying Θ to each of the E_i's, i.e., $S \cdot \Theta=\{E_1 \cdot \Theta, \ldots, E_n \cdot \Theta\}$

Definition 1.3.6. Let $\Theta=\{u_1/s_1, \ldots, u_m/s_m\}$ and $\sigma=\{v_1/t_1, \ldots, v_n/t_n\}$ be substitutions. By $\Theta \cdot \sigma$, the composition of Θ and σ, we mean the substitution obtained from the following set: $\{u_1/s_1 \cdot \sigma, \ldots, u_m/s_m \cdot \sigma, v_1/t_1, \ldots, v_n/t_n\}$ by deleting any element $u_i/s_i \cdot \sigma$, in which $u_i=s_i \cdot \sigma$ and deleting also any element v_j/t_j for which $v_j \varepsilon \{u_1, \ldots, u_m\}$.

Definition 1.3.7. Let S be either a finite set of terms or a finite set of atoms. A substitution, Θ, is a unifier for S if $S \cdot \Theta$ is a singleton.

A unifier Θ is said to be a most general unifier (mgu) for S, if for each unifier σ of S, there exists a substitution δ such that $\sigma = \Theta \cdot \delta$.

Definition 1.3.8. Let E and F be expressions. By saying that E is a variant of F, we mean that there is a substitution θ such that $E=F\cdot\theta$.

Now, we define the meaning of a single SLD-resolution step.

Definition 1.3.9. Let C be a definite clause $A \leftarrow B_1,\ldots,B_q$, and let G be a definite goal $\leftarrow A_1,\ldots,A_m,\ldots,A_k$, which shares no variables with C. Then G' is derived from G and C using mgu θ if the following hold:

(a) A_m is an atom, call the selected atom in G;

(b) θ is an mgu of A_m and A;

(c) G' is the goal $\leftarrow(A_1,\ldots,A_{m-1},B_1,\ldots,B_q,A_{m+1},\ldots,A_k)\cdot\theta$.

The following is the definition of the SLD-derivation process.

Definition 1.3.10. Let P be a definite program, and let G be a definite goal. An SLD-derivation of $P\cup\{G\}$ consists of a sequence $G_0=G,G_1,\ldots$ of goals, a sequence $C_1,C_2,\ldots$ of variants of program clauses of P and a sequence $\theta_1,\theta_2,\ldots$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

Definition 1.3.11. Let P be a definite program, and let G be a definite goal. An SLD-refutation of $P\cup\{G\}$ is a finite SLD-derivation of $P\cup\{G\}$ which has the empty clause as the last goal in the derivation.

Note: Procedurally, an empty clause includes no sub-goals, so we have nothing to solve; declaratively, it denotes a logical falsehood, so the set involved is unsatisfiable by the resolution principle.

An SLD-refutation is also called a successful SLD-derivation. A (finitely) failed SLD-derivation is one that terminates with a non-empty clause such that the selected atom doesn't unify with the head of any program clause. Both failed and successful SLD-derivation are finite. Infinite SLD-derivations are also possible. For example, if we have a program consisting of only one clause, i.e. $p \leftarrow p$ and the goal $\leftarrow p$; then the derivation for this program and goal is infinite.

Given a program P, and a goal,G, we can also define the notion of SLD-tree as follows:

An SLD-tree for $P \cup \{G\}$ is a directed tree with G as the root. Every node in the SLD-tree is a goal with a particular literal selected. There is an edge from a node G_1 to another node G_2 provided G_2 is derivable from G_1 and some clause C in P by an SLD-resolution step.

It is obvious that in an SLD-tree corresponding to a program, P and a goal, G, any path starting from G and ending with the empty goal (which must then be a leaf of the tree)

corresponds to an SLD-refutation of P∪{G}; any path starting from G and ending with a non-empty goal for which the selected atom doesn't unify with the head of any program clause (which is thus a leaf in the tree) corresponds to a (finitely) failed SLD-derivation of P∪{G} and any infinite path starting from G corresponds to an infinite SLD-derivation of P∪{G}. It is well known([21]) that any two SLD-trees for the same program and goal will have the same number of branches which terminate with the empty goal(i.e. success branches).

1.3.3. Correspondence between the Two Approaches.

In the last two sub-sections, we defined declarative and procedural semantics for the class of definite programs. Now, we examine the relationship between them.

Definition 1.3.12. Let P be a definite program, and let G be a definite goal. An answer for P∪{G} is a substitution for all the free variables in G.

The following two definitions are fundamental in discussing the relationship between the procedural semantics and the declarative semantics.

Definition 1.3.13. Let P be a definite program, G be a definite goal $\leftarrow A_1,\ldots,A_m$, and let Θ be an answer for P∪{G}. By saying that Θ is a correct answer for P∪{G}, we mean that $\forall((A_1\wedge\ldots\wedge A_m)\cdot\Theta)$ is a logic consequence of P.

Definition 1.3.14. Let P be a definite program, and let G be a definite goal. By saying that Θ is a computed answer for P∪{G}, we mean that Θ is the substitution obtained by restricting the concatenation $\Theta_1 \cdot \ldots \cdot \Theta_n$ to the variables of G, where $\Theta_1, \ldots, \Theta_n$ is the sequence of mgu's used in an SLD-refutation of P∪{G}.

Now, we exhibit the intimate relationship between SLD-resolution and the first order model semantics for definite programs([2],[15],[21]).

Theorem 1.3.1. Let P be a definite program and let G be a definite goal. Then every computed answer for P∪{G} is a correct answer for P∪{G}.

Theorem 1.3.2. Let P be a definite program and let G be a definite goal. Then every correct answer for P∪{G} is an instance of a computed answer of P∪{G}.

1.4. Semantics for Normal Programs.

In last section, we discussed the class of definite programs and the associated declarative and procedural semantics. As we observed there, for that class of logic programs, the two semantics fit to each other perfectly: everything supported by the declarative semantics is supported by the procedural one, and vice versa.

But, the class of definite programs is quite limited, as no occurrence of negation in the bodies of clauses is allowed. However, one often needs to express negation in a logic program. For example, let X and Y be two sets. In order to test whether they are different, we can use the following logic program:

diff(x,y) ← member(u,x), ¬member(u,y)

diff(x,y) ← member(u,y), ¬member(u,x)

We present a definition of a highly useful class of programs which is more general than the class of definite programs:

Definition 1.4.1. A normal clause is any clause with at least one positive literal. It is usually written in the form $L \leftarrow L_1,\ldots,L_n$, where L is an atom and the L_i's are arbitrary literals.

A normal program is a finite set of normal clauses.

Definition 1.4.2. By a normal goal, we mean any clause written in the form $\leftarrow L_1,\ldots,L_k$, where the L_i's are arbitrary literals.

When we add negation in logic programming, there are two things we have to consider: an effective computation rule to implement negation and a proper semantics to support that rule.

1.4.1. Negation and SLDNF-Resolution.

From the point of view of the logician, the ideal way to include negation in logic programming is to implement classical negation, which is well defined and understood. But, the negation usually used in logic programming is not the classical one, as that won't lead to an efficient procedure.

There are various practical approaches available, such as Clark's Negation as Failure([8]), Reiter's Closed World Assumption([26]), Gabbay's Negation as Inconsistency([14]), Fitting's Negation as Refutation([12]), and many others. Each of them has its own advantages and drawbacks.

Arguably, Negation as Failure is one of the most popular approaches. Under this computational rule, a sentence is false if every possible proof leads to failure which can be tested in finite time. Although it is weaker than some of the other alternative rules for negation, it's very efficient and is supported by a natural semantics, i.e. Clark's completed program semantics. Thus, it finds wide application in logic programming and database practice. It has been implemented in the popular logic programming language PROLOG, as an augmentation to SLD-resolution to deal with negative sub-goals. The augmented computational procedure is called SLDNF-

resolution(SLD-resolution plus Negation as Failure).

In SLDNF-resolution, the technique for dealing with positive sub-goals is exactly the same as that given in SLD-resolution. The only difference is that if the selected sub-goal is negative and ground, say $\neg p(t)$, we will recursively carry out SLDNF-resolution with respect to the goal, $\leftarrow p(t)$ and the program, P: if the process succeeds, we will say the original goal, i.e. $\neg p(t)$ fails; otherwise, if the associated (SLDNF-)tree for $P \cup \{\leftarrow p(t)\}$ is finitely failed, we will say that the original goal succeeds. And if during the process, we meet with other negative sub-goals, the same strategy will be applied. The notion of "SLDNF-tree" is defined analogously to that of SLD-tree.

For more details of SLDNF-resolution, Lloyd's book([21]) is a classical reference.

1.4.2. Completed Program Semantics.

We need to have a declarative semantics to justify the application of SLDNF-resolution. It's well known that no negative literal could be a logical consequence of any program(even normal), as the corresponding Herbrand base, in which every ground atom is regarded as true, is a model of the program; but the negative literal couldn't be true there.

Therefore, SLDNF-resolution is not supported by the classical semantics.

It turns out that SLDNF-resolution, in particular, the Negation as Failure rule, is supported by another semantics: completed program semantics introduced by Clark([8]).

Completed program semantics is based on a syntactical transformation of the original program: basically, it replaces the conditional connective in each clause by the bi-conditional; moreover, as this transformation involves the use of the equality symbol, the completed program also includes a set of equality axioms. More formally, we have the following:

Definition 1.4.2. Let P be a normal program, let p be a predicate symbol defined in P, and let C: $p(t_1,\ldots,t_n)\leftarrow L_1,\ldots,L_m$ be a clause in P with p as the predicate in its head. By Comp(p,C,P) (the completion of p in P with respect to C), we mean the following formula:
$\forall x_1\ldots\forall x_n(p(x_1,\ldots,x_n) \leftrightarrow \exists y_1\ldots\exists y_k((x_1=t_1)\wedge\ldots\wedge(x_n=t_n)\wedge L_1\wedge\ldots\wedge L_m))$, where x_i's are new variables and y_i's are variables in the original clause.

Definition 1.4.3. Let P be a normal program, p be a predicate symbol defined in P. If there are k clauses in P with p as the predicate in the head, and the completion of p in P with respect to one of those k clauses, C_j, is the following: $\forall x_1\ldots\forall x_n(p(x_1,..,x_n) \leftrightarrow E_j)$, then by Comp(p, P) (the

completion of p in P), we mean the following formula:
$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \leftrightarrow E_1 \vee E_2 \vee \ldots \vee E_k)$.

If p is a predicate occurring in P, but there is no clause with p as the predicate in its head, then Comp(p, P) is defined as $\forall x_1 \ldots \forall x_n \neg p(x_1, \ldots, x_n)$.

As the equality symbol is involved in the definition of the completion of a predicate in a program, the following set of equality axioms is needed:

Definition 1.4.4. By Clark's equality theory, we mean the following axiom schemata(from[21]):

(1) $c \neq d$. for any distinct constant symbols.

(2) $\forall (f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m))$ for any function f, g.

(3) $\forall (f(x_1, \ldots, x_n)) \neq c$ for any function f and constant c.

(4) $\forall (t[x] \neq x)$ if t[x] contains x and is different from x.

(5) $\forall ((x_1 \neq y_1) \vee \ldots \vee (x_n \neq y_n)) \rightarrow f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n)$.

(6) $\forall (x=x)$.

(7) $\forall ((x_1=y_1) \wedge \ldots \wedge (x_n=y_n)) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, x_n)$.

(8) $\forall ((x_1=y_1) \wedge \ldots \wedge (x_n=y_n)) \rightarrow (p(x_1, \ldots, x_n) \rightarrow p(y_1, \ldots, y_n))$.

Finally, we provide the definition of the completed program:

Definition 1.4.5. Let P be a normal program. By Comp(P)(the completion of P), we mean the union of the completion of all the predicates occurring in P, and Clark's equality theory.

Now, we can say what is precisely the declarative semantics usually assigned to normal program.

Definition 1.4.6. Let P be a normal program. By the completed program semantics, we mean the semantics determined by Comp(P), i.e., a sentence is regarded as true if and only if it is a logical consequence of Comp(P).

Note: When dealing with definite programs, we only consider Herbrand models, as only a set of clauses is involved. This is no longer true when we deal with normal program, as what is involved is Comp(P) which is not a set of clauses and equality is involved as well. So, we must consider all the models of Comp(P) when we want to decide whether any sentence is a logical consequence of Comp(P). For a profound discussion, the section on The Completed Database in Shepderson's paper([31]) is suggested.

The great acceptance of the completed program semantics is due to the fact that it's natural([8]) and it supports Negation as Failure. However, this semantics also has its problems, the most crucial of which is the incompleteness of SLDNF-resolution with respect to it. In the following chapters, we will have some discussions dealing with this incompleteness problem. For a detailed discussion of other problems with this semantics, Przymusinski([25]) is a representative reference.

1.4.3. Correspondence between the Two Approaches.

By saying that a semantics supports SLDNF-resolution, we mean that everything we can compute by using the computational procedure is regarded as true by that semantics. More formally, we would say that SLDNF-resolution is sound with respect to that semantics. We will see in this subsection that the completed program semantics is such a semantics.

Definition 1.4.7. Let P be a normal program, G be a normal goal $\leftarrow L_1,\ldots,L_n$, and let Θ be an answer for $P\cup\{G\}$. By saying that Θ is a correct answer for $Comp(P)\cup\{G\}$, we mean that $\forall((L_1 \ldots L_n)\cdot\Theta)$ is a logical consequence of $Comp(P)$.

We also have a procedural correspondent to the above definition, as follows:

Definition 1.4.8. Let P be a normal program, G be a normal goal, and let Θ be an answer for $P\cup\{G\}$. By saying that Θ is a computed answer for $P\cup\{G\}$, we mean that Θ is the substitution obtained by restricting the concatenation $\Theta_1\cdot\ldots\cdot\Theta_n$ to the variables of G, where $\Theta_1,\ldots,\Theta_n$ is the sequence of substitutions used in the SLDNF-refutation of $P\cup\{G\}$.

Note: If the i^{th} selected literal in the refutation is negative, the Θ_i is the identity substitution, ε.

Now, we can formally state the soundness results for Negation as Failure([8]) and for SLDNF-resolution([21]).

Theorem 1.4.1. Let P be a normal program and let G be a normal goal. If P∪{G} has a finitely failed SLDNF-tree, then G is a logical consequence of Comp(P).

Theorem 1.4.2. Let P be a normal program and let G be a normal goal. Then every computed answer for P∪{G} is a correct answer for Comp(P)∪{G}, as well.

However, there is no general completeness result available for SLDNF-resolution with respect to the completed program semantics. In the following chapters, we will discuss some completeness results with respect to the completed program semantics when some additional conditions are met.

1.5. Conclusion.

In this chapter, we have discussed declarative and procedural semantic issues for first order language, and in particular, for logic programs.

We presented the least model semantics and SLD-resolution for definite programs as the declarative semantics and procedural semantics, respectively. In parallel, we presented the completed program semantics and SLDNF-resolution for

normal programs as the declarative and procedural semantics, respectively. In each case, we discussed the corresponding relationship.

We observed that for definite programs, the declarative semantics and procedural semantics correspond to each other perfectly; but this is not true for normal programs: SLDNF-resolution is not complete with respect to the completed program semantics.

In the next chapter, as a preparation for discovering completeness results for the class of normal programs, we will have a look at some of the causes for incompleteness.

Chapter 2. CAUSES OF INCOMPLETENESS.

2.1. The Completeness Problem.

A logic program may be viewed as a theory in a first order language([22],[32]), i.e. a set of first order sentences. One of the important properties of a logic program is that the meaning of such a set can be precisely defined by using the well-developed first order model theory as a declarative semantics. In addition, given that intended semantics, from the programming point of view, we would like to have a corresponding procedure semantics to define the consequences of such a set of formulas([25]). Naturally, we want this procedural semantics to fulfil some requirements: first of all, when a sentence and a program is given, if the response from the procedural semantics is positive, then the sentence should be a consequence of the program with respect to the declarative semantics. This is the soundness requirement and is a necessity for any such procedural semantics([21]). Secondly, it is usually expected that the procedural semantics be complete in the sense that for any consequence of the program with respect to the declarative semantics, the procedural semantics will give a positive response([21]).

It is a well known result that SLD-resolution is both sound and complete for definite programs and definite goals with respect to the least fixed point, declarative semantics ([15],[21]). However, when negation is taken into consideration, problems arise. As we discussed before, there are different treatments of negation: the farthest is to interpret any uncertainty as false, i.e., anything which cannot be proved will be regarded as false. This kind of treatment leads to a theoretically complete set of consequences with respect to the fixed semantics. The unfortunate thing is that it has been shown that if the consequence set supported by this extreme sort of semantics, besides being complete, is also consistent and contains no positive results other than those obtainable under the classical semantics, then no effective procedure for such a semantics can be complete ([1],[31]). Reiter's Closed World Assumption (CWA)([26]) and the minimum supported model semantics for stratified program of Apt et al([1]) are two examples of such semantics.

Another possible interpretation is the finite simulation of the foregoing, i.e. " Negation as Failure" (NAF). As NAF leads to negative result, it's surely not sound with respect to the classical semantics. But, it has been proved that this computation rule is sound with respect to the completed program semantics([8]).

The completed program semantics generally doesn't lead to a theoretically complete set of consequences. Thus, theoretically, it's possible to find a procedure which is both complete and effective corresponding to this semantics([1]). Usually, SLDNF-resolution is associated with Clark's semantics. One of the major reasons for this association is that it has been proved that SLDNF-resolution is sound with respect to Clark's semantics ([8]). Unfortunately, it has been shown that SLDNF-resolution is not complete even for very simple programs with respect to Clark's semantics.

As SLDNF-resolution is the mostly used computational procedure in logic programming and the completed program semantics is one of the widely accepted semantics in logic programming([4]), it is worthwhile to do a systematic study of the completeness problem of SLDNF-resolution with respect to the completed program semantics. As a first step, we need to clarify the situations in which incompleteness occurs. In the following sections, we'll discuss various causes for the incompleteness of SLDNF-resolution with respect to Clark's semantics and the corresponding techniques to avoid them. In particular, we will present an alternative explanation for a well-known cause in section 2.4 and introduce a seldom mentioned cause in section 2.5 for the incompleteness. For both of them, we will suggest some effective methods to avoid them.

2.2. Inconsistent Completion of a Logic Program.

Any logic program is consistent, as the corresponding Herbrand base in which every ground atom is regarded as true is a model for it; even a normal program is no exception([31]). But, when we consider the completion of a logic program, this nice property no longer holds([31]).

For example, suppose we have the following program:

P: $p \leftarrow \neg p$

The completion of P is $\{ p \leftrightarrow \neg p \}$ together with proper equality theory. Obviously, Comp(P) could not be consistent.

Shepherdson pointed out a simple property of SLDNF-resolution : If a goal G succeeds under a computation rule R, it cannot fail under any computation rule([31]). Based on this fact, Cavedon showed that completeness fails for both SLDNF-resolution and Negation as Failure for a program P and any goal G when Comp(P) is inconsistent([4]).

In the above example, Comp(P) is inconsistent, thus it implies anything in the language of Comp(P); in particular, it implies p, i.e. 'yes'(or the identity substitution) is a correct answer for $Comp(P) \cup \{\leftarrow p\}$. But, it is obvious that the identity substitution isn't an instance of any computed answer for $P \cup \{\leftarrow p\}$, as the latter has no SLDNF-refutation at all.

Therefore, when we take the completed program semantics as the intended one for logic programming, we do expect the consistency of Comp(P). Now, given that the underlying language of a program in most cases is a first order language, i.e. non-propositional, it's generally not effectively decidable whether the completed program is consistent ([22],[31],[32]). However, one often will encounter a condition which guarantees the consistency of the completed program([28]). Therefore, a completeness result assuming such consistency can be frequently applied.

2.3. Floundering.

Most logic programs we deal with are in first order language and will include variables. When there are also negative literal occurring , we may have some problems: in applying SLDNF-resolution, there is no step applicable if the current goal contains only non-ground negative literals([8],[21]). Such a goal is usually called a floundering goal. In fact, it's generally accepted as a safeness condition(to ensure soundness) that whenever one selects a sub-goal in the process of applying SLDNF-resolution, it's required that if the sub-goal is negative, then it must be ground, i.e., it contains no variables([21]). This requirement leads to another cause of incompleteness. In

the practice of logic programming we do have floundering goals, and when floundering occurs, we cannot continue computation due to the safeness requirement.

For example, given the following program and goal:

P: $p(x) \leftarrow \neg s(x)$

$r(a)$

G: $\leftarrow p(x)$;

Comp(P) is $\{ p(x) \leftrightarrow \neg s(x), r(x) \leftrightarrow x = a, \neg s(x) \}$, so we have that Comp(P) $\nvdash \forall x\ p(x)$. But, when applying SLDNF-resolution to $P \cup \{\leftarrow p(x)\}$, we immediately get into a floundering goal.

The desire to avoid floundering leads to the further requirement of "weak-allowedness"([30],[31]), which requires that every variable occurring in a negative literal in any goal must occur also in some positive literal in the same goal. This is motivated by the fact that when applying SLDNF-resolution to solve a goal, only a positive sub-goal can cause binding of variables occurring there. On the one hand, if a variable occurring in a negative literal also occurs in a positive literal, then after that positive literal gets solved, that variable may possibly be associated with a ground term so that the negative literal might become selectable. On the other hand, if some variable occurs in a negative literal but in no positive literal, then due to the safeness condition

this negative sub-goal could never be selected and it won't be solved. Obviously, in this case the derivation for the original goal will be blocked at some point. A floundering goal must be reached.

For example, if we have a program consisting of only one clause and a goal as follows:

P: $p(x) \leftarrow \neg q(y)$

G: $\leftarrow p(a)$;

Then after one derivation step, we have $\leftarrow \neg q(y)$, which is a floundering goal. Hence, the condition of weak-allowedness is necessary for a successful resolution.

After more careful consideration, one will discover that this condition of weak-allowedness isn't sufficient, however to avoid floundering. This is because it's quite possible that some variable occurring in a positive literal doesn't get grounded when that literal is solved, so its occurrence in a negative literal remains.

For example,

P: $p(x) \leftarrow$

G: $\leftarrow p(x), \neg q(x)$

Then, although G is weakly-allowed, after only one step, we will flounder.

To overcome this problem, a fairly strong condition of allowedness was introduced([1],[4],[21]&[31]). It not only

deals with a goal, but also relates to the whole syntactical structure of the involved program. Basically, it requires that the goal must be weakly-allowed, and for every clause in program, every variable occurring in that clause must occur in some positive literal on the right hand side of that clause. It has been proved that if this condition of allowedness is met, then no floundering will occur in an SLDNF-derivation([21],[30]).

However, in the following sections we will see that generally speaking, the bare condition of allowedness is not sufficient for completeness. Thus, in this sense, allowedness is not strong enough. In another sense, however, it is too strong; for even if a goal does contain a non-ground negative literal, if some positive literal in the same goal fails before the selection of the negative literal, then the negative literal won't lead to a floundering goal. The allowedness condition is not satisfactory as it disallows some quite useful programs such as the standard one for the membership relation; moreover, as already pointed out it's neither sufficient nor necessary for successful resolution. Several possible ways to weaken the condition of allowedness have been suggested in the literature ([18],[30],[4]); however, not one of them is both effectively testable and easily applicable. The condition of allowedness remains the best sufficient condition for preventing floundering so far.

Therefore, it's often taken as a precondition when one considers the completeness problem.

2.4. The Regularity Condition.

It happens that the relationship between program and goal can also affect the completeness of SLDNF-resolution, as the following frequently-quoted example shows([1],[8],[31]):

P: $p \leftarrow q$

$p \leftarrow \neg q$

$q \leftarrow q$

G: $\leftarrow p$

As Comp(P) is $\{ p \leftrightarrow q \vee \neg q,\ q \leftrightarrow q\}$; the query is obviously a logic consequence of Comp(P); but $\leftarrow p$ isn't refutable from program P due to the fact that every path in any possible SLDNF-tree is infinite.

The cause for this kind of incompleteness is usually explained by noting that p depends both positively and negatively on q. As a solution, the condition of strictness has been put forward, a condition which basically requires that a goal not depend both positively and negatively on the program ([1], [6], [19]).

The strictness condition guarantees that truth can never be derived under the semantics using the rule of excluded

middle, which is a problem in the consideration of completeness for SLDNF-resolution. As Cavedon pointed out, "SLDNF-resolution is unable to combine information from computations corresponding to different branches in a SLDNF-tree"([4]).

The usefulness of the strictness condition has been shown in the work done by Cavedon and Kunen, respectively ([4],[6],[19]). They have proved the Completeness of SLDNF-resolution for both stratified and call-consistent programs with respect to Clark's semantics, with the condition that the goal be strict with respect to the program. Their work will be presented in the next chapter.

The strictness condition is indeed "strict" in the sense that it disallows some useful programs, e.g., Cavedon showed that a program implementing if-then-else isn't strict([4]).

As another possible solution for non-strictness cause of incompleteness, Cavedon introduced the notion of atomically decidable program, which basically requires that any ground atom either succeeds or finitely fails and he proved([4]) the completeness of SLDNF-resolution with respect to this class of programs. As Cavedon himself pointed out that the class of atomically decidable programs isn't effectively decidable in polynomial time. This is true of another class of programs

which Cavedon introduced to weaken the condition of strictness, i.e. the "well-behaved" programs, which satisfies the condition that every ground atom with predicate depending both positively and negatively on another predicate should either succeed or finitely fail([4]).

We feel that the standard explanation for this kind of incompleteness isn't quite adequate: besides the dependency problem, the recursiveness of q, a descendent of p, should also be given some credit for causing the incompleteness, as it's obvious that if we delete the last clause from the program in the above example, then we will have a refutation for that goal, although p is still not strict with respect to the program. Motivated by the above discussion, a weaker condition of "regularity" of goals relative to programs is introduced, which requires that if some predicate, p, occurring in a goal isn't strict with respect to a program, then no descendants of p should be recursive.

The regularity condition is effectively decidable in polynomial time and weaker than the condition of strictness. A completeness result based on this condition will be presented in chapter 4, which is a slight extension of the completeness result achieved by Kunen for call-consistent programs.

2.5 Semantic Coincidence.

So far, we've discussed some causes for incompleteness relevant to the program structure, the occurrence of variables and the relationship between programs and goals and we have presented corresponding remedies. In this section, we will discuss one more cause for incompleteness, the lack of "semantic coincidence", which involves the relationship between SLDNF-resolution and the completed program semantics, which are the basic elements of the completeness problem.

When we consider the completeness problem, we always have a declarative and a procedural semantics in mind([25]). In the current context, the completed program semantics is the declarative semantics and SLDNF-resolution is the procedural correspondent.

From soundness of SLDNF-resolution with respect to the completed program semantics, we know that every answer which is computable by using SLDNF-resolution is correct with respect to the completed program semantics. In other words, everything supported by the procedural semantics is also supported by the declarative semantics. By completeness, we are considering the opposite, i.e. every answer which is correct according to the completed program semantics should be computable by applying SLDNF-resolution, i.e., everything supported by the declarative semantics should be supported by

the procedural semantics, as well. Therefore, the two semantics are intended to be equivalent.

Now, by the definition of the completed program semantics, it is always defined for the whole program([8]). It is built upon everything in the program. However, this is not true for SLDNF-resolution: usually, it is not defined for the whole program. Actually, it only deals with the part of program which is relevant to the goal, i.e., only those clauses the head of each of which either occurs in the involved goal or is a descendant of some predicate occurring in that goal([8]). Therefore, the set of clauses of a program on which declarative and procedural semantics are defined can be different; although as we discussed before, the two semantics are supposed to be equivalent.

As semantics are assigned on syntactical parts, if the syntactical parts are different, it is quite plausible that the semantics defined on them will be different also. Note, as we already have soundness for SLDNF-resolution with respect to Clark's semantics([8],[21]), the only possible discrepancy in semantics will be that something supported by the declarative semantics is not supported by the procedural one, as the latter contains less syntactical information than the former, which leads to the incompleteness of SLDNF-resolution with respect to Clark's semantics.

We have the following example to demonstrate the situation we have been discussing so far.

P: $p \leftarrow \neg p$

$p \leftarrow q$

$q \leftarrow q$

G: $\leftarrow q$

It's not hard to see that Comp(P), which is consistent and is equivalent to $p \wedge q$. G is strict (regular as well) with respect to P, and trivially $P \cup \{G\}$ is allowed as the program is in propositional language.

Although we have $Comp(P) \nvdash q$, there is no SLDNF-refutation for $P \cup \{G\}$.

In the above example, although Comp(P) is defined for the whole program, the procedural semantics only involves part of it: as $q \leftarrow q$ is the sole clause in P dealing with that goal, it's the only part used to test the computability of the identity answer in SLDNF-resolution. So the information we have to test whether or not q follows from Comp(P) is not what we have to judge whether or not q is supported by Comp(P). Thus, there is no surprise this program/goal pair exemplifies incompleteness.

The above situation is quite different from the previously discussed causes and it deserves separate treatment. In Chapter 5, we will discuss it in more detail and based on the discussion of this cause of incompleteness, we will present a new characterization of completeness for a

fairly large class of programs and goals. We will also provide an effectively testable and sufficient condition to prevent this cause of incompleteness from occurring.

2.6 Conclusion.

In this chapter, we have discussed the general completeness problem; specifically, we have discussed the causes of incompleteness for SLDNF-resolution with respect to the completed program semantics. For each of the causes, we either present the existing approaches or introduce new ones to deal with them.

In order to achieve the completeness of SLDNF-resolution with respect to the completed program semantics, one standardly must have a consistent completion. It is also standard to assume the strong condition of allowedness to guarantee that the resolution process won't flounder. Additionally, it is useful to let the program and the goal satisfy the regularity condition so some of the infinite deductions could be prevented. Finally, but not trivially, we should make sure that the semantics defined on sometimes different syntactical parts coincide.

So far, all of our discussions are in the domain of two-valued logic. It turns out that we could also consider the

problem from the point of view of 3-valued logic ([15],[18],[31]). We would like to mention this alternative approach as it provides an interesting explanation of the incompleteness of SLDNF-resolution. As to be discussed in the next chapter, it will be clear that SLDNF-resolution is also sound with respect to Clark's semantics under 3-valued interpretation, which says that every computed answer for P∪{G} is also a correct answer for Comp(P)∪{G} when we consider all the 3-valued consequences ([18],[31]). This result can be used to explain the incompleteness of applying SLDNF-resolution to a program, P and a goal, G.

For example, when P is the program consisting of the following clauses { $p \leftarrow \neg p$, $p \leftarrow q$, $q \leftarrow q$} and G is the goal { $\leftarrow p$ }: as Comp(P) ={ $p \leftrightarrow \neg p \vee q$, $q \leftrightarrow q$} together with a proper equality theory, there is a 3-valued model of Comp(P), in which both p and q are assigned "undefined". Hence p isn't a 3-valued consequence of Comp(P). By the soundness of SLDNF-resolution with respect to the completed program semantics under the 3-valued approach, it's certain that we couldn't have a successful SLDNF-resolution for P∪{G}.

At the beginning of the next chapter, we will discuss in full about the 3-valued logic approach and the corresponding completeness results. Then, we will present some currently known completeness results for SLDNF-resolution with respect to the completed program semantics.

Chapter 3. Currently Known Completeness Results.

In the last chapter, we discussed some causes for the incompleteness of SLDNF-resolution with respect to the completed program semantics, and we presented corresponding methods to prevent those problems form occurring. Now, we will present several well-studied classes of logic programs, i.e., hierarchical, stratified and call-consistent programs as well as associated completeness results obtained in line with the results of the last chapter.

Before we do these things, we feel it would be quite helpful to begin with a discussion of an alternative declarative semantics of normal program using a 3-valued approach and the associated general completeness result; as the latter is, or can be used as a foundation for the classical 2-valued completeness results which are to be presented in this chapter.

3.1. Three-Valued Semantics.

Given a normal program P, and a ground atom, p(t), the application of SLDNF-resolution to P$\cup\{\leftarrow p(t)\}$, as we noted

before, could lead to either a successful refutation, or finite failure, or an infinite loop.

In the declarative semantics based on a 2-valued approach, we regard p(t) as 'true' if SLDNF-resolution leads to a successful refutation, and as 'false' if the resolution leads to a finite failure, but have no interpretation for the case of an infinite loop.

In a 3-valued approach, this gap will be closed: the truth value corresponding to the infinite loop will be regarded as 'undefined', while keeping the interpretations of refutation and finite failure the same as those given in the 2-valued semantics.

3.1.1. Operations on Truth Values in a 3-valued Approach.

The 3-valued logic on which the new semantics is based has the three truth values: 't' for truth, 'f' for false and 'u' for undefined. The logical operations follow from Kleene's truth tables([17]), to wit: given a propositional formula, its truth value is defined as 't', if all possible ways of replacing the occurrences of 'u' with either 't' of 'f' leads to 't' as computed in ordinary 2-valued logic; its truth value is 'f' if the value of the negation of that formula is 't'; otherwise, its truth value will be 'u'. One exception is the interpretation of the bi-conditional, which is defined as 't'

when the left hand side and the right hand side have the same truth values. This exceptional assignment is due to Łukasiewicz. This is so defined that we can have the consistency of Comp(P) in the three-valued approach([16],[28], [9]).

3.1.2. Declarative Semantics Based on a 3-valued Approach.

The declarative semantics based on the 3-valued logic assigned to a logic program is constructed by using an associated operator, Φ_p, defined on the corresponding Herbrand base.

Definition 3.1.1. Let S be a set of closed formulas, B_s be the corresponding Herbrand base. A 3-valued Herbrand interpretation of S is an ordered pair I = (T, F), where T, F are disjoint sets of ground atoms in B_s. For every $A \varepsilon B_s$, $Val_I(A)$ is t (A is true) if $A \varepsilon T$, is f (A is false) if $A \varepsilon F$, is u if $A \varepsilon B_s - (T \cup F)$.

The valuation function Val_I extends to the entire language over the alphabet of S in the standard way, but using the aforementioned modification of Kleene's truth table, instead of the usual 2-valued truth tables.

By saying a 3-valued interpretation for S is a 3-valued model for S, we mean that every closed formula in S is true under the interpretation.

Now, we define the associated operator:

Definition 3.1.2. Let P be a normal program, B_p be the corresponding Herbrand base, and let (T_1, F_1) be a 3-valued Herbrand interpretation for P. Then Φ_p is defined as the following:

$\Phi_p((T_1,F_1))=(T_2,F_2)$, where

$T_2=\{\alpha \varepsilon B_p$ | there is a ground instance of a clause in P:

$\alpha \leftarrow L_1,\ldots,L_n$, and $L_1 \wedge \ldots \wedge L_n$ is true in (T_1,F_1),

which means every L_i is true in $(T_1, F_1)\}$.

$F_2=\{\alpha \varepsilon B_p$ | for all ground instances of clauses in P:

$\alpha \leftarrow L_1,\ldots,L_n$, and $L_1 \wedge \ldots \wedge L_n$ is false in (T_1,F_1),

which means at least one L_i is false in $(T_1,F_1)\}$.

We have one more definition.

Definition 3.1.3. Let P be a normal program, Φ_p be the associated operator and let α be an ordinal. By $\Phi_p \uparrow \alpha$, we mean:

$\Phi_p \uparrow 0 = (\phi, \phi)$

$\Phi_p \uparrow (\alpha+1) = \Phi_p(\Phi_p \uparrow (\alpha))$ if α is not a limit ordinal

$\Phi_p \uparrow (\alpha+1) = \cup_{(\beta<\alpha)}(\Phi_p \uparrow \beta)$ if α is a limit ordinal.

The following theorem is due to Fitting([10],[31])

Theorem 3.1.1.. Let P be a normal program, Φ_p be the associated operator. Then we have:

(1) Φ_p is monotone;

(2) Φ_p has a least fixed point, $\Phi_p \uparrow \alpha$ for some ordinal α.

(3) The least fixed point of Φ_p is a 3-valued model of Comp(P).

From the above theorem, we can see that the 3-valued semantics guarantees the consistency of Comp(P), just as the classical semantics guarantees the consistency of P.

But, it turns out that Φ_p is not always continuous. Thus, although we know that there exists an ordinal α such that $\Phi_p \uparrow \alpha$ is the least fixed point of Φ_p, α could be very big([10],[18]) . As Kunen pointed out that Φ_p can be discontinuous even for very simple program([18]), generally speaking, this semantics doesn't lead to a recursively decidable set.

In order to solve this problem, Kunen cuts off the operator at ω, i.e., he takes $\Phi_p \uparrow \omega$ as the intended semantics assigned to a program P, in other words he regards any ground atom p(t) as true if and only if it is true in $\Phi_p \uparrow n$ for some finite n. This approximation is justified by the following important theorem due to Kunen([18],[19]).

Definition 3.1.4. Let Φ be a sentence, and let S be a set of sentences. By saying that Φ is a 3-valued logical consequence of S, we mean that every 3-valued model of S is also a 3-valued model of Φ.

Theorem 3.1.2.(Kunen([18]) Let P be a normal program, and A be a sentence. There is an n such that A becomes true in $\Phi_p \uparrow n$, if and only if A is a 3-valued logical consequence of Comp(P).

3.1.3. The Declarative verse the Procedural Semantics.

We present some important results concerning relationships between 3-valued semantics and SLDNF-resolution.

Definition 3.1.5. Let P be a normal program, $G(\leftarrow L_1, \ldots, L_n)$ be a normal goal, and let Θ be an answer for $P \cup \{G\}$. By saying that Θ is a 3-valued correct answer for $Comp(P) \cup \{G\}$, we mean that $\forall((L_1 \wedge \ldots \wedge L_n) \cdot \Theta)$ is a 3-valued consequence of Comp(P).

The definition of computed answer is the same as before.

It turns out that both Negation as Failure and SLDNF-resolution are sound with respect to Kunen's semantics([31]).

Theorem 3.1.3. Let P be a normal program, and let G be a normal goal. Then if $P \cup \{G\}$ has a finitely-failed tree, then G is a 3-valued consequence of Comp(P).

Theorem 3.1.4. Let P be a normal program and G be a normal goal. Then any computed answer for $P \cup \{G\}$ is also a 3-valued correct answer for $Comp(P) \cup \{G\}$.

Kunen also shows that under the condition of allowedness of the involved program and goal, we also have the completeness of SLDNF-resolution, in particular, of Negation as Failure with respect to his semantics([19]).

Theorem 3.1.5. Let P be a normal program, and let G be an allowed normal goal. Then every 3-valued correct answer of Comp(P)∪{G} is ground; moreover, it is also a computed answer for P∪{G}. If G is a logical consequence of Comp(P), then there is a finitely-failed tree for P∪{G}.

3.2. Hierarchical Programs.

When we introduced the regularity condition in the last chapter, we did so in the context of the problem of the existence of infinite paths in every SLDNF-tree for the involved program and goal.

Assume we can find a condition that for a given allowed program and a goal, it's guaranteed that every path in every SLDNF-tree either terminates with an empty clause or finitely fails; i.e. every SLDNF-tree is finite (In other words, those trees satisfy the "finite tree property".), then this condition of finite tree property is sufficient for a completeness result([8],[31]).

If there is an infinite path in an SLDNF-tree, then because of the finiteness of the involved program, there is only a finite number of distinct predicate symbols in this program, in particular, along the concerned path. Due to the "pigeon hole" principle, there must be repetition of predicate

symbols in that path. From the definition of SLDNF-derivation, there must be a predicate which depends on itself, i.e., there exists recursion in the definition of predicates in that program.

On the other hand, if during the SLDNF-resolution process, a recursive predicate is involved, then there must be an infinite path in the corresponding SLDNF-tree.

Therefore, if there is no recursive predicate in program, then there won't be any infinite path in any SLDNF-tree. We will have the finite tree property satisfied for the involved program and goal([30],[31]).

The class of hierarchical programs satisfies the above condition([8],[30],[31]).

Definition 3.2.1. A level mapping of a normal program is a mapping from its set of predicate symbols to the non-negative integers. The value of a predicate under this mapping is referred to as the level of that predicate symbol.

Definition 3.2.2. A normal program is hierarchical if it has a level mapping such that, in every program clause with a body, $p(t_1,\ldots, t_n) \leftarrow L_1,\ldots,L_n$, the level of the predicate symbol of every literal in the body is less than the level of predicate symbol p.

For example, the program consisting of the clauses: $\{p \leftarrow q,\ p \leftarrow \neg q\}$ is a hierarchical program.

Obviously, as in any clause of a hierarchical program the level of the predicate in the head is strictly greater than the level of any predicate occurring in the body, thus no predicate could be recursive in any hierarchical program due to the finiteness assumption. Actually, it's true that a program is hierarchical if and only if every predicate defined in the program is not recursive.

The class of hierarchical programs is a special case of the class of stratified programs(which will be discussed in the next section). As it has been shown that the completion of every stratified program is consistent([1]), the completion of every hierarchical program is also consistent.

From the discussion about the finite tree property and the connection between non-recursiveness and the hierarchical condition, one should expect the completeness of SLDNF-resolution with respect to Clark's semantics defined for hierarchical programs, when the allowedness condition is met as well. As a matter of fact, this completeness result is one of the earliest. It's given in the same paper in which the completed program semantics is introduced([8],[30]), as an application of the latter.

We present the completeness proof for hierarchical programs, following Kunen's 3-valued logic approach([19]). At first, we have a lemma due to Kunen:

Lemma 3.2.1. Let P be a hierarchical normal program. Then every 3-valued model of Comp(p) is also a 2-valued model of Comp(P).

Proof: By induction on levels of predicates defined in program.

Theorem 3.2.2. Let P be a hierarchical normal program, G be a normal goal and let $P \cup \{G\}$ be allowed. Then every correct answer of Comp(P) is an instance of a computed answer of $P \cup \{G\}$.

Proof: Let G be $\leftarrow L_1, \ldots L_n$, and let Θ be a correct answer for $Comp(p) \cup \{G\}$, so that $Comp(P) \models \forall((L_1 \wedge \ldots \wedge L_n) \cdot \Theta)$. From the preceding lemma, we have $\forall((L_1 \wedge \ldots \wedge L_n) \cdot \Theta)$ is also a 3-valued consequence of Comp(p). Due to Kunen's general completeness result for the completed program semantics in 3-value logic approach, the conclusion of the theorem follows.

Obviously, the hierarchical condition is fairly strong as it doesn't allow any kind of recursion. A natural weakening of the hierarchical condition is the stratifiability condition.

3.3. Stratified Programs.

The notion of stratified programs is a natural development from that of hierarchical programs and has

undergone a systematic study from various angles([1],[23]). Stratifiability disallows recursion through negation, i.e., any atomic formula, before it's used negatively, must be well defined already. Therefore, it extends the class of hierarchical programs to some extent.

Definition 3.3.1. A normal program is stratified if it has a level mapping such that, in every program clause with a body, $p(t_1,\ldots, t_n) \leftarrow L_1,\ldots,L_n$, the level of the predicate symbol of every positive literal in the body is less than or equal to the level of p, and the level of the predicate symbol of every negative literal in the body is less than the level of p.

For example, the program consisting of clauses {$p \leftarrow q$, $p \leftarrow \neg q$, $q \leftarrow q$} is stratified; although it's not hierarchical as q depends on itself, which is a recursive definition.

In the paper by Apt et al([1]), a unique minimal supported model semantics is assigned to any stratified program and it has been shown to be equivalent to the least fixed point of an iterative operator constructed for that stratified program. Due to a well-known result that such a fixed point is a model of the completed program ([21]), the consistency of completion for any stratified program is proved([1]).

Unfortunately, SLDNF-resolution is not complete even for this class of programs. As we discussed in the section of the regularity condition in last chapter, if we have the stratified program { $p \leftarrow q$, $P \leftarrow \neg q$, $q \leftarrow q$} and the goal $\leftarrow p$. then SLDNF-resolution is not complete with respect to the completed program semantics assigned to this program.

It turns out that if we add an additional condition of strictness, then a completeness result could be obtained for both SLDNF-resolution and Negation as Failure for the completed program semantics assigned to a stratified program([6]).

First, we give the formal definition of positive and negative dependency of one predicate on another in a normal program.

Definition 3.3.2. Let P be a normal program, and let p, q be two predicates. By $p \geq_{+1} q$ (resp. $p \geq_{-1} q$) (p depends positively (resp. negatively) on q), we mean that either there is a clause such that p occurs in the head of that clause, and q occurs in a positive (resp. negative) literal in the body, or p depends positively (resp. negatively) on another predicate r such that there is a clause in P with r in the head and q occurs in a positive literal in the body, or p depends negatively (resp. positively) on a predicate r, and there is a clause in P with r in the head, and q occurs in a

negative literal in the body.

Moreover, by $p \geq q$ (p depends on q), we mean that either $p \geq_{+1} q$, or $p \geq_{-1} q$. We will call q a descendant of p.

For example, in the program with the following clauses $\{ p \leftarrow q,\ p \leftarrow \neg q,\ q \leftarrow q\}$, we have $p \geq_{+1} q$, $p \geq_{-1} q$ and $q \geq_{+1} q$.

Now, we can define the strictness condition of a goal with respect to a program([1],[6],[19]):

Definition 3.3.3. Let P be a normal program, and let G be a normal goal $\leftarrow L_1,\ldots,L_n$; moreover, in the goal, let p_i be the predicate of L_i. Then:

(1) $\leftarrow L_1,\ldots,L_n$ depends positively on a predicate r via p_i in case either L_i is an atom and $p_i \geq_{+1} r$, or L_i is a negative literal and $p_i \geq_{-1} r$.

(2) $\leftarrow L_1,\ldots,L_n$ depends negatively on a predicate r via p_i in case either L_i is an atom and $p_i \geq_{-1} r$, or L_i is a negative literal and $p_i \geq_{+1} r$.

(3) $P \cup \{G\}$ is strict in case G doesn't depend on any predicate in P both positively and negatively via any predicate.

We shall say G is strict with respect to P when $P \cup \{G\}$ is strict.

For example, given the following program $\{ p \leftarrow q,\ p \leftarrow \neg q\}$ and goal $\leftarrow p$; then the goal is not strict with respect to the program, as p depends on q both positively and negatively.

In this case, the condition of strictness amounts to prevent SLDNF-resolution from running into $q \vee \neg q$.

The following completeness results are due to Cavedon and Lloyd([6],[4]).

Theorem 3.3.1. Let P be an allowed, stratified normal program and G an allowed normal goal such that G is strict with respect to P. Then every correct answer for Comp(P)∪{G} is a computed answer for P∪{G}, as well.

Theorem 3.3.2. Let P be an allowed, stratified normal program and G an allowed normal goal such that G is strict with respect to P. If Comp(P)⊭ G, then there exists a fair SLDNF-tree for P∪{G} and every fair SLDNF-tree for P∪{G} is finitely failed.

These results have been extended by Kunen to the class of call-consistent programs, which properly contains the class of stratified programs([19]). The result dealing with call-consistent programs will be discussed in the following section.

In addition, the condition of strictness could be weakened to the condition of regularity. The condition of regularity and the associated completeness results will be discussed in Chapter 4.

3.4. Call-consistent Program.

The class of call-consistent programs is introduced independently by Sato and Kunen([27],[19]). It disallows any predicate which depends negatively on itself. In other words, it "permits recursion through an even number of negation"([6]) but disallows recursion through an odd number of negation.

Considering the program { $p \leftarrow \neg q$, $q \leftarrow \neg p$ }. It is not stratified, but it is call-consistent as neither p nor q depends on itself negatively. However, the program consisting of only one clause {$p \leftarrow \neg p$} is not call-consistent. Call-consistency is weaker than stratifiability in that it allows some kind of recursion through negation. The condition is less intuitive, however, than stratifiability.

The call-consistency condition seems to be directly concerned with the consistency of the completed program, as it is obvious that if a program contains such clause as $p \leftarrow \neg p$, then its completion couldn't be consistent. The important thing is that Sato and Kunen proved that this condition is actually sufficient for the consistency of the completed program.

Note: from the exemplifying program of {$p \leftarrow \neg p$}, we can see that the real problem in the inconsistency of completed

program is the negative dependency of a ground atom on itself. So, the call-consistency property is a bit too strong to safeguard the consistency property of the completed program. It has been shown that disallowing of negative dependency of any ground atom on itself is already sufficient for the consistency of completed program. The corresponding condition is called local call-consistency([4]).

Definition 3.4.1. Let P be a normal program. By saying that P is call-consistent, we mean that no predicate defined in P depends negatively on itself.

Cavedon suggests an equivalent definition for call-consistency, which has more flavor of the level mapping approach we followed to define both hierarchical and stratified programs ([4]).

Definition 3.4.2. A normal program p is call-consistent provided it has a level mapping such that for any two predicate symbols p and q in P, if p depends on q then level(p) $\geq$ level(q); and if p depends both positively and negatively on q, then level(p) > level(q).

We will also present some new finding of a syntactical property of call-consistent programs, which adds a bit more to the intuitiveness of this condition. As this is somewhat

digressing from the main topic of this paper, we put it in the Appendix.

Kunen showed that for any call-consistent program, there is a partition for all predicate symbols in that program such that any two predicates in each partition depend on each other, but not both positively and negatively. Based on this property, Kunen proved a model expansion lemma(Lemma 4.2.4) for the class of call-consistent programs. As a special case of the lemma, the consistency of the completed program of any call-consistent program is proved([19]).

By using the general completeness results obtained in the track of 3-valued logic, Kunen proved that with the condition of strictness, SLDNF-resolution is actually complete with respect to the completed program semantics (2-valued) assigned to the class of call-consistent programs([19]).

The following are the completeness results due to Kunen:

Lemma 3.4.1. Let P be an allowed, call-consistent normal program, G ($\leftarrow L_1, \ldots L_n$) be an allowed normal goal and G be strict with respect to P; let Θ be an answer for $P \cup \{G\}$. If $\forall((L_1 \wedge \ldots \wedge L_n)\cdot\Theta)$ is a 2-valued consequence of Comp(P), then it is also a 3-valued consequence of Comp(P).

Theorem 3.4.1. Let P be an allowed, call-consistent normal program, G be an allowed normal goal and let G be

strict with respect to P. Then every correct answer of Comp(P)∪{G} is an instance of a computed answer of P∪{G}.

Proof: Straightforward from the lemma and the general completeness result dealing with 3-valued consequence.

Kunen, after presenting his proof of the above theorem, suggests a possible extension to it, which is formally put forward as the following theorem by Cavedon([4]).

Theorem 3.3.2. Let P be an allowed, call-consistent normal program and G a normal goal $\leftarrow L_1,\ldots,L_n$. Such that the set of clauses defining those predicates in G which depend positively and negatively on some predicates in P is a hierarchical program. If $\forall((L_1\wedge\ldots\wedge L_n)\cdot\theta)$ is a 2-valued consequence of Comp(P) then it is also a 3-valued consequence.

Actually, what we will do in Chapter 4 leads to a syntactical condition for the involved program such that the condition in the above theorem will be met, though our approach is different from his: Kunen's idea is developed based on the coincidence of 2-valued and 3-valued consequences of the completed program, a semantic property; while ours is motivated by the prevention of recursiveness of some dependents of a predicate, a syntactical approach and our result is proved in the classical, 2-valued logic way. However we also make out a proof of our results following Kunen's approach, so it is more consistent with the main results of

Kunen.

3.5. Conclusion.

By a discussion of the condition of being definite, hierarchical, stratified and call-consistent programs, we obtain that they are developed by successive liberalization pertaining to the restriction on the occurrence of recursion and negation: from the total disallowing of negation in definite programs, to disallowing of recursion in hierarchical programs, to disallowing recursion through negation in stratified programs, to disallowing only recursion through an odd number of recursion in call-consistent programs. The call-consistency condition is already a bit subtle. After that, it seems quite difficult to find out a meaningful characterization of a broader class of programs for which a completeness result can be proved.

The line followed here is some proper relation between predicates. One approach for going forward is to study a similar relationship at the atomic level. For example, dependency of ground atoms in the grounded program which is formed by collecting all ground instances of every clause in the original program, by using all the constant symbol and function symbols. Some important and interesting results have

been made in this direction, for the locally hierarchical, locally stratified and locally call-consistent programs([4],[5],[23],[31]). However, the existence of function symbols in a program immediately leads to an infinite grounded program ([25],[18]), which could cause problems to both theoretical study and practical application.

An alternative approach is to insist on some general principles. It is obvious that all solutions to the completeness problem considered so far share some common ground: consistency of the completed program, allowedness of programs and goals, and strictness of goals with respect to programs.

After slightly extending one of the foregoing completeness results in chapter 4, we will embark on a new approach towards obtaining completeness results for SLDNF-resolution with respect to the completed program semantics. We will start with those basic requirements mentioned above and analyze the relationship between the basic elements of the completeness problem: SLDNF-resolution as the procedural semantics, and the completed program semantics as the declarative correspondent relative to the respective syntactical parts of a program on which the two semantics are defined. We will end up with a new characterization of completeness of a fairly large class of programs and provide

an effective condition to test the completeness for member of a broader class of programs than the class of call-consistent ones.

Chapter 4. AN EXTENSION FOR A COMPLETENESS RESULT.

4.1. Motivation.

It is well known that in general, SLDNF-resolution is not complete for normal programs. One of the frequently-quoted examples ([8],[30],[1]) is: {p ← q, p ← ¬q, q ←q}. Its completion is {p←>q ∨ ¬q, q←>q}. As q ∨ ¬q is true under any model of Comp(P), so is p; thus p is a logical consequence of Comp(P). On the other hand, there is no successful SLDNF-resolution tree for P∪{←p}:

Notice that the program mentioned above is stratified, so generally speaking, SLDNF-resolution is not complete even for stratified programs.

Looking at the above program, we may note that the predicate p is the head of two different clauses, while another predicate, q, occurs both positively and negatively in those two clauses, a phenomenon which could be generalized. Motivated by this fact, Apt et al([1]) defined the strict programs to be stratified programs, in which no such kind of dependency will be allowed, and presented([1]) the conjecture that SLDNF-resolution is complete for strict programs also

satisfying the allowedness condition([1]).

This conjecture has been solved positively by Cavedon & Lloyd[6] with extensions on the type of goals; their result was further extended by Kunen([19]) using the 3-valued logic approach; and moreover, Kunen gave a positive result for call-consistent programs, a larger class than the stratified programs, but still requiring the condition of the program being strict with respect to a goal.

We approach this problem from a different angle by treating the preceding three-clause program as a whole. Namely, we not only note the dependency property, but also pay attention to the recursive property of the predicate on which another predicate depends both positively and negatively, feeling that recursiveness plays an important part in this situation. A simple supporting example is: $p \leftarrow q$, $p \leftarrow \neg q$. On the one hand, it is still true that $Comp(P) \nvdash p$; on the other hand, there does exist a successful SLDNF-refutation tree for $P \cup \{\leftarrow p\}$.

In our approach, we require that, in the programs under study, if a predicate p depends on another one both positively and negatively, then all predicates on which p depends should be non-recursive. This condition is weaker than the one discussed by Apt et al. Moreover, motivated by Kunen's

result, we will consider the class of call-consistent programs instead of the smaller class of stratified ones.

4.2. Basic Definitions and Lemmas.

We have the following definitions.

Definition 4.2.1. Let P be a normal program, and p be a predicate. By Def(p, P) (the definition of p in P), we mean the set of all clauses, in P, with p as the predicate of the head.

For example, let P be the following program (taken from [16]):

$q(X) \leftarrow \neg p(X)$

$p(X) \leftarrow isc(X)$

$p(X) \leftarrow nonc(X)$

$isc(c)$

$nonc(X) \leftarrow \neg isc(X)$

Then, $Def(p,P) = \{ p(X) \leftarrow isc(X),\ p(X) \leftarrow nonc(X) \}$

Definition 4.2.2. Let P be a normal program, and let R be a set of predicates occurring in P. By Res(P,R) (the restriction of P on R), we mean the sub-program consisting of exactly the definitions of those predicates defined in P which are either in R, or are descendants of some predicates in R.

For example, let P be the same program given before, and

let R={p}. Then, Res(P, R) = { p(X) ← isc(X), p(X) ← nonc(X), isc(c)←, nonc(X) ← ¬isc(X) }.

Now, we can begin to introduce the condition of regularity.

Definition 4.2.3. Let P be a normal program, and let p be a predicate in P. By saying that p is lenient, we mean that there is a predicate, q, in P such that p depends on q both positively and negatively.

Recall: Definition 3.2.2 gives out the definition of dependency of one predicate on another predicate.

For example, in above program, both p and q are lenient, while nonc is not.

Definition 4.2.4. Let P be a normal program, and let R be a set of predicates of P. By saying that P is regular for R, we mean that for each lenient predicate p in R, no descendant of p is recursive. We say that P is regular, if P is regular for the set of all predicates in P.

For example, the program mentioned before is regular, and stratified as well, while the program p ← q, p ← ¬q, q ← q is not regular for the set {p, q}.

Definition 4.2.5. Let P be a normal program, and let G be a normal goal $\leftarrow L_1, L_2, \ldots, L_n$; moreover, in the goal, G, let p_i be the predicate of L_i. Then:

(1) $\leftarrow L_1,\ldots,L_n$ <u>depends positively on a predicate</u> r <u>via</u> p_i in case either L_i is an atom and $p_i \geq_{+1} r$, or L_i is a negative literal and $p_i \geq_{-1} r$.

(2) $\leftarrow L_1,\ldots,L_n$ <u>depends negatively on a predicate</u> r <u>via</u> p_i in case either L_i is an atom and $p_i \geq_{-1} r$, or L_i is a negative literal and $p_i \geq_{+1} r$.

(3) $P\cup\{G\}$ is <u>regular</u> in case for all i,j, if there is a predicate r such that G depends positively on r via p_i and negatively on r via p_j, then no descendent of p_i or p_j is recursive.

We shall say that G is <u>regular with respect to</u> P(or P is <u>regular with respect to</u> G) when $P\cup\{G\}$ is regular.

For example, if we have the following program and goal

P: a(X) $\leftarrow$ p(X), q(X);

b(X) $\leftarrow \neg$p(X);

G: $\leftarrow$ a(X), b(X);

then $P\cup\{G\}$ is regular; but if we add one more clause,

p(X)$\leftarrow$ p(f(X)), to P, then $P\cup\{G\}$ would not be regular.

The next definition is due to Kunen([19]):

<u>Definition 4.2.6</u>. Let P be a normal program, and let R be a set of predicates occurring in P. We say that R is <u>downward closed</u>, if for every predicate p in R and every other predicate q occurring in P, if $p \geq q$, then q is also in R.

In order to prove a completeness result for goals which

are regular with respect to call-consistent programs, we need the following basic lemmas.

First, we have the following obvious result:

Lemma 4.2.1. Let P be a normal program, then P is hierarchical iff no predicate in P is defined recursively.

Next, we have a result revealing a property of the completed program:

Lemma 4.2.2. Let P be a normal program, and let R be a set of predicates occurring in P. Then Comp(Res(P,R)) is a subset of Comp(P).

Proof: Let p be a predicate which occurs in Res(P,R). If it is defined in Res(P, R), then by definition of Res(P,R), the completion of p in Comp(P, R) is the same as that in Comp(P). If p isn't defined in Res(P, R), then by definition of Comp(Res(P, R), $\forall x \neg p(x)$ is in Comp(Res(P, R)). We claim that it is also in Comp(P): assume that it is not the case, then p must be defined in P. However, as p occurs in Res(P, R), it must be a descendant of some predicate in R. By the definition of Res(P, R), the definition of p should be in Res(P, R), as well. In other words, $\forall x \neg P(x)$ wouldn't be in Comp(Res(P,R)), a contradiction.

Finally, noting that Clark's equality theory([8],[21]) is independent of the program. the conclusion follows.

Corollary 4.2.3. Let P be a normal program, let

$G(\leftarrow L_1, \ldots, L_n)$ be a normal goal, and let Θ be a substitution for the variables in G. Then Θ is a correct answer of Comp(Res(P,R)) implies that it is also a correct answer for Comp(P).

Proof: Let M be any model of Comp(P). By Lemma 4.2.2, Comp(Res(P, R)) is a subset of Comp(P). Hence M is a model of Comp(Res(P, R)) as well. But we also have the following fact

$Comp(Res(P, R)) \models \forall((L_1 \wedge \ldots \wedge L_n)\cdot\Theta)$; thus $\forall((L_1 \wedge \ldots \wedge L_n)\cdot\Theta)$ is true under M. As M is arbitrary, we have the conclusion that $Comp(P) \models \forall((L_1 \wedge \ldots \wedge L_n)\cdot\Theta)$, i.e. Θ is correct for $Comp(P) \cup \{G\}$.

Now, we are ready to prove the main result of this section, which shows the importance of Res(P, R). The proof is largely based on the following important result due to Kunen.

Lemma 4.2.4. ("Kunen's Theorem", see [19], Th. 3.4): Let P be a normal program, and let Q_1 and Q_2 be disjoint subsets of the set of all the predicates of P, such that both Q_1 and $Q_1 \cup Q_2$ are downward closed. Suppose also that P is call-consistent on Q_2 and that M is a 2-valued model for $Comp(Res(P,Q_1))$. Then M has an expansion to a 2-valued model for $Comp(Res(P,Q_1 \cup Q_2))$.

Theorem 4.2.5. Let P be a call-consistent, normal program, let $G(\leftarrow L_1, \ldots, L_n)$ be a normal goal, and let R be the set of predicates occurring in G. Then Θ is a correct answer for $Comp(P) \cup \{G\}$ implies that it is also a correct answer for

Comp(Res(P, R))∪{G}.

Proof: As Θ is a correct answer for Comp(P)∪{G}, we have that Comp(P) $\models \forall((L_1 \wedge \ldots \wedge L_n)\cdot\Theta)$. We want to show that it is also the case that comp(Res(P, R)) $\models \forall((L_1 \wedge \ldots \wedge L_n)\cdot\Theta)$.

Let Q_1 be the set of predicates which are either members of R or are descendants of predicates in R; and let Q_2 be the set of the predicates not in Q_1. Then $Q_1 \cap Q_2 = \phi$, Q_1 and $Q_1 \cup Q_2$ are both downward closed . Now P is call-consistent, so no predicate symbol p in P satisfies $p \geq_{-1} p$; in particular this is true for predicates in Q_2. Therefore, due to the preceding lemma, every model of Comp(Res(P,R)) can be expanded to a model of Comp(P); and from the fact that Comp(P) $\models \forall((L_1 \wedge \ldots \vee L_n)\cdot\Theta)$, $\forall((L_1 \wedge \ldots \wedge L_n)\cdot\Theta)$ is true in every model of Comp(Res(P, R)). Therefore, the conclusion follows.

Corollary 4.2.6. Let P be a call-consistent normal program, let G be a normal goal and let R be the set of predicates in G. Then, Θ is a correct answer for Comp(P)∪{G} iff it is also a correct answer for Comp(Res(P, R))∪{G}.

Proof: Immediate from Corollary 4.2.3 and Theorem 4.2.5.

Finally, we state Lloyd's lifting lemma for normal program ([6]) as a reference. It is an extension of a similar result for definite programs([21]).

Lemma 4.2.7.(Lloyd[6]): Let P be an allowed, normal

program, G an allowed, normal goal and Θ a substitution. Suppose there exists an SLDNF-refutation of $P \cup \{G\Theta\}$. Then there exists an SLDNF-refutation of $P \cup \{G\}$ of the same length such that, if $\Theta_1, \ldots, \Theta_n$ are the mgu's from the SLDNF-refutation of $P \cup \{G\Theta\}$ and $\Theta_1', \ldots, \Theta_n'$ are the mgu's from the SLDNF-refutation of $P \cup \{G\}$, then there exists a substitution σ such that $\Theta \cdot \Theta_1 \cdot \ldots \cdot \Theta_n = \Theta_1' \cdot \ldots \cdot \Theta_n' \cdot \sigma$.

4.3. Completeness Results.

At first, we present a completeness result for a simple goal:

Lemma 4.3.1. Let P be an allowed, call-consistent normal program, and let L be an allowed literal for which P is regular and such that $Comp(P) \models \forall L$. Then there is an SLDNF-refutation for $P \cup \{\leftarrow L\}$.

Proof: Let P_L be $Res(P, \{p\})$, where p is the predicate of L. Since $Comp(P) \models \forall L$, we have by Theorem 4.2.5 that $Comp(P_L) \models \forall L$.

Suppose that p is not lenient. Then P_L is strict. Hence, by Kunen[19], $P_L \cup \{\leftarrow L\}$ has an SLDNF-refutation, and thus so does $P \cup \{\leftarrow L\}$.

Suppose, on the other hand, that $p \geq_{+1} r$ and $p \geq_{-1} r$ for some predicate, r. Then since P is regular for L, all predicates of P_L are non-recursive. Hence, by Lemma 4.2.1, P_L

is hierarchical. Therefore, $P_L \cup \{\leftarrow L\}$ has an SLDNF-refutation by Theorem 3.1.2, so does $P \cup \{\leftarrow L\}$.

Next, we relate a lemma dealing with a property of the condition of allowedness.

Lemma 4.3.2.(Lloyd[18] Proposition 15.1): Let P be a normal program and G a normal goal. Suppose that $P \cup \{G\}$ is allowed. Then every computed answer for $P \cup \{G\}$ is a ground substitution for all variables in G.

Now, we have the following completeness result for call-consistent programs:

Theorem 4.3.3. Let P be an allowed, call-consistent program, $G(\leftarrow L_1, L_2, \ldots, L_n)$ be an allowed goal, and let G be regular with respect to P. If $Comp(P) \models \forall(L_1 \wedge L_2 \wedge \ldots \wedge L_n)$. Then there is a refutation for $P \cup \{G\}$.

Proof: First consider any positive literal L_k in G: as $\leftarrow L_k$ is an allowed goal, P is regular with respect to L_k and $Comp(P) \models \forall L_k$, all conditions for Lemma 4.3.1 are satisfied, and thus there is an SLDNF-refutation for $P \cup \{\leftarrow L_k\}$; by Lemma 4.3.2, the computed answer grounds all variables in L_k.

Then, as G is allowed, for every negative literal L_j, if it is not ground then there must be some positive literals $L_{k1}, \ldots, L_{km}$ such that each L_{ki} contains a variable which occurs in L_j, and each of the variables of L_j occurs in L_{ki} for some i. By the preceding discussion, there will be refutations with

computed answers which ground all the variables in the L_{k1}'s, and hence in L_j. Then, following these km refutations, Lemma 4.3.1 could be applied to the grounded L_j's.

Therefore, if we apply a computation rule for which all the positive sub-goals are solved first, then we have a refutation for P∪{G}.

When the correct answer is also ground, we have the following:

Theorem 4.3.4. Let P be an allowed and call-consistent program, G be an allowed goal and let G be regular with respect to P. Then every correct ground answer, Θ, for P ∪ {G} is a computed answer for P ∪ {G}.

Proof: Let G be $\leftarrow L_1,\ldots,L_n$, and suppose that Θ is a ground substitution with $\text{Comp}(P) \models \forall((L_1 \wedge \ldots \wedge L_n)\cdot\Theta)$. Since $G\cdot\Theta$ is ground, it is immediate from Theorem 4.3.3 that $P\cup\{G\cdot\Theta\}$ has a refutation(with identity substitution ε as computed answer). By Lemma 4.2.7, there is a refutation of P∪{G} with computed answer Θ' such that for some substitution, σ, $\Theta=\Theta\cdot\varepsilon=\Theta'\cdot\sigma$. By Lemma 4.3.2, Θ' is ground, and thus $\Theta = \Theta'$.

Note: If the condition in Theorem 4.3.4 that Θ be ground is omitted, similar reasoning would show that an instance of the correct answer Θ is computable. Moreover, if the correct answer is not ground, then by Lemma 4.3.2 it cannot be computed.

4.4. An Alternative Proof.

Kunen once proved the following general result by using 3-valued logic approach([19]):

Theorem 4.4.1. Let P be a normal program, and let G be an allowed goal. If Θ is a correct answer for Comp(P)$\cup${G} with respect to the 3-valued completed program semantics; then GΘ is ground, and Θ is an instance of a computed answer for P$\cup${G}.

In particular, by showing that for P, a call-consistent program and G, a goal, which is strict with respect to P; every correct answer for Comp(P)$\cup${G} in 2-valued completed program semantics is also a correct answer for Comp(P)$\cup${G} with respect to the 3-valued completed program semantics, Kunen obtained a completeness result for call-consistent program with goals which are strict with respect to the program([19]).

In section 4.3, we showed that the above result could be extended slightly, using the condition of regularity of goals with respect to programs, which is weaker than the strictness condition. But, it seems that using the classical approach, we couldn't show that every correct answer must be ground, as well.

The following presents the details of a "suggested" proof for the strengthened result via Kunen's 3-valued logic approach([19]).

The following lemma is repeated from Chapter 3(Lemma 3.1.1).

Lemma 4.4.2. Let P be a normal program. If P is also hierarchical, then every 3-valued model of its completed program is a 2-valued model of that, too.

Note: the above lemma equivalently says that every 2-valued consequence of the completed program of a hierarchical program is also its 3-valued consequence.

The next theorem is a strengthened completeness result:

Theorem 4.4.3. Let P be a call-consistent, normal program, and let G be allowed and regular with respect to P. Then every correct answer of Comp(P)$\cup${G} is ground and is an instance of a computed answer of P$\cup${G}.

Proof: Let H be the set of predicates defined in P of which no descendant of any of them is recursive. Obviously, Res(P, H) is hierarchical.

Let G be $\leftarrow L_1, L_2, \ldots, L_n$; and let Θ be a correct answer for Comp(P)$\cup${G}.. By definition, Comp(P) $\models \forall((L_1 \wedge \ldots \wedge L_n)\cdot\Theta)$, hence, Comp(P) $\models \forall((L_1\cdot\Theta \wedge \ldots \wedge L_n)\cdot\Theta)$; therefore, we have that for all i, Comp(P) $\models \forall L_1\cdot\Theta$.

Let q_1 be the predicate occurring in L_1. We have 2 cases:

(1) L_i is strict with respect to P: Kunen([19]) showed that in this case $L_i \cdot \Theta$ is a 3-valued consequence of Comp(P); moreover, by Theorem 4.4.1, Θ is ground.

(2) L_i isn't strict with respect to P, i.e., q_i depends both positively and negatively on some predicate in P: As G is regular with respect to P, so is L_i with respect to P. By definition, q_i must be defined in Res(P, H).

As P is call-consistent, by Kunen's theorem, we have that $Comp(Res(P, H)) \models \forall L_i \cdot \Theta$. Finally, by Lemma 4.4.2, we have that $L_i\Theta$ is also a 3-valued consequence of Comp(Res(P, H)). Moreover, by Theorem 4.4.1, $L_i \cdot \Theta$ is ground also.

Therefore, we have the following:

$Comp(P) \models \forall((L_1 \wedge \ldots \wedge L_n) \cdot \Theta)$ iff $Comp(P) \models ((L_1 \wedge \ldots \wedge L_n) \cdot \Theta)$ iff $Comp(P) \models (L_1 \cdot \Theta \wedge \ldots \wedge L_n \cdot \Theta)$ iff for every i, $Comp(P) \models L_i \cdot \Theta$.

As each L_i falls into case (1) or case (2) discussed above, we have that every $L_i \cdot \Theta$ is a 3-valued consequence of Comp(P).

By Theorem 4.4.1, for each i, $G \cup \{L_i \cdot \Theta\}$ could be refuted; as each L_i is ground, we could combine all the refutations for $P \cup \{\leftarrow L_i \cdot \Theta\}$ into one for $P \cup \{G \cdot \Theta\}$. Finally, Lloyd's lifting lemma (Lemma 4.2.7) tells us that there is a computed answer Θ' for $P \cup \{G\}$ such that $\Theta = \Theta' \cdot \delta$ for some δ.

4.5. Conclusion.

The results we obtained in 4.3 are slightly stronger than both Cavedon & Lloyd's and Kunen's completeness results. Regularity is a weaker condition than strictness. Also, our restriction to ground answers in Theorem 4.3.4 is not a true limitation. With regard to Lloyd's result, his Lemma 3([6]) shows that the only correct answers for allowed goals with respect to allowed, stratified programs are ground ones. With regard to Kunen's result, his theorems 3.6, 4.1 and 4.2([18]&[19]) show that the only correct answers for allowed goals which are strict with respect to a call-consistent(i.e. near-regular) and allowed program are ground, as well.

The alternative result obtained in 4.4 is carried out in Kunen's approach, so it's more consistent with his results and also seems to be a natural extension to these results.

Chapter 5. A New Approach.

5.1. Introduction.

Since Clark presented the completed program semantics and solved the first completeness problem with respect to hierarchical programs([8]), several more results for stratified, call-consistent and other types of program, with respect to various types of goal have been made ([19],[6]&[3]).

In order to find a stronger completeness result, we need to have a look at the causes for incompleteness to gain some inspiration. In Chapter 2, we have noted that the inconsistency of the completed program and the floundering of goals are two major causes for incompleteness of SLDNF-resolution with respect to Clark's semantics ([30],[25]&[28]); thus it makes sense to consider only those programs with a consistent completed program and those goals which do not flounder.

In Chapter 4, we discussed the condition of strictness and regularity as solutions for other causes of the incompleteness which are involved with dependency of a

predicate on another predicate, as well as recursive definition of some dependents in the program. This work was inspired by the following famous incompleteness example([8],[1],[30])

P_0: $p \leftarrow q$

$p \leftarrow \neg q$

$q \leftarrow q$

G: $\leftarrow p$.

Using the same "incompleteness example" approach, it should not be surprising that the consideration of yet other such examples will lead to completeness theorems as well. The main results of this Chapter are, in fact, motivated by the following example.

Example 1:

P_1: $p \leftarrow \neg p$.

$p \leftarrow q$.

$q \leftarrow q$.

G: $\leftarrow q$.

It's not difficult to see that the completed program of P1 $\{ p \leftrightarrow \neg p \vee q,\ q \leftrightarrow q \}$ is consistent, and is equivalent to $\{ p,\ q \}$. G is actually strict with respect to P1 and trivially G is not floundering.

But, although we have $Comp(P_1) \models q$; there is no SLDNF-refutation for $P_1 \cup \{\leftarrow q\}$.

Different from the other examples, the incompleteness in

this case is due to the fact that although q follows from $Comp(P_1)$, it really shouldn't in a certain sense.

The point is that $q \leftarrow q$ is the only part of the program to be used in SLDNF-resolution; if q were computed in the process, then by the soundness of SLDNF-resolution with respect to the completed program semantics, q would be a consequence of $Comp(q \leftarrow q)$, or $q \leftrightarrow q$, which is false.

One of the possible explanations for this is that although $q \leftarrow q$ is the only part of the program used to test the computability of the answer 'yes' by symbolic manipulation, it's not the part that decide the truth of q in the completed program - the whole program does. In other words, the information we need to judge one part of the story isn't the same as what we have to test the other part, which is intended to be equivalent to the first part.

Based on the above example and discussion, we have a new approach towards completeness results - that is, to coincide the semantics decided by the "procedurally-relevant" part and the "declaratively-relevant" part of a program with respect to a goal.

5.2. Procedurally-Relevant vs. Declaratively-Relevant Part.

The use of SLDNF-resolution determines that the only

clauses in the symbolic manipulation process we could use to decide whether an answer could be computed for a program P and a goal G are just those for which the predicate in the head either occurs in G, or is a descendant of some predicate occurring in G([21]). We have the following:

Definition 5.2.1. Let P be a normal program, and let G be a normal goal. By Res(P, G), we mean the sub-program of P, consisting of exactly the definitions of those predicates which either occur in G, or are descendants of predicates occurring in G.

We had a similar definition in the last Chapter for Res(P, R) where R is a set of predicates defined in P.

We will call Res(P, G) the procedurally-relevant part of the program P with respect to the goal G because of the following simple observation:

Lemma 5.2.1. Let P be a normal program, G be a normal goal, and let Θ be an answer for G; then Θ is a computed answer of $P \cup \{G\}$ iff Θ is a computed answer of $Res(P, G) \cup \{G\}$.

Proof: Straightforward from the definition of Res(P, G) and that of SLDNF-resolution.

From example 1, we notice that although Res(P, G) is the procedurally-relevant part, it doesn't constitute the

"declaratively-relevant part", i.e. it couldn't decide whether Θ is a correct answer of $Comp(P_1) \cup \{G\}$. Which part of the program does that?

Again, from example 1, we find out that we must also include the ancestor of q, that is p, to decide the truth of q in the whole program. The following example shows that this is still not enough.

Example 2.

P_2: $p \leftarrow \neg p$.

$p \leftarrow r$.

$q \leftarrow r$.

$r \leftarrow r$.

$G: \leftarrow q$.

$Comp(P_2)$ is as follows:

$\{p \leftrightarrow \neg p \vee r,\ q \leftrightarrow r,\ r \leftrightarrow r\}$

$Res(P_2, G)$ is: $q \leftarrow r.\ r \leftarrow r.$, its completion is $\{q \leftrightarrow r,\ r \leftrightarrow r\}$.

It's obvious that $\{p,q,r\}$ is equivalent to $Comp(P_2)$, and $Comp(P_2) \nvdash q$; but q isn't computable.

Note: there is no ancestor of q existing in P_2, but there does exist a predicate p, which is an ancestor of a descendent of q , i.e., r.

We are ready to define the declarative correspondent for Res(P, G). First, we have the following:

Definition 5.2.2. Let P be a normal program. By Pred(P), we mean the set of all predicates occurring in P.

Definition 5.2.3. Let P be a normal program, and let G be a normal goal. By Prior(P, G) (prior predicates of G with respect to P), we mean the set of all predicates which properly depend on either predicates occurring in G, or their descendants, i.e., Prior(P, G) consists of exactly the proper ancestors of the descendants of the predicates of G. Formally, we have that $Prior(P, G)=\{ p \in (Pred(P)-Pred(Res(P, G))) \mid \exists q (q \in Pred(Res(P, G)) \& P \geq q)$.

For example, $Prior(P_1, \{\leftarrow q\}) = Prior(P_2, \{\leftarrow q\}) = \{p\}$.

Definition 5.2.4. Let P be a normal program, and let G be a normal goal. By Ext(P, G) (the extension of G in P), we mean the sub-program of P, consisting of exactly the definition of all predicates in G and Prior(P, G) and all their descendants. Formally, we have: $Ext(P, G) = Res(P, Prior(P, G)) \cup Res(P, G)$.

Example: $Ext(P_1, \{\leftarrow q\}) = P_1$, $Ext(P_2, \{\leftarrow q\}) = P_2$.

In examples 1 and 2, the extension of G in P is the same as P, generally, it need not be. For the sake of simplicity, we make the following

Assumption S: *In any program, P, no predicate outside* Ext(P, G) depends on any other predicate inside Ext(P, G).

In Section 5.6, we will discuss the general situation.

Definition 5.2.5. Let P_1 and P_2 be two normal programs such that P_1 is a subset of P_2. By saying that P_1 is well-complemented in P_2, we mean that:

(1) P_1 and P_2-P_1 share no function symbols, or

(2) For every predicate, p, in P_2-P_1, p doesn't depend negatively on itself.

Obviously, the condition of well-complementation is decidable in polynomial time.

The significance of Ext(P, G) is expressed in the upcoming Lemma 5.2.5.

Recall that Res(P, R) denotes the sub-program of P, consisting of exactly the definitions of those predicates which either occur in R, or are descendants of some predicates in R.

Lemma 5.2.2. Let P be a normal program, and suppose that R_1 and R_2 are sets of predicates such that Pred(Res(P, R_1)) and Pred(Res(P, R_2)) are two disjoint subsets of Pred(P). If both Comp(Res(P, R_1)) and Comp(Res(P, R_2)) are consistent and share no function symbols, then every model of Comp(Res(P, R_1)) can be expanded to a model of Comp(Res(P, $R_1 \cup R_2$)).

Proof: Let M_1, M_2 be models of Comp(Res(P, R_1)) and Comp(Res(P, R_2)), respectively.

Now, Mendelson shows on p. 73 of [22] that if α and ß are

two ordinal numbers with $\alpha \leq ß$, then any model of cardinality α of a first order theory, T, expands to a model of T of cardinality ß. Thus, without loss of generality, M_1 and M2 can be assumed to have the same cardinality. Let $M_1=<D_1, J>$, $M_2=<D_2, K>$. As both D_1 and D_2 have the same cardinality, there is a 1-1 mapping between D_1 and D_2, let it be f: D_2 -> D_1.

Define M=<D,I> for all constants, function symbols and predicate symbols in either Res(P, R_1) or Res(P, R_2) in the following way:

$D=D_1$.

For every constant c, if it occurs in Res(P, R_1), $c^I = c^J$; otherwise, $c^I = f(c^K)$. If c occurs in Res(P, R_1), then c^I should be compatible with c^J; as we have only finite number of constants, it can always be done.

For every function symbol g , if it occurs in Res(P, R_1), then g^I: D ->D , $g^I(t_1,\ldots,t_n) = g^J(t_1,\ldots,t_n)$; otherwise, $g^I(t_1,\ldots,t_n) = f(g^K(t_1,\ldots,t_n))$.

For every predicate symbol p, if it occurs in Res(P, R_1), then p^I is a subset of D, $p^I = \{(d_1,\ldots,d_n) \mid (d_1,\ldots,d_n) \in p^J\}$; otherwise, $p^I = \{(f(d_1),\ldots,f(d_n)) \mid (d_1,\ldots,d_n) \in p^K\}$.

Now, from the construction and the fact that R_1 and R_2 are disjoint, we have the following:

M_1 is isomorphic to M restricted to Res(P, R_1) (actually, they are the same), while M_2 is isomorphic to M restricted to Res(P, R_2); thus, for any formula, A, in either Comp(Res(P, R_1)), or Comp(Res(P, R_2)), $M \models A$.([22], sec. 2.11)

Therefore, M is a model of both Comp(Res(P, R_1)) and Comp(Res(P, R_2)). Finally, as they have no predicate in common, M is a model of Comp(Res(P, $R_1 \cup R_2$)). Obviously, M expands both M_1 and M_2.

Lemma 5.2.3. Let P_1 and P_2 be normal programs such that P_1 is a subset of P_2, and both Comp(P_1) and Comp(P_2-P_1) are consistent.

If the predicates in P_1 and P_2 are both downward closed, the predicates in P_2-P_1 and those in P_1 are disjoint, and P_1 is well-complemented in P_2, then every model of Comp(P_1) can be expanded to a model of Comp(P_2).

Proof: Let Q_1 be Pred(P_1) and Q_2 be Pred(P_2-P_1), we have that $Q_1 \cap Q_2 = \phi$ and $Q_1 \cup Q_2$ = Pred(P_2).

If P_2-P_1 is call-consistent, i.e., P_2 is call-consistent on Q_2, then as both Q_1 and $Q_1 \cup Q_2$ are downward closed and $Q_1 \cap Q_2 = \phi$, by Kunen's theorem, every model of Comp(Res(P_2, Q_1)) can be expanded to a model of Comp(Res(P_2, $Q_1 \cup Q_2$)).

As Res(P_2, Q_1) = P_1 and Res(P_2, $Q_1 \cup Q_2$) = P_2, we have the result for this case.

Otherwise, Res(P_2, Q_2) shares no function symbol with P_1 by the well-complementation condition. Therefore, by Lemma 5.2.2, we have the expected result.

Definition 5.2.6. Let P be a normal program and let G be a normal goal. By saying that G is well-placed for P, we mean

that Ext(P, G) is well-complemented in P.

Corollary 5.2.4. Let P be a normal program and G be a normal goal with Comp(P) consistent. If G is well-placed for P, then every model of Comp(Ext(P, G)) expands to a model of Comp(P).

Proof: As both Pred(Ext(P, G)) and Pred(P) are downward closed , and Pred(Ext(P, G)) and Pred(P-Ext(P, G)) are disjoint because Assumption S and, moreover, G is well-placed for P, so we have that Lemma 5.2.3 is applicable.

Finally, we have the following:

Lemma 5.2.5. Let P be a normal program with Comp(P) consistent, let G be a normal goal such that G is well-placed for P, and let Θ be an answer for P$\cup${G}. Then Θ is a correct answer for Comp(P)$\cup${G} iff Θ is a correct answer for Comp(Ext(P, G))$\cup${G}.

Proof: It is straightforward that Comp(Ext(P, G)) is a subset of Comp(P).

Thus, "Sufficiency" is trivial by the definition of being a correct answer and "Necessity" follows from the preceding corollary.

Just as Lemma 5.2.5 actually shows that Comp(P) is a conservative extension of Comp(Ext(P, G)), it's also true that Comp(Ext(P, G)) is a conservative extension of Comp(Res(P,G)).

The same relationship of course exists among P, Ext(P, G) and Res(P, G).

As Ext(P, G) has the property shown in Lemma 5.2.5, we will call it the declaratively-relevant part of the program P with respect to the goal G.

Now that, we have found both the procedurally-relevant and declaratively-relevant parts, we are interested in their relationship. The following theorem is the main result of this section.

Theorem 5.2.6. Let P be a normal program with a consistent completion, and let G be a normal goal which is well-placed for P. Then: if completeness holds for P∪{G}, i.e., every correct answer for Comp(P)∪{G} is an instance of a computed answer for P∪{G}, then, for every answer, Θ, for P∪{G}, Θ is correct for Comp(Res(P, G))∪{G} iff it is correct for Comp(Ext(P, G))∪{G}.

Proof: As Comp(P) is assumed to be consistent, by Lemma 4.2.2 , so are Comp(Ext(P, G)) and Comp(Res(P, G)).

On the one hand, as Comp(Res(P, G)) is a subset of Comp(Ext(P, G)), every correct answer for Comp(Res(P, G))∪{G} is trivially a correct answer for Comp(Ext(P, G))∪{G}.

On the other hand, let Θ be a correct answer for Comp(Ext(P, G))∪{G}; by lemma 5.2.5, it is correct for Comp(P)

$\cup\{G\}$. By the assumption of this theorem, θ is an instance of a computed answer for $P\cup\{G\}$, and by Lemma 5.2.1, θ is also an instance of a computed answer for $Res(P, G)\cup\{G\}$. By the soundness of SLDNF-resolution with respect to Clark's semantics, we have that θ must be also a correct answer for $Comp(Res(P, G))\cup\{G\}$.

Theorem 5.2.6 tells us that if we have a completeness result, then, $Res(P, G)$ decides the declarative part also. In other words, the semantics decided by the declaratively-relevant part of P with respect to G coincides with that decided by the corresponding procedurally-relevant part. The following is a simple demonstration of Theorem 5.2.6 for the case that $Comp(Res(P, G))$ is not the same as $Comp(Ext(P, G))$:

Example 3:

P_3: $r_1 \leftarrow r_1$

$r \leftarrow q$

$p \leftarrow q$

$p \leftarrow \neg q$

G: $\leftarrow p$

$Comp(P_3) = \{ r_1 \leftrightarrow r_1,\ r \leftrightarrow q,\ p \leftrightarrow q \vee \neg q,\ \neg q\}$.

As $\{p\}$ is a Herbrand model of $Comp(P_3)$, $Comp(P_3)$ is consistent. Obviously, we have that $Comp(P_3) \nvdash p$; and the answer 'yes' could be obtained from applying SLDNF-resolution to $P_3\cup\{\leftarrow p\}$. Therefore, the conditions for the theorem are satisfied.

Moreover, we have $Res(P_3, G)=\{p \leftarrow q,\ p \leftarrow \neg q\}$, and

$Comp(Res(P_3, G)) \models p$; and also, $Ext(P_3, G) = \{r \leftarrow q, p \leftarrow q, p \leftarrow \neg q\}$, and $Comp(Ext(P_3, G)) \models p$. In other words, the identity substitution, as a correct answer, is supported by both $Comp(Res(P, G))$ and $Comp(Ext(P, G))$.

Note: For a program, P, with consistent completion:
(1) Theorem 5.2.6 shows that "completeness" is sufficient for the coincidence of the declarative semantics decided by the procedurally-relevant and that decided by the declaratively-relevant parts of P with respect to G.
(2) Program P_1 (of Example 1) shows that the coincidence of the two semantics may sometimes fail (of course only for incompleteness examples).
(3) Program P_0 shows that "completeness" is not necessary for coincidence of the semantics decided by the two respective parts, i.e., the converse of Theorem 5.2.6 is not generally true.

In the next section, we will show that under some reasonable conditions, we can obtain the converse of Theorem 5.2.6, as well as a stronger completeness result. In section 5.4, we will have a look at some examples, displaying some properties of the results we obtained in section 5.3. In section 5.5 and section 5.6, we will discuss the necessity of the condition of well-complementation and the generalization of declaratively-relatively part, respectively. Finally,

section 5.7 sums up the results of this chapter.

5.3. A New Characterization of Completeness.

We provide the following lemma to show that the condition of regularity provides us with a nice working environment.

Lemma 5.3.1. Let P be a normal program. If G is regular with respect to P, then Res(P, G) is call-consistent.

Proof: Just suppose not: then there is a predicate p defined in Res(P, G) such that $p \geq_{-1} p$. By the definition of Res(P, G), p either occurs in G or is a descendent of a predicate occurring in G. In the first case, as we always have $p \geq_{+1} p$, p won't be regular with respect to P; in the other case, there must be a predicate, q, in G such that $q \geq p$; then we also get the contradiction that G couldn't be regular with respect to P, as q will depend on p both positively and negatively, while p is recursive.

We are ready to give the proof of the converse of Theorem 5.2.6.

Theorem 5.3.2. Let P be a normal program with a consistent completion, and let G be a normal goal, allowed and regular with respect to P, and which is well-placed for P. Then: If the correct answers for Comp(Ext(P, G))∪{G} are exactly the correct answers for Comp(Res(P, G))∪{G}, then

every correct answer for Comp(P)∪{G} is an instance of a computed answer for P∪{G}.

Proof: Let Θ be a correct answer for Comp(P)∪{G}, then by Lemma 5.2.5, it is correct for Comp(Ext(P, G))∪{G}, and thus, by assumption, also for Comp(Res(P, G))∪{G}. By Lemma 5.3.1 and Theorem 4.4.3, it's an instance of a computed answer for Res(P, G)∪{G}. Finally, by Lemma 5.2.1, it is an instance of a computed answer for P∪{G}.

Combining Theorem 5.2.6 and Theorem 5.3.2, we obtain the following new characterization of completeness in a fairly broad context.

Theorem 5.3.3. Let P be a normal program with a consistent completion, and let a normal goal G be allowed and regular with respect to P, and which is well-placed for P. Then SLDNF-resolution is complete with respect to the completed program semantics assigned to P∪{G} iff the semantics determined by the procedurally-relevant part of P with respect to G coincides with that determined by the declaratively-relevant part.

The following presents a condition which is sufficient for the hypothesis of Theorem 5.3.2.

Lemma 5.3.4. Let P be a normal program with a consistent completion, and let G be a goal. Then if the following Condition C is met: "Every model of Comp(Res(P, G)) can be

expanded to a model of Comp(Ext(P, G))", then the correct answers for Comp(Ext(P, G))∪ {G} are exactly the correct answers for Comp(Res(P, G))∪{G}.

Proof: Straightforward.

As condition C is not effectively decidable, it is not quite ideal. In order to have a useful completeness result, we have to find an effective condition which implies Condition C. It is understandable that such an effective condition will be fairly strong.

One such condition is that the whole program be call-consistent. In that case, Kunen's theorem(Lemma 4.2.4) guarantees that condition C will be met. Actually, that's the method we used to prove Theorem 4.4.3. It's given as Corollary 4.2.6.

The next example shows that it is not necessary that we must have a call-consistent program in order to satisfy Condition C.

Example 4:

P_4: $p \leftarrow r$.

$p \leftarrow \neg q$.

$q \leftarrow q$.

$r \leftarrow \neg r$.

$r \leftarrow q_1$.

$q_1 \leftarrow q_1$.

G: $\leftarrow q$.

Comp(P_4) = { $p \leftrightarrow r \vee \neg q$, $q \leftrightarrow q$, $r \leftrightarrow \neg r \vee q_1$, $q_1 \leftrightarrow q_1$}, which is consistent. (There are exactly two models: p, r and q_1 must be assigned "true", and q may be assigned either "true" or "false"). G is regular with respect to P.

On the one hand, G is still not refutable, on the other hand, Comp(P_4) does not imply q.(Make q_1, r and p "true" and q "false".)

We notice that both models of Comp(Res(P_4, G)), (Assign q "false" or "true".) can be expanded to models of Comp(Ext(P_4, G)) which is the same as Comp(P_4), but P_4 isn't call-consistent.

The above example leads us to Lemma 5.3.5, which gives another effective sufficient condition for Condition C.

Definition 5.3.1. Let P be a normal program. By CIC(P), we mean the set of all predicates defined in P such that each of those depends negatively on itself.

Definition 5.3.2. Let P be a normal program, and let G be a normal goal. By $Q_r(P, G)$, we mean the set of all proper descendants of Prior(P, G) which aren't in Pred(Res(P, G)).

Formally, we have that

$Q_r(P, G)$ = Pred(Ext(P,G)) - (Prior(P,G) $\cup$ Pred(Res(P,G))).

For example, $Q_r(P_4, \{\leftarrow q\}) = \{r, q1\}$.

Lemma 5.3.5. Let P be a normal program with a consistent completion, and let G be a normal goal such that Res(P, G) is well-complemented in Res(P, G)$\cup$Res(P, Q_r(P, G)). The following effective Condition E is sufficient for Condition C.

Condition E: Prior(P, G) $\cap$ CIC(P) = ϕ

Proof:

As Pred(G), Prior(P, G) and Q_r are all subsets of Pred(P), then by Lemma 4.2.2 and the fact that Comp(P) is consistent, so is the completion of each of the corresponding sub-programs ,i.e.,

(1) Comp(Res(P, G)), Comp(Ext(P, G)) and Comp(Res(P, Q_r)) are all consistent.

From the definition of Prior(P, G) and that of Q_r, we have that Prior(P, G) and the set of all predicates in Res(P, G) are disjoint; so are Prior(P, G) and the set of all predicates in Res(P, Q_r); thus, we have

(2) Prior(P, G) is disjoint from the set of all the predicates in Res(P, G) and from the set of all predicates in Res(P, Q_r).

An immediate consequence of Condition E is

(3) P is call-consistent on the predicates in Prior(P, G).

From the way we define Prior(P, G), Pred(G) and Q_r, we also have the following fact:

(4) Pred(Res(P, G))$\cup$pred(Res(P, Q_r)) and Pred(Res(P, Prior(P, G)) are both downward closed.

Letting M be a model for Comp(Res(P, G)). By the given definition of Q_r and Res(P, G), the assumption of the well-complementation and (1), above, Lemma 5.2.3 is applicable: M expands to M_1, a model for Comp(P, Res(Pred(G)$\cup Q_r$(P, G))), which is actually the completed program restricted to the set of all descendants of Prior(P, G).

Moreover, by the preceding facts (2)-(4), Kunen's theorem (Lemma 4.2.4) is applicable, thus, M_1 can be expanded further to a model of Comp(Res(P, Pred(G)$\cup Q_r$(P, G)$\cup$Prior(P, G))), which is the same as Comp(Ext(P, G)).

Definition 5.3.3. Let P be a normal program and G be a normal goal. Let NBD(P, G) denote Res(P, G)$\cup$Res(P, Q_r(P, G)). Then, by saying that G is strongly well-placed for P, we mean that Res(P, G) is well-complemented in NBD(P, G) and Ext(P, G) is well-complemented in P.

Thus, relying on Lemma 5.3.5, Lemma 5.3.4, and Theorem 5.3.2, we have proven the following:

Theorem 5.3.6. Let P be a normal program with a consistent completion, and let G be allowed and regular with respect to P, and is strongly well-placed for P. Then if no predicate in Prior(P, G) depends negatively on itself, (i.e., Prior(P, G) $\cap$ CIC(P) = ϕ) then every correct answer for Comp(P)$\cup${G} is an instance of a computed answer for P$\cup${G}.

Theorem 5.3.6 is quite general, in the sense that it covers nearly all the completeness results known so far:

(1) P is definite([21]).

The consistency of Comp(P) is well known([2],[1]). and every goal is regular with respect to P, as no negative literal could occur; and trivially it's allowed. Moreover, CIC(P) is empty, so condition E is met.

(2) P is hierarchical([8],[29]).

It has been proved that Comp(P) is consistent([1]). As no predicate in P could be recursive, any goal will be regular with respect to P. CIC(P) is also empty, so we have completeness result for any allowed goal.

(3) P is stratified([6]).

It also has been shown that Comp(P) is consistent ([6]). And the strictness condition in the existing result is stronger than the regularity condition; and CIC(P) is empty, as well.

(4) P is call-consistent([19]).

Similar to (3).

In all the above cases, we have CIC(P)=ϕ; thus, Condition E is trivially met. Recently, L. Cavedon proved the consistency of locally call-consistent programs ([31],[4] &[5]), which actually requires that no ground atom will depend on itself negatively. This being the case, then local call-consistency is weaker than the call-consistency. Thus, CIC(P)

needn't be empty for this class of programs. Theorem 5.3.6 provides an effectively calculable condition to test the completeness property for that class of programs and for those even weaker, when consistency of the completion is provable.

5.4. Some Further Examples.

It is still possible to find weaker conditions for completeness, based on the approach of semantic coincidence of the procedurally-relevant and declaratively-relevant parts of a program with respect to a goal, inasmuch as neither Condition C nor Condition E is necessary for completeness. The following examples will show this.

Example 5.

P_5: $r \leftarrow \neg r$.

$r \leftarrow p, q$.

$p \leftarrow \neg q_1$.

$p \leftarrow r_1, q$.

$q \leftarrow q$.

G: $\leftarrow p$.

Comp(P_5), which is the same as Comp(Ext(P_5, G)), is the following: $\{r \leftrightarrow \neg r \vee (p \wedge q),\ p \leftrightarrow \neg q_1 \vee (r_1 \wedge q),\ q \leftrightarrow q,\ \neg q_1,\ \neg r_1\}$

Res(P_5, G) is: $\{\ p \leftarrow \neg q_1;\ p \leftarrow r_1,\ q;\ q \leftarrow q\}$

Comp(Res(P_5, G)) is: $\{\ p \leftrightarrow \neg q_1 \vee (r_1 \wedge q),\ q \leftrightarrow q,\ \neg q_1,\ \neg r_1\}$. As the only model of Comp(P) makes r, p and q "true" and r_1

and q_1 "false", $Comp(P_5) \nvdash p$; and also we have $Comp(Res(P_5, G)) \nvdash p$, as p is true in both models of $Comp(Res(P_5, G))$,i.e. one which makes exactly p and q "true", and the one which assigns "true" to only p. So, the answer 'yes' is supported by both parts.

But, the latter model cannot be expanded to a model of $Comp(P_5)$.

That is to say, Condition C is not necessary for the semantic coincidence.

Example 6.

P_6: $r \leftarrow \neg r$.

$r \leftarrow p$.

$p \leftarrow q$.

$p \leftarrow \neg q$.

G: $\leftarrow p$.

$Comp(P_6)$ is $\{ r \leftrightarrow \neg r \vee p,\ p \leftrightarrow q \vee \neg q,\ \neg q\}$ and it has a model in which r and p are exactly assigned "true".

$Res(P_6, G)$ is the follows: $\{ p \leftarrow q,\ p \leftarrow \neg q\}$. Now $Comp(Res(P_6, G))$ has a unique model. (It makes p (only) "true".)

$Prior(P_6, G)=\{r\}$, so is $CIC(P_6)$, thus the Condition E is not met, but the sole model of $Comp(Res(P_6, G))$ can be expanded to a model of $Comp(P_6)$. That is to say, Condition E is not necessary for Condition C.

Although, Example 6 shows that Condition E is not necessary for Condition C, Example 1 and the following Example 7 show that no improvement could be achieved by just putting simple dependency restrictions on programs.

Example 7.

P_7: $p \leftarrow \neg p$.

$p \leftarrow \neg q$.

$q \leftarrow q$.

G: $\leftarrow q$.

P_7 is quite similar to P_1, the only difference is that in P_1, p depends positively on q, but here, the dependency is explicitly negative. One can argue similarly as we did in Example 1 that some model of Comp(Res(P_7, G)) cannot be expanded to a model of Comp(P_7).

5.5. On the Condition of Well-Complementation.

In the proof of Lemma 5.2.5 and Lemma 5.3.6, we need the condition of well-complementation. The reason can be shown by the following example:

Example:

P: $p(x, f(x))$

$q(x, f(x)) \leftarrow \neg q(x, x)$

G: $\leftarrow p(u, v)$

We have the following:

Comp(P)={p(x, f(x)), q(x, f(x)) ↔ ¬q(x, x)}.

Let M=<D, I> such that D={a, b}, $f^I(a)=b$, $f^I(b)=a$, $p^I=\{(a, b), (b, a)\}$ and $q^I=\{(a, b), (b, a)\}$. Obviously, M is a model of Comp(P).

Moreover, Res(P, G)=Ext(P, G)={p(x, f(x))}.

Let M_1=<D, J> such that $f^J(a)=a$, $f^J(b)=b$ and $p^J=\{(a, a), (b, b)\}$. It's also obvious that M_1 is a model of Comp(Ext(P, G)); but it cannot be expanded to a model of Comp(P), as the interpretation of f in M_1 doesn't fit the same function symbol occurring outside Ext(P, G).

Note, In this example, the condition of well-complementation is not satisfied.

It is obvious that if interpretation of function symbols is always fixed, e.g., in practical logic programming, one usually only works with the associated Herbrand Universe; then we do not need the condition of well-complementation. But, generally, when we consider models of Comp(P), we have to consider all the possible interpretation of function symbols([31]), so we need this condition in the general case.

From the above example, we can see that if the condition fails, Ext(P, G) may not be the declaratively-relevant part. But, if we redefine the whole program as the declaratively-relevant part, then semantic coincidence is still equivalent to completeness, and the Condition C provided by Lemma 5.3.4

is still sufficient for semantic coincidence. But, it's no longer trivial to find an effective, sufficient condition similar to condition E for Condition C.

5.6. The Declaratively-Relevant Part in General.

In discussing the declaratively-relevant part in Section 5.2, we made the assumption S that no predicate outside Ext(P, G) depends on any predicate inside Ext(P, G), based on which we showed that every model of Comp(Ext(P, G)) expands to a model of Comp(P). Thus Ext(P, G) completely decides the declarative aspects of P with respect to G under the Completed Program Semantics; so we call it the declaratively-relevant part of the program with respect to the goal.

Generally, the assumption may not be true as the following example shows:

P: $p_1 \leftarrow \neg p_1,\ r$

$p_1 \leftarrow q_1$

$p \leftarrow \neg p,\ q_1$

$p \leftarrow q,\ q_1$

$q \leftarrow q$

$q_1 \leftarrow q_1$

$r \leftarrow$

G: $\leftarrow q$

We have that: Res(P, G)={ q ← q}, Prior(P, G)={p}, Ext(P, G)={ p ← ¬p, q_1; p ← q, q_1; q ← q; q_1 ← q_1} and Q_r(P, G)={q_1}.

Moreover, we have that

Comp(Ext(P, G))={ p ↔ (¬p∧q_1)∨(q∧q_1), q ↔ q, q_1 ↔ q_1 } and Comp(P)={ p_1 ↔ (¬p_1∧r)∨q_1, r}∪Comp(Ext(P, G)).

As M={r, p_1, q_1, p, q} is a model of Comp(P), Comp(P) is consistent. Actually, M is the only model of Comp(P). Therefore, we have that Comp(P) ⊨ q, identity substitution is a correct answer for Comp(P)∪{G}.

On the other hand, there are three models for Comp(Ext(P, G)): ϕ, {q} and {q1, q, p}. Among those models, ϕ cannot expand to the unique model of Comp(P). It's also easy to see that it's not the case that Comp(Ext(P, G)) ⊨ q, so identity is not a correct answer for Comp(Ext(P, G))∪{G}. Obviously, Ext(P, G) should not be regarded as the declaratively-relevant part of P with respect to G.

Therefore, in general, Ext(P, G) doesn't completely determine the declarative behavior under the given semantics. Our theory needs to be generalized.

The problem here is that some other predicates outside Ext(P, G) are involved, and they are possibly relevant to the declarative semantics. We should include them in "the general declaratively-relevant part" as well.

In this particular example, we should regard the whole program as the declaratively-relevant part. A possible process starting from the goal to extend to all the relevant part can be described as the following:

Initially, we have Res(P, G); we extend upward to find Prior(P, G), then downward to find Q_r(P, G); finally, we construct Ext(P, G). If no predicate outside Ext(P, G) depends on Ext(P, G), we should and will stop right here. But as this is not the case in this example, we should extend upward again to find p_1, which is an ancestor of a predicate inside Ext(P, G), and extend downward to find all the descendants of p_1. As we then have exhausted the whole program, we stop here. We may also stop because nothing else depends on or be depended by what we have already found.

In the above process, we extend twice; generally, it can be more. But, as the program is finite, the extension must be finite as well.

We have the following definitions:

Definition 5.6.1. Let P be a program and let R be a subset of pred(P). By Des(P, R), we mean the set consisting of all the descendants of predicates in R. Formally, we have

$$Des(P, R)=\{ q \in Pred(P) \mid \exists p \in R,\ p \geq q\}$$

Note: As we always have $p \geq p$, so R is a subset of Des(P, R).

For example, let P be the program given at the beginning

of this section and R be {p, r, q}, then Des(P, R) will be $R \cup \{q_1\}$.

Definition 5.6.2. Let P be a program and let R be a subset of Pred(P). By Anc(P, R), we mean the set consisting of all the ancestors of predicates in R. Formally, we have

$Anc(P, R) = \{ p \in Pred(P) \mid \exists q \in R, p \geq q \}$.

Similarly, we have that Anc(P, R) is a subset of R.

For example, Let P be the program given before and let R be {q}, then Anc(P, R) will be $R \cup \{p\}$.

Definition 5.6.3. Let P be a normal program and let G be a normal goal. We define a predicate-extension function, ext, as follows:

ext(0, P, G)=Des(P, Pred(G))

ext(i, P, G)=Anc(P, ext(i-1, P, G)) if $i \geq 1$ and i is odd,

ext(i, P, G)=Des(P, ext(i-1, P, G)) if $i \geq 1$ and i is even.

Moreover, let i_0 be the minimum number such that ext(i, P, G) is the same as ext(i+1, P, G); we define the general extension of G with respect to P, denoted EXT(P, G), to be the restriction of P on $ext(i_0, P, G)$, i.e. $EXT(P, G) = Res(P, ext(i_0, P, G))$.

From this definition, we have the following:

ext(0, P, G)=Des(Pred(G))=Pred(Res(P, G));

ext(1, P, G)=Anc(P, Pred(Res(P, G)))

$$=Prior(P, G)\cup Pred(Res(P, G));$$

$$ext(2, P, G)=Des(P, Prior(P, G)\cup Pred(Res(P, G)))$$

$$=Prior(P, G)\cup Pred(Res(P, G))\cup Q_r(P, G)$$

$$Then\ Ext(P, G)=Res(P, Prior(P, G))\cup Res(P, G)$$

$$=Res(P, Prior(P, G)\cup Pred(Res(P, G)))$$

$$=Res(P, Prior(P, G)\cup Pred(Res(P, G))\cup Q_r(P,G)),$$

$$=Res(P, ext(2, P, G))$$

Actually, $ext(2, P, G)=Pred(Ext(P, G))$. Now, if we assume that no predicate outside Ext(P, G) depends on any predicate inside Ext(P, G), then we have $Anc(P, Pred(Ext(P, G))) = Pred(Ext(P, G))$, or $ext(3, P, G)=Anc(P, ext(2, P, G))= ext(2, P, G)$. By definition, we have $i_0=2$, so $EXT(P, G)=Res(P, ext(2, P, G))=Ext(P, G)$. So, the definition we gave before in section 5.2 is really a special case of this general definition.

We said before that as P is finite, i_0 exists. Moreover, as $ext(i_0, P, G)$ is a fixed point of this extension process, it is true that nothing outside EXT(P, G) will be relevant to the predicates inside; in other words, no predicate outside EXT(P, G) will either depend on or be depended on by any predicate inside EXT(P, G). Therefore, predicates in P-EXT(P, G) are disjoint from those in EXT(P, G). It is easy to check that conditions for both Lemma 5.2.3 and Lemma 5.2.5 are met, thus it make sense to call EXT(P, G) <u>the general declaratively-relevant part of</u> P <u>with respect to</u> G.

It is also easy to check that all the results in Section 5.3 up to Lemma 5.3.4 are still valid if we replace Ext(P, G) with EXT(P, G) and replace "declaratively-relevant part" with "general declaratively-relevant part".

As far as a generalization of Lemma 5.3.5 is concerned, we present the following discussion:

Definition 5.6.4. Let P be a normal program and let G be a normal goal. By the prior predicates at i^{th} extension, for i odd, denoted by Prior(i, P, G), we mean the set of proper ancestors of the predicates in ext(i-1, P, G). Formally, we have: Prior(i, P, G)=ext(i, P, G)-ext(i-1, P, G), in which i is odd.

By the extra descendants at ith extension(Q_r(i, P, G)) for i even, we mean the set of predicates which are the descendants of predicates in ext(i-1, P, G), but not in ext(i-1, P, G). Formally, we have: Q_r(i, P, G)=ext(i, P, G)-ext(i-1, P, G), in which i is even.

We have that Prior(P, G)=Prior(1, P, G) and Q_r(P, G)=Q_r(2, P, G).

Notice, Res(P, ext(0, P, G))=Res(P, G). In Lemma 5.3.5, we showed, by using the condition of well-complementation of Res(P, G) in Res(P, G)$\cup$Res(P, Q_r(P, G), how a model of Comp(Res(P, G)) expands to a model of Comp(P, Res(Q_r(2, P, G) $\cup$ ext(0, P, G)), and further expands to a model of Comp(P,

$Res(Q_r(2, P, G) \cup ext(0, P, G) \cup Prior(1, P, G)))$, which is $Comp(P, Res(ext(2, P, G)))$ and can be simplified as $Comp(P, Ext(P, G))$.

Let's assume we have a model, M_i, for $Comp(ext(i, P, G))$, with i even. If i is not i_0, then we can construct $ext(i+1, P, G)$. If $Q_r(i+2, P, G)$ is not empty, then by definition of Q_r, it is disjoint from $ext(i, P, G)$, so if the well-complementation condition holds for $Res(P, Q_r(i+2, P, G))$ in $Res(P, Q_r(i+2, P, G)) \cup Res(P, ext(i, P, G)$, then by the same argument we gave in Lemma 5.2.3, M_i expands to a model of $Comp(Res(P, Q_r(i+2, P, G) \cup ext(i, P, G)))$, and further expands to a model M_{i+2} of $Comp(Res(P, ext(i+2)))$ if P is call-consistent on $Prior(i+1, P, G)$, by the definition of $Prior(i+1, P, G)$ and Kunen's theorem(Lemma 4.2.4). If $Q_r(i+2, P, G)$ is empty, then by the second part of the above argument, M_i can expand to a model of M_{i+1} if the conditions listed there satisfied. incidentally, in the second case, $i_0=i+1$.

The argument for i being odd is similar to the first part of the preceding paragraph, when $Q_r(i+1, P, G)$ is not empty.

Therefore, we have proved the following Theorem 5.6.1, which is a general form of Theorem 5.3.6.

Theorem 5.6.1. Let P be a normal program with a consistent completion, and let G be a normal goal such that G is allowed and regular with respect to P, which is extremely

well-placed for P. Then if for all $i \geq i_0$, no predicate in Prior(i, P, G) depends negatively on itself, then every correct answer for Comp(P)$\cup${G} is an instance of a computed answer for P$\cup${G}.

In the above theorem, by G <u>is extremely well-placed for</u> P, we mean that for all even $i \leq i_0$, ext(i-2, P, G) is well-complemented in Res(P, ext(i-2, P, G)$\cup Q_r$(i, P, G)) and EXT(P, G) is well-complemented in P.

We note that the condition of extreme well-placement and all the other conditions to be satisfied in an application of the preceding theorem are effectively decidable in polynomial time.

We end this section by analyzing the program/goal pair presented at the beginning of this section:

The predicate extension process is as follows:

ext(0, P, G)={q}

ext(1, P, G)={p, q}

ext(2, P, C)={p, q, q_1}

ext(3, P, G)={p_1, p, q, q_1}

ext(4, P, G)={p_1, r, p, q, q_1}

i_0=4.

EXT(P, G)=Res(P, ext(i0, P, G))=P.

As pεPrior(1, P, G) and p $\geq_{-1}$ p, so the sufficient condition in Theorem 5.6.1 fails, the completeness cannot be

guaranteed; actually, it does fail in this case.

We can see that $\{q_1\}$, a model of Comp(Res(P, ext(0, P, G)) $\cup$ Comp(Res(P, Q_r(2, P, G))) (which is Comp(Res(P, G)) $\cup$ Comp(Res(P, Q_r(P, G)))) cannot expand to a model of Comp(Res(P, ext(2, P, G)) (which is Comp(Ext(P, G))) as in any model of the latter, q must be assigned "true". Needless to say that it cannot expand to a model of Comp(EXT(P, G)). Semantic coincidence fails.

5.7. Conclusion.

We start with an interesting example of incompleteness, then define the concepts of procedurally and declaratively-relevant parts of a program with respect to a goal, and develop the idea of the coincidence of the declarative semantics determined by the declaratively-relevant part and the corresponding procedurally-relevant part, and finally obtain an equivalent condition for completeness under a set of reasonable conditions for the programs and goals under study.

In the case when the condition of extreme well-placement is satisfied, we supplied an effective condition for completeness, one which is general and weaker than the conditions previously known.

It is obvious that model expansion is the bottleneck in obtaining any stronger results following this new approach. For general normal program, including those with function symbols, we would like to see conditions weaker than those of Lemma 5.2.3 under which a model can be expanded. That is where further efforts should be made. Other questions for future work include that of finding conditions weaker than Condition E in the case when the extreme well-placement condition is met, and that of finding an effectively testable and sufficient condition for semantic coincidence when well-complementation fails.

Appendix. CALL-CONSISTENCY AND PARADOX.

In logic programming applications, when we apply SLDNF-resolution to solve a goal with respect to a logic program, sometimes there is a derivation starting with that goal, leading to sub-goals, among which is the negation of the original goal.

Example 1:

P: $p \leftarrow q, r$	G_0: $\leftarrow p$
$q \leftarrow q_1, \neg r_1$	G_1: $\leftarrow q, r$
$q1 \leftarrow \neg p, q_1$	G_2: $\leftarrow q_1, \neg r_1, r$
G: $\leftarrow p$	$G_3 \leftarrow \neg p, q_1, \neg r_1, r$

We have an SLDNF-derivation as shown above, it starts with the original goal as G_0, in G_3 $\neg p$ appears.

More formally, we have the following:

Definition 1. Let P be a normal program, p be a predicate symbol defined in P , and let t, t_1 be terms. By saying that p(t) leads to a near-paradox, we mean there is an SLDNF-derivation starting with p(t) and ending with a goal which contains $\neg p(t_1)$ as one of its sub-goals.

We call the derivation $(\leftarrow p(t); \ldots; \leftarrow L_1, \ldots, \neg p(t_1), \ldots, L_n)$ a near-paradox for predicate p in program P. If no such derivation exists for any predicate in P , we call P a near-paradox free program.

Roughly speaking, a near-paradox requires that to solve $p(t)$, we have to solve $\neg p(t_1)$ first. It seems that in some cases near-paradox doesn't make sense. We will make an analysis of near-paradox and consider sufficient conditions to avoid its occurrence.

If we are taking a propositional language as the underlying language of our program, then it's clear that no derivation containing a near-paradox could lead to a refutation.

Intuitively, in this case a near-paradox would require that to show that p is true, you need to show $\neg p$ is true, which obviously leads to nowhere.

Formally, we notice that in order to have a refutation for $P \cup \{\leftarrow p\}$, we must have a finitely-failed tree for $P \cup \{\leftarrow p\}$ first, and, as the near-paradox is always one of the choices to develop the goal $\leftarrow p$, no such tree exists.

Finally, if Comp(P) is consistent, then with the soundness of both SLDNF-resolution and Negation as Failure, we could obtain a contradiction to the assumption of the existence of a near-paradox.

Next, we analyze the situation when a first order language is taken as the underlying language of our program.

With the occurrence of variables, we will see that near-paradox isn't fatal. This is due to the fact that if a

predicate isn't valid, then for some ground terms it could be "true" and for some others it could be false. Therefore, it makes allowable sense that the truth of a ground atomic formula depends on the falsehood of the same predicate with a different ground term. We have the following:

Example 2:

P: $p(a) \leftarrow r(x), \neg p(b)$

$r(a)$

G: $\leftarrow p(x)$

G_0: $\leftarrow p(a)$

G_1: $\leftarrow r(x_1), \neg p(b)$

G_2: $\leftarrow \neg p(b)$ --> $\leftarrow p(b)$

succeeds <-- fails

G_3: $\leftarrow$

Here, we have a near-paradox: (G_0, G_1), which still leads to a refutation. We notice that $p(b)$ is not an instance of $p(a)$; but, even when $p(t)$ is an instance of $P(t_1)$, a near-paradox may lead to a refutation, as shown in the following example:

Example 3:

P: c_1 $p(x) \leftarrow r(x), q(x)$

c_2 $q(a) \leftarrow \neg p(b)$

G: $\leftarrow p(x)$

G_0: $\leftarrow p(x_0)$ c_1, $\Theta_1=\{x_0/x_1\}$

G_1: $\leftarrow r(x_1), q(x_1)$ c_3, $\Theta_2=\{x_1/a\}$

G_2: $\leftarrow q(a)$ c_2, $\Theta_3=\{\}$

G_3: $\leftarrow \neg p(b)$ --> $\leftarrow p(b)$

succeeds <-- fails

G_4: $\leftarrow$

Here we see that $(G_1,\ldots,G_3)$ is a near-paradox , leading to a refutation, even though $p(b)=p(x_0)\cdot\{x_0/b\}$. We notice that $p(x_0)\cdot\Theta_1\cdot\Theta_2\cdot\Theta_3=p(a)$, which is not the same as $p(b)$.

As a special case of near-paradox, we have the notion of a paradox.

Definition 2. Let P be a normal program, p be a predicate symbol defined in P; and let t, t_1, be two terms. By saying that p(t) leads to a paradox in P, we mean that there is an SLDNF-derivation with length n, starting with $\leftarrow p(t)$ to a goal containing $\neg p(t_1)$ as one of its sub-goals, with $p(t)\cdot\Theta = p(t_1)$; here Θ is $\Theta_1\cdot\ldots\cdot\Theta_n$, and each Θ_i is the mgu(most general unifier) generated at step i for the derivation.

We call this derivation $(\leftarrow p(t); \ldots; \leftarrow L_1, \ldots, \neg p(t_1), \ldots, L_n$ a paradox for predicate p in program P. If no such derivation exists for any predicate in P, we call P a paradox -free program.

We have the following lemma to show the need to avoid paradox.

Lemma 1. Let P be an allowed normal program with Comp(p) consistent, and let p(t) be a literal with p defined in P. If p(t) leads to a paradox, then no path in any SLDNF-tree containing the paradox as an initial segment could lead to an SLDNF-refutation.

Proof: Just suppose in an SLDNF-resolution tree for $P\cup\{G\}$, there is path starting with $\leftarrow p(t); \ldots; \leftarrow L_1, \ldots, \neg p(t_1), \ldots, L_n$ which leads to the empty clause. Then we would have refutations for both $P\cup\{\leftarrow p(t)\}$ and

$P \cup \{\leftarrow L_1, \ldots, \neg p(t), \ldots, L_n\}$.

On the one hand, as $P \cup \{G\}$ is allowed, so is every Gi, shown below:

G_0: $\leftarrow p(t)$

G_n: $\leftarrow L_1, \ldots, \neg p(t_1), \ldots, L_n$

G_N: where $\neg p(t1) \cdot \Theta'$ is picked up

G_f: $\leftarrow$

So all variables in t1 must occur in some positive sub-goal, say, L_1; as there is a refutation for $P \cup \{Gn\}$, there must be refutations for $P \cup \{\leftarrow L_1\}$ for all those L_1's. Moreover, as $P \cup \{\leftarrow L_1\}$ is allowed, so by Lloyd's lemma (Lemma 4.3.2.[21]), all variables in L_1 will be grounded as results of those refutations. So are those variables in $p(t_1)$. Therefore, as a result as those refutations, there exists such a goal G_N, $p(t_1) \cdot \Theta'$ is grounded, $\Theta' = \Theta_1 \cdot \ldots \cdot \Theta_N$, here Θ_1 is the mgu produced at step i of the SLDNF-derivation.

From step N on, the safeness condition is always met, thus the ground sub-goal $\neg p(t_1) \cdot \Theta'$ could be selected at any step. As it must be solved to get a refutation for G_N, it has to be selected at some step. Without loss of generality, say it's selected at step N, and a finitely-failed tree surely exists. By the soundness of Negation as Failure, $Comp(P) \models \neg p(t1) \cdot \Theta'$.

On the other hand, suppose $\sigma = \Theta_{N+1} \cdot \ldots \cdot \Theta_f$. Then as we know, $\Theta \cdot \Theta' \cdot \sigma$ is the concatenation of all mgu's produced in this refutation of $P \cup \{\leftarrow p(t)$. By the soundness of SLDNF-resolution,

we have that $Comp(P) \models \forall(p(t)\cdot\Theta\cdot\Theta'\cdot\sigma)$, from the definition of paradox we have $p(t)\cdot\Theta=p(t_1)$, and also $p(t_1)\cdot\Theta'$ is ground. So we have that $Comp(P) \models p(t_1)\cdot\Theta'$.

These results contradict the assumption of the consistence of Comp(P). Therefore, the conclusion of the lemma has been proved.

Next, we'll show that call-consistency is sufficient for the near-paradox freeness. First, we have the following:

Lemma 2. Let P be a normal program, and let p be a predicate symbol defined in P. Suppose there is an SLDNF-derivation in the form of $(\leftarrow p(t), \ldots, \leftarrow L_1, .., L_k, .., L_n)$. Let q be the predicate symbol in L_k, we claim that for all k

(1) if L_k is a positive literal, then $p \geq_{.1} q$, i.e., p depends positively on q;

(2) if L_k is a negative literal, then $p \geq_{-1} q$, i.e., p depends negatively on q.

Proof: By induction on m, the length of the given derivation. If m = 1: then we have a derivation in only one step, as shown below:

G_0: $\leftarrow p(t)$

G_1: $\leftarrow L_1, \ldots, L_k, \ldots, L_n$

Then by definition, p(t) is an instance of the head of a clause in P. The result is obvious by the definition of predicate dependency.

Inductively, we assume the result holds for any

derivation with length less than m.

By definition of SLDNF-resolution, for k, L_k could be obtained in one of only two ways:

(1) L_k is inherited from G_m, then by the inductive assumption, we have the expected result.

(2) $G_{m+1} \leftarrow L_1, .., L, .., L_p$, and L_k is introduced by resolving L with an instance of some clause in P whose head has the same predicate as that in L. From the fact that L must be positive, the inductive assumption and the definition of the dependency relation, we have the result for this case also.

The following result provides a sufficient condition for near-paradox freeness.

Theorem 3. Let P be a normal program. If P is call-consistent, then P is near-paradox free.

Proof: Just suppose not. By definition of paradox, we will have a literal p(t), leading to a goal $\leftarrow L_1, .., \neg p(t_1), .., L_n$ in some SLDNF-resolution tree.

Then, by the above lemma, we immediately have that p depends negatively on p.

As paradox freeness is a special case of near-paradox freeness, we have that call-consistency is sufficient for avoiding paradox in program.

We have the following example to show that in general

call-consistency is not necessary for a program to be paradox-free.

Example 4:

P: $p \leftarrow \neg r, q$

$r \leftarrow q_1, p$

G: $\leftarrow p$

Then, as $p \geq_{-1} r$, $r \geq_{-1} p$, so $p \geq_{-1} p$. So P is not call-consistent. Nevertheless, we don't have a near-paradox for predicate p in program P, i.e., P is paradox free.

Although, paradox is not very likely to occur, and other near-paradoxes shouldn't be disallowed in the practice of logic programming, it is interesting that the call-consistent programs share the property of being near-paradox free.

We could generalize the idea of near-paradox in a natural way as follows.

Definition 3. Let P be a normal program, and G a normal goal. By saying that G leads to a general near-paradox, we mean that there is a derivation starting with G, ending with G', which is another normal goal such that G contain P(t), and G' contains $\neg p(t_1)$, for some terms t and t_1.

We have results for general near-paradox similar to what we have obtained for near-paradox. In particular, we have the

following sufficient condition for general near-paradox freeness.

Theorem 4. Let P be a normal program, G be a normal goal, and let G be strict with respect to P, then G won't lead to a general near-paradox in P.

Proof: Just suppose not. Let $G(\leftarrow L_1,\ldots,p(t),\ldots,L_n)$;...; $G'(\leftarrow L_1',\ldots,\neg p(t_1),\ldots,L_m')$ be the shortest general near-paradox.

As negative literal cannot introduce new literals, the derivation must fall into one of the following two cases:
(1) $p(t)$ is the selected literal and $\neg p(t_1)$ is introduced by either $p(t)$ or its descendant. Then Lemma 2 applies, we have $p \geq_{-1} p$. As G depends positively on p via p and negatively on p via p, G is not strict with respect to P.
(2) $\neg p(t_1)$ is introduced by either L_i for some i, or its descendant. Then L_i must be positive, then by Lemma 2, we have that the predicate in L_i depends on p negatively. So we have that G depends positively on p via p and negatively on p via L_i.

As those are the only possible cases, we have the expected conclusion.

Bibliography.

[1] Apt, K.R., Blair, H. & Walker, A. [1988] *Towards a Theory of Declarative Knowledge,* in Foundations of Deductive Database & Logic Programming (J. Minker ed.), Morgan Kaufmann Publisher, Los Altos, CA, 89-148.

[2] Apt, K.R. & Emden, M.H.van [1982] *Contribution to the Theory of Logic Programming,* J. ACM 29, 841-863.

[3] Barbuti, R. & Martelli, M. [1986] *Completeness of SLDNF-Resolution for a Class of Logic Programs,* in Proceedings of the 3rd International Conference on Logic Programming, London.

[4] Cavedon, L. [1988] *On the Completeness of SLDNF-Resolution,* Technical Report 88/17, Dept. of Computer Science, Univ. of Melbourne.

[5] Cavedon, L. [1988] *Properties of Logic Programs Defined via Atomic Level Mapping,* Technical Report 88/33, Dept. of Computer Science, Univ. of Melbourne.

[6] Cavedon, L. & Lloyd, J. [1989] *A Completeness Theorem for SLDNF-Resolution,* J. Logic Programming 1989:7:177-191.

[7] Chang, C.L. & Lee, R.C.L [1973] *Symbolic Logic and Mechanical Theorem Proving,* Academic Press, New York.

[8] Clark, K.L. [1978] *Negation as Failure,* in Logic & Data Bases (H. Gallaire & J. Minker ed.), Plenum Press, NY, 293-322.

[9] Emden, M.H.van & Kowalski, R.A. [1976] *The Semantics of Predicate Logic as a Programming Language,* J. ACM 1976:23:733-742.

[10] Fitting, M.C. [1985] *A Kripke-Kleene Semantics for Logic Programming,* J. Logic Programming, 4:295-312.

[11] Fitting, M.C. [1986] *Notes on the Mathematical Aspects of Kripke's Theory of Truth,* Notre Dame J. Formal Logic, 1986:27(1).

[12] Fitting, M.C. [1988] *Negation as Refutation,* Dept. of Mathematics and Computer Science of Lehman College, CUNY.

[13] Fitting, M.C. *Bilattices and the Semantics of Logic Programming,* forthcoming in J. Logic Programming.

[14] Gabbay, D.M. & Sergot, M.J. [1986] *Negation as Inconsistency,* J. Logic Programming, 3(1):1-36.

[15] Hill, R. [1974] *Lush-Resolution and its Completeness*, DCL Memo 78, Dept. of Artificial Intelligence. Univ. of Edinburgh.

[16] Jaffar, J., Lassez, J.L. & Lloyd, J.W. [1983] *Completeness of the Negation as Failure Rule*, in Proceedings of the 8th International Joint Conference on Artificial Intelligence, Karlsruhe.

[17] Kleene, S.C. [1953] *Introduction to Mathematics*, North-Holland, Amsterdam.

[18] Kunen, K. [1987] *Negation in Logic Programming*, J. Logic Programming, 1987:4:289-308.

[19] Kunen, K. [1989] *Signed Data Dependencies in Logic Programs*, J. Logic Programming 1989:7:231-245.

[20] Lifschitz, V. [1988] *On the Declarative Semantics of Logic Program with Negation*, in Foundation of Deductive Databases and Logic Programming(J. Minker ed.), Morgan Kaufmann Publisher, Los Altos, CA, 177-192.

[21] Lloyd, J.W. [1987] *Foundation of Logic Programming* (2nd ed.) Spring-Verlag, Symbolic Computation Series.

[22] Mendelson, E. [1987] *Introduction to Mathematical Logic* 3rd edition, Wadsworth,Inc, Montenry, CA.

[23] Minker, J. [1988] *Foundations of Deductive Database & Logic Programming*, Morgan Kaufmann Publisher, Los Altos, CA.

[24] Przymusinska, H. & Przymusinski, T. *Semantic Issues in Deductive Databases & Logic Programs*, Dept. of Computer Sciences, Univ. of Texas at El Paso.

[25] Przymusinski, T. [1989] *On the Deductive & Procedural Semantics of Logic Programs*, J. Automated Reasoning, 5:167-205.

[26] Reiter, R. [1978] *On Closed World Databases*, in Logic and Databases (H. Gallaire & J. Minker ed.) Plenum, NY, 55-76.

[27] Sato, T. [1987] *On Consistency of First Logic Programs*, Tech. Rep. 87/12, Electrotechnical Lab. Ibaraki, Japan.

[28] Sato, T. [1988] *Completed Logic Programs and Their Consistency*, manuscript, Electrotechnical Lab, Ibaraki, Japan. To appear in the Journal of Logic Programming.

[29] Shepherdson, J.C. [1984] *Negation as Failure: A Comparison of Clark's Program Database and Reiter's Closed World Assumption*, J. Logic Programming, 1(1): 51-87.

[30] Shepherdson, J.C. [1988] *Negation as Failure II*, J. Logic Programming 3, 185-202.

[31] Shepherdson, J.C. [1988] *Negation in Logic Programming*, Foundations of Deductive Database & Logic Programming, Minker,J.(ed), Morgan Kaufmann Publisher, Los Altos, CA, 19-88.

[32] Shoenfield, J. [1967] *Mathematic Logic*, Addison-Wesley Publishing Company.

[33] Tarski, A. [1955] *A Lattice-theoretical Fixpoint Theorem and Its Applications*, Pacific Journal of Mathematics, 1955:5:285-309.

[34] *Logic Programming Newsletter*, ALP [1989] Dept. of Computing, Imperial College, 3(1), Jul/Aug.