DISJUNCTIVE DEDUCTIVE DATABASES

by

Charlie Obimbo and Bruce Spencer

TR96-110, August 1996

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
E-mail: fcs@unb.ca
www: http://www.cs.unb.ca

Contents

	Topic	***Page***
1.	Introduction	1
	1.1 Historical Background	2
2.	First-Order Logic	4
	2.1 Syntax	4
	2.2 Semantics	8
3.	Logic Programs	13
	3.1 Syntax	13
	3.2 Semantics	18
	3.3 Substitutions and Unifiers	20
	3.4 Fixpoints	24
4.	Interpretations of Rules	27
	4.1 Proof-theoretic	27
	4.2 Model-theoretic	28
	4.3 Computational Interpretation	30
	4.4 Basic Inference Mechanisms	30
5.	Refutation Procedures	32
	5.1 SLD Resolution	32
	5.2 SLI Resolution	39
	5.3 Clause Trees	49
6.	Negative Information	53
	6.1 Nonmonotonic Reasoning	54
	6.2 Closed World Assumption	55
	6.3 Generalized Closed World Assumption	57
	6.4 Circumscription	60
	6.5 Negation as Failure	61
	6.6 Disjunctive Logic Programs	62

Topic		Page

7. PROLOG 63
 7.1 Components of a PROLOG Program 64
 7.2 Lists 66
 7.3 Recursion 67
 7.4 The CUT (!) Predicate 67

8. DATALOG 69
 8.1 Extensional and Intensional Predicates 70
 8.2 Atomic Formulae 70
 8.3 Dependency Graphs and Recursion 71
 8.4 Safe Rules 73

9. Deductive Databases 76
 9.1 Disjunctive Deductive Databases 77
 9.2 Queries and Answers 78
 9.3 Minimal Models and Deductive Databases 81
 9.4 Stratification 91

Conclusion 92
 Open Problem 92

References 95

Appendices
 Appendix I 103
 Appendix II 105
 Appendix III 108

1 Introduction

... this we do affirm - that if truth is to be sought in every division of Philosophy, we must, before all else, possess trustworthy principles and methods for the discernment of truth. Now the Logical branch is that which includes the theory of criteria and proofs; so it is with this that we ought to make our beginnings.

- *Sextus Empiricus*

The fields of deductive databases and logic programming are intimately related. Theoretical developments in one area have impacted on the other. This is not surprising as both subjects are an outgrowth of work in *automated reasoning, AR* (formerly known as automated theorem proving [*ATP*]). Much exciting work is being done in both of the areas (Minker [*46*]).

In this paper, a brief account is given on the historical background of *Logic Programming* and *Deductive Databases,* starting from their origins in Philosophical and Mathematical Logic. Next we look at a broad overview on the fields including *First-order Logic Theory*; the *Syntax* and *Semantics of Logic Programming*, *Proof-theoretic*, *Model-theoretic* and *Computational Interpretations of rules*; *Proof* (or *Refutation*) *procedures*; *Negative Information* including *General Closed World Assumption* (*GCWA*) and *Non-monotonic reasoning*; use of *PROLOG* and *DATALOG* in the areas; and *Disjunctive Deductive Databases*.

Finally, some related open problems in the literature will be discussed.

1.1 *Historical Background*

Some formalisms gain a sudden success and it is not always immediately clear why. Consider the case of logic programming. It was introduced in an article by Kowalski [*32*] in 1974 and for a long time - in the case of computer science - not much happened. But, twenty-one years later, already the Journal of Logic Programming and Annual Conferences on the subject have been introduced and hundreds of articles on it have been published.

Its success can be attributed to at least two circumstances. First of all, logic programming is closely related to PROLOG. In fact, logic programming constitutes its theoretical framework. This close connection led to the adoption of logic programming as the basis for the influential Japanese Fifth Generation Project. Secondly, in the early eighties a flurry of research on alternative programming styles started and suddenly it turned out that some candidates already existed and had for a considerable time. This led to a renewed interest in logic programming and its extensions.

The power of logic programming stems from two reasons. First, it is an extremely simple formalism. And next, it relies on mathematical logic which developed its own methods and techniques and which provides a rigorous mathematical framework. It should be stated, however, that the main basis of logic programming is *automated theorem proving* which was developed in a large part by computer scientists (Apt [*3*]).

The modern era of automated reasoning is due to the development of the *Robinson Resolution Principle*, described in the landmark paper by *J. Alan Robinson* [57]. The resolution principle is a single rule of inference that permits deductions to be automated in a uniform and simple manner. The ***resolution principle*** simply states as follows:

Let q be an atom (*Definition 2.3(1)*), and let $p_1,\dots,p_n$ be literals (*Definition 2.5*) then

q is a ***logical consequence*** of

$p_1,\dots,p_n$ ***iff*** $(\neg q\ \mathit{AND}\ p_1\ \mathit{AND} \dots \mathit{AND}\ p_n)$ is ***FALSE***.

This is based on "*Proof by contradiction*" (or *argumentum absurdum*). All the same it is difficult not to remember its much earlier origin from philosophical logic - by renown philosophers like *Socrates, Aristotle* and *Plato*. In *Appendix I* we have listed the Rules of Inference and Replacement used in Philosophical logic.

2 *First-Order Logic*

Logic programs are a subset of first-order logic. A first-order logic (theory) consists of an alphabet, a first-order language, a set of axioms, and a set of inference rules. The first-order language consists of the well-formed formulae (*Definition 2.3*) of the theory. The axioms are a designated subset of well-formed formulae. The axioms and rules of inference are used to derive the theorems of the logic (*Lloyd* [*34*]).

First-order logic has two aspects: its syntax and its semantics. The syntactic aspect is concerned with well-formed formulae admitted by the grammar of a formal language, as well as deeper proof-theoretic issues. The semantics is concerned with the meanings attached to the symbols in the well-formed formulae.

In order to define logic programs, we will first have a brief overview of the syntax and semantics of first-order logic.

2.1 *First-Order Logic - syntax*

The syntax of first-order logic is based on an alphabet and the language defined over the alphabet (*LMR* [*35*]).

Definition 2.1 An ***alphabet*** consists of the following classes of symbols:

1. ***variables***, denoted by the upper case English letters (e.g. *W*, *X*, *Y*, *Z*);
2. ***function symbols***, denoted by a finite sequence of the lower case English letters (e.g. *f*, *g*, *h*, *factorial*);

3. ***predicate*** (or ***relation***) ***symbols***, denoted by a finite sequence of the lower case English letters (e.g. *p, q, r, path, top*);
4. ***propositional constants***, *true* and *false*;
5. ***connectives***, ¬ (negation), ∨ (disjunction), ∧ (conjunction), ← or → (implication), and ↔ (equivalence);
6. ***quantifiers***, ∃ (there exists, or *existential quantifier*) and ∀ (for all, or *universal quantifier*);
7. ***punctuation symbols***, '(', ')', ',' and ';'.

Each function and relation symbol has a fixed ***arity***, that is the number of arguments. A function of arity *0* is called a ***constant symbol***. A predicate (or relation) with arity *0* is called a ***propositional symbol***. Each language is said to have an infinite but fixed set of variables.

To avoid having formulas cluttered with brackets, the following precedence hierarchy has been adopted, with the highest precedence at the top:

$$\neg,\ \forall,\ \exists$$

$$\wedge$$

$$\vee$$

$$\rightarrow,\ \leftrightarrow$$

We now turn to the definition of the first order language given by an alphabet.

Definition 2.2 A ***term*** is defined as follows:

1. A variable is a term.
2. A constant is a term.
3. If f is an n-ary function symbol and $t_1,\ldots,t_n$ are terms, then $f(t_1,\ldots,t_n)$ is a term.

Definition 2.3 A ***(well-formed) formula (wff)*** is defined as follows:

1. If p is an n-ary predicate symbol and $t_1,...,t_n$ are terms, then $p(t_1,...,t_n)$ is a formula (called an *atomic formula*, or simply an *atom*).
2. *true* and *false* are formulae.
3. If F and G are formulae, then so are $(\neg F)$, $(F \vee G)$, $(F \wedge G)$, $(F \leftarrow G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$.
4. If F is a formula and X is a variable, then $(\exists XF)$ and $(\forall XF)$ are formulae.

Definition 2.4 A ***first-order language*** is defined as the set of all well-formed formulae constructible from a given alphabet.

Example 2.1 Consider the formulae:

$$\{ (\forall X(\forall Y(\forall Z(path(X,Y,via(Z)) \rightarrow getto(X,Y))))),$$
$$(path(vancouver,\ saskatoon,\ via(calgary)) \vee$$
$$(path(vancouver,\ saskatoon,\ via(edmonton))) \}$$

By dropping pairs of brackets where no confusion is possible and using the above precedence convention, we can write the formulae more simply as:

$$\{ \ \forall X \forall Y \forall Z(path(X,Y,via(Z)) \rightarrow getto(X,Y),$$
$$path(vancouver,\ saskatoon,\ via(calgary)) \vee$$
$$path(vancouver,\ saskatoon,\ via(edmonton)) \ \}$$

which means that for all cities X, Y, and Z, if there is a path from X to Y via Z then one can get from X to Y, and there exists a path from *Vancouver* to *Saskatoon* either via *Calgary* or via *Edmonton*.

Definition 2.5 A ***literal*** is either an atom (say A) or its negation (or compliment $\neg A$). A ***positive literal*** is an atom. A ***negative literal*** is the negation of an atom.

Definition 2.6 A ***ground term*** is a term that does not contain variables. A ***ground atom*** is an atom that does not contain variables. A ***ground literal*** is a literal constructed from a ground atom. A ***ground formula*** is a formula where no variable occurs in the formula.

Example 2.2

teaches(jim, physics) $\vee$ **teaches (jim, math).** - is a ground formula;

whereas,

female(X) $\leftarrow$ **mother(X).** - is not.

In the second part of the example - the formula is by default universally quantified and can thus be read as - for all X, if X is a mother, then X is a female, or in simple terms - all mothers are female.

Definition 2.7 The formula over which a quantifier applies is called the ***scope*** of the quantifier. The *scope* of $\forall X$ (respectively $\exists X$) in $\forall XF$ (respectively $\exists XF$) is F. A ***bound occurrence*** of a variable in a formula is an occurrence within the scope of the quantifier, which has the same variable immediately after the quantifier. Any other occurrence of the variable is ***free***.

Example 2.3 In the formula $\exists X\ path(X,Y) \wedge \exists Y\ city(Y)$ the only occurrence of X and the second occurrence of Y are bound. The first occurrence of Y is free. Also X is bound in F, whereas, Y is both free and bound.

Definition 2.8 A formula is in ***prenex conjunctive normal form*** (***PCNF***) if it has the form

$$Q_1X_1 \ldots Q_kX_k(\ (L_{11}\vee\ldots\vee L_{1m})\wedge\ldots\wedge(L_{n1}\vee\ldots\vee L_{nm})\ )$$

where $Q_i \in \{\ \exists, \forall\ \}$ and L_{ij} is a literal (i.e. a conjunction of disjuncts.)

Definition 2.9 A ***closed formula*** is a formula with no free variable. If F is a formula, then $\forall F$ denotes a ***universal closure***, which is a closed formula obtained by adding a universal quantifier for every variable having a free occurrence in F. The ***existential closure*** of F, $\exists F$ is defined similarly.

Example 2.4 If F is $p(X,Y) \wedge q(X)$,

then $\forall F$ is $\forall X \forall Y (p(X,Y) \wedge q(X))$,

while $\exists F$ is $\exists X \exists Y (p(X,Y) \wedge q(X))$.

2.2 *First-Order Logic - Semantics*

The semantics of first-order logic provides the meaning of the theory based on some interpretation. Interpretations provide specific meaning to the symbols of the language and are used to provide meaning to a set of well-formed formulae (*LMR* [35]).

An interpretation simply consists of some *domain* of discourse over which the variables range, the assignment of each function to a mapping on the domain, and the assignment of each predicate to a relation on the domain. Each interpretation thus specifies a meaning for each symbol in the formula. We are particularly interested in interpretations for which the formula expresses *a true statement* in that interpretation. Such an interpretation is called a ***model*** of the formula (*Lloyd* [34]).

The logic programming systems in which we are interested use the resolution rule as the only inference rule. Suppose we want to prove that the formula

$$\exists y_1,\ldots,\exists y_r (B_1 \wedge \ldots \wedge B_n)$$

is a logical consequence of a program P. Now resolution theorem provers (otherwise known as automated theorem provers) are refutation systems. That is, the negation of the formula to be proved

is added to the axioms and a contradiction is derived. If we negate the formula we want to prove, we obtain the goal

$$\leftarrow B_1,\ldots,B_n$$

Working top-down from the goal, the system derives successive goals. If the empty clause is eventually derived, then a contradiction has been obtained and later results assure us that

$$\exists y_1,\ldots,\exists y_r\,(B_1 \wedge\ldots\wedge B_n)$$

is indeed a logical consequence of P.

From theorem proving point of view, the only interest is to demonstrate *logical consequence* (*Definition 2.18*). However, from a programming point of view, we are much more interested in the bindings that have been made for variables $y_1,\ldots,y_r$, because these give us the output from running the program (*Lloyd* [*34*]).

We now give the definitions of interpretations and models, but first we define a pre-interpretation (all definitions from *LMR* [*35*]).

Definition 2.10 A ***pre-interpretation J*** of a first-order language ***L*** consists of:

1. A non-empty set D, called the *domain* or the *universe* of the pre-interpretation.
2. For each constant L, the assignment of an element in D.
3. For each n-ary function symbol in L, the assignment of a mapping from D^n to D.

Definition 2.11 An ***interpretation I*** of a first-order language ***L*** consists of:

1. A pre-interpretation J with domain D of L.
2. For each n-ary predicate symbol in L a mapping from D^n to $\{true, false\}$.

It is meant that I is based on J.

Definition 2.12 Let J be a pre-interpretation of a first-order language L. A ***variable-assignment*** (with respect to J) is an assignment to each variable in L of an element in the domain of J.

Definition 2.13 Let J (with domain D) be a pre-interpretation of a first-order language L and let V be a variable assignment with respect to J. The ***term-assignment*** (with respect to J and V) of terms in L is defined as follows:

1. Each variable is given an assignment according to V.
2. Each constant is given an assignment according to J.
3. If $t',...,t'$ are term assignments of $t_1,...,t_n$ and f' is the assignment according to J of function symbol f, then $f'(t',...,t') \in D$ is the term assignment of $f(t_1,...,t_n)$.

Definition 2.14 Let I (based on J with domain D) be an interpretation of a first-order language L and let V be a variable assignment with respect to J. Then a formula in L can be given a ***truth value***, *true* or *false* (with respect to I and V) as follows:

1. If the formula is an atom $p(t_1,...,t_n)$ then the truth value is obtained by calculating the value $p'(t',...,t')$ where p' is the mapping assigned to p by I and $t',...,t'$ are atoms assignments of $t_1,...,t_n$ with respect to J and V.

2. If the formula is of the form $\neg F$, $F \vee G$, $F \wedge G$, $F \rightarrow G$, or $F \leftrightarrow G$, then the value of the formula is obtained using the following table:

F	G	$\neg F$	$F \vee G$	$F \wedge G$	$F \rightarrow G$	$F \leftrightarrow G$
true	true	false	true	true	true	true
true	false	false	true	false	false	false
false	true	true	true	false	true	false
false	false	true	false	false	true	true

3. If the formula is of the form $\exists XF$, then the truth value of the formula is *true*, if there exists $d \in D$ such that F has truth value *true* with respect to I and $V(X/d)$, where $V(X/d)$ is V except that X is assigned d; otherwise, its truth value is *false*.

4. If the formula is of the form $\forall XF$, then the truth value of the formula is *true*, for all $d \in D$, F has truth value *true* with respect to I and $V(X/d)$; otherwise its truth value is *false*.

Definition 2.15 Let I be an interpretation of a first-order language L and let F be a formula in L. Then,

1. F is ***satisfiable*** in I if $\exists(F)$ is *true* with respect to I.
2. F is ***valid*** in I if $\forall(F)$ is *true* with respect to I.
3. F is ***unsatisfiable*** in I if $\exists(F)$ is *false* with respect to I (i.e. if F is *not satisfiable*).
4. F is ***nonvalid*** in I if $\forall(F)$ is *false* with respect to I (i.e. if F is *not valid*).

Definition 2.16 Let I be an interpretation of a first-order language L and let F be a formula in L. I is a ***model*** of F if F evaluates to *true* with respect to I. Let S be a set of closed formulae, then I is a *model* of S if I is a model of each formula of S. This is denoted as $I \models S$.

Example 2.5 (*Lloyd* [*34*]) Let $F = \forall X(\exists Y(p(X,Y)))$ be a formula. Consider an interpretation I with domain D, the set of non-negative integers, and let p be assigned the relation $<$. Then I is a model of F. In I, F expresses the true statement that for every non-negative integer, there exists a non-negative integer which is strictly larger than it. On the other hand if $G = \exists Y(\forall X(p(X,Y)))$, then our interpretation I above would not be a model of G.

Definition 2.17 Let S be a set of closed formulae. When S has a model, it is ***consistent*** (or *satisfiable*). Otherwise, when S has no model, it is ***inconsistent*** (or *unsatisfiable*).

When every interpretation is a model of S then S is valid.

Definition 2.18 Let S be a set of closed formulae of a first-order language L and let F be a closed formula in L. F is a logical consequence of S if for each model M of S, M is also a model of F. This is denoted as $S \models F$.

Let S' be a set of closed formulae of a first-order language L. S' is a ***logical consequence*** of S if each formula in S' is a logical consequence of S. This is denoted as $S \Rightarrow S'$.

Proposition 2.1 Let S be a set of closed formulae of a first order language L and let F be a closed formula of L. F is a *logical consequence* of S iff $S \cup \{\neg F\}$ is unsatisfiable.

Proof (*Lloyd* [*34*]) Suppose that F is a *logical consequence* of S. Let I be an interpretation of L and suppose that I is also a model for S. Then I is also a model for F. Hence I is not a model for $S \cup \{\neg F\}$. Thus $S \cup \{\neg F\}$ is unsatisfiable.

Conversely, suppose $S \cup \{\neg F\}$ is unsatisfiable. Let I be an interpretation of L and suppose that I is a model for S. Since $S \cup \{\neg F\}$ is unsatisfiable, I cannot be a model for $\neg F$. Thus I is a model for F and so F is a logical consequence of S.

Example 2.6 (*Lloyd* [*34*]) Let $S = \{ p(a), \forall X(p(X) \rightarrow q(X)) \}$ and F be $q(a)$. We show that F is a logical consequence of S. Let I be any model for S. Thus $p(a)$ is true wrt I. It follows from this, and the truth of $\forall X(p(X) \rightarrow q(X))$ that $q(a)$ must be true wrt I.

3 Logic Programs

Logic programming began in the early 1970's as a direct outgrowth of earlier work in *ATP* and *AI*. The credit for the introduction of logic programming goes mainly to *Kowalski* [*32*] and *Colmerauer* [*16*], although *Green* [*25*] and *Hayes* [*26*] should be mentioned in this regard (*Lloyd* [*34*]).

Logic programs are simply sets of certain formulae of first-order language (discussed earlier see pp 6-7).

3.1 Logic Programs - Syntax

Logic programming deals with a specific class of *wff* called clauses. Each statement clause in a logic program can be viewed as consisting of two distinct parts: an antecedent and a consequent. Such delineation provides a declarative meaning for a clause in that the consequent is *true* when the antecedent is *true*. It also translates into procedural meaning where the consequent can be viewed as a problem which is to be solved by reducing it to a set of sub-problems given by the antecedent (*LMR* [*35*]). Below are precise definitions of the syntax of logic programs.

Definition 3.1 A ***disjunctive clause*** (or simply a ***clause***) is a formula of the form

$$\forall X_1 \cdots \forall X_n (L_1 \vee \ldots \vee L_m)$$

where $X_1,\cdots,X_n$ are all the variables that occur in literals $L_1,\ldots,L_m$.

The universal quantifier is normally omitted while writing a disjunctive clause. When $L_1,\ldots,L_m$ are atoms the clause is called ***positive disjunctive clause***. When $L_1,\ldots,L_m$ are compliments of atoms the clause is called a ***negative disjunctive clause***. A *ground disjunctive clause* is a disjunctive clause formed with ground literals. A *positive disjunctive ground clause* is a disjunctive clause formed with ground atoms. A *negative disjunctive ground clause* is a disjunctive clause formed with negative literals. A *subclause* of a clause C is a clause formed by eliminating one or more literals from C.

Definition 3.2 A ***program clause*** is a special representation of a clause. Let a clause be of the form

$$\forall X_1 \cdots \forall X_n(A_1 \vee \ldots \vee A_k \vee \neg B_1 \vee \ldots \vee \neg B_m)$$

where $A_1,\ldots,A_k,B_1,\ldots,B_m$ are atoms. Then the corresponding program clause representation is

$$A_1 \vee \ldots \vee A_k \leftarrow B_1, \ldots, B_m$$

with $k \geq 1$ and $m \geq 0$. This is also known as the *clausal form* (or *clausal notation*).

Thus all variables in a program clause are considered to be universally quantified. The commas between atoms after the implication sign denote conjunction symbols. $A_1 \vee \ldots \vee A_k$ is called the *head* of the program clause (sometimes called the *conclusion*) and $B_1, \ldots, B_m$ is called its *body* (*premises*).

Example 3.1 Thus the formula

$$\forall X \forall Y(p(X) \vee \neg a \vee \neg q(Y) \vee b)$$

looks in clausal form as:

$$p(X) \vee b \leftarrow a, q(Y).$$

If a clause has only one conclusion ($k=1$), then it is called a ***definite clause***. When the set of premises of a program clause is empty ($n=0$), then we talk of a ***unit clause***. They have the form $A\leftarrow$. (or simply A.)

When the set of conclusions is empty ($k=0$), then we talk of a ***goal*** or a ***negative clause***. They have the form $\leftarrow B_1,...,B_n$. Finally, when both the sets of premises and conclusions is empty then we talk of the ***empty clause*** and denote it by $\square$. It is interpreted as a contradiction.

Definition 3.3 A ***definite*** (*logic program*) ***clause*** (or a ***horn clause***) is a program clause of the form:

$$A \leftarrow B_1, ..., B_m$$

with $m \geq 0$ where $A, B_1, ..., B_m$ are atoms. If $m = 0$, it is called a ***definite assertion***, or simply an ***assertion***.

Definition 3.4 A ground clause $\boldsymbol{A = A_1 \vee ... \vee A_n}$ is a subclause of a ground clause $\boldsymbol{B = B_1 \vee ... \vee B_m}$ if for each A_i, $1 \leq i \leq n$, there is a B_j such that $A_i = B_j$. The clause $\boldsymbol{A}$ is said to be a ***proper subclause*** of $\boldsymbol{B}$ if it is a subclause and there is a $\boldsymbol{B_j}$, $1 \leq j \leq \boldsymbol{m}$, such that for all A_i, $1 \leq i \leq n, B_j \neq A_i$.

Example 3.2 If $\boldsymbol{A = p(X,Y) \vee q}$ and $\boldsymbol{B = p(X,Y) \vee q \vee t(f(Y),Z)}$ then $\boldsymbol{A}$ is a proper subclause of $\boldsymbol{B}$. We say that A ***subsumes*** B.

Definition 3.5 An *indefinite* or ***disjunctive logic program clause*** is a clause of the form:

$$A_1 \vee ... \vee A_k \leftarrow B_1, ..., B_m$$

with $k \geq 1$ and $m \geq 0$, where $A_1, ..., A_k, B_1, ..., B_m$ are atoms. When $m = 0$, it is called a ***disjunctive assertion***.

While representing definite and disjunctive assertions, normally, the implications signs is not included and is assumed to be present after the clause. Thus:

$$A_1 \vee ... \vee A_k \leftarrow$$

is written as

$$A_1 \vee ... \vee A_k.$$

Definition 3.6 A ***definite logic program*** (or *Horn program*) is a finite set of definite logic program clauses.

Definition 3.7 A ***disjunctive logic program*** is a finite set of program clauses.

That is a disjunctive logic program contains either definite logic clauses, disjunctive clauses or both definite and disjunctive clauses.

Definition 3.8 A ***logic program*** is either a definite or a disjunctive logic program. The letters ***P*** and ***Q*** will normally be used to denote logic programs.

Definition 3.9 In a logic program the set of all program clauses with the same predicate ***p*** in the head is called the ***definition*** of ***p***.

Example 3.3 Let ***P*** be a disjunctive logic program, defined as follows:

P = {
goto(X,Y) ∨ *goto(X,Z)* ← *at(Y), path(X,Y,via(Z)), day(sunday);*
at(Y) ← *day(sunday);*
at(Y) ∨ *day(sunday)*
}

The definition of the predicate symbol ***goto*** is

goto(X,Y) ∨ *goto(X,Z)* ← *at(Y), path(X,Y,via(Z)), day(sunday);*

The definition of the predicate symbol ***at*** is

$$at(Y) \leftarrow day(sunday);\ \ at(Y) \vee day(sunday)$$

The definition of the predicate symbol ***day*** is

$$at(Y) \vee day(sunday)$$

There is no definition of the predicate symbol ***path***.

Definition 3.10 A ***goal clause*** denoted by G (with or without subscripts) is a clause of the form:

$$\leftarrow B_1, \ldots, B_m$$

with $m > 0$. That is the head is empty. Each B_i ($i = 1,\ldots,m$) is a literal and is called a ***subgoal*** of the clause.

The concept of a query of a logic program is given next.

Definition 3.11 A ***query*** is a formula of the form:

$$\exists(A_1 \wedge \ldots \wedge A_n)$$

where $n > 0$ and $A_1,\ldots,A_n$ are atoms with the variables existentially quantified.

Observe that a goal clause

$$\leftarrow A_1, \ldots, A_n$$

is the negation of a query defined above.

3.2 Logic Programs - Semantics

Applying the definitions above to programs, we see that when we give a goal G to the system to show that the set of clauses $P \cup \{G\}$ is unsatisfiable. In fact, if G is the goal $\leftarrow B_1,...,B_m$ with variables $y_1,...y_r$ then *Proposition 2.1 (p 12)* states that showing $P \cup \{G\}$ is unsatisfiable is exactly the same as showing that $\exists y_1,...,\exists y_r(B_1 \wedge ... \wedge B_m)$ is a logical consequence of P.

Thus the basic problem is that of determining the unsatisfiability, or otherwise, of $P \cup \{G\}$, where P is a program and G is a goal. According to the definition, this implies showing *every* interpretation of $P \cup \{G\}$ is not a model. Needless to say, this seems to be a formidable problem. However, it turns out that there is a much smaller and more convenient class of interpretations, which are all that need to be investigated, to show unsatisfiability. These are the so-called ***interpretations***, which we now proceed to study.

Definition 3.12 Let L be a first order language, and let P be the disjunctive logic program in L. The ***Herbrand Universe*** for L (resp. P), denoted by U_L (resp. U_P), is the set of all ground terms (*Definition 2.6*) which can be formed from the constants and function symbols appearing in L. If L has no constants, we add some constant, say a, to from ground terms.

Example 3.4 Consider the program

```
{
        p(X) ← q(f(X), g(X)).
        r(X).
}
```

which has an underlying first-order language L based on the predicates P, q, and r and the functions f and g. Then the *Herbrand Universe* for L is

$$\{ a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a)), ... \}.$$

Observe that the Herbrand Universe of a first-order language L and the Herbrand Universe of a disjunctive logic program in L are the same. When the underlying language of a logic program P is not mentioned we assume that the language is formed by the constants and function symbols that appear in the program. For our purposes, henceforth, we will simplify the definitions - referring to the language L taking into consideration that the same applies to the program P as well.

Definition 3.13 Let L be a first-order language. The ***Herbrand Base***, HB_L, for L is the set of all ground atoms which can be formed by using predicates from L with ground atoms from the *Herbrand Universe* of arguments.

Example 3.5 For the previous example, the Herbrand base for L is

$$\{\ p(a),\ q(a,a),\ r(a),\ p(f(a)),\ p(g(a)),\ q(a,f(a)),\ q(f(a),a),\ \ldots\ \}.$$

Definition 3.14 Let L be a first-order language and let P be a disjuncive logic program in L. The ***disjunctive Herbrand base*** of L (resp. P), denoted by DHB_l (resp. DHB_P) is the set of all positive disjunctive ground clauses which can be formed using *distinct* ground atoms from the Herbrand base L (resp. P), such that no two logically equivalent clauses are in the set.

Example 3.6 For the program in Example 3.4, the disjunctive Herbrand base for L is

$$
\begin{aligned}
\{ & \\
& p(a),\ q(a,a),\ r(a), \\
& p(a) \vee q(a,a),\ p(a) \vee r(a),\ q(a,a) \vee r(a), \\
& p(a) \vee q(a,a) \vee r(a), \\
& p(f(a)),\ p(g(a)),\ q(a,f(a)),\ q(f(a),a), \\
& p(f(a)) \vee g(a,a), \\
& \ldots \\
\}. &
\end{aligned}
$$

Definition 3.15 Let L be a first-order language and let P be a disjuncive logic program in L. A ***Herbrand state*** (or simply a ***state***) S for L (resp. P) is a subset of the disjunctive Herbrand base L (resp. P).

Definition 3.16 Let L be a first-order language. An interpretation for L is a ***Herbrand interpretation*** if the following conditions are satisfied:

a. The domain of the interpretation is the Herbrand Universe U_L.
b. Constants in L are assigned to "themselves" in U_L.
c. If f is an n-ary function in L, then f is assigned to the mapping from $(U_L)^n$ into U_L defined by

$(t_1,...,t_n) \rightarrow f(t_1,...,t_n)$.

3.3 Substitutions and Unifiers

The substitution of variables for terms in a formula plays an important role in the theory of logic programs. Answers for queries posed to logic programs (including those in deductive databases) are usually represented by substitutions. We hereby introduce some basic concepts of substitution and unifiers that are normally used to derive answers.

Definition 3.17 A ***substitution*** θ is a finite mapping from variables to terms and is written as

$$\theta = \{ X_1 = t_1, ..., X_n = t_n \}.$$

The pair $X_i = t_i$ is called a ***binding***. The variables $X_1,...,X_n$ become bound to the terms $t_1,...,t_n$ respectively. The X_i's are all distinct variables and each term t_i is a term distinct from X_i. If $t_1,...,t_n$ are distinct variables then θ is called a ***renaming substitution***.

Substitutions operate on expressions defined as follows:

Definition 3.18 An ***expression*** is either a term, a literal, or a conjunction or disjunction of literals. A ***simple expression*** is either a term or an atom.

Definition 3.19 Let $\theta = \{ v_1/t_1, ..., v_n/t_n \}$ be a substitution and E be an expression. Then $E\theta$, the ***instance*** of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variable v_i in E by the term t_i ($i = 1,...,n$). If $E\theta$ is ground, then $E\theta$ is called a ***ground instance*** of E.

Example 3.7 Let $E = p(X, Y, f(a))$ and $\theta = \{X/b, Y/Z\}$.
Then $E\theta = p(b, Z, f(a))$.

If $S = \{E_1,...,E_n\}$ is a finite set of expressions and θ is a substitution, then $S\theta$ denotes the set $\{E_1\theta,...,E_n\theta\}$.

Definition 3.20 Two expressions are ***standardized apart*** if they do not have any variable symbols in common.

Definition 3.21 (*Lloyd* [32]) Let $\theta = \{u_1/s_1,...,u_m/s_m\}$ and $\sigma = \{v_1/t_1,...,v_n/t_n\}$ be substitutions. Then the ***composition*** $\theta\sigma$ of θ and σ is the substitution obtained from the set

$$\{u_1/s_1\sigma,...,u_m/s_m\sigma, v_1/t_1,...,v_n/t_n\}$$

by deleting any bindings $u_i/s_i\sigma$ for which $u_i = s_i\sigma$ and deleting any bindings v_j/t_j for which $v_j = \{u_1,...,u_m\}$.

Example 3.8 Let $\theta = \{X/f(Y), U/Z\}$ and $\sigma = \{X/a, Y/b, Z/W\}$. Then $\theta\sigma = \{X/f(b), U/W, Y/b, Z/W\}$.

Definition 3.22 Let E and F be expressions. We say that E and F are ***variants*** if there exist substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$. We also say that E is a variant of F or F is a variant of E.

Example 3.9 The expression *p(f(X,Y), g(Z), a)* is a variant of *p(f(Y,X), g(U), a)*[1]. However, *p(X,X)* is not a variant of *p(X,Y)*.

We will be particularly interested in substitutions which unify a set of expressions, that is, make each expression in the set syntactically identical. Below we define *unifiers* and related terms.

Definition 3.23 Let S be a finite set of simple expressions. A substitution θ is called a ***unifier*** for S if $S\theta$ is a singleton. A unifier θ for S is called a ***most general unifier*** (***mgu***) for S, if for each unifier σ of S, there exists a substitution τ such that $\sigma = \theta\tau$.

Example 3.10 {*p(f(X),Z), p(Y,a)*} is unifiable, since σ = {*Y/f(a), X/a, Z/a*} is a unifier. A most general unifier is θ = {*Y/f(X), Z/a*}. Note that $\sigma = \theta$\{*X/a*\}.

Definition 3.24 A clause C ***subsumes*** a clause D iff there is a substitution θ such that $C\theta$ is a subclause of D. A clause C, θ-***subsumes*** a clause D iff $C\theta$ subsumes $D\theta$.

Procedural semantics describe syntactic transformation rules that permit us to obtain logical consequences from programs. The next definition describes a very simple rule which is the fundamental tool used in procedural semantics.

Definition 3.25 (*Loveland* [37]) (***Binary Resolution***) Let A and B be clauses which are standardized apart, where each clause is represented as a set of literals, such that L_1 and L_2 are complimentary

[1] Of course there arises a necessity of standardizing them apart first.

literals with $L_1 \in A$ and $L_2 \in B$. Let θ be a unifier for the atoms in L_1 and L_2. Then a ***(binary) resolvent*** of A and B is given by $(A\theta - \{L_1\theta\}) \vee (B\theta - \{L_2\theta\})$.

The notation X - Y implies that the literals from the clause Y are removed from the clause X (i.e. set difference).

Definition 3.26 (*Loveland* [*37*]) Let A and B be clauses and let C be a binary resolvent of A and B. then C is an ***immediate consequence*** of A and B.

Definition 3.27 Let S be a set of clauses. A derivation of the empty clause $\square$ for S is called a ***refutation***, or ***proof*** of S.

Proposition 3.1 Let A and B be in clausal form and let C be an immediate consequence of A and B, then C is a logical consequence of $\{A, B\}$.

Proof: See *Chang* and *Lee* [*10*] p 17.

Corollary 3.1 Let C be a clause derivable from a set S of clauses. Then C is a logical consequence of S.

3.4 Fixpoints

Van Emden and *Kowalski* [67] proposed an elegant way of studying logic programs without negation. Their definitions still make perfect sense in the presence of negation. Their idea was to use a natural closure operator and equate the models of a program P with the pre-fixed points of the operator, which are simpler to analyze. This operator is usually called $\boldsymbol{T_p}$. It maps interpretations of P into interpretations of P and is defined as follows:

$A \in T_p(I)$ iff for some clause $A_1 \leftarrow L_1 \vee ... \vee L_m$ in P and substitution θ, $I \models L_1 \vee ... \vee L_m$ and $A = A_1\theta$.

Intuitively, $T_p(I)$ is the set of immediate conclusions of I, i.e. those which can be obtained by applying a rule from P ***only once***. Note that $A \in T_p(I)$ iff there exists a clause in *ground(P)* with head A whose body is true in I.

In this section we introduce the requisite concepts of monotonic mappings and their fixpoints.

Definition 3.28 Let S be a set. A binary ***relation*** R on S is a subset $S \times S$. Infix notation is used to denote a relation - thus $(x,y) \in R$ becomes xRy.

Definition 3.29 A relation R on a set S is a ***partial order*** if the following conditions are satisfied:

(a) $xRx, \ \forall x \in S$.
(b) xRy and yRx imply $x=y \ \ \forall x,y \in S$. *(Reflexive)*
(c) xRy and yRz imply $xRz \ \ \forall x,y,z \in S$. *(Transitive)*

We adopt the standard notation and use $\leq$ to denote a partial order. Thus we have $x \leq x$; $x \leq y$ and $y \leq x$ imply $x = y$; and $x \leq y$ and $y \leq z$ imply $x \leq z \ \ \forall x,y,z \in S$.

Definition 3.30 Let S be a set with partial order $\leq$. Let X be a subset of S. Then $a \in S$ is an ***upper bound*** of X if $x \leq a$, $\forall x \in X$. Similarly $b \in S$ is an ***lower bound*** of X if $b \leq x$, $\forall x \in X$.

Definition 3.31 Let S be a set with partial order $\leq$. Let X be a subset of S. Then $a \in S$ is the ***least upper bound*** of X (denoted by ***lub(X)***) if a is an upper bound of X, and for all upper bounds a' of X, $a \leq a'$. Similarly $b \in S$ is the ***greatest lower bound*** of X (denoted by ***glb(X)***) if b is a lower bound of X, and for all lower bounds b' of X, $b' \leq b$.

The *lub(X)* and the *glb(X)* are unique if they exist.

Definition 3.32 A partially ordered set L is a ***complete lattice*** if *lub(X)* and *glb(X)* exist for every subset X of L.

The symbol _ is used to denote the *top element*, *lub(L)* and the symbol $\perp$ is used to denote the *bottom element*, *glb(L)*, of a complete lattice L.

Example 3.11 Let S be a set and 2^S be the set of all subsets of S. Then set inclusion, $\subseteq$, is a partial order on 2^S. Also, 2^S is a complete lattice under $\subseteq$. The *least upper bound* of a collection of subsets of S is their *union* and the *greatest lower bound* is their *intersection*. The top element is S and the bottom element is $\varnothing$.

Definition 3.33 Let L be a complete lattice and $T: L \rightarrow L$ be a mapping. The mapping T is ***monotonic*** if for all $a_1, a_2 \in L$, $a_1 \leq a_2$ implies $T(a_1) \leq T(a_2)$.

Definition 3.34 Let L be a complete lattice and X be a subset of L. X is a ***directed set*** if every finite subset of X has an upper bound in X.

Definition 3.35 Let L be a complete lattice and $T: L \rightarrow L$ be a mapping. T is a ***continuous mapping*** if $T(lub(X)) = lub(\{T(a) \mid a \in X\})$ for every directed subset X of L.

Every continuous mapping is monotonic. However, the converse is not true.

Definition 3.36 Let L be a complete lattice and $T: L \rightarrow L$ be a mapping. Then $a \in L$ is a ***fixpoint*** of T (denoted by $fp(T)$) if $T(a) = a$. Furthermore, a is the ***least fixpoint*** of T (denoted by $lfp(T)$) if a is a fixpoint of T and for all fixpoints b of T, $a \leq b$. The ***greatest fixpoint*** of T (denoted by $gfp(T)$) is defined similarly. An element $a \in L$ is a ***pre-fixpoint*** of T if $T(a) \leq a$.

Theorem 3.1 (*Knaster* [30], *Tarski* [65]) Let L be a complete lattice and $T: L \rightarrow L$ be a monotonic mapping. Then T has a least fixpoint, $lfp(T)$, and a greatest fixpoint, $gfp(T)$.

Proof: See *Lobo, Minker* and *Rajasekar* [35] p50.

Definition 3.37 Let L be a complete lattice and $T: L \rightarrow L$ be a monotonic mapping. Then the ***powers*** of T are defined by

$$T \uparrow 0 = \perp$$
$$T \uparrow \alpha = T(T \uparrow (\alpha - 1)), \text{ if } \alpha \text{ is a successor ordinal}$$
$$T \uparrow \alpha = lub(T \uparrow \beta \mid \beta < \alpha), \text{ if } \alpha \text{ is a limit ordinal}$$
$$T \downarrow 0 = \top$$
$$T \downarrow \alpha = T(T \downarrow (\alpha - 1)), \text{ if } \alpha \text{ is a successor ordinal}$$
$$T \downarrow \alpha = glb(T \downarrow \beta \mid \beta < \alpha), \text{ if } \alpha \text{ is a limit ordinal.}$$

4 Interpretation of Rules

There are two main alternatives for interpreting the *theoretical meaning* of rules:

- ***proof-theoretic*** and
- ***model-theoretic***.

A third approach to interpreting the meaning of rules involves defining an inference mechanism that is used by the system to deduce facts from the rules. This inference mechanism is a computational procedure, and hence provides a ***computational interpretation*** to the meaning of the rules (*Elmasri* [*18*]).

4.1 Proof-theoretic Interpretation

In this interpretation, we consider the facts and rules to be true statements, or ***axioms***. ***Ground axioms*** contain no variables. The facts are ground axioms that are given to be true. Rules are called ***deductive axioms***, since they can be used to deduce new facts from existing facts.

The proof-theoretic interpretation gives us a procedural or computational approach for computing an answer to the Datalog query.

4.2 *Model-theoretic Interpretation*

Here given a *finite* or an *infinite* domain of constant values, we assign to a predicate every possible combination of values as arguments. We then determine for which arguments the predicate is true and for which it is false. It is sufficient to specify the combinations of arguments that make the predicate true, and to state that all other combinations make the predicate false. If this is done for every predicate, it is called an interpretation of the set of predicates.

An interpretation is called a ***model*** for a *specific set of rules* if those rules are *always true* under that interpretation. In the following example, we illustrate a model theoretic interpretation for the given set of rules (*Elmasri* [*18*]).

Information

(i) Los Angeles, New York, Chicago, Atlanta, Frankfurt, Paris, Jakarta, and Sydney are cities.
(ii) The following flights exist: LA to NY, NY to Atlanta, Atlanta to Frankfurt, Frankfurt to Atlanta, Frankfurt to Jakarta, and Jakarta to Sydney.

[*Note*: No flight in reverse direction can automatically be assumed.]

Consider the following rules:

```
reachable(X, Y) :- flight(X, Y).
reachable(X, Z) :- flight(X, Y), reachable(Y, Z).
```

An Interpretation that is a model for the two rules listed:

Rules

reachable(X, Y) :- flight(X, Y).
reachable(X, Z) :- flight(X, Y), reachable(Y, Z).

Known facts

```
flight(la       ,  ny       ) is true.
flight(ny       ,  atlanta  ) is true.
flight(atlanta  ,  frankfurt) is true.
flight(frankfurt,  atlanta  ) is true.
flight(frankfurt,  jakarta  ) is true.
flight(jakarta  ,  sydney   ) is true.
```

Interpretation

Derived facts

```
reachable(la       , ny       ) is true.
reachable(ny       , atlanta  ) is true.
reachable(atlanta  , frankfurt) is true.
reachable(frankfurt, atlanta  ) is true.
reachable(frankfurt, jakarta  ) is true.
reachable(jakarta  , sydney   ) is true.
reachable(la       , atlanta  ) is true.
reachable(la       , frankfurt ) is true.
reachable(la       , jakarta  ) is true.
reachable(la       , sydney   ) is true.
reachable(atlanta  , jakarta  ) is true.
reachable(atlanta  , sydney   ) is true.
reachable(frankfurt, atlanta  ) is true.
reachable(X, Y) is false for all other possible (X,Y) combinations.
```

4.3 Computational Interpretation

The third way to define the meaning of logical rules is to provide an algorithm for *executing* them, to tell whether a potential fact (predicate with constants for its arguments) is *true* or *false*. For example, PROLOG defines the meaning of rules in this way, using a particular algorithm that involves searching for proofs of the potential fact. Unfortunately, the set of facts for which PROLOG finds a proof this way is not necessarily the same as the set of all facts for which a proof exists. Neither is the set of facts PROLOG finds true, necessarily a model. However, in many cases, PROLOG will succeed in producing the unique minimal model for a set of rules when those rules are run as a PROLOG program.

4.4 Basic Inference Mechanisms

There are two main inference mechanisms that are based on the *proof-theoretic* interpretation of rules.

4.4.1 Bottom-Up Inference Mechanisms (Forward Chaining)

In bottom-up inference (also called ***bottom-up resolution***), the inference engine starts with the facts and applies the rules to generate new facts. As facts are generated, they are checked against the query predicate goal for a match.

4.4.2 Top-Down Inference Mechanisms (Backward Chaining)

The top-down inference mechanism is the one used in PROLOG interpreters. ***Top-down resolution*** starts with the query predicate goal and attempts to find matches to the variables that lead to valid facts in the database. The term *backward chaining* indicates that the inference moves backward from the intended *goal* to determine the rules and facts that would satisfy the goal. In this approach, facts are not explicitly generated, as they are in forward chaining.

5 Refutation Procedures

There are many refutation procedures based on the resolution inference rule, which are refinements of the original procedure of *Robinson* [*56*]. We hereby discuss two of them, one for definite (*Horn*) clauses - the *SLD Resolution*, and the other for indefinite clauses - the *SLI Resolution*. We also use two data structures to explain the resolution steps, and these are *t-clauses* (*LMR* [*35*]) and *clause trees* introduced in the paper by *Horton* and *Spencer* [*29*].

5.1 SLD Resolution

SLD[2] resolution provides a way to compute answers for a goal from a *definite logic program*. It starts with a goal clause, $\leftarrow A_1,\ldots,A_n$ and provides a refutation by deriving an empty goal clause. In the process of refuting the goal, the composition of substitutions made, restricted to the variables in the goal clause, provides an answer substitution for the goal clause. Answers computed with *SLD* resolution are sound and complete with respect to the correct answers provided by the declarative meaning. That is, all the computed answers are correct answers and all the correct answers can be computed using *SLD* resolution (*Lloyd* [*34*]).

We first define answers and correct answers:

Definition 5.1 Let P be a disjunctive logic program and let G be a goal of the form $\leftarrow A_1,\ldots,A_n$. An ***answer*** for $P \cup \{G\}$ is a set of substitutions for variables of G.

Definition 5.2 Let P be a disjunctive logic program and let G be a goal of the form $\leftarrow A_1,\ldots,A_k$, where A_i, $1 \le i \le k$, are atoms. Let $\{\theta_1,\theta_2,\ldots,\theta_n\}$ be an answer for $P \cup \{G\}$. The answer $\{\theta_1,\theta_2,\ldots,\theta_n\}$ is a ***correct answer*** for $P \cup \{G\}$ if

$$\forall((A_1\wedge\ldots\wedge A_k)\,\theta_1 \vee (A_1\wedge\ldots\wedge A_k)\,\theta_2 \vee\ldots\vee (A_1\wedge\ldots\wedge A_k)\,\theta_n)$$

---

[2] *SLD* - ***S****elected rule-driven* ***L****inear resolution for* ***D****efinite clauses.*

is a logical consequence of P.

Definition 5.3 A ***computation rule*** is a function from a set of goals to a set of atoms, such that the value of the function for a goal is always an atom, called the *selected* atom, in that goal.

Definition 5.4 Let G_i be $\leftarrow A_1,\ldots,A_k,\ldots,A_m$, C_{i+1} be $A \leftarrow B_1,\ldots,B_q$ and $\boldsymbol{R}$ be a computation rule. Then G_{i+1} is derived from C_{i+1} using mgu θ_{i+1} via $\boldsymbol{R}$ if the following conditions hold:

(a) A_k is the selected atom given by the computation rule $\boldsymbol{R}$.
(b) $A_k\theta_{i-1} = A\theta_{i-1}$ (that is, θ_{i-1} is an mgu of A_k and A).
(c) G_{i+1} is the goal $\leftarrow(A_1,\ldots,A_{k-1},B_1,\ldots,B_q,A_{k+1},\ldots,A_n)\theta_{i+1}$.

In resolution terminology, G_{i+1} is a ***resolvent*** of G_i and C_{i+1}.

Definition 5.5 Let P be a program, G a goal and R a computation rule. An ***SLD-derivation*** of $P \cup \{G\}$ via R consists of a (finite or infinite) sequence $G_0,\ldots,G_k$, of goals, a sequence $C_1,\ldots,C_m$ of variants of program clauses in P and a sequence $\theta_1,\ldots,\theta_n$ of *mgu*'s, such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1} via R.

Each C_i is a suitable variant of the corresponding program clause so that C_i does not have any variables which already appear in the derivation up to G_{i-1}. This can be achieved, for example, by subscripting variables in G by 0 and variables in C_i by i. This process of renaming variables is called *standardizing* the variables *apart* (*Definition 3.18*). It is necessary, otherwise for example, we would not be able to derive all the clauses we could obtain by unifying $p(X) \vee q(X)$ with $\neg p(X) \vee \neg q(Y)$.

Each program clause variant $C_1,\ldots,C_m$ is called an *input clause* of the derivation.

Definition 5.6 An ***SLD-refutation*** of $P \cup \{G\}$ via R is a finite *SLD-derivation* of $P \cup \{G\}$ via R which has the empty clause, □, as the last goal in the derivation. If G_n = □, we say that the refutation has *length n*.

SLD-derivations can be *finite* or *infinite*. A finite SLD-derivation can be successful or failed. A *successful* SLD-derivation is one that ends in the empty clause. In other words, we call a *successful derivation* a *refutation*. A *failed* SLD-derivation is one that ends in a non-empty goal with the property that the selected atom in this goal does not unify with the head of any program clause.

Definition 5.7 Let P be a definite program. The ***success set*** of P is the set of all $A \in HB_p$[3] such that $P \vee \{\neg A\}$ has an *SLD-refutation* (using some computation rule depending on A.)

The success set is the procedural counterpart of the least Herbrand model. We shall see later that the success set of P is in fact equal to the least Herbrand model of P.

Definition 5.8 Let P be a program, G a goal and R a computation rule. Then the ***SLD-tree*** of $P \cup \{G\}$ via R is defined as follows:

(a) Each node of the tree is a goal (possibly empty).

(b) The root node is G.

(c) Let $\leftarrow A_1, \ldots, A_k, \ldots, A_n$, $(n \geq 1)$ be a node in the tree and suppose that A_k is the atom selected by R. Then this node has a descendent for each input clause $A \leftarrow B_1, \ldots, B_q$ such that A_k and A are unifiable. The descendent is

$$\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_q, A_{k+1}, \ldots, A_n)\theta.$$

where θ is an *mgu* of A_k and A.

(d) Nodes which are the empty clause have no descendants.

---

[3] B_p - Herbrand Base of P.

Each branch of the SLD-tree is a derivation of $P \cup \{G\}$. Branches corresponding to successful derivations are called ***success branches***, branches corresponding to failed derivations are called ***failed branches***.

Example 5.1 Consider the program

1. $p(X,Z) \leftarrow q(X,Y), p(Y,Z).$
2. $p(X,X) \leftarrow .$
3. $q(a,b) \leftarrow .$

and the goal $\leftarrow p(X,b)$. *Figures 5.1* and *5.2* show two SLD-trees for this program and goal. The SLD-tree in *Figure 5.1* comes from the standard *PROLOG* computation rule (i.e. *leftmost derivation*). The SLD-tree in *Figure 5.2* comes from the computation rule which always selects the rightmost atom. The selected atoms are underlined and the success, failure, and infinite branches are shown. Note that the first tree is finite, while the second tree is infinite. Each tree has two success branches corresponding to the answer $\{X/a\}$ and $\{X/b\}$.

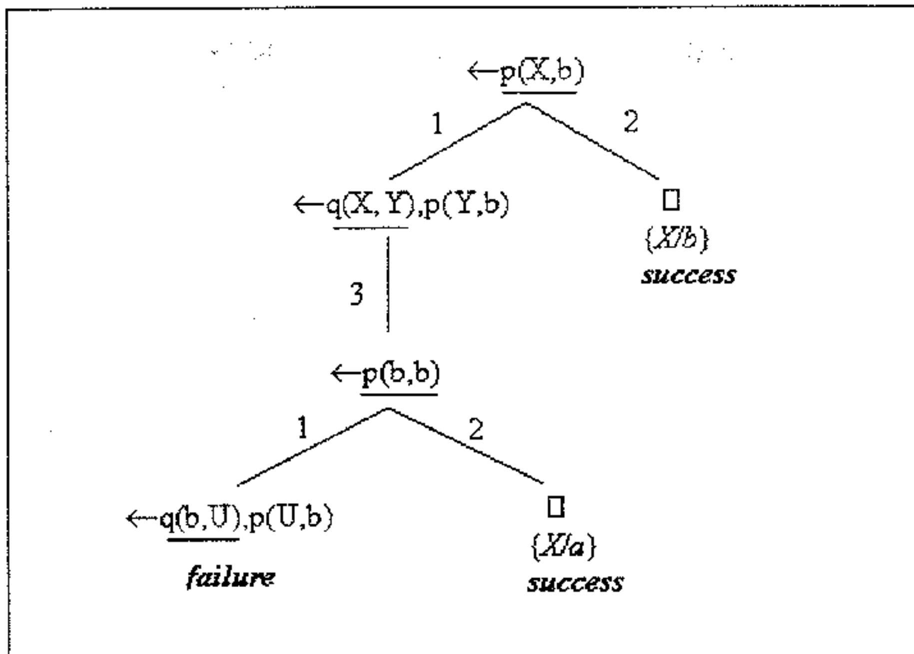


Figure 5.1 A finite SLD-tree

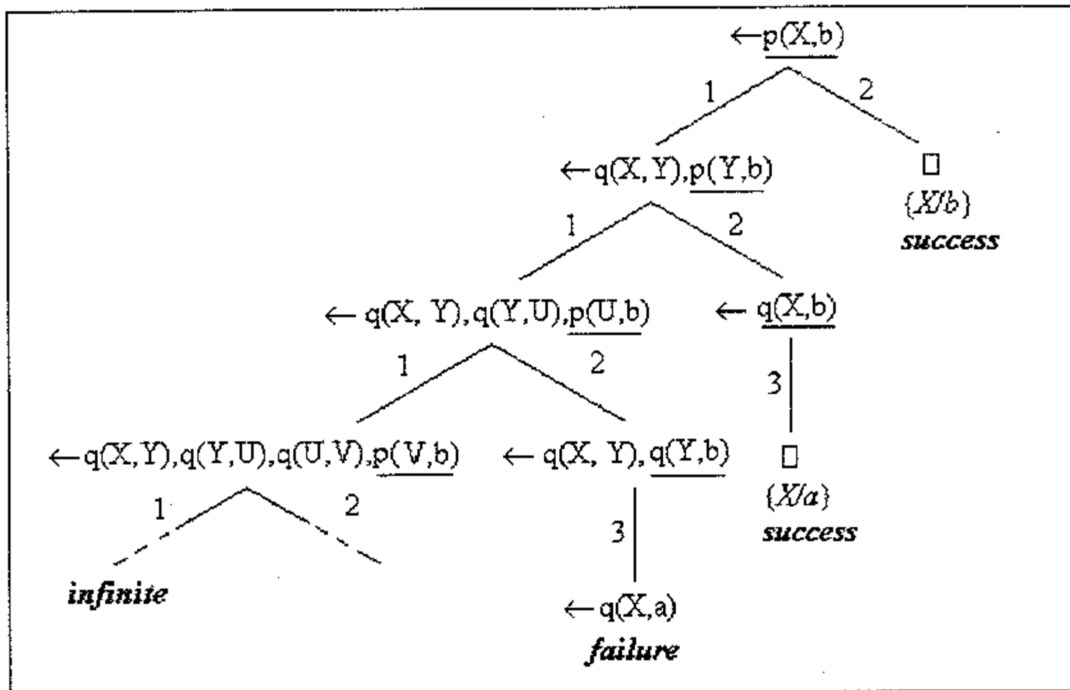


Figure 5.2 An infinite SLD-tree

This example shows that the choice of computation has a great bearing on the size and structure of the corresponding SLD-tree. However, no matter what the choice of computation rule, if $P\cup\{G\}$ is unsatisfiable, then the corresponding SLD-tree does have a success branch.

The procedural counterpart of a correct answer is defined next.

Definition 5.9 Let P be a definite program and G a definite goal. A *computed answer* $\theta = \theta_1,\ldots,\theta_n$ for $P\cup\{G\}$ is the substitution obtained by restricting the composition $\theta_1, \ldots, \theta_n$ to the variables of G, where $\theta_1,\ldots,\theta_n$ is the sequence of *mgu*'s used in an *SLD-refutation* of $P\cup\{G\}$.

Example 5.2 In *Example 5.1* above, $\{X/a\}$ is a computed answer, and the success set is $\{\{X/a\}, X/b\}\}$.

The first soundness result is that computed answers are correct. In the form below, this result is due to Clark [*12*].

Theorem 5.1 ***(Soundness of SLD-Resolution)***
Let P be a definite program and G a definite goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.

Proof Let G be the goal $\leftarrow A_1, \ldots, A_k$ and $\theta_1, \ldots, \theta_n$ be the sequence of *mgu*'s used in a refutation of $P \cup \{G\}$. We have to show that $\forall((A_1 \wedge \ldots \wedge A_k)\theta_1 \ldots \theta_n)$ is a logical consequence of P. The result is proved by induction on the length of the refutation.

Suppose first that $n=1$. This means that G is a goal of the form $\leftarrow A_1$, the program has a unit clause of the form $A\leftarrow$ and $A_1\theta_1 = A\theta_1$. Since $A_1\theta_1\leftarrow$ is an instance of a unit clause of P, it follows that $\forall(A_1\theta_1)$ is a logical consequence of P.

Next suppose that the result holds for computed answers which come from refutations of length $n-1$. Suppose $\theta_1, \ldots, \theta_n$ is the sequence of *mgu*'s used in refutation of $P \cup \{G\}$ of length n. Let $A\leftarrow B_1, \ldots, B_q$ $(q\geq 0)$ be the first input clause and A_m the selected atom of G. By the induction hypothesis,

$$\forall((A_1 \wedge \ldots \wedge A_{m-1} \wedge B_1 \wedge \ldots \wedge B_q \wedge A_{m+1} \wedge \ldots \wedge A_k)\theta_1 \ldots \theta_n)$$

is a logical consequence of P. Thus, if $q > 0$, then $\forall((B_1\wedge\ldots\wedge B_q)\theta_1\ldots\theta_n)$ is a logical consequence of P. Consequently, $\forall(A_m\theta_1\ldots\theta_n)$, which is the same as $\forall(A\theta_1\ldots\theta_n)$, is a logical consequence of P. Hence $\forall((A_1\wedge\ldots\wedge A_k)\theta_1\ldots\theta_n)$ is a logical consequence of P. ■.

Corollary 5.1 Let P be a definite program and G a definite goal. Suppose there exists an *SLD-refutation* of $P \cup \{G\}$. Then $P \cup \{G\}$ is unsatisfiable.

Proof. Let G be the goal $\leftarrow A_1, \ldots, A_k$. By theorem , the computed answer θ coming from the refutation is correct. Thus $\forall((A_1 \wedge \ldots \wedge A_k)\theta_1)$ is a logical consequence of P. It follows that $P \cup \{G\}$ is unsatisfiable. ■.

Corollary 5.2 The success set of a definite program is contained in its least Herbrand model.

Proof. Let the program be P, let $A \in HB_P$ and suppose $P \cup \{\leftarrow A\}$ has a refutation. By *theorem 5.1*, A is a logical consequence of P. Thus A is in the least Herbrand model of P. ■.

It is possible to strengthen *corollary 5.2*. We can show that if $A \in HB_P$ and $P \cup \{\leftarrow A\}$ has a refutation of length n, then $A \in T_P \uparrow n$. This result is due to *Apt* and *van Emden* [*4*].

Definition 5.10 A ***search rule*** is a strategy for searching SLD-trees to find success branches. An ***SLD-refutation procedure*** is specified by a computation rule together with a search rule.

From this definition and the above we say that the SLD-refutation procedure is complete, and the proof can be found in *Lloyd* [*34*].

5.2 SLI Resolution

The SLD resolution procedure discussed above cannot be used for disjunctive logic programs since the application of an SLD derivation step to a goal does not necessarily result in another goal as the following example shows.

Example 5.3 (*LMR* [*35*]) Let P be a disjunctive logic program

$$P = \{ \mathit{path(washington, newyork)} \vee \mathit{path(washington, philadelphia)} \}$$

and let G be the goal $\leftarrow$*path(washington, X)*. If we use the SLD refutation procedure to resolve the goal with the program clause we obtain two possible derived consequences *path(washington, newyork)* with binding X = *philadelphia*, and *path(washington, philadelphia)* with binding X = *newyork*. Furthermore, SLD does not permit one to make a copy of the goal clause and use it a second time.

SLI resolution[4] is based on the linear resolution principle. Linear resolution operates on clauses. It starts with a clause, (binary) resolves it against another clause to obtain a resolvent. The resolvent is used in another resolution step against another clause until the empty clause, $\Box$, is obtained.

Definition 5.11 (*Loveland* [*37*]) Let S be a set of clauses and let $C_0 \in S$. A ***linear derivation*** of C_n from S with top-clause C_0 is a finite sequence of clauses $C_0,\ldots,C_n$ such that C_{i-1} is either a factor of C_i, or a resolvent of C_i and a clause B_i for some i, $0 \leq i \leq n\text{-}1$, where B_i is either a factor of a clause in S or a clause C_j for some j, $0 \leq j \leq i$.

SLI resolution is defined using trees as the basic representation for clauses. Program and goal clauses are represented using *t-clauses* which have a tree structure. Each node in the trees is a literal

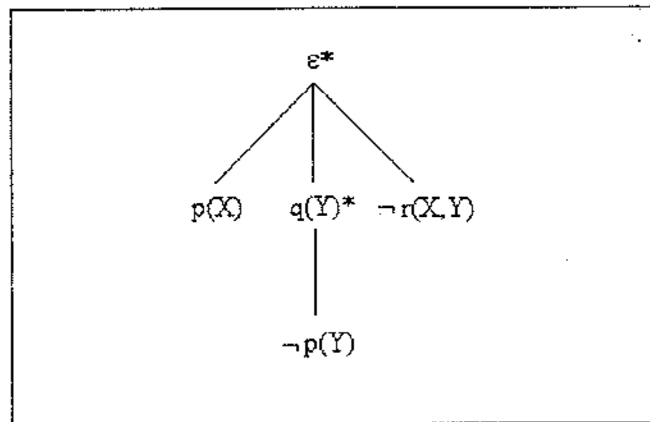
[4] ***SLI*** - *S*elected rule-driven *L*inear resolution for *I*ndefinite clauses.

from the language of the program or the special symbol ε. The literals in a *t-clause* are classified into two types: *marked literals*, and *unmarked literals*. A non-terminal node is always a marked literal. We now give the definition of a *t-clause*.

Definition 5.12 (*Minker* and *Zanon* [47]) A ***t-clause*** ζ, is an ordered pair $\langle C, m \rangle$ where

1. C is a labelled tree whose root is labelled with the distinguished symbol e, and whose other nodes are labelled with literals; and
2. m is a marking (unary) relation on the node such that every non-terminal node in ζ, is marked.

For convenience, we sometimes use a well-parenthesized pre-order representation of the tree denoted by a *t-clause* in *Definition 5.12*. The *t-clause*



in a parenthesized pre-order representation is given by (ε^* $p(X)$ $(q(Y)^*$ $\neg p(Y))$ $\neg r(X,Y)$). A program clause of the form $A_1 \vee \ldots \vee A_n \leftarrow B_1, \ldots, B_m$, is represented by a *t-clause* (ε^* $A_1 \ldots A_n$ $\neg B_1 \ldots \neg B_m$), which has only one marked literal, the distinguished literal ε. A goal clause is also transformed into a *t-clause*. It is called the *top t-clause* and is used to start an SLI derivation. During an SLI derivation the top t-clauses are renamed with distinct names. For each use of a goal clause, a substitution is maintained which records the renaming of the goal variables. We call these substitutions *renaming substitutions* and use them to extract the answer from the SLI derivation.

In contrast to definite logic programs where an SLD derivation maps a goal clause to a goal clause, an SLI derivation maps a t-clause to a t-clause, which may not necessarily be a goal t-clause. We need to define two sets, γ_L and δ_L for a literal, L, in a t-clause which are used while performing ancestry resolution and factoring.

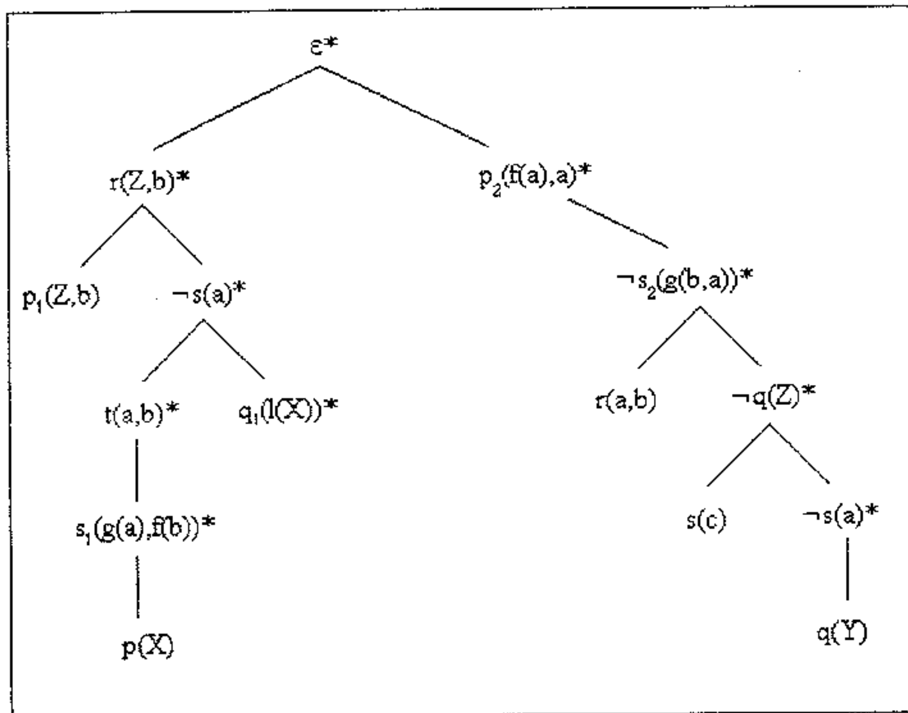
Definition 5.13 Let L be an unmarked literal in a t-clause.

$\delta_L = \{ N$: where N is a marked literal and an ancestor of L $\}$

$\gamma_L = \{ M$: where M is an unmarked literal and a sibling of an ancestor of L $\}$.

Note that the special symbol ε is an ancestor of a literal L, but not a literal, and therefore is not in δ_L.

Example 5.4 Consider the t-clause



Then

$\gamma_{p(X)} = \{ p_1(Z,b) \}$.

$\gamma_{q(Y)} = \{ r(a,b), s(c) \}$.

$\delta_{p(X)} = \{ s_1(g(a), f(b)), t(a,b), \neg s(a), r(Z,b) \}$.

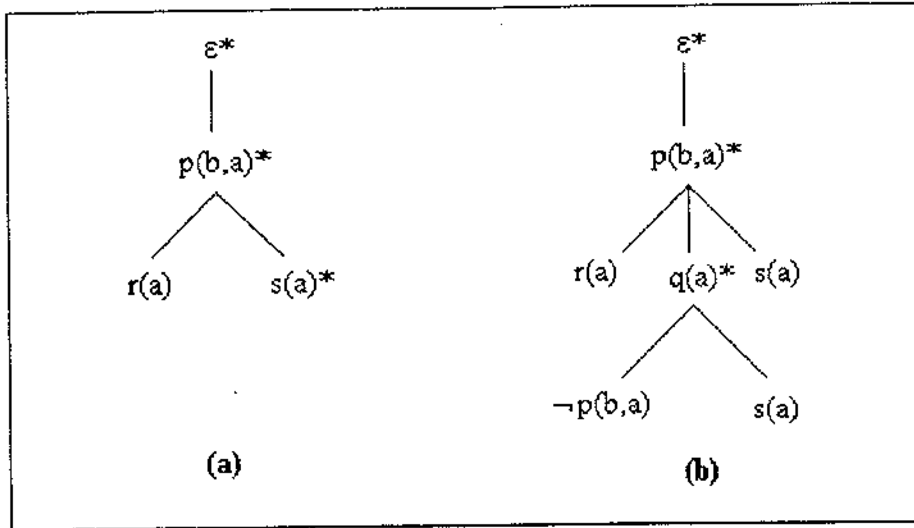
$\delta_{q(Y)} = \{ \neg s(a), \neg q(Z), \neg s_2(g(b,a)), p_2(f(a),a) \}$.

γ_L denotes a set of literals which need to be checked to see if factoring can be done on literal L. That is, L can be factored (removed from the t-clause) if L unifies with a literal in γ_L. The set γ_L is also useful in detecting the derivation of a tautology. The set δ_L contains the literals which need to be checked to see if ancestry resolution can be done on literal L. That is, L can be ancestry resolved (removed from the t-clause) if L is equivalent, modulo variable renaming, to the negation of a literal L' in δ_L. The set δ_L is also used to check for infinite derivations. If L appears in the set of its ancestors, δ_L, we can stop the derivation since it implies a *loop* in the derivation. We also have two conditions that make precise the use of γ_L and δ_L. A t-clause is defined as ***admissible***[5] if no tautologies or loops are present in the t-clause, i.e. no two literal from γ_L and L have identical atoms, and no two literals from δ_L have identical atoms. It is also important that a t-clause satisfies the ***minimality condition*** (*MC*). This occurs if there is no marked literal in the t-clause which is a terminal node.

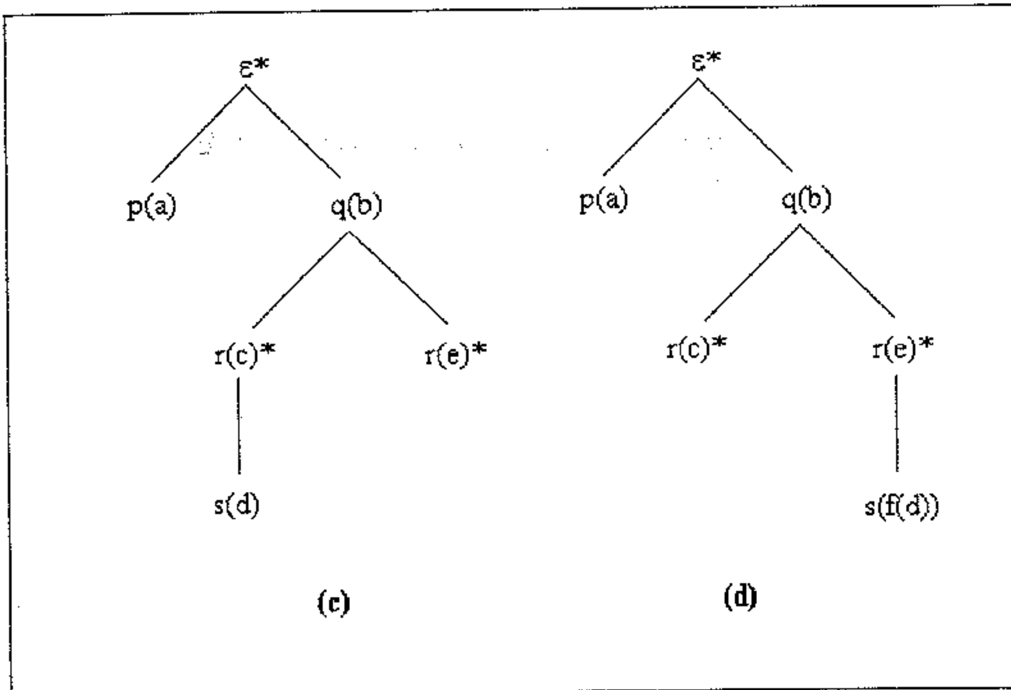
A marked terminal node in a t-clause corresponds to a literal that has been solved and hence may be removed from the t-clause. When a marked terminal literal is removed from a t-clause it is said that ***truncation*** has taken place.

[5] i.e. satisfies the ***admissibility condition*** (*AC*).

Example 5.5 Consider the following *t-clauses.*



The *t-clause* (a) satisfies *AC* whereas the *t-clause* (b) does not because there is a $\gamma_{s(a)}$ that contains s(a).



The *t-clause* (c) satisfies *MC* whereas the *t-clause* (d) does not because *r(c)** is a terminal node.

The MC ensures that truncation is performed as soon as possible on a derived t-clause. The framework has now been laid for describing SLI resolution. We first define a ***tranfac-derivation*** which performs ***tr***uncation, ***an***cestry resolution and ***fac*** on a t-clause.

Definition 5.14 Let C_0 be a t-clause. The t-clause C_n is a ***tranfac-derivation*** (truncation, ancestry and factoring) of C_0 when there is a sequence of t-clauses $C_0, C_1, ..., C_n$ and substitutions $\theta_0, \theta_1, ..., \theta_n$ such that for all i, $0 \leq i \leq n$, C_{i+1} is obtained from C_i by either t-factoring, t-ancestry, or t-truncation with substitution θ_i.

C_{i+1} is obtained from C_i by t-factoring iff

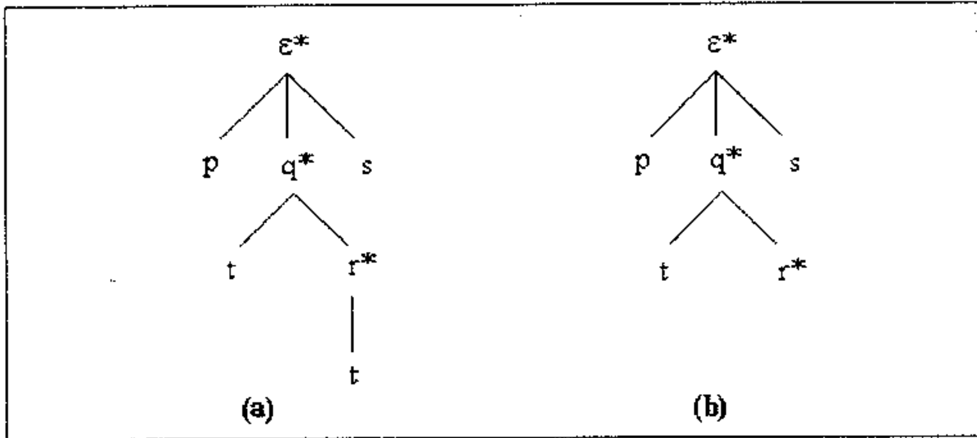
1. C_i is $(\alpha_1 L \alpha_2 M \alpha_3)$ or C_i is $(\alpha_1 M \alpha_2 L \alpha_3)$;
2. $M\theta_i = L\theta_i$, where θ_i is a substitution;
3. L is in γ_M (that is, L is an unmarked sibling of an ancestor of M);
4. C_{i+1} is $(\alpha_1 L \alpha_2 \alpha_3)\theta_i$ or C_{i+1} is $(\alpha_1 \alpha_2 L \alpha_3)\theta_i$.

C_{i+1} is obtained from C_i by t-ancestry iff

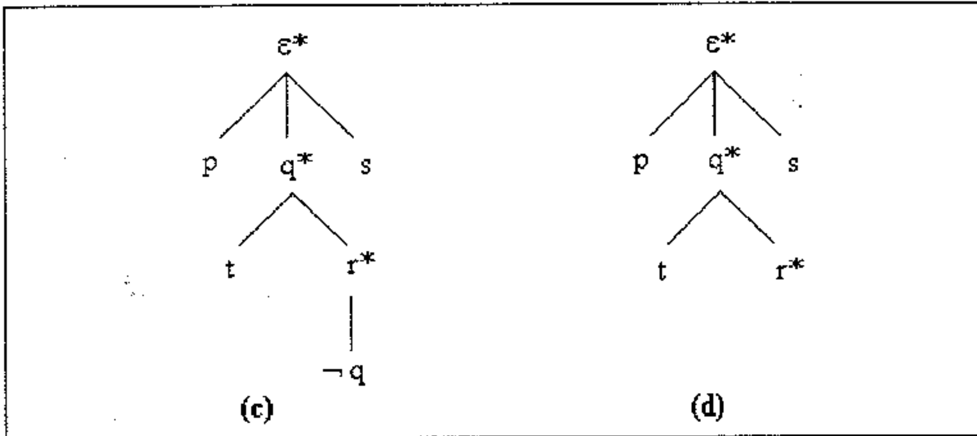
1. C_i is $(\alpha_1 (L^* \alpha_2 (\alpha_3 M \alpha_4) \alpha_5) \alpha_6)$;
2. $L\theta_i = \neg M\theta_i$, where q_i is a (most general) substitution;
3. L is in δ_M;
4. C_{i+1} is $(\alpha_1 (L^* \alpha_2 (\alpha_3 \alpha_4) \alpha_5) \alpha_6)$;

C_{i+1} is obtained from C_i by t-truncation with q_i equal to the identity substitution iff either C_i is $(\alpha(L^*)\beta)$ and C_{i+1} is $(\alpha\beta)$ or C_i is (ε^*) and C_{i+1} is $\Box$.

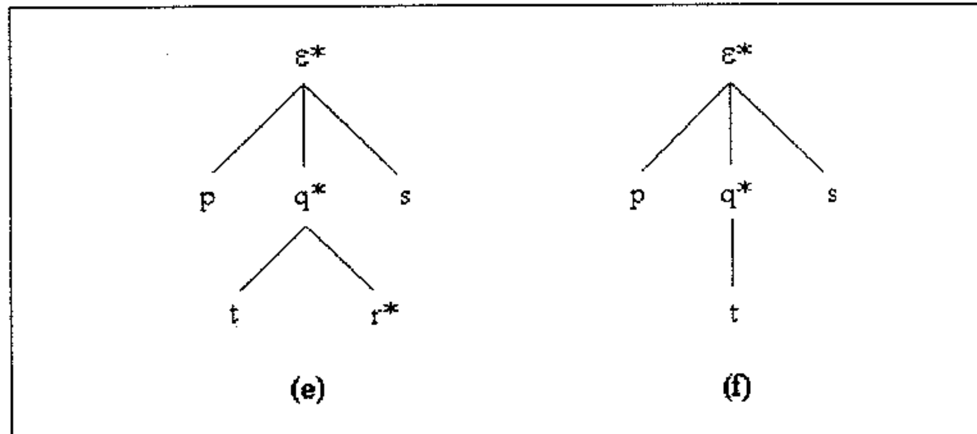
Example 5.6 Consider the following *t-clauses*.



The t-clause (b) is obtained from (a) after *t-factoring*.



The t-clause (d) is obtained from (c) after *t-ancestry*.



The t-clause (f) is obtained from (e) after *t-truncation*.

A derivation resolves an admissible and minimal t-clause with an input t-clause to obtain an admissible and minimal t-clause.

Definition 5.15 Let $C_i = (\varepsilon^* \alpha_1 L \beta_1)$ be a t-clause. Let $B_{i+1} = (\varepsilon^* \alpha_2 M \beta_2)$ be another t-clause standardized apart from C_i. Then C_{i+1} is ***t-derived*** from C_i and B_{i+1} using the substitution θ_i if the following conditions hold:

1. $L\theta' = \neg M\theta'$, where θ' is a (most general) substitution;
2. C'_{+1} is $(\varepsilon^* \alpha_1 (L^* \alpha_2 \beta_2) \beta_1)\theta'$;
3. C_{i+1} is either a tranfac-derivation of C'_{+1} with substitution θ'' or directly C'_{+1} and for this case $\theta'' = \varepsilon$;
4. $\theta_i = \theta' \cdot \theta''$;
5. C_{i+1} must satisfy the admissibility and minimality conditions.

Definition 5.16 Let S be an input set of t-clauses and let C be a t-clause in S. An ***SLI-refutation*** from S with top t-clause C is an SLI derivation of the empty clause □. If there is an SLI refutation of a clause C with input set S, then we write $S \vdash_{SLI} \neg C'$, where C' is the clause representation of the t-clause C.

The following example illustrates the SLI refutation procedure.

Example 5.7 Let P be a disjunctive logic program given in t-clause form as:

(1) { (ε^* a ¬c ¬d ¬e);
(2) (ε^* ¬d);
(3) (ε^* f e ¬g);
(4) (ε^* a ¬f);
(5) (ε^* d);
(6) (ε^* g);

and let the goal clause be:

(7) (ε^* ¬*a*);

We show that there is an SLI refutation of the goal clause (the bold literal is the one chosen to be resolved upon in the proceeding step.)

(7)	$(\varepsilon^* \neg \mathbf{a})$;	goal clause
(8)	$(\varepsilon^* (\neg a^* \neg c \neg d \neg \mathbf{e}))$;	derivation (7,1)
(9)	$(\varepsilon^* (\neg a^* \neg \mathbf{c} \neg d (\neg e^* f \neg g)))$;	derivation (8,3)
(10)	$(\varepsilon^* (\neg a^* (\neg c^* \neg d) \neg d (\neg e^* f \neg g)))$;	derivation (9,2)
(11)	$(\varepsilon^* (\neg a^* (\neg c^*) \neg d (\neg e^* \mathbf{f} \neg g)))$;	t-factoring (10)
(12)	$(\varepsilon^* (\neg a^* \neg d (\neg e^* \mathbf{f} \neg g)))$;	t-truncation (11)
(13)	$(\varepsilon^* (\neg a^* \neg d (\neg e^* (f^* \mathbf{a}) \neg g)))$;	derivation (12,4)
(14)	$(\varepsilon^* (\neg a^* \neg d (\neg e^* (\mathbf{f}^*) \neg g)))$;	t-ancestry (13)
(15)	$(\varepsilon^* (\neg a^* \neg \mathbf{d} (\neg e^* \neg g)))$;	t-truncation (13)
(16)	$(\varepsilon^* (\neg a^* (\neg \mathbf{d}^*) (\neg e^* \neg g)))$;	derivation (15,5)
(17)	$(\varepsilon^* (\neg a^* (\neg e^* \neg \mathbf{g})))$;	t-truncation (16)
(18)	$(\varepsilon^* (\neg a^* (\neg e^* (\neg g^*))))$;	derivation (17,6)
(19)	$\Box$.	t-truncations (18)

Lastly we provide the meaning of an SLI computed answer.

Definition 5.17 Let P be a disjunctive logic program and let G be a goal in t-clause form and used as the top t-clause in an SLI refutation. Let the goal t-clause be used n times during the SLI refutation with the corresponding renaming substitutions $\sigma_1, \ldots, \sigma_n$. Let the composition of substitutions computed for the variables in G during the SLI refutation be θ and let $\theta_1, \ldots, \theta_n$ be the substitutions such that $\forall i$, $i = 1, \ldots, n$ θ_i, is obtained by restricting q to the variables in s_i. Then an ***SLI computed answer*** is given as:

$$\{ \theta_1 \cdot \sigma_1, \ldots, \theta_n \cdot \sigma_n \}.$$

Example 5.8 Let P be the disjunctive logic program

$$P = \{ \, path(washington,newyork) \vee path(washington,philadelphia) \, \}$$

The corresponding t-clause is

(1) (ε^* *path(washington,newyork) path(washington,philadelphia)*)

Let $G = \neg path(washington, X)$ be a goal with the corresponding t-clause

(2) ($\varepsilon^* \neg path(washington, X)$)

The SLI computed answer would be given by

$$\{ \, \{ \, X_1 = \text{newyork} \, \} \cdot \{ \, X_1 = X \, \}, \{ \, X_2 = \text{philadelphia} \, \} \cdot \{ \, X_2 = X \, \} \, \}.$$

That is the answer computed is { { X = newyork },{ X = philadelphia } }, which states that the disjunction *path(washington,newyork)* $\vee$ *path(washington,philadelphia)* is a correct answer.

5.3 Clause Trees

Horton and Spencer [*29*] have come up with a data structure - *clause trees* - for understanding and implementing resolution. The clause trees can be defined as follows:

Definition 5.18

A ***clause tree T*** on a set S of clauses is a 4-tuple <*N,E,L,M*>, where N is a set of nodes divided into clause nodes and atome nodes, E is a set of edges, each of which joins a clause node to an atom node, L is a labelling of $N \cup E$ which assigns to each clause node a clause node of S, to each atom node an instance of an atom of some clause S, and to each edge either + or -. M is the set of chosen merge paths.

An instance of a claue C in S is represented by an elementary clause tree which consists of a single clause node labelled by S and joined to atom nodes.

Example 5.9 Consider the two clauses {*a, b, ~d*} and {*~b, c, ~d*}. These can be represented by clause trees as shown in *Figure 5.3* (*a*) and (*b*).

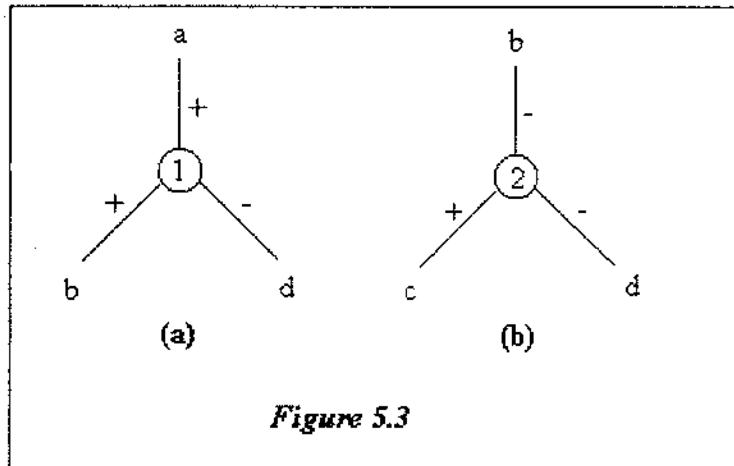


Figure 5.3

These two clauses can be resolved on *b* getting {*a, c, ~d*}. This can also be done using clause trees, first by identifying the leaf nodes that represent complementary literals from the two different

clauses and joining the two trees to form one as shown in *Figure 5.4(a)* and then by joining the two atom nodes that correspond to the same literal with a *merge path* as in *Figure 5.4(b)*. The node (containing the literal) at the tail of the merge path is considered closed and therefore the literal is no longer considered to be a literal of the corresponding clause tree. On the other hand, the node at the head of a merge path is open.

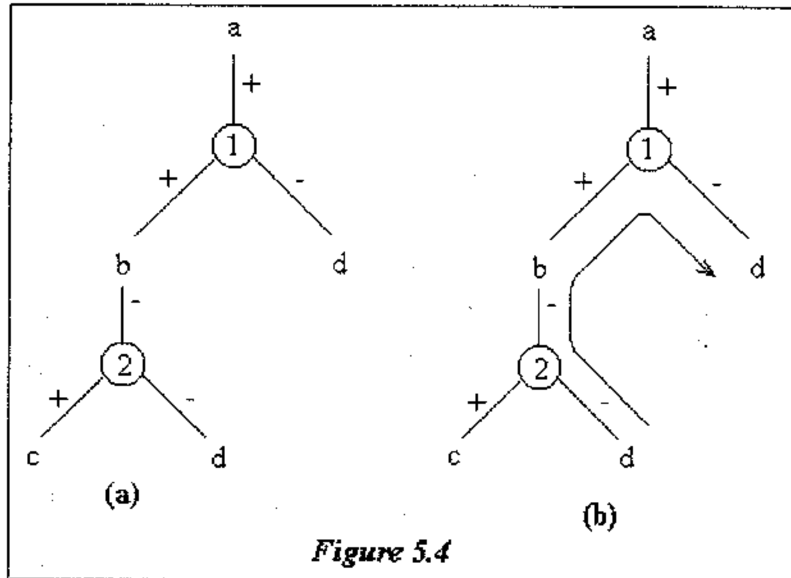


Figure 5.4

As seen in the paper (Horton and Spencer *[29]*), the sequence of resolutions, on a set of clauses, can have a significant impact on the results obtained. This can be resolved by adding merge paths from open leafs to internal nodes, thus producing the same minimal and most general results by using clause trees. Merge paths, however, should be introduced with caution and the general principles have been laid out in the paper.

Example 5.10 Applying the clause trees to the program and query in *Example 5.7*, we come up with the following proof:

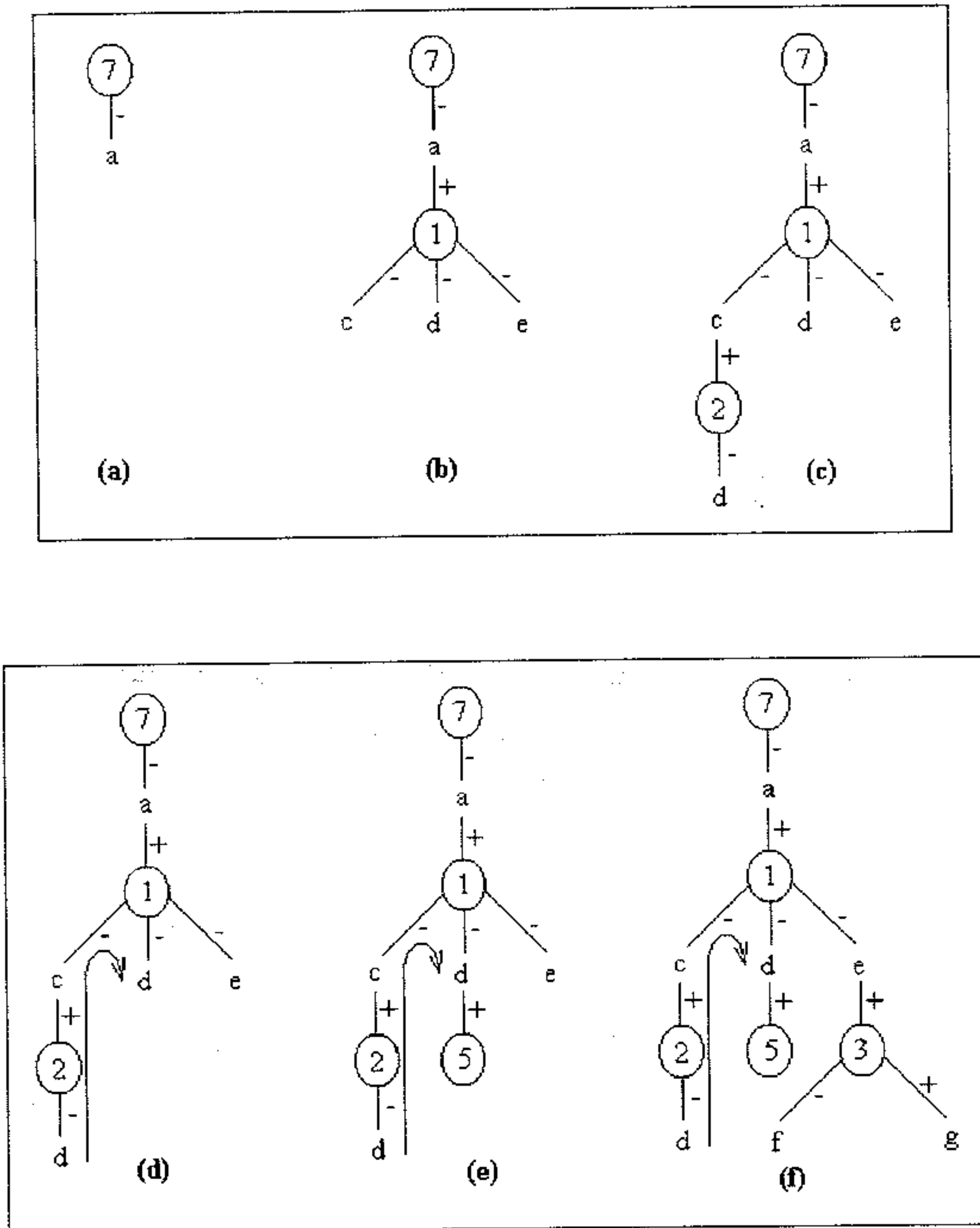


Figure 5.5

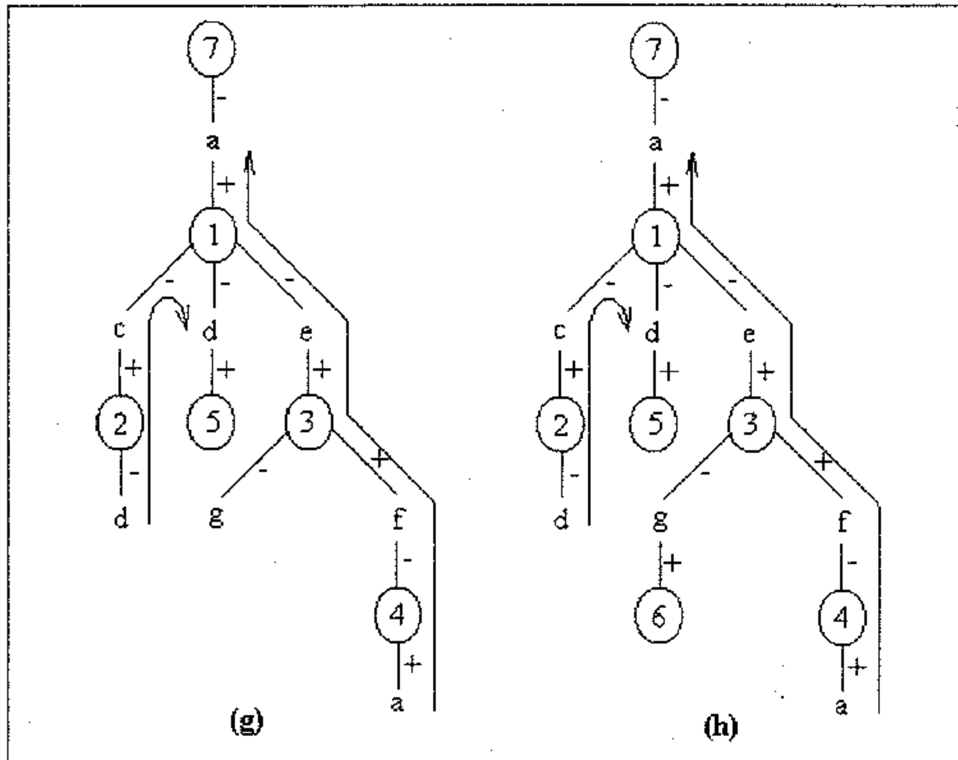


Figure 5.5

The clause tree in *Figure 5.5 (h)* is a *closed tree*, with no open leaf (see *Horton* and *Spencer* [*29*] *p 12*), and therefore the proof is complete.

6 *Negative Information*

It is not possible to derive explicit negative information from a definite or a disjunctive logic program. Fixpoint theory, model-theoretic semantics and proof procedures for logic programs define sets which contain only positive clauses. One alternative is to allow for explicit representation of negated information in the program. Such an explicit representation is not feasible in many applications such as deductive databases and artificial intelligence, where the amount of negative information may overwhelm the system. The solution to this problem has been to use default rules to infer negated facts implicitly from the system. Resolution systems augmented with such default rules offer attractive inference systems for logic programs.

The implicit definition of negation in logic programs is strongly tied to the notion of *absence of something factual*. This mechanism is called the *closed world semantics* and consists of three sets of formulae: *success*, *failure*, and *unknown*. The set *success* consists of formulae which can be proven to be *true*. The *failure* set consists of the formulae which can be assumed to be *false* under the closed world semantics. The set *unknown* captures all other possible formulae. The different meanings and definitions of *success* and *failure* provide a rich variation of semantics for negation.

In the case of definite logic programs, the closed world semantics is given by the *closed world assumption* (*CWA*). Another area where negative information is derived and used is *Commonsense Reasoning* - otherwise known as *Nonmonotonic Reasoning*. We shall first look at this area.

6.1 Nonmonotonic Reasoning

6.1.1 Historical Background

The start of the field of nonmonotonic reasoning is an outgrowth of McCarthy's 1958 paper on commonsense reasoning [*42*]. The paper by Hayes [*26*] is another important early development. The PROLOG programming language developed by Colmerauer and his students [*16*] and the PLANNER language developed by Hewitt [*28*] were the first languages to have a nonmonotonic component. The ***not*** operator in PROLOG and ***THNOT*** capability in PLANNER provided default rules for answering questions about data where the facts did not appear explicitly in the program.

The formalization of the field of nonmonotonic reasoning as we know it today started approximately in 1975/6 with papers published in the 1977-79 time period. Two important papers, one by Reiter [*54*] and the other by Clark [*11*], appeared in the book *Logic and Databases*, edited by Gallaire and Minker [*19*]. Reiter set forth the rule of negation called the *closed world assumption* (*CWA*) that states that in Horn logic theories if we cannot prove an atom *p*, then we can assume ***not*** *p*. Clark related negation to the ***only if*** counterpart of ***if*** statements in a logic program. The ***if-and-only-if*** (***iff***) statements form a theory in which negated atoms can be proven using a full theorem prover. The importance of Clark's observation is that he showed that, for ground atoms, an inference system called *SLDNF* resolution, operating on the ***if*** statements of logic programs, was sufficient to find the ground negated atoms in the ***iff*** theory that can be assumed *true*. These two *rules of negation* are probably, the first formalization of nonmonotonic reasoning. McCarthy first introduced the concept of *circumscription* in 1977 [*41*], and Doyle developed his *truth maintenance systems* in 1979 [*17*]. Reiter gave preliminary material on default reasoning in 1978 [*55*].

Nonmonotonic reasoning obtained its impetus in 1980 with the publication of an issue of the *Artificial Intelligence Journal* devoted to nonmonotonic reasoning. In that seminal issue the initial theories of nonmonotonic logic were presented. As noted by Bobrow in his "Editor's Preface" to the journal, the approaches to nonmonotonic reasoning can be characterized broadly as falling in two different classes. The first approach extends the logic system in different ways. This is characterized in papers by McCarthy [*40*], who formalized his theory of *circumscription* introduced earlier [*41*]; by Reiter [*56*], who introduced his theory of *default reasoning*; and by McDermott and Doyle [*44*], who used *modal logic* to handle nonmonotonicity. The second approach views logic as an object and extends the reasoning system with metadevices. This is explored by Welrauch [*68*], and Winograd [*69*].

6.1.2 The Concept of Nonmonotonic Reasoning

SLD-resolution is an example of a *sound* method of reasoning because only true facts can be deduced using it. More precisely, we say that a reasoning method "$\vdash$" is sound if, for all variable-free formulae φ, $P \vdash \varphi$ implies $P \models \varphi$, where $P \vdash \varphi$ denotes that φ can be proved from a program P. And we call "$\vdash$" weakly sound if $P \vdash \varphi$ implies consistency of $P \cup \{\varphi\}$. Now, putting $P \vdash_{SLD} \exists x_1 \ldots \exists x_s(A_1 \wedge \ldots \wedge A_k)$ iff there exists an *SLD*-refutation of $P \cup (\leftarrow A_1,\ldots,A_k)$ (see subsection *5.1*). We see that $\vdash_{SLD}$ is sound by virtue of the Soundness Theorem (Apt [*3*]).

SLD-resolution is also an example of a *monotonic* method of reasoning. We call here a reasoning method "$\vdash$" *monotonic* if, for any two consistent programs P and Q,

$$P \vdash \varphi \text{ implies } P \cup Q \vdash \varphi.$$

Otherwise, "$\vdash$" is called *nonmonotonic*. Clearly, if there exists an *SLD*-refutation of $P \cup \{N\}$ then there also exists an *SLD*-refutation of $P \cup Q \cup \{N\}$.

However, *SLD*-refutation is a very restricted form of reasoning, because only positive facts can be deduced using it. This restriction cannot be overcome if soundness or monotonicity is to be maintained. More precisely, the following simple yet crucial observation holds.

Lemma 6.1 (Apt [*3*]) Let P be a definite logic program and let "$\vdash$" be a reasoning method, such that, $P \vdash \neg A$ for some negative ground literal $\neg A$. Then "$\vdash$" is not sound. Moreover, if "$\vdash$" is weakly sound, then it is not monotonic.

Proof

Note that a subset of the Herbrand base is a model of P, but no subset thereof is a model of $\neg A$. Thus "$\vdash$" is not sound. Suppose it is monotonic. Then we get $P \cup \{A\} \vdash \neg A$. But $P \cup \{A\} \cup \neg A$ is inconsistent, so "$\vdash$" is not weakly sound. □.

However, in some applications it is natural to require that negative information can also be deduced.

Example 6.1 (Apt [*3*]) Consider the program

```
P = {
        element(fire).
        element(air).
        element(water).
        element(earth).
        stuff(mud)
    }
```

Then we naturally expect that $\neg$*element(mud)*, $\neg$*stuff(fire)* can be deduced, and similarly with other elements.

By *Lemma 6.1* any such extension of *SLD*-resolution leads to nonmonotonic reasoning.

6.2 Closed World Assumption (CWA)

A useful example that can be used to describe *closed world assumption* is the familiar - "*missionaries and cannibals*" puzzle - which is also used by McCarthy to explain *circumscription*.

> *Three missionaries and three cannibals must cross a river using a boat that can hold only two persons; if the cannibals outnumber the missionaries on either bank of the river, the missionaries will be eaten. How can the crossing be arranged safely?*

Here we can see that the puzzler is expected to recognize certain ground rules, such as that the boat does not have a leak or any other incapacity for transporting people. Moreover, there are no additional cannibals or missionaries lurking in the background, who may upset otherwise sound plans, even though it was not specifically stated that there are *only* three cannibals and three missionaries. It is as if there is an implicit assumption that if something is not mentioned in the puzzle, then it is not to be considered, and this is the idea we sometimes refer to as *closed world assumption*. It was first considered by Reiter [*54*].

The declarative definition of the *CWA* defines *success* to be the set of ground atoms in the least Herbrand model of a definite logic program. The set of ground atoms not in the least Herbrand model of the program are taken as *failure*. The *CWA* has a proof-theoretic definition where the *success* set of ground atoms that are logical consequences of the program and the *failure* set of ground atoms that are not logical consequences of the program. The *unknown* set is empty (LMR [*35*]).

The metarule used for *CWA* can be written as follows:

$$\frac{A \text{ cannot be proved from } P}{\neg A}$$

where A is a ground atom. The notion of provability referred to in the hypothesis is that in first-order logic. In the case of definite programs it is sufficient to know that it is equivalent here to provability by means of the SLD-resolution.

Given now a program P, we can state its *CWA* closure as:

$$CWA(P) = \{\neg A: A \text{ is a ground atom for which there does not exist an SLD-refutation of } P \cup \{\leftarrow A\} \}.$$

Example 6.2 Let P be a definite logic program given by:

$$P = \{ \; male(jack); female(rita) \; \}.$$

Using the *CWA* we obtain: *success* = { *male((jack), female(rita)* }.

since the atoms, *male(jack)* and *female(rita)* are logical consequences of P.

failure = { *female(jack), male(rita)* }.

since the atoms, *female(jack)* and *male(rita)* are not logical consequences of P. That is, literals $\neg$*female(jack)* and $\neg$*male(rita)* can be assumed to be true under the closed world assumption.

The *CWA* leads to inconsistent results, however, when applied to disjunctive logic programs. For example, consider the program $P = \{ p \vee q \}$. From the *CWA* one can infer both $\neg p$ and $\neg q$ as true since neither p nor q is a logical consequence of P. But this is inconsistent with the fact that $p \vee q$ is true in P.

6.3 Generalized Closed World Assumption (GCWA)

The problem of inconsistency can be solved by extending the definition of the *CWA* to disjunctive logic programs. For disjunctive logic programs, there is no unique least Herbrand model, but there is a set of models, the set of minimal Herbrand models, *MM(P)*, which captures the intended meaning of logical consequences from a disjunctive logic program *P*. If a clause is *true* in all minimal models then it is in the success set. The *failure* set can be formed by ground clauses that are false in all minimal models. That is, a positive ground clause $C = A_1 \vee A_2 \vee ... \vee A_n$ is considered to be a failure if none of its atomic components are in any model in *MM(P)*. Note that some clauses in the disjunctive Herbrand base are neither in the *success* set nor in the *failure* set.

Example 6.3 Consider the program $P = \{p \vee q\}$. The atomic clause p is not a member of either the *success* set or the *failure* set, since p belongs to one but not all minimal models of P.

This inference about p is consistent with logical consequences since p and $\neg p$ are not provable from the program P. The closed world semantics defined above is the "*strongest*" negation semantics possible for disjunctive logic programs based on minimal models and is known as the ***generalized closed world assumption*** (***GCWA***).

6.4 Circumscription

While dealing with a database, since it is not possible to explicitly assert all negative information, it is sometimes advisable to use metarules that can be used to derive them. It is important to minimize the number of objects having certain properties. One way of doing this is by *circumscription.*

Circumscription involves the use of an axiom schema in a first-order language, intended to express the idea that certain formulae (*wffs*) have the smallest extensions consistent with certain axioms (*Perlis* [*48*]).

To illustrate this, let us look at one of McCarthy's applications of circumscription to commonsense reasoning problems - his use of a predicate *ab* for abnormal aspects of entities.

> Typically birds can fly. The idea is to minimize (as a conjectural assumption) the objects that are abnormal with respect to any given aspect, for instance, birds that are abnormal with respect to flying (such as penguins and ostriches). This allows the expression of default reasoning to be given a uniform treatment, in which the predicate *ab* is circumscribed provided that other predicates as desired may be considered variable. For instance letting *ab(b, f, X)* stand for *X* is an abnormal bird with respect to flying, then from the following axioms,
>
> $$flies(X) \leftarrow bird(X) \wedge \neg ab(b, F, X)$$
>
> $$ab(b, f, X) \leftarrow ostrich(X)$$
> $$bird(X) \leftarrow ostrich(X)$$
> $$bird(tweety).$$
>
> one can prove by formula circumscription that *tweety* can fly (and consequently *tweety* is not an ostrich) (Minker [*46*]).

6.5 Negation as Failure

Whilst using the *CWA* we may not able to derive some of the negative answers because we may get into an infinite loop. A more helpful way out is to adopt a more restrictive form of unprovability. A natural possibility is to consider $\neg A$ proved when an attempt to prove A using SLD-resolution fails *finitely*. This leads to the following definitions.

An SLD-tree (*Definition 5.8*) is *finitely failed* if it is finite and contains no empty clause. Thus all branches of a finitely failed SLD-tree are failed SLD-derivations. Given a program P, its *finite failure* set is the set of all ground atoms A such that there exists a finitely failed SLD-tree with $\leftarrow A$ as root.

We now replace CWA by the following rule:

$$\frac{A \text{ is in the finite failure set of } P}{\neg A}$$

introduced in Clark [*10*] and called the *negation as failure* rule. (A more appropriate name would be negation as a *finite failure rule*.) This likewise applies to disjunctive logic programs with *SLI-refutation*. The modified algorithms are known as *SLDNF* and *SLINF* respectively.

6.6 Disjunctive Logic Programs

In the area of disjunctive logic programming, there are also several approaches to negation: *the generalized closed world assumption* (*GCWA*) developed by Minker [*46*] (discussed earlier is subsection 6.2) and a derivative of this theory, the *extended closed world assumption* (Gelfond et. al. [*22*]), Henschen and Park *27*], that apply to disjunctive logic programs. The model-theoretic definition of the *GCWA* states that one can conclude the negation of a ground atom if it is false in all minimal Herbrand models. The proof-theoretic definition states that one can conclude ***not a*** if for all disjunctions $a \vee B_i$, provable from a program P, B_i is provable from P. The proof-theoretic and model-theoretic definitions are equivalent (Minker [*46*]). The concept of minimal models is closely related to McCarthy's circumscription, which also deals with minimal models. Minker and Rajasekar [*44*] describe how one can compute this theory. The GCWA is needed for disjunctive theories since Reiter [*54*] has shown that the CWA is consistent with respect to disjunctive theories. A weaker form of the GCWA, the *weak generalized closed world assumption* (*WGCWA*) was developed by Rajasekar, Lobo and Minker [*53*]. The complexity of the WGCWA is the same as that of the CWA. The WGCWA is equivalent to the *disjunctive database rule* (***DDR***), developed by Ross and Topor [*58*]. Lobo [*36*] has extended constructive negation to apply to normal disjunctive logic programs.

7 *Prolog*

PROLOG has been described as relational [Malpas [*38*]), descriptive (Genesereth [*23*]) and declarative (Coelho and Cotta [*15*], Brookshear [*7*]). Both the relational and descriptive views consider the organization of the database, or set of PROLOG facts. PROLOG is considered declarative in that one describes to it what one wants to accomplish, such as, "*sort([5,3,7,2], Answer)*", with little regard to the procedure accomplishing the sorting task, which returns "*Answer = [2,3,5,7]*". Of course, we must describe further what we mean by "*sort*" (see *Appendix III*).

PROLOG is also called a language for programming in logic (Colingaert [*8*], Ghezzi [*24*]). This last description may be the most accurate one, but PROLOG itself is only logic based and does not produce all the same proofs possible from methods using the full power of predicate calculus.

PROLOG comes in several dialects (Sosnowski [*61*]). The original version of Colmarauer and Roussel is Edinburgh syntax, also called *DEC-10 PROLOG*, due to its early implementation on *DEC-10* computers running on the *TOPS-10* operating system. Micro-Prolog, MS-Prolog and PD-Prolog are other dialects, available for microcomputers. Quintus and Eclipse, which are more powerful ones, with vast built-in libraries, built-in *C*-functions and abilities to embed *C*-subroutines within the programs, are available for 32-bit Unix based machines.

7.1 Components of a PROLOG Program

Syntax

A PROLOG program is a list of statements consisting of ***facts*** and ***rules***. The general form of a statement is: ***Head :- Body.***, where *Head* is a single structure, and *Body* is comprised of zero or more structures, called subgoals, separated by commas, meaning "***and***" or semicolons, meaning "***or***". A fact is a statement with no body, while a rule contains both a head and a body. A query is a fact preceded by "***?-***", and returns either ***true*** or ***false***. If a query contains variables, values are printed that make the query true.

The form of a ***structure*** is just that of a PROLOG fact, ***functor***$(t_1,\ldots,t_n)$, where t_i is a term, and can either be a constant, a variable or a structure.

Functors[6] are predicate symbols, operators, or relation names. A predicate symbol generally takes on the values ***true*** or ***false***. For example, <=(*2,4*) is *true* whereas <=(*4,2*) is *false*.

A ***constant*** is thought of as naming a specific object or relation and is either an atom or an integer. A constant atom is a string of letters and digits beginning with a lowercase letter and

[6] This is not necessarily a *function*.

containing no signs other than the underscore. For example, *bruce_spencer*, *x*, *y*, and *map2* are all constants whereas *2X*, *Mary*, and *new-york* are not. However, any combination of characters may be used to form a constant if placed between single quotes. Thus '*New-York*' is a constant.

An ***atom*** may also be composed entirely of signs, but these are reserved for special purposes. Two of these special atoms are "*:-*", which means "***if***" and "*?-*", signalling a ***query***.

The special characters used in the PROLOG alphabet are: { + - * / \ ^ < > ~ : . ? @ # $ & }, some of whose semantics are version dependent. A ***relation-name*** is also an atom, e.g., the "<" in <(*2,4*) or the ***loves*** in *loves(john, mary)*, the latter being a predicate which may be used to represent the fact - "*John loves Mary.*"

A ***variable*** is an atom preceded by either a capital letter or the underscore. *Who*, *Salary_Amt*, *X*, and *_2_brothers* are all variables, while *Last-Name* and *2ndBase* are not. PROLOG also has a special anonymous variable "_". The query, *?-has(tom, _)*., for example, will return the answer "*yes*" (which means *true*) if there exists an atom satisfying the "*has*" relationship, with "*tom*" as the first term, or if there is a rule, with the same, whose body can be proved. All queries, facts and rules are terminated by a period "."

Comments in PROLOG are indicated by the sign "%", before the comment as in:

% tree(LeftTree, Info, RightTree)

Some implementations, such as *PD-Prolog*, enclose comments between "/*" and "*/".

7.2 Lists

One of the important elements of the PROLOG language is the *list*. In PROLOG, a ***list*** is an ordered series of items that can be accessed as a single argument (Garavaglia [*20*]). Lists are important because they can be used to conceptualize just about every representation of data (or data structures). An example of a list is a shopping list:

eggs, tea, milk, steak, spinach, toothpaste

In PROLOG this is written as:

[*eggs, tea, milk, steak, spinach, toothpaste*]

The items in the list are called *elements* (Konigsberger [*31*]). The first element in the list is called its head and the remaining elements comprise the tail. In PROLOG, the head is usually separated from the tail by the long vertical bar "|". In referring to a list with the use of variables in a rule or expression, the vertical bar is used as follows:

[*H*|*T*]

If this were used to reference a shopping list then

H = *eggs*
T = [*tea, milk, steak, spinach, toothpaste*]

As can be seen, the tail of a list is a list.

Another way we can represent lists in PROLOG is by the use of the functor "./*2*". The "." is the function, and the *2* indicates that it has an arity of *2*. A one member list would thus be represented as:

There are numerous procedures that can be used for manipulating lists - the *sort* above being one of them. The two procedures shown in *Figure 7.1* can be used to determine membership of an element in a list, and appending one list to another. Other procedures include subtraction of two lists, subsets, bubblesort, splitting a list, quicksort, implementation of stacks and queues, graph implementation and many others (see Coelho & Cotta [*15*]).

```
member(A, [A|_]).                    append([], L, L).
member(A, [_|Tail]) :-           append([X|L1], L2, [X|L3]) :-
        member(A, Tail).                     append(L1, L2, L3).
```

Figure 7.1

7.3 Recursion

PROLOG allows the definition of recursive predicates. A predicate is a *recursive* if its definition involves the predicate itself. An example is the *member* predicate above (*Figure 7.1*). The first argument of the predicate *member* is a variable, the second argument is a list. The first fact says that *X* is a member of a list if it is the head of the list. The second clause says that *X* is a member of a list if it is a member of its tail. Notice the use of the anonymous variable "_", and the list concatenation operator "|" (Ceri, Gottlab and Tanca [*9*]).

7.4 The CUT (!) Predicate

The built in predicate ***cut*** (*!/0*), always succeeds and prevents reevaluation of any clause that precede it. PROLOG searches for all possible solutions to a query. If we are aware that there is only

one, *cutting* off further search after the single solution has ben found to save both time and space (Appleby [*1*]).

Below is a modified procedure for append.

```
append2([], L, L) :- !.
append2([X|L1], L2, [X|L3]) :-
        append(L1, L2, L3).
```

PROLOG will stop the search the first time it satisfies append2([], L, L). Such a one solution procedure would be useful if we always were to use it with two ground clause lists as the first two arguments, as in

?- append2([1,2,3], [4,5,6], Z).

Which would give us *Z = [1,2,3,4,5,6]*. However, if we want to find all possible sublists as in:

?- append2(X, Y, [1,2,3,4,5,6]).

PROLOG would return only one answer:

X = []; Y = [1,2,3,4,5,6].

The *cut* would prevent any further search.

The *cut* predicate may also be used to implement the ***if-then-else*** statement. for example, the clause:

a1 :- b1, !, b2.
a1 :- b3.

is equivalent to the following

to prove a1 if you can prove b1, then prove b2 ***else*** *prove b3.*

8 Datalog

In this chapter, we will discuss *Datalog* and its similarities and differences with both PROLOG and relational models[7] in DBMS. The name "*Datalog*" was coined to suggest a version of PROLOG suitable for database systems. It differs from PROLOG in several respects:

1. Datalog does not allow function symbols in arguments. It only allows variables and constants as arguments of predicates.

2. The semantics of Datalog programs follow the model-theoretic point of view, or when equivalent, the proof-theoretic approach. PROLOG, however, has a computational "meaning," which may deviate in some cases from either the model-theoretic or proof-theoretic meanings (Ullman [*66*]).

The underlying mathematical model of data for datalog is essentially that of the relational model. Predicate symbols in datalog denote relations. However, as in the formal definition of relational algebra, these relations do not have attributes with which to name their columns. Rather they are relations in the set-of-lists sense, where the components appear in fixed order, and reference to a column is only by its position among the arguments of a given predicate symbol. For example, if p is a predicate symbol, then we may refer to $p(X, Y, Z)$, and variable X will denote the first component of some tuple in the relation corresponding to predicate p.

---

[7] For purposes of understanding *relational models*, we would advice reference to *Elmasri* [*18*] pp. 137-165, *Ullman* [*66*] pp. 43-65 and *Codd* [*14*].

8.1 *Extensional and Intensional Predicates*

Another distinction between the relational and datalog models is that in datalog, there are two ways relations can be defined. A predicate whose relation is stored in the database is called an ***extensional database*** (***EDB***) relation, while one defined by logical rules is called an ***intensional database*** (***IDB***) relation.

In the relational model, all relations are EDB relations. The capability to create *views* in models like the relational model is somewhat analogous to the ability in datalog to define *IDB* relations. However, the view definition facility in relational DBMS's does not compare in power with logical rules as a definition mechanism.

8.2 *Atomic Formulae*

Datalog programs are built from ***atomic formulae*** which are predicate symbols with a list of arguments, e.g. $p(A_1,...,A_n)$, where p is the predicate symbol. An argument in datalog can either be a variable or a constant. Constants - as in PROLOG - begin with lowercase letters whereas variables begin with uppercase letters. Numbers are also used as constants. Each predicate symbol is associated with a particular number of arguments that it takes and $p^{(k)}$ is used to denote a predicate of arity k.

An atomic formula denotes a relation. It is the relation of its predicate restricted by:

1. Selecting for equality between a constant and the component(s) in which the constant appear, e.g. in *customer(joe, Address, Balance)*,

2. Selecting for equality between components that have the same variable. e.g. in *includes(X,Item,X)*.

Notice that although there are no names for attributes in the datalog model, selecting suggestive variables like *Address* (above) help remind us what is going on. However, as in relational algebra, we must remember the intuitive meaning of each position in a list of arguments.

8.3 Dependency Graphs and Recursion

We frequently need to discuss the way predicates in a logic program depend on one another. To do so, we draw a ***dependency graph***, where nodes are the ordinary predicates. There is an arc from a predicate q if there is a rule with a subgoal whose predicate is p and with a head whose predicate is q. *Figure 8.1* shows an example of a datalog program. Rule(1) may be read as "*for all X, Y, and Z, if Z is a parent of both X and Y, and X is not Y, then X is a sibling of Y.*" In this case we are regarding all variables as universally quantified.

```
(1) sibling(X, Y) :- parent(X, Z), parent(Y, Z), X ≠ Y.

(2) cousin(X, Y) :- parent(X, Xp), parent(Y, Yp),
                    sibling(Xp, Yp).

(3) cousin(X, Y) :- parent(X, Xp), parent(Y, Yp),
                    cousin(Xp, Yp).

(4) related(X, Y) :- sibling(X, Y).
(5) related(X, Y) :- related(X, Z), parent(Y, Z).
(6) related(X, Y) :- related(Z, Y), parent(X, Z).
```

Figure 8.1 A Datalog Program

The *dependency graph* for *Figure 8.1* is depicted in *Figure 8.2*.

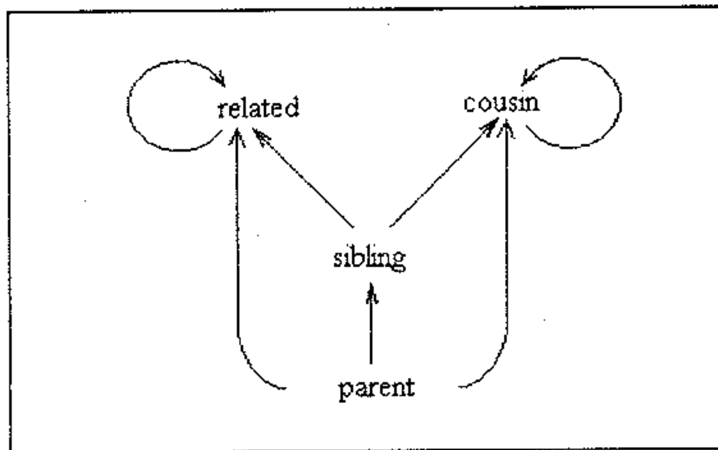


Figure 8.2 Dependency Graph for Figure 8.1

A logic program is ***recursive***, if its dependency graph has one or more cycles[8].

All the predicates that are on one or more cycles are said to be ***recursive predicates***. A logic program with an acyclic dependency graph is *nonrecursive*. Clearly all predicates in a nonrecursive program are nonrecursive. We also call a predicate *nonrecursive* if it is in a recursive program but is not part of any cycle in the dependency graph.

In *Figure 8.2*, there are two cycles, one involving only *cousin* and the other involving *related*. Thus, these predicates are recursive, and the predicates *parent* and *sibling* are nonrecursive. Therefore the program of *Figure 8.1* is recursive, because it has some recursive predicates.

---

[8] A *cycle* consisting of an arc from a node to itself makes the program recursive. One-node cycles are more common than multinode cycles.

8.4 Safe Rules

A program or a rule is said to be ***safe*** if it generates a *finite* set of facts. There are constraints that must be placed on the form of datalog rules if they are to be safe. One situation where we get unsafe rules that can generate an infinite number of facts arises when one of the variables in the rule can range over an infinite domain of values, and that variable is not limited to ranging over a finite relation.

Example 8.1 The rule

biggerThan(X, Y) :- X > Y.

defines an infinite relation, if *X* and *Y* are allowed to range over the integers, or any infinite set. Also the rule

loves(X, Y) :- lover(Y).

i.e., "*all the world loves a lover,*" defines an infinite set of pairs *loves(X, Y)* even if the relation *lover* is a finite set, as long as the first argument of *loves* ranges over an infinite set. The rule is unsafe because the variable *X* appears only at the head of the rule. Another problem arises due to the fact that some rules that are *theoretically safe*, such as:

big_salary(Y) :- Y > 60000, employee(X), salary(X, Y).

are *computationally unsafe* using top-down, left-to-right inference mechanisms like PROLOG. In the rule above, there would first be a search for a value for Y and then PROLOG would check whether it is salary of an employee. In order to rectify this problem, the same rule could be written in the form:

$$big_salary(Y) :- employee(X), salary(X, Y), Y > 60000.$$

Albeit we can try to avoid rules that create infinite relations from finite (i.e. *unsafe rules*) ones by insisting that each variable appearing in the rule be "*limited*." The intuitive idea is that we assume all the ordinary (non-built-in) predicates appearing in the body correspond to finite relations. After making that assumption, we need assurance that for each variable X, there is a finite set of values V_X such that in any assignment of values to the variables that make the body true, the value of X must come from V_X. We formally define *limited* variables for a given rule as follows:

1. Any variable that appears as an argument in an ordinary predicate of the body is limited.
2. Any variable X that appears in subgoal $X = a$ or $a = X$, where a is a constant, is limited.
3. Variable X is limited if it appears in a subgoal $X = Y$ or $Y = X$, where Y is variable already known to be limited (Ullman [*66*]).

Note that (*1*) and (*2*) form basis for the definition, and (*3*) can be applied repeatedly to discover more limited variables.

A rule is said to be *safe* if all its variables are limited. The critical issue is whether variables appearing in the head and variables appearing in subgoals with built-in predicates either appear in some subgoal with an ordinary predicate, are equated to constants, or are equated to other limited variables through the recursive use of (*3*).

Example 8.2 The first rule of *Example 8.1* is not safe because none of its variables are limited. The second is not safe because, although Y is limited by its occurrence in the subgoal *lover(Y)*, there is no way to limit X. In general, a variable appearing only in the head of a rule cannot be limited, so its rule cannot be safe.

Rule (1) of *Figure 8.1* is safe because X, Y, and Z are limited by their occurrences in the two *parent* subgoals. Note that the built-in predicate $X \neq Y$ cannot result in an infinite number of siblings, because X and Y are already limited to be individual that appear in the first component of the *parent* relation. All the other rules in Figure 8.1 are likewise safe.

For a more complex example, consider the rule

$$p(X, Y) :\!- q(X, Z), W = a, Y = W.$$

X and Z are limited by rule (1), because of the first subgoal in the body, W is limited by the rule (2), because of the second subgoal, and therefore (3) tells us Y is limited because of the third subgoal. As all variables are limited, the rule is safe (Ullman [*66*]).

9 Deductive Databases

In this chapter we discuss the topic of deductive databases and in particular disjunctive deductive databases. A *deductive database* is a logic program with no function symbols. The major difference between a logic program and a deductive database is the query evaluation process. In databases, the user is generally interested in all possible answers to a query in contrast to a single answer from an *SLI refutation*. Considering the restriction of function free programs, it is possible to develop query answering procedures adapted to answer database queries (*LMR* [*35*]).

The main differences between *deductive databases* and *knowledge-based systems* (also known as *expert-systems*) are two-fold:

1. Knowledge-based systems have traditionally assumed that the data needed resides in the main memory; hence, secondary storage management is not an issue. Deductive database systems attempt to change this restriction so that either a DBMS is enhanced to handle an expert system interface or an expert system is enhanced to handle secondary storage resident data.

2. The knowledge in an expert system is extracted from application experts and refers to an application domain rather than to knowledge inherent in the data (*Elmasri* [*18*]).

9.1 Disjunctive Deductive Databases

A disjunctive deductive database (*DDDB*) is a collection of data separated into two parts. One part is an extensional database which represents the set of (possibly indefinite) facts known about the world. It is a finite set of ground positive disjunctions known to be *true*. The second part is an intensional database that consists of a finite set of disjunctive program clauses that represent general rules or conditions from which new facts can be deduced. If the set of rules and facts used are definite clauses, then we have definite deductive databases, or sometimes simply called deductive databases. Because part of the theoretical foundation for some deductive database systems is mathematical logic, they are often referred to as *logic databases.*

We may formally define disjunctive databases as follows:

Definition 9.1 (*LMR* [*35*]) Let *L* be a function-free first order language with a finite number of constant symbols. A disjunctive logic program *DB* defined in *L* is called a ***disjunctive deductive database*** (**DDDB**). The *intensional disjunctive database* part of *DB*, $IDDB_{DB}$, is the set of program clauses in *DB* with at least one atom in the body of each clause. The *extensional disjunctive database* part of *DB*, $EDDB_{DB}$, is the set of clauses in *DB* with an empty body. The following is an example of a simple disjunctive database:

Example 9.1 Let *DB* be a disjunctive deductive database. *DB* describes a world of blocks and spheres. The $EDDB_{DB}$ consists of:

E1: blk(1).	*E5: red(1).*
E2: blk(2) sph(2).	*E6: blue(3).*
E3: blk(3).	*E7: blue(4).*
E4: sph(4).	*E8: top(1,2).*
	E9: top(2,3).

The $IDDB_{DB}$ consists of:

I1: red(X) blue(X) ← blk(X).
I2: redblue(X,Y) ← top(X,Y), red(X), blue(Y).
I3: blk(X) ← top(X, Y).

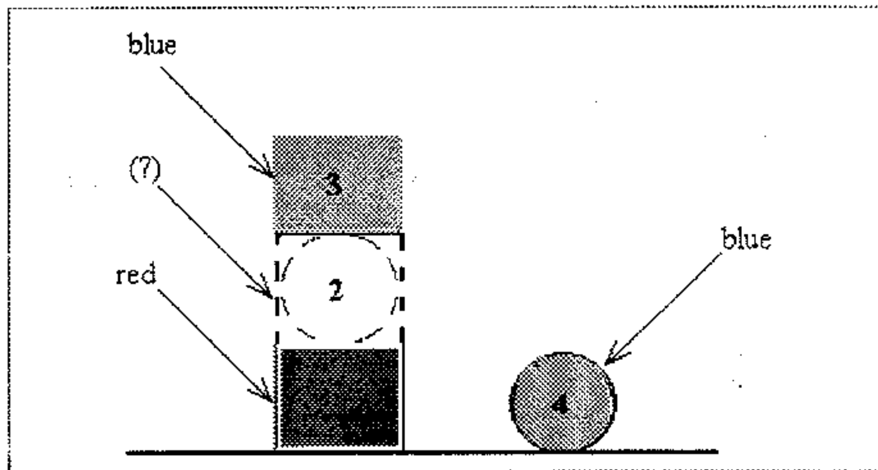


Figure 9.1 The world of blocks and spheres of *Example 9.1*

9.2 Queries and Answers

In Chapter 5 (*Definition 5.2*), we define a *correct answer* to a goal $\leftarrow A_1, ..., A_k$ in a disjunctive logic program P to be a set of substitutions $\{\theta_1,\theta_2,...,\theta_n\}$ such that $\forall((A_1\wedge...\wedge A_k)\,\theta_1 \vee (A_1\wedge...\wedge A_k)\,\theta_2 \vee...\vee (A_1\wedge...\wedge A_k)\,\theta_n)$ is a logical consequence of P. In the case of *DDDB*s we are interested not only in one of these answers but in the set of *all* possible answers to the query. A typical query to a database such as the one in *Example 9.1* is to know what objects in the database are spheres. For this query the answer is given by the set of all correct answers for the goal $\leftarrow$*sph(X)*. Answering this query using *SLI* resolution requires the collection of all *SLI refutations* for

($\varepsilon^* \neg sph(X)$). In many cases, these refutations may contain several repetitions of the same derivation.

An alternative to this procedure is to start with the facts and then to evaluate each clause as follows. When the body of a clause is proven to be *true*, then the head clause can be concluded to be *true*. When all rules have been evaluated, we can repeat this process and perform deduction with the rules using the original facts and those derived by application of the rules until all clauses that imply the query are obtained. This mechanism is similar to the operations that is performed by the operator T^S_P and simultaneously computes all the answers to a query. Computations such as the one performed by the operator T^s_P are called *bottom-up* procedures. Procedures such as *SLI* resolution that start with the query and go through the rules until the refutation is obtained are called *top-down* procedures (*LMR* [35]).

Normally most *DDDB*s are concerned with *bottom-up* procedures. As mentioned earlier the data that is to be processed is resident on secondary storage. If we are to use the *top-down* tuple-at-a-time processing methods it would take a tremendously long time to get the set of all possible answers to our queries.

According to the definition of a correct answer (*Definition 5.2*), given the goal $\leftarrow q(X)$, and the $DB = \{q(a);\ q(b) \leftarrow q(c)\}$, the following are possible sets of answers from the query: $\{\{X=a\}\}$, $\{\{X=a\}, \{X=b\}\}$, $\{\{X=a\}, \{X=c\},\}$, and $\{\{X=a\}, \{X=b\}, \{X=c\}\}$. The meaning of these answers is that $q(a)$, $q(a) \vee q(b)$, $q(a) \vee q(c)$, $q(a) \vee q(b) \vee q(c)$ are true in the database. All of these are correct answers for the query $\exists X(q(X))$, but the answer $\{X = a\}$ is sufficient since all other answers are particular cases of $\{X = a\}$. Notice that $q(a)$ subsumes $q(a) \vee q(b)$, $q(a) \vee q(c)$, and $q(a) \vee q(b) \vee q(c)$. However, checking for *minimality* of an answer to a database query with the known algorithms, may require among other things, a subsumption check which is computationally expensive. The redundancy of the answer will normally depend on the set of disjunctions the

procedure generates to represent the answers. The answers for the query will be a set of disjunctions that imply the query. Formally, these concepts are described by the following definitions.

Definition 9.2 (LMR [35]) Let *DB* be a *DDDB* and let *S* be a *state*[9] of *DB*. *S* ***characterizes*** *DB* if for every ground clause *C* in the disjunctive Herbrand base of *DB*, DHB_{DB}, the following condition holds:

$$DB \models C \text{ iff } (\exists C' \in S \text{ and } C' \Rightarrow C).$$

In other words, a state characterizes *DB* if it contains only positive ground clauses that are logical consequences of *DB* and for every positive ground clause *C* that is a logical consequence of *DB* and does not belong to *S*, there is a clause in *S* that implies *C*.

Definition 9.3 (*LMR* [35]) A ***database query*** is a positive (possibly ground) disjunctive clause.

The use of disjunctions in database queries instead of conjunctions neither increases nor reduces the expressive power of queries. A disjunctive query of the form $Q_1 \vee ... \vee Q_n$ is equivalent to the query $\exists X_1,...,\exists X_m q(X_1,...,X_m)$ where q is a predicate symbol that does not occur in *DB*, if the rules

$$q(X_1,...,X_m) \leftarrow Q_1$$

$$\cdots$$

$$q(X_1,...,X_m) \leftarrow Q_n$$

[9] *Definition 3.15.*

are added to the database and X_1, ..., X_m are all the variables that appear in the $Q_1 \vee ... \vee Q_n$ (*Shepherdson* [*70*]). But the fact that queries are in disjunctions of atoms usually simplifies the description of algorithms (*LMR* [*35*]).

Definition 9.4 (*LMR* [*35*]) Let *DB* be a disjunctive database and let *C* be a positive (possibly ground) disjunctive clause, called a *database query*. Let *S* be a state such that *S* characterizes *DB*. A ***database query*** for *C* in *S* is a ground clause *C′* in *S* such that ($C' \Rightarrow \exists C$). A subset *S′* of *S* is a ***complete set of database answers*** for *C* in *S* if and only if for every database answer *C′* in S there is a database answer *C″* in *S′* such that $C' \Rightarrow C''$.

Example 9.2 From *Definition 9.4*, a database answer to a database query is a set of positive ground clauses known to be ***true*** in the disjunctive deductive database (i.e. part of characterization state of the database) such that each answer implies the existential closure of the database query. Using this notion, let $S = \{\ q(a),\ q(a) \vee q(b),\ q(a) \vee q(c),\ q(a) \vee q(b) \vee q(c)\ \}$ be a characterization state of a database *DB*. Then *q(a)* is a database answer to the database query *q(X)*. The remaining disjunctions in the set are also database answers for the query but they are not necessarily members of the complete set, since *q(a)* implies all of them. Moreover, {*q(a)*} and any subset of *S* that contains *q(a)* is a complete set of database answers.

9.3 Minimal Models and Deductive Databases

Traditionally, the declarative semantics of logic programming and deductive databases has been studied based on the notion of *minimal models*. For instance, the *least Herbrand model semantics* for Horn logic programs (*van Emden* [*67*]), the *perfect model semantics* for stratified logic programs (*Przymusinski* [*50*]), and the *stable model semantics* for normal logic programs (*Gelfond*

[*21*]) are all minimal models. The minimal models reflect the so-called *Occam's razor* such that "only those objects should be assumed to exist which are minimally required by the context." Such a *principal of minimality* plays a fundamental role in the area of not only logic programming but also *nonmonotonic reasoning* in artificial intelligence (*McCarthy* [*40*]). Therefore, it has been recognized that the principle of minimality is one of the most basic and indispensable criteria that each semantics for commonsense reasoning should obey (*Schlipf* [*60*]).

9.3.1 Minimal Model Semantics

The model-theoretic approach is particularly well understood in the case of the so called *Horn* (or *definite*) databases, otherwise known as *positive logic programs*, i.e. databases consisting of clauses of the form

$$C \leftarrow A_1, \ldots, A_m$$

with $m \geq 0$, and C and A_i denoting atomic formulae (atoms).

Example 9.3 Let our database *DB* consist of the following clauses:

good_mathematician ← physicist(X).
physicist(einstein).
businessman(perot).

This database has several different models, the largest of which is the model M_{max} in which both men are at the same time businessmen, physicists and good mathematicians. This model hardly seems to correctly describe the intended meaning of *DB*. In particular, there is nothing in this

database to imply that *Perot* is a physicist or that *Einstein* is a businessman. In fact, we are inclined to believe that the lack of such information indicates that we can assume the contrary.

The database also has a smallest model M_{min}, in which only *Einstein* is a physicist and good mathematician and only *Perot* is a businessman. This model seems to correctly reflect the semantics of *DB*, at the same time incorporating the classical case of the *closed-world assumption*, namely *Reiter's CWA*: if no reason exists for some positive statement to be true, we are allowed to infer that it is false (*Reiter* [*54*]).

Every Horn database has exactly one minimal (i.e. smallest) model. The *unique minimal model of a Horn database DB provides a natural semantics of DB, incorporating a suitable form of the CWA.*

Horn database constitute a fairly restricted class of deductive databases and do not allow the full expressive power of first-order logic to be utilized. A natural extension of this class of databases is obtained by allowing disjunctions of atoms in the consequents of the clauses, thereby permitting general forms of disjunctive information. We will call such databases *positive disjunctive* databases. A positive disjunctive database therefore consists of a set of clauses of the form

$$C_1, \ldots, C_p \leftarrow A_1, \ldots, A_m$$

where $m \geq 0, p \geq 1$ and A_i and C_k denoting atoms.

Example 9.4 Suppose that a database *DB* consists of the clauses

successful_businessman(X) ∨ renown_scientist(X) ← famous_man(X).
famous_man(smith).
good_mathematician(jones).

Again this database has several models. In the largest model M_{max}, both men, *Smith* and *Jones*, are successful businessmen, renown scientists, good mathematicians, and famous men. This model obviously does not represent the intended meaning of *DB*.

It is, however, also easy to see that this database does not have a smallest model. How are we supposed to define the semantics of *DB*? The problem fortunately has a natural solution: Even though *DB* does not have a smallest model, it has two models which are *minimal*, i.e. they do not contain smaller models. In both of them, *Smith* is the only famous man, *Jones* is the only good mathematician. In one of them M_1, *Smith* is the only successful businessman, whereas in the other M_2, *Smith* is famous and a renown scientist, while *Jones* is neither. Both of these models seem to correctly capture *one aspect* of the intended meaning of *DB*, and together M_1 and M_2 define a proper semantics for *DB*: A sentence (formula) F is true in *DB* iff it is true in both models. In particular, it is true that *Smith* is either a famous businessman or renown scientist, but we do not know which. On the other hand, it is also true that *Jones* is neither famous, nor a successful businessman, nor a renown scientist, since all of these are false in both minimal models.

It is well known that for every model N of a positive disjunctive database *DB* there is a minimal model M that is contained in N (*Bossu* and *Siegel* [*6*]). In fact, every positive disjunctive database has at least one minimal model. The minimal model semantics corresponds to a natural form of the closed-world assumption, namely the so called *Extended Closed-World Assumption* (*ECWA*) (see *Gelfond et al.* [*22*]), which extends *Minker's Generalized Closed-World Assumption* (*Minker* [*46*]) (see also *Yahya* and *Henschen* [*71*]). For Herbrand models, it is also equivalent to the semantics of parallel circumscription (with all predicates minimized) (*McCarthy* [*39*], *Lifshitz* [*33*]). We can hence assert the following:

> *The set MIN(DB) of all minimal models of a positive disjunctive database DB provides the intended semantics of DB, incorporating a suitable form of the closed world assumption.*

9.3.2 Insufficiency of Minimal Model Semantics for General Databases

The expressive power of positive disjunctive databases is not sufficient for general databases. In practical applications permitting negation in the premises of the rules greatly increases their expressiveness. Consider a disjunctive database consisting of a set of clauses of the form:

$$C_1, \ldots, C_p \leftarrow A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n$$

where $p \geq 1$, $m,n \geq 0$, and A_i, B_j and C_k denoting atoms[10]. If $p = 1$ for all clauses, then such a database is called a *(general) logic program* (see *Lloyd* [*34*] p68).

With disjunctive databases the situation becomes more complex.

Example 9.5 Suppose that we know that a typical businessman tends to avoid using advanced mathematics in his work, unless somehow he happens to be a good mathematician, and that *Perot* is a businessman, and that *Einstein* is a physicist. We can express these facts using negation as follows:

avoids_math(X) ← businessman(X), ¬good_mathematician(X). (1)
businessman(perot).
physicist(einstein).

This database *DB* has two minimal models M_1 and M_2. In both of them *Perot* is the only businessman and *Einstein* is the only physicist; but in M_1 *Perot* avoids advances mathematics and in M_2 he is a good mathematician, instead. Do both these models capture the intended meaning of the *DB*?

---

10 Such databases are called *positivistic* by *Bidoit* and *Hull* [5].

Certainly not! By placing the negated predicate *good_mathematician(X)* among the premises of the rule, we most likely intended to say that businessmen, in general, do not use advanced mathematics unless they happen to be good mathematicians. Since we have no reason to believe that *Perot* is a good mathematician, we are forced to infer that he does not use advanced mathematics. Therefore, only the first minimal model M_1 corresponds with the intended meaning of *DB*.

The reason for this asymmetry is easy to explain. The first clause (*1*) of *DB* is logically equivalent to the following clause (*2*) without negation:

$$good_mathematician(X) \vee avoids_math(X) \leftarrow businessman(X). \qquad (2)$$

and models M_1 and M_2 are therefore also minimal models of the positive disjunctive database DB^*, obtained from *DB* by replacing (*1*) by (*2*). These models provide a correct semantics of DB^* because DB^* does not assign different priorities to the predicate *good_mathematician* and *avoids_math* treating them as equally plausible. The database *DB*, on the other hand, gives a *higher priority* to the predicate *good_mathematician* than to the predicate *avoids_math*[11].

We can easily imagine the above priorities reversed. This is for instance the case in the following database:

good_mathematician(X) ← *physicist(X)*, ¬*avoids_math(X)*.
businessman(perot).
physicist(einstein).

[11] According to the established convention (see *Lifshitz* [*33*]), a *higher priority for minimization*, i.e. predicate *A* has a higher priority than predicate *B* if *A* is supposed to be minimized before *B* is.

which says that if X is a *physicist* and if we have no specific evidence showing that he *avoids mathematics*, then we are allowed to assume that he is a *good mathematician*. This shows that relative priorities among the predicates in the database are determined by the syntax of its clauses, with *consequents having lower priority than negated premises*.

From the preceding example we can see that the set $MIN(DB)$ of all minimal models of a disjunctive database *may contain models which do not properly interpret the declarative meaning of the database* and therefore we need to properly identify the class of models of DB which provides the correct, intended meaning of DB. The class of *perfect models* described in the next section fulfills those needs.

9.3.3 Perfect Model Semantics

In his paper, Przymusinski [*50*] introduces the concept (or class) of *perfect models*. This is a subclass of the class of minimal models enjoying many of its natural properties. This concept is based on *relative prioritization* in the database.

As we observed above, the syntax of clauses determines the relative *priorities* among the predicates in the database and that:

I. Negative premises should have *higher* priority than consequents.

Moreover we can assume that

II. Positive premises should have priority *higher than* or *equal* to that of consequents.

Indeed, if $B \leftarrow A$, then minimizing B immediately results in A being minimized too. Consequently, A is always minimized before or at the same time that B is.

III. Predicates appearing in the consequent of a given clause should have the *same* priority.

This is true otherwise those predicates whose priority is higher should be converted into negative premises.

To formalize conditions I - III, Przymusinski introduces a *priority relation* $\prec$ between elements A and B of the Herbrand Base H_B and an auxiliary relation $\preceq$. If $B \prec A$, then we say that *B has a priority higher than A* and if $B \preceq A$, then we say that the *priority of A is less than or equal to that of B.*

Definition 9.5 Relations $\prec$ and $\preceq$ are defined by the following rules:

(*PR1*) (*Condition I*) $B \prec C$, if B is a negative premise and C is one of the consequents in a ground instance of a clause from DB.

(*PR2*) (*Condition II*) $C \preceq A$, if A is a positive premise and C is one of the consequents in a ground instance of a clause from DB.

(*PR3*) (*Condition III*) $C \preceq C'$, if C' and C are both consequents in a ground instance of a clause from DB.

augmented by transitivity and closure rules:

(*PR4*) (*transitivity of* $\prec$) if $B \preceq A$ and $C \prec B$ (resp. $A \prec D$), then $C \preceq A$ (resp. $B \prec D$);

(*PR5*) (*transitivity of* $\prec$) if $B \prec A$ and $C \preceq B$, then $C \preceq A$;

(*PR6*) ($\prec$ *implies* $\preceq$) if $B \prec A$ then $B \preceq A$;

(*PR7*) (*closure axiom*) Nothing else satisfies $\prec$ or $\preceq$.

Example 9.6 In the database given by

$a(X) \vee c(X) \leftarrow b(X), \neg g(X).$

$g(X) \leftarrow p(X).$

$b(t).$

we have $g(t) \prec a(t)$, $g(t) \prec c(t)$ (by *PR1*); $b(t) \preceq a(t)$, $b(t) \preceq c(t)$, $p(t) \preceq g(t)$, (by *PR2*); and $a(t) \preceq c(t)$, $c(t) \preceq a(t)$ (by *PR3*). Consequently, $p(t) \prec a(t)$, $p(t) \prec c(t)$ (by *PR5*), but neither $c(t) \prec p(t)$ nor $b(t) \prec a(t)$ is true (by *PR7*).

Definition 9.6 Suppose that M and N are two *distinct* models of a disjunctive database *DB*. We say that a N is ***preferable*** to M (or $N \lll M$), if for every ground atom A in N-M there is a ground atom B in M-N, such that $A \prec B$. We say that a model M of *DB* is ***perfect*** if there are no models preferable to M.

We call the relation $\lll$ the ***preference relation*** between models. If $M = N$ or $N \lll M$ then we write $N \underline{\lll} M$.

Example 9.7 Only model M_1 in *Example 9.5* is perfect. Indeed:

$M_1 = \{$ *businessman(perot), physicist(einstein), avoids_math(perot)* $\}$,

$M_2 = \{$ *businessman(perot), physicist(einstein), good_mathematician(perot)* $\}$,

and we know that *good_mathematician(perot)* $\prec$ *avoids_math(perot)* and therefore $M_1 \lll M_2$. Consequently, M_1 is perfect, but M_2 is not.

Not every disjunctive database - nor even a logic program - has a perfect model:

Example 9.8 The database:

$$q(a) \leftarrow \neg p(a) \text{ and } p(a) \leftarrow \neg q(a) \quad (3)$$

has only two minimal models $M_1 = \{ p(a) \}$ $M_2 = \{ q(a) \}$ and since $p(a) \prec q(a)$ and $q(a) \prec p(a)$ we have $M_1 \prec\!\!\prec M_2$ and $M_2 \prec\!\!\prec M_1$, thus none of our models is perfect[12]. The cause of this pecularity is quite clear: our semantics is based on relative priorities between ground atoms and therefore we have to be consistent when assigning those priorities to avoid priority conflict (cycles), which could render our semantics meaningless.

Definition 9.7 A relation $<$ is said to be *noetherian* if there is *no* infinite increasing sequence $A_0 < A_1 < A_2 < \cdots$.

The non-existence of a perfect model in this example above and the fact that the preference relation $\prec\!\!\prec$ was not transitive (and therefore not partial order) turns outto be caused by the fact that the priority relation $\prec$ was not *noetherian*.

[12] It also shows that in this case the preference relation $\prec\!\!\prec$ is not transitive.

9.4 Stratification

A deductive database query language can be enhanced by permitting negated literals in the bodies of rules in programs. Once we permit negated literals in the rules, however, we lose an important property of the rules called the minimal model, which we discussed earlier. In the presence of negated literals, a program may not have a least model. For example, the program

$$p(a) \leftarrow \neg p(b).$$

has two minimal models: $\{ p(a) \}$ and $\{ p(b) \}$.

One important class of negation that has been extensively studied is stratified negation (*Apt*, *Blair* and *Walker* [2], Przymusinski [50]). A program is said to be stratifiable if it has no recursion through negation. Programs in this class have a very intuitive semantics and can be efficiently evaluated. The example that follows describes a stratified program (*Elmasri* [18]). Consider the following program P_1:

$$r_1 : ancestor(X, Y) \leftarrow parent(X, Y).$$
$$r_2 : ancestor(X, Z) \leftarrow parent(X, Y), ancestor(Y, Z).$$
$$r_3 : nocyc(X, Y) \leftarrow ancestor(X, Y), \neg ancestor(Y, X).$$

Notice that the third rule has a negative literal in its body. This program is stratifiable because the definition of the predicate *nocyc* depends (negatively) on the definition of *ancestor*, but the definition *ancestor* does not depend on the definition of *nocyc*. A bottom-up evaluation of P_1 would first compute a fixpoint of rules r_1 and r_2 (the rules defining *ancestor*). Rule r_3 is applied only when all the *ancestor* facts are known.

Conclusion

Open Problems

Two aspects need to be considered in the computation of minimal models as stated in *LMR* [*35*], while designing a data structure to store those minimal models. The first aspect is the replication of atoms in different models that can be produced when new disjunctions are introduced and a new set of models is created. The second aspect is the duplication of proofs. These both consume a lot of time and space, which are critical in large databases.

Problem 1: Avoiding Duplicate Proofs

In building a Deductive DBMS it is desirable to avoid having duplicate proofs. Wos [*70*] identified the problem of recomputing redundant information in automated reasoning. Spencer [*62*] considered it in the setting of logic programming and came up with the foothold refinement algorithm. This is a refinement of linear resolution that admits fewer duplicate proofs than Loveland's popular *MESON* format (Loveland [*37*]). Spencer's paper builds upon his original presentation of the foothold format [*64*], with a more precise definition and more rigorous proofs. After his original foothold paper, he proposed a different restriction on *ME*[13] *reduction* by the use of *ordered clauses* [*63*].

One of the open problems is to consider this in the setting of databases. Here as was mentioned earlier (*p 80*), the *subsumption check* is an expensive operation that should be avoided. It

[13] *ME - Model Elimination* - Loveland's algorithm as presented in [*37*].

is therefore necessary to have an algorithm that would minimize redundant proofs as they are built, without losing completeness.

Problem 2: Minimization of Proofs

In this we consider the minimization of the size of the proof, in contrast to the number of proofs. Horton and Spencer [*29*] have come up with a data structure - *clause trees* (see *subsection 5.3 p 50*). This can be used with the Bottom-up algorithm to quickly evaluate answers as follows:

1. Form the set T of all clause trees, from the intensional database ($IDDB_{DB}$), with only literals from the query, or extensional database ($EDDB_{DB}$) as open leaves (i.e. form all *ACTs*).

2. Form the set S of ground instances of clause trees from T using $EDDB_{DB}$.

Example:

Let P be the function free disjunctive logic program:

IDB: {*p(X), r(X), ~c(X)*}.
{*p(X), ~r(X)*}.

EDB: *c(1).* *c(2).*
c(5). *c(6).*

Thus the query:

Query $\leftarrow p(X)$.

may be solved using clause trees as depicted in *Figure 10.1*.

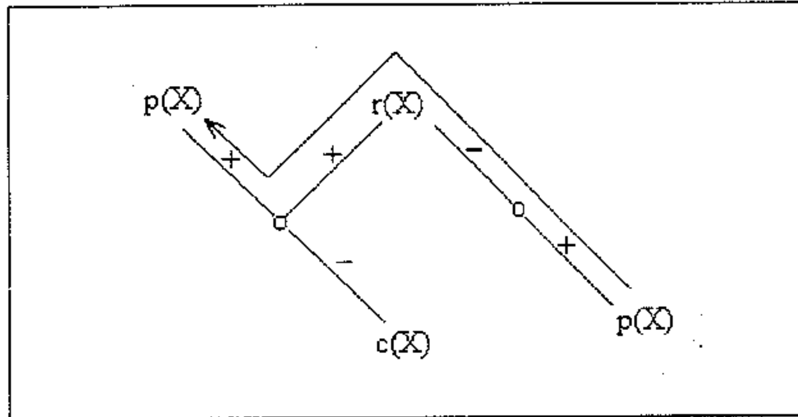


Figure 10.1 Using Clause-trees for Bottom-up evaluation of Disjunctive Database Queries

Here we see that $cl(T_i) = \{p(X), \neg c(X)\}$, and so applying this to the *EDB* we get

$$X \in \{1, 2, 5, 6\}.$$

References

1. Appleby Doris. *Programming languages: Paradigm and Practice*. McGraw-Hill, Inc 1991.

2. Apt K., Blair H. and Walker A. *Towards a theory of declararive knowledge*. Preprints of Workshop on Foundations of Deductive Databases and Logic Programming, J. Minker (ed) Washington D.C., August 1986 547-623.

3. Apt K. R. *Logic Programming*, in J. van Leewen (ed) Handbook of Theoretical Computer Science, Vol. B (North Holland, Amsterdam, 1990) 495-574.

4. Apt K. R. and van Emden M. H. *Contributions to the Theory of Logic Programming*, J. ACM **29**, 3 (July 1982), 841-862.

5. Bidoit N. and Hull R. *Positivism vs Minimalism in Deductive Databases*, Proceedings ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Cambridge, MA 123-132, 1986.

6. Bossu G. and Siegel P. *Saturation, Nonmonotonic Reasoning and the Closed World Assumption*, Artificial Intelligence 25(1) 13-63, 1985.

7. Brookshear J. Glenn. *Computer Science: An Overview* (3rd ed.) Benjamin/ Cummings Publishing, Inc 1991.

8. Calingaert P. *Program Translation Fundamentals: Methods of Issues*. Rockville, MD: Computer Science Press, 1988.

9. Ceri S., Gottlab G. and Tanca L. *Logic Programming and Databases*. Springer Verlag, Berlin 1990.

10. Chang C. L. and Lee R. C. T. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

11. Clark K. L. *Negation as Failure*, in H. Gallaire & J. Minker (eds) Logic and Databases. Plenum Press, NY, 1978 293-322.

12. Clark, K. L., *Predicate Logic as a Computational Formalism*, Research Report DOC 79/59. Department of Computing, Imperial College, 1979.

13. Clocksin W. F. and Mellish C. S. *Programming in Prolog*. Springer Verlag, Berlin 1994.

14. Codd E. F. *A Relational Model for Large Shared Data Banks*. Comm. ACM 13:6 1970 377-387.

15. Coelho H. and Cotta J. C. *Prolog by Example: How to learn, teach and use it*. Symbolic Computation, AI. Springer-Verlag, Berlin 1988.

16. Colmerauer A., Kanoui H., Roussel P. and Pasero R. *Un Systeme de Communication Homme-Machine en Français*, Groupe de Recherche en Intelligence Artificielle, Univerisite d'Aix-Marseille 1973.

17. Doyle J. *A Truth Maintenance System*, Artrificial Intelligence, 12:23 1-272 1979.

18. Elmasri R. and Navathe S. *Fundamentals of Database Systems* (2nd Ed.) The Benjamin/Cummings Publishing Company, Inc (1994).

19. Gallaire H. and Minker J. (eds.), *Logic and Databases*, Plenum Press, New York, 1978.

20. Garavaglia Susan. *Prolog: Programming Techniques and Applications*. Harper and Row Publishers, Inc 1987.

21. Gelfond M. and Lifshitz V. *The Stable Model Semantics for Logic Programming*. In R. A. Kowalski and K. A. Bowen, editors, Proc. 5th International Conference and Symposium on

Logic Programming pages 1070-1080, Seattle Washington, August 15-19 1988 (MIT Press 1988).

22. Gelfond M., Przymusinska H. and Przymusinski T. *The Extended Closed World Assumption and Its Relation to Parallel Circumscription*, Proceedings SIGACT-SIGMOD Symposium on Principles of Database Systems, Cambridge, MA 133-139, 1986.

23 Genesereth M. R. and Ginsberg M. L. *Logic Programming*. CACM 28(9): 933-941 1985.

24 Ghezzi C. and Jazayeri M. *Programming Language Concepts* (2nd Ed.) John Wiley: New York 1987.

25 Green C. *Applications of Theorem Proving Solving*, IJCAI- 69, Washington, 1969, 219-239.

26 Hayes P. J. *Computation and Deduction*, In Proceedings of the Second Symposium on Mathematical Foundation of Computer Science, Czechoslovkian Academy of Sciences, 1973, 289-300.

27 Henschen L. and Park H. *Compiling the GCWA In Indefinite Deductive Databases*, in: J Minker (ed.) Foundations of Deductive Databases and Logic Programming. Morgan Kauffman Publishers, Los Altos, CA, 395-439, 1987.

28 Hewitt C. E. *PLANNER: A Language for Proving Theorems in Robots*, First International Joint Conference on Artificial Intelligence, 1969, 295-301.

29. Horton J. D. and Spencer E. B. Clause Trees: *A Tool for Understanding and Implementing Resolution in Automated Reasoning* (submitted).

30. Knaster B. *Un Theorem sur les Functions d'ensmbles*. Annals Soc. Pol. Math, 6: 133-134, 1928.

31. Konigsberger H. and de Bruyn F. *PROLOG from the Beginning*. McGraw Hill 1990.

32. Kowalski R. A. *Predicate Logic as a Programming Language,* in: Proc IFIP '74. (North Holland, Amsterdam, 1974) 569-574.

33. Lifschitz V. *On the Declarative Semantics of Logic Programs with Negation,* In Foundations of Deductive Databases and Logic Programming. J. Minker (ed.) Morgan Kauffman Publishers, Los Altos, CA, 177-192, 1987.

34. Lloyd J. W. *Foundations of Logic Programming.* Springer-Verlag, Berlin 1984.

35. Lobo J., Minker J., Rajasekar A. *Foundations of Disjunctive Logic Programming.* The MIT Press Cambridge, Massachusetts, 1992.

36. Lobo J. *On Constructive Negation for Disjunctive Logic Programs, in:* Proceedings of the North American Conference on Logic Programming, Austin Texas, 1990, 704-718.

37. Loveland D. W. *Automated Theorem Proving: A Logical Basis.* North-Holland Publishing Co., 1978.

38. Malpas J. *PROLOG: A Relational Language and Its Applications.* Prentence Hall, Englewood Cliffs, NJ 1987.

39. McCarthy J. *Applications of Circumsciption to Formalizing Common Sense Knowledge.* AAAI Workshop on Nonmonotonic Reasoning, 1984 295-323.

40. McCarthy J. *Circumscription-a form of nonmonotonic reasoning.* Artificial Intelligence 13 **(1&2)** (1980) 27-39.

41. McCarthy J. *Epistemological Problems of Artificial Intelligence,* in: Proceedings of the International Joint Conference on Artificial Intelligence, 1977, 223-227.

42. McCarthy J. *Programs with Common Sense,* in: Mechanization of Thought Processes, Proceedings of the Symposium of the National Physics Laboratory, 1, London, 1958, 77-84 (Reprinted in Semantic Information Processing, MIT Press, Cambridge, MA, 1968, 403-418).

43. McDermott D. and Doyle J. *Non-monotonic Logic I,* Artificial Intelligence, 25:4172 1980.

44. Minker J. and Rajasekar *A. A Fixpoint for Disjunctive Logic Programs,* J. of Logic Programming, 9(1): 45-74 1990.

45. Minker J. ed. *Symposium on Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers Inc., Los Altos CA (1988).

46. Minker J. *On Indefinite Databases and The Closed World Assumption. Proc 6th* International Conference on Automated Deduction, Lecture Notes in Computer Science 138, Springer Verlag, 1982, 292-308.

47. Minker J. and Zanon G. An *Extension To Linear Resolution With Selection Function.* Information Processing Letters, 14(3): 191-194, June 1982.

48. Perlis D. (University of Maryland). *Circumscription* (Definition), In: Encyclopedia on AI.

49. Przymusinski T. C. *Stable semantics for disjunctive programs.* New Generation Computing 9 (3&4) 199 1 40 1-424.

50. Przymusinski T. C. *On the Declarative Semantics of Deductive Databases and Logic Programs,* In Foundations of Deductive Databases and Logic Programming. J. Minker (ed.) Morgan Kauffman Publishers, Los Altos, CA, 193-216, 1987.

51. Przymusinski T. C. *On the Semantics of Stratified Deductive Databases*. Workshop on Foundations of Deductive Databases and Logic Programming, J. Minker (ed.) Washington D.C., August 1986 433-443.

52. Przymusinski T. C. *Stable Semantics of Disjunctive Logic Programs and Deductive Databases*. Proc 2nd International Conference on Deductive and ObjectOriented Databases. Lecture Notes in Computer Science 566, Springer-Verlag, 1991 85-107.

53. Rajasekar A., Lobo J. and Minker J. *Weak Generalized Closed World Assumption*, J. Automated Reasoning 5 (1989), 293-307.

54. Reiter R. *On Closed World Databases*, In: H. Gallaire & J. Minker (eds) Logic and Databases. Plenum Press, NY, 1978 55-76.

55. Reiter R. *On Reasoning by Default Logic*, In: Proceedings of TINLAP-2, Theoretical Issues in Natural Language Proceeding-2, University of Illinois at Urbana Champaign, 1978, 210-218.

56. Reiter R. *A Logic for Default Reasoning*, Artificial Intelligence, 13(1 and 2): 81132 1980.

57. Robinson J. A. *A machine-oriented logic based on the resolution principle*. J.ACM, 12(1): 23-41, January, 1965.

58. Ross K. A. and Topor R. W. *Inferring Negative Information from Disjunctive Databases*, J. Automated Reasoning 4 (2) (1988), 397-424.

59. Sakama C. and Inoue K. *An alternative approach to the Semantics of Disjunctive logic Programs and Deductive Databases*. J Automated Reasoning 13 (1994), 145-172.

60. Schlipf J. S. *Formalizing a logic for Logic Programming*. Ann. Mathematics & Artificial Intelligence 5 (1992) 279-302.

61. Sosnowski R. A. *PROLOG Dialects: A déjà vu of BASIC-s.* ACM SIGPLAN Notices 22(6): 39-48 1987.

62. Spencer Bruce. *Avoiding Duplicate Proofs With The Foothold Refinement.* Annals of Mathematics and Artificial Intelligence, 12 117-140 1994.

63. Spencer Bruce. *Linear Resolution With Ordered Clauses*, in: Proc. Workshop on Disjunctive Logic Programming Int. Symposium on Logic Programming, San Diego, California, 1991.

64. Spencer Bruce. *Avoiding Duplicate Proofs*, in: Proc. North American Conference on Logic Programming, eds. S. K. Debray and M. Hermenogildo. MIT Press 569-584 1990.

65. Tarski A. *A Lattice-Theoeretical Fixpoint Theorem and Its Application.* Pacific J. Math., 5: 285-309, 1955.

66. Ullman J. D. *Principles of Database and Knowledge-Base Systems vol 1.* Computer Science Press, 1988.

67. Van Emden M. H. and Kowalski R. A. *The Semantics of Predicate logic as a Programming Language.* J. ACM 23 (4) 1976 733-742.

68. Weyrauch R. W. *Prologomena to a Theory of Mechanized Formal Reasoning*, Artificial Intelligence, 13(1/2): 133-170 1980.

69. Winograd T. *Extended Inference Modes In Reasoning by Computer Systems*, Artificial Intelligence, 13(1/2): 5-26 1980.

70. Wos L. Automated Reasoning: *33 Basic Research Problems.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

71. Yahya A. and Henschen L. *Deduction in Non-Horn in Databases*, Journal of Automated Reasoning 1(2): 141-160 (1985).

72. Shepherdson J. C. *Negation in Logic Programming*, in Foundations of Deductive Databases and Logic Programming (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 19-88.

Appendices

Appendix I

Rules of Inference

1. *Modus Ponens* (M.P.)

$p \supset q$
p
$\therefore q$

2. *Modus Tollens* (M.T.)

$p \supset q$
$\sim q$
$\therefore \sim p$

3. *Hypothetical Syllogism* (H.S.)

$p \supset q$
$q \supset r$
$\therefore p \supset r$

4. *Disjunctive Syllogism* (D.S.)

$p \vee q$
$\sim p$
$\therefore q$

5. *Constructive Dilemma* (C.D.)

$(p \supset q) \cdot (r \supset s)$
$p \vee r$
$\therefore q \vee s$

6. *Absorption* (Abs.)

$p \supset q$
$\therefore p \supset (p \cdot q)$

7. *Simplification* (Simp.)

$p \cdot q$
$\therefore p$

8. *Conjunction* (Conj.)

p
q
$\therefore p \cdot q$

9. *Addition* (Add)

p
$\therefore p \vee q$

Rules of Replacement

Any of the following logically equivalent expressions can replace each other wherever they occur:

10. De Morgan's Theorems (De M.):

$$\sim(p \cdot q) \equiv (\sim p \vee \sim q)$$
$$\sim(p \vee q) \equiv (\sim p \cdot \sim q)$$

11. Commutation (Com.)

$$(p \vee q) \equiv (q \vee p)$$
$$(p \cdot q) \equiv (q \cdot p)$$

12. Association (Assoc.):

$$[p \vee (q \vee r)] \equiv [(p \vee q) \vee r]$$
$$[p \cdot (q \cdot r)] \equiv [(p \cdot q) \cdot r]$$

13. Distribution (Dist.):

$$[p \cdot (q \vee r)] \equiv [(p \cdot q) \vee (p \cdot r)]$$
$$[p \vee (q \cdot r)] \equiv [(p \vee q) \cdot (p \vee r)]$$

14. Double Negation (D.N.):

$$p \equiv \sim\sim p$$

15. Transposition (Trans.)

$$(p \supset q) \equiv (\sim q \supset \sim p)$$

16. Material Implication (Impl.):

$$(p \supset q) \equiv (\sim p \supset q)$$

17. Material Equivalence (Equiv.):

$$(p \equiv q) \equiv [(p \supset q) \cdot (q \supset p)]$$
$$(p \equiv q) \equiv [(p \cdot q) \vee (\sim p \cdot \sim q)]$$

18. Exportation (Exp.):

$$[(p \cdot q) \supset r] \equiv [p \supset (q \supset r)]$$

19. Tautology (Taut.)

$$p \equiv p \vee p$$
$$p \equiv p \cdot p$$

Appendix II

Logical Deduction and the Resolution Principal

Suppose we already know that either John is at school or John is sick. If we are then told that John is not sick, we could conclude that John is at school. This is an example of a deductive-reasoning principle called resolution. To better understand this principle, let us first agree to represent statements by single letters and the negation of a statement by preceding the letter representing the statement with the symbol $\neg$. For instance, we might represent the statement "John is at school" by P, the statement "John is sick" by Q, and the statement "John is not sick" by $\neg Q$. Then, reasoning described above could be summarized as:

In a more general form, the ***resolution*** principle says that if P, Q and R are statements, then:

$$P \text{ or } Q$$

and

$$R \text{ or } \neg Q$$

collectively imply the statement:

$$P \text{ or } R$$

as represented in *Figure A2-1*. In this case, we say that the two original statements resolve to produce the third statement, which is called the ***resolvent***.

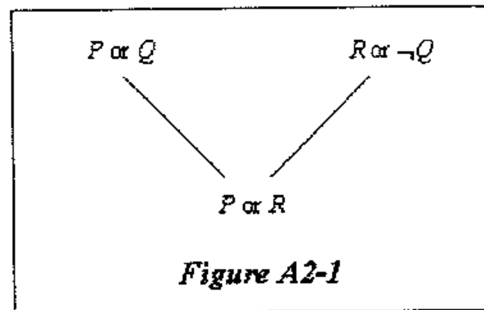


Figure A2-1

It is important that resolution can only be applied to pairs of statements that are written as disjunctive clauses - that is, statements whose elementary components are connected by the word *or* (see p 13), for example $P \vee Q \vee R$. A statement of the form *if P then Q* is equivalent to $Q \vee \neg P$ in this form.

Of course, resolution is only one of many inference rules used by logicians. Another rule is called *conjunction* (or *and introduction* [see *Appendix I*]). It says that from a statement P and another statement Q one can derive the statement $P \wedge Q$.

Still another inference rule is *modus ponens*. It says that from statements *if* P *then* Q and P, one can conclude the statement Q.

There is, however, something special about resolution. By using resolution, and only resolution, we can confirm the inconsistency of a collection of contradictory clause form statements. More precisely, a collection of clauses is contradictory if and only if repeated applications of resolution can ultimately lead to an empty resolvent (the result of applying resolution to two clauses of the form P and $\neg P$). For example, *Figure A2-2* indicates how this could arise in the case of the initial clauses

$$P \vee Q \qquad R \vee \neg Q \qquad \neg R \qquad \neg P$$

To appreciate the significance of this, suppose a collection of statements implies some other statement P and we wish to demonstrate this implication. One approach would be to show that the original collection is inconsistent with the statement $\neg P$. Indeed, implying the statement P is the same as contradicting the statement $\neg P$. Thus, to demonstrate that the original collection of statements implies P, we do not need to apply several different inference rules. All we need to do is express the resolution until an empty resolvent occurs. This later approach is much more conducive to automation. A program for proving consequences of collections of statements does not need to decide which rule, among many inference rules, to apply. It needs merely to apply resolution over and over again.

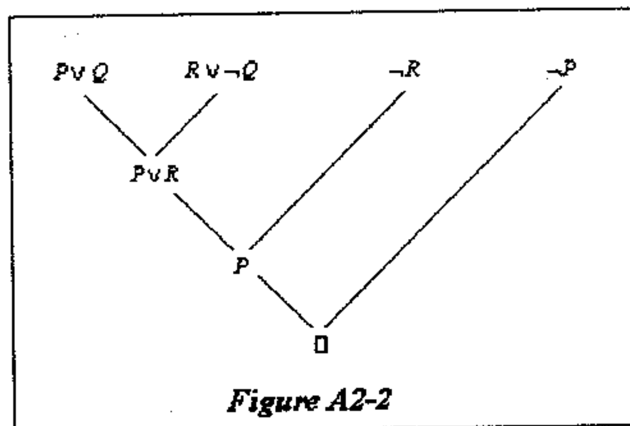


Figure A2-2

One final point remains to be discussed before we are ready to apply resolution in an actual programming environment. Suppose we have the two statements

if Mary is at X then Mary's lamb is at X too

and

Mary is at home

where *X* is intended to represent any location. As disjunctive clauses the two statements become

(*Mary's lamb is at home*) $\vee$ $\neg$(*Mary is at home*)

which can be resolved with the statement

(*Mary is at home*)

to produce the statement

(*Mary's lamb is at home*).

The process of assigning values to variables (such as assigning the value *home* to *X*) so that resolution can be performed is called *unification* (or *instantiation*). It is this process that allows general statements to be applied to specific applications in a deductive system. It also provides a technique for extracting information from the system. For example, it is by means of unification that we are able to conclude that *Bill* is a grandparent of *Tom* from the statements

Bill is a parent of Joan.
Joan is a parent of Tom.
If Y is a parent of X and Z is a parent of Y,
then Z is a grandparent of X.

Appendix III

```
sort(L, S) :- permutation(L, S), ordered(S).

permutation([], []).
permutation(Q, [X|T]) :- append(A, [X|B], Q),
                         append(A, B, P),
                         permutation(P, T).

ordered([]).
ordered([_|[]]).
ordered([A|[B|T]]) :- A < B, ordered([B|T]).

append([], Z, Z).
append([X|L], M, [X|N]) :- append(L, M, N).
```

Figure A3-1 A Prolog program for sorting a list of numbers