J. W. Lloyd

Foundations of Logic Programming

Second, Extended Edition

Springer-Verlag
Berlin Heidelberg New York
[illegible]

Chapter 1

PRELIMINARIES

This chapter presents the basic concepts and results which are needed for the theoretical foundations of logic programming. After a brief introduction to logic programming, we discuss first order theories, interpretations and models, unification, and fixpoints.

§1. INTRODUCTION

Logic programming began in the early 1970's as a direct outgrowth of earlier work in automatic theorem proving and artificial intelligence. Constructing automated deduction systems is, of course, central to the aim of achieving artificial intelligence. Building on work of Herbrand [44] in 1930, there was much activity in theorem proving in the early 1960's by Prawitz [84], Gilmore [39], Davis, Putnam [26] and others. This effort culminated in 1965 with the publication of the landmark paper by Robinson [88], which introduced the resolution rule. Resolution is an inference rule which is particularly well-suited to automation on a computer.

The credit for the introduction of logic programming goes mainly to Kowalski [48] and Colmerauer [22], although Green [40] and Hayes [43] should be mentioned in this regard. In 1972, Kowalski and Colmerauer were led to the fundamental idea that *logic can be used as a programming language.* The acronym PROLOG (PROgramming in LOGic) was conceived, and the first PROLOG interpreter [22] was implemented in the language ALGOL-W by Roussel, at Marseille in 1972. ([8] and [89] describe the improved and more influential version written in FORTRAN.) The PLANNER system of Hewitt [45] can be regarded as a predecessor of PROLOG.

The idea that first order logic, or at least substantial subsets of it, could be used as a programming language was revolutionary, because, until 1972, logic had only ever been used as a specification or declarative language in computer science. However, what [48] shows is that logic has a *procedural interpretation*, which makes it very effective as a programming language. Briefly, a program clause $A \leftarrow B_1,\ldots,B_n$ is regarded as a *procedure definition*. If $\leftarrow C_1,\ldots,C_k$ is a goal, then each C_j is regarded as a *procedure call*. A program is run by giving it an initial goal. If the current goal is $\leftarrow C_1,\ldots,C_k$, a step in the computation involves unifying some C_j with the head A of a program clause $A \leftarrow B_1,\ldots,B_n$ and thus reducing the current goal to the goal $\leftarrow (C_1,\ldots,C_{j-1},B_1,\ldots,B_n,C_{j+1},\ldots,C_k)\theta$, where θ is the unifying substitution. Unification thus becomes a uniform mechanism for parameter passing, data selection and data construction. The computation terminates when the empty goal is produced.

One of the main ideas of logic programming, which is due to Kowalski [49], [50], is that an algorithm consists of two disjoint components, the logic and the control. The logic is the statement of *what* the problem is that has to be solved. The control is the statement of *how* it is to be solved. Generally speaking, a logic programming system should provide ways for the programmer to specify each of these components. However, separating these two components brings a number of benefits, not least of which is the possibility of the programmer only having to specify the logic component of an algorithm and leaving the control to be exercised solely by the logic programming system itself. In other words, an ideal of logic programming is purely declarative programming. Unfortunately, this has not yet been achieved with current logic programming systems.

Most current logic programming systems are resolution theorem provers. However, logic programming systems need not necessarily be based on resolution. They can be non-clausal systems with many inference rules [11], [41], [42]. This account only discusses logic programming systems based on resolution and concentrates particularly on the PROLOG systems which are currently available.

There are two major, and rather different, classes of logic programming languages currently available. The first we shall call "system" languages and the second "application" languages. These terms are not meant to be precise, but only to capture the flavour of the two classes of languages.

For "system" languages, the emphasis is on AND-parallelism, don't-care non-determinism and definite programs (that is, no negation). In these languages, according to the *process interpretation* of logic, a goal $\leftarrow B_1,\ldots,B_n$ is regarded as a system of concurrent processes. A step in the computation is the reduction of a process to a system of processes (the ones that occur in the body of the clause that matched the call). Shared variables act as communication channels between processes. There are now several "system" languages available, including PARLOG [18], concurrent PROLOG [93] and GHC [106]. These languages are mainly intended for operating system applications and object-oriented programming [94]. For these languages, the control is still very much given by the programmer. Also these languages are widely regarded as being closer to the machine level.

"Application" languages can be regarded as general-purpose programming languages with a wide range of applications. Here the emphasis is on OR-parallelism, don't-know non-determinism and (unrestricted) programs (that is, the body of a program statement is an arbitrary formula). Languages in this class include Quintus PROLOG [10], micro-PROLOG [20] and NU-PROLOG [104]. For these languages, the automation of the control component for certain kinds of applications has already largely been achieved. However, there are still many problems to be solved before these languages will be able to support a sufficiently declarative style of programming over a wide range of applications.

"Application" languages are better suited to deductive database systems and expert systems. According to the *database interpretation* of logic, a logic program is regarded as a database [35], [36], [37], [38]. We thus obtain a very natural and powerful generalisation of relational databases. The latter correspond to logic programs consisting solely of ground unit clauses. The concept of logic as a uniform language for data, programs, queries, views and integrity constraints has great theoretical and practical power.

The distinction between these two classes of languages is, of course, by no means clearcut. For example, non-trivial problem-solving applications have been implemented in GHC. Also, the coroutining facilities of NU-PROLOG make it suitable as a system programming language. Nevertheless, it is useful to make the distinction. It also helps to clarify some of the debates in logic programming, whose source can be traced back to the "application" versus "system" views of the participants.

The emergence of these two kinds of logic programming languages has complicated the already substantial task of building parallel logic machines. Because of the differing hardware requirements of the two classes of languages, it seems that a difficult choice has to be made. This choice is between building a predominantly AND-parallel machine to directly support a "system" programming language or building a predominantly OR-parallel machine to directly support an "application" programming language.

There is currently substantial effort being invested in the first approach; certainly, the Japanese fifth generation project [71] is headed this way. The advantage of this approach is that the hardware requirements for an AND-parallel language, such as GHC, seem less demanding than those required for an OR-parallel language. However, the success of a logic machine ultimately rests on the power and expressiveness of its application languages. Thus this approach requires some method of compiling the application languages into the lower level system language.

In summary, logic provides a single formalism for apparently diverse parts of computer science. It provides us with general-purpose, problem-solving languages, concurrent languages suitable for operating systems and also a foundation for deductive database systems and expert systems. This range of application together with the simplicity, elegance and unifying effect of logic programming assures it of an important and influential future. Logical inference is about to become the fundamental unit of computation.

§2. FIRST ORDER THEORIES

This section introduces the syntax of well-formed formulas of a first order theory. While all the requisite concepts from first order logic will be discussed informally in this and subsequent sections, it would be helpful for the reader to have some wider background on logic. We suggest reading the first few chapters of [14], [33], [64], [69] or [99].

First order logic has two aspects: syntax and semantics. The syntactic aspect is concerned with well-formed formulas 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 well-formed formulas and the symbols they contain.

We postpone the discussion of semantics to the next section.

A *first order theory* consists of an alphabet, a first order language, a set of axioms and a set of inference rules [69], [99]. The first order language consists of the well-formed formulas of the theory. The axioms are a designated subset of well-formed formulas. The axioms and rules of inference are used to derive the theorems of the theory. We now proceed to define alphabets and first order languages.

Definition An *alphabet* consists of seven classes of symbols:
(a) variables
(b) constants
(c) function symbols
(d) predicate symbols
(e) connectives
(f) quantifiers
(g) punctuation symbols.

Classes (e) to (g) are the same for every alphabet, while classes (a) to (d) vary from alphabet to alphabet. For any alphabet, only classes (b) and (c) may be empty. We adopt some informal notational conventions for these classes. Variables will normally be denoted by the letters u, v, w, x, y and z (possibly subscripted). Constants will normally be denoted by the letters a, b and c (possibly subscripted). Function symbols of various arities > 0 will normally be denoted by the letters f, g and h (possibly subscripted). Predicate symbols of various arities ≥ 0 will normally be denoted by the letters p, q and r (possibly subscripted). Occasionally, it will be convenient not to apply these conventions too rigorously. In such a case, possible confusion will be avoided by the context. The connectives are $\sim$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$, while the quantifiers are $\exists$ and $\forall$. Finally, the punctuation symbols are "(", ")" and ",". To avoid having formulas cluttered with brackets, we adopt the following precedence hierarchy, with the highest precedence at the top:

$$\sim, \forall, \exists$$
$$\vee$$
$$\wedge$$
$$\rightarrow, \leftrightarrow$$

Next we turn to the definition of the first order language given by an alphabet.

Definition A *term* is defined inductively as follows:
(a) A variable is a term.
(b) A constant is a term.
(c) If f is an n-ary function symbol and $t_1,...,t_n$ are terms, then $f(t_1,...,t_n)$ is a term.

Definition A (*well-formed*) *formula* is defined inductively as follows:
(a) 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, more simply, an *atom*).
(b) If F and G are formulas, then so are $(\sim F)$, $(F\wedge G)$, $(F\vee G)$, $(F\rightarrow G)$ and $(F\leftrightarrow G)$.
(c) If F is a formula and x is a variable, then $(\forall x\ F)$ and $(\exists x\ F)$ are formulas.

It will often be convenient to write the formula $(F\rightarrow G)$ as $(G\leftarrow F)$.

Definition The *first order language* given by an alphabet consists of the set of all formulas constructed from the symbols of the alphabet.

Example $(\forall x\ (\exists y\ (p(x,y)\rightarrow q(x))))$, $(\sim(\exists x\ (p(x,a)\wedge q(f(x)))))$ and $(\forall x\ (p(x,g(x))\leftarrow(q(x)\wedge(\sim r(x)))))$ are formulas. By dropping pairs of brackets when no confusion is possible and using the above precedence convention, we can write these formulas more simply as $\forall x\exists y\ (p(x,y)\rightarrow q(x))$, $\sim\exists x\ (p(x,a)\wedge q(f(x)))$ and $\forall x\ (p(x,g(x))\leftarrow q(x)\wedge\sim r(x))$. We will simplify formulas in this way wherever possible.

The informal semantics of the quantifiers and connectives is as follows. $\sim$ is negation, $\wedge$ is conjunction (and), $\vee$ is disjunction (or), $\rightarrow$ is implication and $\leftrightarrow$ is equivalence. Also, $\exists$ is the existential quantifier, so that "$\exists x$" means "there exists an x", while $\forall$ is the universal quantifier, so that "$\forall x$" means "for all x". Thus the informal semantics of $\forall x\ (p(x,g(x)) \leftarrow q(x)\wedge\sim r(x))$ is "for every x, if q(x) is true and r(x) is false, then p(x,g(x)) is true".

Definition The *scope* of $\forall x$ (resp. $\exists x$) in $\forall x\ F$ (resp. $\exists x\ F$) is F. A *bound occurrence* of a variable in a formula is an occurrence immediately following a quantifier or an occurrence within the scope of a quantifier, which has the same variable immediately after the quantifier. Any other occurrence of a variable is *free*.

Example In the formula $\exists x\ p(x,y)\wedge q(x)$, the first two occurrences of x are bound, while the third occurrence is free, since the scope of $\exists x$ is p(x,y). In $\exists x\ (p(x,y)\wedge q(x))$, all occurrences of x are bound, since the scope of $\exists x$ is $p(x,y)\wedge q(x)$.

Definition A *closed formula* is a formula with no free occurrences of any variable.

Example $\forall y\exists x\ (p(x,y)\wedge q(x))$ is closed. However, $\exists x\ (p(x,y)\wedge q(x))$ is not closed, since there is a free occurrence of the variable y.

Definition If F is a formula, then $\forall(F)$ denotes the *universal closure* of F, which is the closed formula obtained by adding a universal quantifier for every variable having a free occurrence in F. Similarly, $\exists(F)$ denotes the *existential closure* of F, which is obtained by adding an existential quantifier for every variable having a free occurrence in F.

Example 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 $\exists x\exists y\ (p(x,y)\wedge q(x))$.

In chapters 4 and 5, it will be useful to have available the concept of an atom occurring positively or negatively in a formula.

Definition An atom A *occurs positively* in A.
If atom A occurs positively (resp., negatively) in a formula W, then A *occurs positively* (resp., *negatively*) in $\exists x\ W$ and $\forall x\ W$ and $W\wedge V$ and $W\vee V$ and $W\leftarrow V$.
If atom A occurs positively (resp., negatively) in a formula W, then A *occurs negatively* (resp., *positively*) in $\sim W$ and $V\leftarrow W$.

Next we introduce an important class of formulas called clauses.

Definition A *literal* is an atom or the negation of an atom. A *positive literal* is an atom. A *negative literal* is the negation of an atom.

Definition A *clause* is a formula of the form

$$\forall x_1...\forall x_s\ (L_1\vee...\vee L_m)$$

where each L_i is a literal and $x_1,...,x_s$ are all the variables occurring in $L_1\vee...\vee L_m$.

Example The following are clauses

$$\forall x\forall y\forall z\ (p(x,z)\vee\sim q(x,y)\vee\sim r(y,z))$$
$$\forall x\forall y\ (\sim p(x,y)\vee r(f(x,y),a))$$

Because clauses are so common in logic programming, it will be convenient to adopt a special clausal notation. Throughout, we will denote the clause

$$\forall x_1 \ldots \forall x_s \, (A_1 \vee \ldots \vee A_k \vee \sim B_1 \vee \ldots \vee \sim B_n)$$

where $A_1,\ldots,A_k,B_1,\ldots,B_n$ are atoms and $x_1,\ldots,x_s$ are all the variables occurring in these atoms, by

$$A_1,\ldots,A_k \leftarrow B_1,\ldots,B_n$$

Thus, in the clausal notation, all variables are assumed to be universally quantified, the commas in the antecedent $B_1,\ldots,B_n$ denote conjunction and the commas in the consequent $A_1,\ldots,A_k$ denote disjunction. These conventions are justified because

$$\forall x_1 \ldots \forall x_s \, (A_1 \vee \ldots \vee A_k \vee \sim B_1 \vee \ldots \vee \sim B_n)$$

is equivalent to

$$\forall x_1 \ldots \forall x_s \, (A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n)$$

To illustrate the application of the various concepts in this chapter to logic programming, we now define definite programs and definite goals.

Definition A *definite program clause* is a clause of the form

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

which contains precisely one atom (viz. A) in its consequent. A is called the *head* and $B_1,\ldots,B_n$ is called the *body* of the program clause.

Definition A *unit clause* is a clause of the form

$$A \leftarrow$$

that is, a definite program clause with an empty body.

The informal semantics of $A \leftarrow B_1,\ldots,B_n$ is "for each assignment of each variable, if $B_1,\ldots,B_n$ are all true, then A is true". Thus, if n>0, a program clause is conditional. On the other hand, a unit clause $A\leftarrow$ is unconditional. Its informal semantics is "for each assignment of each variable, A is true".

Definition A *definite program* is a finite set of definite program clauses.

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

Example The following program, called slowsort, sorts a list of non-negative integers into a list in which the elements are in increasing order. It is a very inefficient sorting program! However, we will find it most useful for illustrating various aspects of the theory.

In this program, non-negative integers are represented using a constant 0 and a unary function symbol f. The intended meaning of 0 is zero and f is the successor function. We define the powers of f by induction: $f^0(x)=0$ and $f^{n+1}(x)=f(f^n(x))$. Then the non-negative integer n is represented by the term $f^n(0)$. In fact, it will sometimes be convenient simply to denote $f^n(0)$ by n.

Lists are represented using a binary function symbol "." (the cons function written infix) and the constant nil representing the empty list. Thus the list [17, 22, 6, 5] would be represented by 17.(22.(6.(5.nil))). We make the usual right associativity convention and write this more simply as 17.22.6.5.nil.

```
SLOWSORT PROGRAM
sort(x,y) ← sorted(y), perm(x,y)
sorted(nil) ←
sorted(x.nil) ←
sorted(x.y.z) ← x≤y, sorted(y.z)
perm(nil,nil) ←
perm(x.y,u.v) ← delete(u,x.y,z), perm(z,v)
delete(x,x.y,y) ←
delete(x,y.z,y.w) ← delete(x,z,w)
0≤x ←
f(x)≤f(y) ← x≤y
```

Slowsort contains definitions of five predicate symbols, sort, sorted, perm, delete and ≤ (written infix). The informal semantics of the definition of sort is "if x and y are lists, y is a permutation of x and y is sorted, then y is the sorted version of x". This is clearly a correct top-level description of a sorting program. Similarly, the first clause in the definition of sorted states that "the empty list is sorted". The intended meaning of the predicate symbol delete is that delete(x,y,z) should hold if z is the list obtained by deleting the element x from the list y. The above definition for delete contains obviously correct statements about the delete predicate.

Definition A *definite goal* is a clause of the form

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

that is, a clause which has an empty consequent. Each B_i (i=1,...,n) is called a *subgoal* of the goal.

If $y_1,\ldots,y_r$ are the variables of the goal

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

then this clausal notation is shorthand for

$$\forall y_1 \ldots \forall y_r\ (\sim B_1 \vee \ldots \vee \sim B_n)$$

or, equivalently,

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

Example To run slowsort, we give it a goal such as

$$\leftarrow \text{sort}(17.22.6.5.\text{nil},y)$$

This is understood as a request to find the list y, which is the sorted version of 17.22.6.5.nil.

Definition The *empty clause*, denoted □, is the clause with empty consequent and empty antecedent. This clause is to be understood as a contradiction.

Definition A *Horn clause* is a clause which is either a definite program clause or a definite goal.

§3. INTERPRETATIONS AND MODELS

The declarative semantics of a logic program is given by the usual (model-theoretic) semantics of formulas in first order logic. This section discusses interpretations and models, concentrating particularly on the important class of Herbrand interpretations.

Before we give the main definitions, some motivation is appropriate. In order to be able to discuss the truth or falsity of a formula, it is necessary to attach some meaning to each of the symbols in the formula first. The various quantifiers and connectives have fixed meanings, but the meanings attached to the constants, function symbols and predicate symbols can vary. An interpretation simply consists of some domain of discourse over which the variables range, the assignment to each constant of an element of the domain, the assignment to each function symbol of a mapping on the domain and the assignment to each predicate symbol of a relation on the domain. An 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. Normally there is some distinguished interpretation, called the *intended* interpretation, which gives the principal meaning of the symbols. Naturally, the intended interpretation of a

formula should be a model of the formula.

First order logic provides methods for deducing the theorems of a theory. These can be characterised (by Gödel's completeness theorem [69], [99]) as the formulas which are logical consequences of the axioms of the theory, that is, they are true in every interpretation which is a model of each of the axioms of the theory. In particular, each theorem is true in the intended interpretation of the theory. 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 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 this 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 a theorem proving point of view, the only interest is to demonstrate logical consequence. However, from a programming point of view, we are much more interested in the bindings that are made for the variables $y_1,\ldots,y_r$, because these give us the *output* from the running of the program. In fact, the ideal view of a logic programming system is that it is a black box for computing bindings and our only interest is in its input-output behaviour. The internal workings of the system should be invisible to the programmer. Unfortunately, this situation is not true, to various extents, with current PROLOG systems. Many programs can only be understood in a procedural (i.e. operational) manner, because of the way they use cuts and other non-logical features.

Returning to the slowsort program, from a theorem proving point of view, we can regard the goal $\leftarrow$sort(17.22.6.5.nil,y) as a request to prove that $\exists y$ sort(17.22.6.5.nil,y) is a logical consequence of the program. In fact, we are much more interested that the proof is constructive and provides us with a specific y which makes sort(17.22.6.5.nil,y) true in the intended interpretation.

We now give the definitions of pre-interpretation, interpretation and model.

Definition A *pre-interpretation* of a first order language L consists of the following:

(a) A non-empty set D, called the *domain* of the pre-interpretation.
(b) For each constant in L, the assignment of an element in D.
(c) For each n-ary function symbol in L, the assignment of a mapping from D^n to D.

Definition An *interpretation* I of a first order language L consists of a pre-interpretation J with domain D of L together with the following:
For each n-ary predicate symbol in L, the assignment of a mapping from D^n into {true, false} (or, equivalently, a relation on D^n).

We say I is *based on* J.

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

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

(a) Each variable is given its assignment according to V.
(b) Each constant is given its assignment according to J.
(c) If $t'_1,\dots,t'_n$ are the term assignments of $t_1,\dots,t_n$ and f' is the assignment of the n-ary function symbol f, then $f'(t'_1,\dots,t'_n) \in D$ is the term assignment of $f(t_1,\dots,t_n)$.

[illegible]

Definition Let J be a pre-interpretation of a first order language L, V a variable assignment wrt J, and A an atom. Suppose A is $p(t_1,\dots,t_n)$ and $d_1,\dots,d_n$ in the domain of J are the term assignments of $t_1,\dots,t_n$ wrt J and V. We call $A_{J,V} = p(d_1,\dots,d_n)$ the *J-instance of* A *wrt* V. Let $[A]_J = \{ A_{J,V} : V$ is a variable assignment wrt J $\}$. We call each element of $[A]_J$ a *J-instance of* A. We also call each $p(d_1,\dots,d_n)$ a *J-instance*.

(p) ?

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

(a) If the formula is an atom $p(t_1,\dots,t_n)$, then the truth value is obtained by calculating the value of $p'(t'_1,\dots,t'_n)$, where p' is the mapping assigned to p by I and $t'_1,\dots,t'_n$ are the term assignments of $t_1,\dots,t_n$ wrt I and V.

(b) If the formula has the form ~F, F∧G, F∨G, F→G or F↔G, then the truth value of the formula is given by the following table:

F	G	~F	F∧G	F∨G	F→G	F↔G
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

(c) If the formula has the form ∃x F, then the truth value of the formula is true if there exists d∈D such that F has truth value true wrt I and V(x/d), where V(x/d) is V except that x is assigned d; otherwise, its truth value is false.

(d) If the formula has the form ∀x F, then the truth value of the formula is true if, for all d∈D, we have that F has truth value true wrt I and V(x/d); otherwise, its truth value is false.

Clearly the truth value of a closed formula does not depend on the variable assignment. Consequently, we can speak unambiguously of the truth value of a closed formula wrt to an interpretation. If the truth value of a closed formula wrt to an interpretation is true (resp., false), we say the formula is true (resp,. false) wrt to the interpretation.

Definition Let I be an interpretation for a first order language L and let W be a formula in L.

We say W is *satisfiable in* I if ∃(W) is true wrt I.
We say W is *valid in* I if ∀(W) is true wrt I.
We say W is *unsatisfiable in* I if ∃(W) is false wrt I.
We say W is *nonvalid in* I if ∀(W) is false wrt I.

Definition Let I be an interpretation of a first order language L and let F be a closed formula of L. Then I is a *model* for F if F is true wrt I.

Example Consider the formula ∀x∃y p(x,y) and the following interpretation I. Let the domain D be the non-negative integers and let p be assigned the relation <. Then I is a model of the formula, as is easily seen. In I, the formula 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, I is not a model of the formula $\exists y \forall x\ p(x,y)$.

The axioms of a first order theory are a designated subset of closed formulas in the language of the theory. For example, the first order theories in which we are most interested have the clauses of a program as their axioms.

Definition Let T be a first order theory and let L be the language of T. A *model* for T is an interpretation for L which is a model for each axiom of T. If T has a model, we say T is *consistent*.

The concept of a model of a closed formula can easily be extended to a model of a set of closed formulas.

Definition Let S be a set of closed formulas of a first order language L and let I be an interpretation of L. We say I is a *model* for S if I is a model for each formula of S.

Note that, if $S = \{F_1,...,F_n\}$ is a finite set of closed formulas, then I is a model for S iff I is a model for $F_1 \wedge ... \wedge F_n$.

Definition Let S be a set of closed formulas of a first order language L.
We say S is *satisfiable* if L has an interpretation which is a model for S.
We say S is *valid* if every interpretation of L is a model for S.
We say S is *unsatisfiable* if no interpretation of L is a model for S.
We say S is *nonvalid* if L has an interpretation which is not a model for S.

Now we can give the definition of the important concept of logical consequence.

Definition Let S be a set of closed formulas and F be a closed formula of a first order language L. We say F is a *logical consequence* of S if, for every interpretation I of L, I is a model for S implies that I is a model for F.

Note that if $S = \{F_1,...,F_n\}$ is a finite set of closed formulas, then F is a logical consequence of S iff $F_1 \wedge ... \wedge F_n \rightarrow F$ is valid.

Proposition 3.1 Let S be a set of closed formulas and F be a closed formula of a first order language L. Then F is a logical consequence of S iff $S \cup \{\sim F\}$ is unsatisfiable.

Proof Suppose that F is a logical consequence of S. Let I be an interpretation of L and suppose I is a model for S. Then I is also a model for F. Hence I is not a model for $S \cup \{\sim F\}$. Thus $S \cup \{\sim F\}$ is unsatisfiable.

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

Example 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. Since $\forall x(p(x) \rightarrow q(x))$ is true wrt I, so is $p(a) \rightarrow q(a)$. Hence q(a) is true wrt I.

Applying these definitions to programs, we see that when we give a goal G to the system, with program P loaded, we are asking the system to show that the set of clauses $P \cup \{G\}$ is unsatisfiable. In fact, if G is the goal $\leftarrow B_1,...,B_n$ with variables $y_1,...,y_r$, then proposition 3.1 states that showing $P \cup \{G\}$ unsatisfiable is exactly the same as showing that $\exists y_1 ... \exists y_r\ (B_1 \wedge ... \wedge B_n)$ is a logical consequence of P.

$\forall y_1 \ldots y_r (\sim B_1 \vee \ldots$

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 Herbrand interpretations, which we now proceed to study.

Definition A *ground term* is a term not containing variables. Similarly, a *ground atom* is an atom not containing variables.

Definition Let L be a first order language. The *Herbrand universe* U_L for L is the set of all ground terms, which can be formed out of the constants and function symbols appearing in L. (In the case that L has no constants, we add some constant, say, a, to form ground terms.)

Example Consider the program
$p(x) \leftarrow q(f(x),g(x))$
$r(y) \leftarrow$
which has an underlying first order language L based on the predicate symbols p, q and r and the function symbols f and g. Then the Herbrand universe for L is

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

Definition Let L be a first order language. The *Herbrand base* B_L for L is the set of all ground atoms which can be formed by using predicate symbols from L with ground terms from the Herbrand universe as arguments.

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

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

Definition Let L be a first order language. The *Herbrand pre-interpretation* for L is the pre-interpretation given by the following:

(a) The domain of the pre-interpretation is the Herbrand universe U_L.

(b) Constants in L are assigned themselves in U_L.

(c) If f is an n-ary function symbol in L, then the mapping from $(U_L)^n$ into U_L defined by $(t_1,...,t_n) \rightarrow f(t_1,...,t_n)$ is assigned to f.

An *Herbrand interpretation* for L is any interpretation based on the Herbrand pre-interpretation for L.

Since, for Herbrand interpretations, the assignment to constants and function symbols is fixed, it is possible to identify an Herbrand interpretation with a subset of the Herbrand base. For any Herbrand interpretation, the corresponding subset of the Herbrand base is the set of all ground atoms which are true wrt the interpretation. Conversely, given an arbitrary subset of the Herbrand base, there is a corresponding Herbrand interpretation defined by specifying that the mapping assigned to a predicate symbol maps some arguments to "true" precisely when the atom made up of the predicate symbol with the same arguments is in the given subset. This identification of an Herbrand interpretation as a subset of the Herbrand base will be made throughout. More generally, each interpretation based on an arbitrary pre-interpretation J can be identified with a subset of J-instances, in a similar way.

Definition Let L be a first order language and S a set of closed formulas of L. An *Herbrand model* for S is an Herbrand interpretation for L which is a model for S.

It will often be convenient to refer, by abuse of language, to an interpretation of a set S of formulas rather than the underlying first order language from which the formulas come. Normally, we assume that the underlying first order language is defined by the constants, function symbols and predicate symbols appearing in

S. With this understanding, we can now refer to the Herbrand universe U_S and Herbrand base B_S of S and also refer to Herbrand interpretations of S as subsets of the Herbrand base of S. In particular, the set of formulas will often be a program P, so that we will refer to the Herbrand universe U_P and Herbrand base B_P of P.

Example We now illustrate these concepts with the slowsort program. This program can be regarded as the set of axioms of a first order theory. The language of this theory is given by the constants 0 and nil, function symbols f and "." and predicate symbols sort, perm, sorted, delete and ≤. The only inference rule is the resolution rule. The intended interpretation is an Herbrand interpretation. An atom sort(l,m) is in the intended interpretation iff each of l and m is either nil or is a list of terms of the form $f^k(0)$ and m is the sorted version of l. The other predicate symbols have the obvious assignments. The intended interpretation is indeed a model for the program and hence a model for the associated theory.

Next we show that in order to prove unsatisfiability of a set of clauses, it suffices to consider only Herbrand interpretations.

Proposition 3.2 Let S be a set of clauses and suppose S has a model. Then S has an Herbrand model.

Proof Let I be an interpretation of S. We define an Herbrand interpretation I' of S as follows:

$$I' = \{p(t_1,...,t_n) \in B_S : p(t_1,...,t_n) \text{ is true wrt } I\}.$$

It is straightforward to show that if I is a model, then I' is also a model. ∎

Proposition 3.3 Let S be a set of clauses. Then S is unsatisfiable iff S has no Herbrand models.

Proof If S is satisfiable, then proposition 3.2 shows that it has an Herbrand model. ∎

It is important to understand that neither proposition 3.2 nor 3.3 holds if we drop the restriction that S be a set of *clauses*. In other words, if S is a set of *arbitrary* closed formulas, it is not generally possible to show S is unsatisfiable by restricting attention to Herbrand interpretations.

Example Let S be $\{p(a), \exists x\, \sim p(x)\}$. Note that the second formula in S is not a clause. We claim that S has a model. It suffices to let D be the set {0, 1}, assign 0 to a and assign to p the mapping which maps 0 to true and 1 to false. Clearly this gives a model for S.

However, S does not have an Herbrand model. The only Herbrand interpretations for S are ∅ (the empty set) and $\{p(a)\}$. But neither of these is a model for S.

The point is worth emphasising. Much of the theory of logic programming is concerned only with clauses and for this Herbrand interpretations suffice. However, non-clausal formulas do arise naturally (particularly in chapters 3, 4 and 5). For this part of the theory, we will be forced to consider arbitrary interpretations.

There are various normal forms for formulas. One, which we will find useful, is prenex conjunctive normal form.

Definition A formula is in *prenex conjunctive normal form* if it has the form

$$Qx_1 \ldots Qx_k \, ((L_{11} \vee \ldots \vee L_{1m_1}) \wedge \ldots \wedge (L_{n1} \vee \ldots \vee L_{nm_n}))$$

where each Q is an existential or universal quantifier and each L_{ij} is a literal.

The next proposition shows that each formula has an "equivalent" formula, which is in prenex conjunctive normal form.

Definition We say two formulas W and V are *logically equivalent* if $\forall(W \leftrightarrow V)$ is valid.

In other words, two formulas are logically equivalent if they have the same truth values wrt any interpretation and variable assignment.

Proposition 3.4 For each formula W, there is a formula V, logically equivalent to W, such that V is in prenex conjunctive normal form.

Proof The proof is left as an exercise. (See problem 5.) ■

When we discuss deductive database systems in chapter 5, we will base the theoretical developments on a typed first order theory. The intuitive idea of a typed theory (also called a many-sorted theory [33]) is that there are several sorts of variables, each ranging over a different domain. This can be thought of as a generalisation of the theories we have considered so far which only allow a single domain. For example, in a database context, there may be several domains of interest, such as the domain of customer names, the domain of supplier cities, and so on. For semantic integrity reasons, it is important to allow only queries and

database clauses which respect the typing restrictions.

In addition to the components of a first order theory, a *typed* first order theory has a finite set, whose elements are called *types*. Types are denoted by Greek letters, such as τ and σ. The alphabet of the typed first order theory contains variables, constants, function symbols, predicate symbols and quantifiers, each of which is typed. Variables and constants have types such as τ. Predicate symbols have types of the form $\tau_1 \times \ldots \times \tau_n$ and function symbols have types of the form $\tau_1 \times \ldots \times \tau_n \to \tau$. If f has type $\tau_1 \times \ldots \times \tau_n \to \tau$, we say f has *range type* τ. For each type τ, there is a universal quantifier $\forall_\tau$ and an existential quantifier $\exists_\tau$.

Definition A *term of type* τ is defined inductively as follows:
(a) A variable of type τ is a term of type τ.
(b) A constant of type τ is a term of type τ.
(c) If f is an n-ary function symbol of type $\tau_1 \times \ldots \times \tau_n \to \tau$ and t_i is a term of type τ_i $(i=1,\ldots,n)$, then $f(t_1,\ldots,t_n)$ is a term of type τ.

Definition A *typed (well-formed) formula* is defined inductively as follows:
(a) If p is an n-ary predicate symbol of type $\tau_1 \times \ldots \times \tau_n$ and t_i is a term of type τ_i $(i=1,\ldots,n)$, then $p(t_1,\ldots,t_n)$ is a typed atomic formula.
(b) If F and G are typed formulas, then so are $\sim F$, $F \wedge G$, $F \vee G$, $F \to G$ and $F \leftrightarrow G$.
(c) If F is a typed formula and x is a variable of type τ, then $\forall_\tau x\, F$ and $\exists_\tau x\, F$ are typed formulas.

Definition The *typed first order language* given by an alphabet consists of the set of all typed formulas constructed from the symbols of the alphabet.

We will find it more convenient to use the notation $\forall x/\tau\, F$ in place of $\forall_\tau x\, F$. Similarly, we will use the notation $\exists x/\tau\, F$ in place of $\exists_\tau x\, F$. We let $\forall(F)$ denote the typed universal closure of the formula F and $\exists(F)$ denote the typed existential closure. These are obtained by prefixing F with quantifiers of appropriate types.

Definition A *pre-interpretation* of a typed first order language L consists of the following:
(a) For each type τ, a non-empty set D_τ, called the *domain of type* τ of the pre-interpretation.
(b) For each constant of type τ in L, the assignment of an element in D_τ.
(c) For each n-ary function symbol of type $\tau_1 \times \ldots \times \tau_n \to \tau$ in L, the assignment of a mapping from $D_{\tau_1} \times \ldots \times D_{\tau_n}$ to D_τ.

Definition An *interpretation* I of a typed first order language L consists of a pre-interpretation J with domains $\{D_\tau\}$ of L together with the following:
For each n-ary predicate symbol of type $\tau_1 \times ... \times \tau_n$ in L, the assignment of a mapping from $D_{\tau_1} \times ... \times D_{\tau_n}$ into {true, false} (or, equivalently, a relation on $D_{\tau_1} \times ... \times D_{\tau_n}$).
We say I is *based on* J.

It is straightforward to define the concepts of variable assignment, term assignment, truth value, model, logical consequence, and so on, for a typed first order theory. We leave the details to the reader. Generally speaking, the development of the theory of first order logic can be carried through with only the most trivial changes for typed first order logic. We shall exploit this fact in chapter 5, where we shall use typed versions of results from earlier chapters.

The other fact that we will need about typed logics is that there is a transformation of typed formulas into (type-free) formulas, which shows that the apparent extra generality provided by typed logics is illusory [33]. This transformation allows one to reduce the proof of a theorem in a typed logic to a corresponding theorem in a (type-free) logic. We shall use this transformation process as one stage of the query evaluation process for deductive database systems in chapter 5.

§4. UNIFICATION

Earlier we stated that the main purpose of a logic programming system is to compute bindings. These bindings are computed by unification. In this section, we present a detailed discussion of unifiers and the unification algorithm.

Definition A *substitution* θ is a finite set of the form $\{v_1/t_1,...,v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i and the variables $v_1,...,v_n$ are distinct. Each element v_i/t_i is called a *binding* for v_i. θ is called a *ground substitution* if the t_i are all ground terms. θ is called a *variable-pure substitution* if the t_i are all variables.

Definition 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 Let $\theta = \{v_1/t_1,...,v_n/t_n\}$ be a substitution and E be an expression. Then Eθ, 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 Eθ is ground, then Eθ is called a *ground instance* of E.

Example Let E = p(x,y,f(a)) and θ = {x/b, y/x}. Then Eθ = p(b,x,f(a)).

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

Definition 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* θσ 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 binding $u_i/s_i\sigma$ for which $u_i = s_i\sigma$ and deleting any binding v_j/t_j for which $v_j \in \{u_1,...,u_m\}$.

Example Let θ = {x/f(y), y/z} and σ = {x/a, y/b, z/y}. Then θσ = {x/f(b), z/y}.

Definition The substitution given by the empty set is called the *identity substitution*.

We denote the identity substitution by ε. Note that Eε = E, for all expressions E. The elementary properties of substitutions are contained in the following proposition.

Proposition 4.1 Let θ, σ and γ be substitutions. Then
(a) θε = εθ = θ.
(b) (Eθ)σ = E(θσ), for all expressions E.
(c) (θσ)γ = θ(σγ).

Proof (a) This follows immediately from the definition of ε.
(b) Clearly it suffices to prove the result when E is a variable, say, x. Let $\theta = \{u_1/s_1,...,u_m/s_m\}$ and $\sigma = \{v_1/t_1,...,v_n/t_n\}$. If $x \notin \{u_1,...,u_m\} \cup \{v_1,...,v_n\}$, then $(x\theta)\sigma = x = x(\theta\sigma)$. If $x \in \{u_1,...,u_m\}$, say $x = u_i$, then $(x\theta)\sigma = s_i\sigma = x(\theta\sigma)$. If $x \in \{v_1,...,v_n\} \setminus \{u_1,...,u_m\}$, say $x = v_j$, then $(x\theta)\sigma = t_j = x(\theta\sigma)$.
(c) Clearly it suffices to show that if x is a variable, then $x((\theta\sigma)\gamma) = x(\theta(\sigma\gamma))$. In fact, $x((\theta\sigma)\gamma) = (x(\theta\sigma))\gamma = ((x\theta)\sigma)\gamma = (x\theta)(\sigma\gamma) = x(\theta(\sigma\gamma))$, by (b). ■

Proposition 4.1(a) shows that ε acts as a left and right identity for composition. The definition of composition of substitutions was made precisely to obtain (b). Note that (c) shows that we can omit parentheses when writing a composition $\theta_1 \ldots \theta_n$ of substitutions.

Example Let $\theta = \{x/f(y), y/z\}$ and $\sigma = \{x/a, z/b\}$. Then $\theta\sigma = \{x/f(y), y/b, z/b\}$. Let $E = p(x,y,g(z))$. Then $E\theta = p(f(y),z,g(z))$ and $(E\theta)\sigma = p(f(y),b,g(b))$. Also $E(\theta\sigma) = p(f(y),b,g(b)) = (E\theta)\sigma$.

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

Example $p(f(x,y),g(z),a)$ is a variant of $p(f(y,x),g(u),a)$. However, $p(x,x)$ is not a variant of $p(x,y)$.

Definition Let E be an expression and V be the set of variables occurring in E. A *renaming substitution* for E is a variable-pure substitution $\{x_1/y_1,\ldots,x_n/y_n\}$ such that $\{x_1,\ldots,x_n\} \subseteq V$, the y_i are distinct and $(V \setminus \{x_1,\ldots,x_n\}) \cap \{y_1,\ldots,y_n\} = \emptyset$.

Proposition 4.2 Let E and F be expressions which are variants. Then there exist substitutions θ and σ such that $E=F\theta$ and $F=E\sigma$, where θ is a renaming substitution for F and σ is a renaming substitution for E.

Proof Since E and F are variants, there exist substitutions θ_1 and σ_1 such that $E=F\theta_1$ and $F=E\sigma_1$. Let V be the set of variables occurring in E and let σ be the substitution obtained from σ_1 by deleting all bindings of the form x/t, where $x \notin V$. Clearly $F=E\sigma$. Furthermore, $E=F\theta_1=E\sigma\theta_1$ and it follows that σ must be a renaming substitution for E. ■

We will be particularly interested in substitutions which unify a set of expressions, that is, make each expression in the set syntactically identical. The concept of unification goes back to Herbrand [44] in 1930. It was rediscovered in 1963 by Robinson [88] and exploited in the resolution rule, where it was used to reduce the combinatorial explosion of the search space. We restrict attention to (non-empty) finite sets of simple expressions, which is all that we require. Recall that a simple expression is a term or an atom.

Definition 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 γ such that $\sigma=\theta\gamma$.

Example $\{p(f(x),a), p(y,f(w))\}$ is not unifiable, because the second arguments cannot be unified.

Example $\{p(f(x),z), p(y,a)\}$ is unifiable, since $\sigma = \{y/f(a), x/a, z/a\}$ is a unifier. A most general unifier is $\theta = \{y/f(x), z/a\}$. Note that $\sigma = \theta\{x/a\}$.

It follows from the definition of an mgu that if θ and σ are both mgu's of $\{E_1,\ldots,E_n\}$, then $E_1\theta$ is a variant of $E_1\sigma$. Proposition 4.2 then shows that $E_1\sigma$ can be obtained from $E_1\theta$ simply by renaming variables. In fact, problem 7 shows that mgu's are unique modulo renaming.

We next present an algorithm, called the unification algorithm, which takes a finite set of simple expressions as input and outputs an mgu if the set is unifiable. Otherwise, it reports the fact that the set is not unifiable. The intuitive idea behind the unification algorithm is as follows. Suppose we want to unify two simple expressions. Imagine two pointers, one at the leftmost symbol of each of the two expressions. The pointers are moved together to the right until they point to different symbols. An attempt is made to unify the two subexpressions starting with these symbols by making a substitution. If the attempt is successful, the process is continued with the two expressions obtained by applying the substitution. If not, the expressions are not unifiable. If the pointers eventually reach the ends of the two expressions, the composition of all the substitutions made is an mgu of the two expressions.

Definition Let S be a finite set of simple expressions. The *disagreement set* of S is defined as follows. Locate the leftmost symbol position at which not all expressions in S have the same symbol and extract from each expression in S the subexpression beginning at that symbol position. The set of all such subexpressions is the disagreement set.

Example Let $S = \{p(f(x),h(y),a), p(f(x),z,a), p(f(x),h(y),b)\}$. Then the disagreement set is $\{h(y), z\}$.

We now present the unification algorithm. In this algorithm, S denotes a finite set of simple expressions.

UNIFICATION ALGORITHM

1. Put $k=0$ and $\sigma_0=\varepsilon$.
2. If $S\sigma_k$ is a singleton, then stop; σ_k is an mgu of S. Otherwise, find the disagreement set D_k of $S\sigma_k$.
3. If there exist v and t in D_k such that v is a variable that does not occur in t, then put $\sigma_{k+1} = \sigma_k\{v/t\}$, increment k and go to 2. Otherwise, stop; S is not unifiable.

The unification algorithm as presented above is non-deterministic to the extent that there may be several choices for v and t in step 3. However, as we remarked earlier, the application of any two mgu's produced by the algorithm leads to expressions which differ only by a change of variable names. It is clear that the algorithm terminates because S contains only finitely many variables and each application of step 3 eliminates one variable.

Example Let $S = \{p(f(a),g(x)), p(y,y)\}$.

(a) $\sigma_0 = \varepsilon$.
(b) $D_0 = \{f(a), y\}$, $\sigma_1 = \{y/f(a)\}$ and $S\sigma_1 = \{p(f(a),g(x)), p(f(a),f(a))\}$.
(c) $D_1 = \{g(x), f(a)\}$. Thus S is not unifiable.

Example Let $S = \{p(a,x,h(g(z))), p(z,h(y),h(y))\}$.

(a) $\sigma_0 = \varepsilon$.
(b) $D_0 = \{a, z\}$, $\sigma_1 = \{z/a\}$ and $S\sigma_1 = \{p(a,x,h(g(a))), p(a,h(y),h(y))\}$.
(c) $D_1 = \{x, h(y)\}$, $\sigma_2 = \{z/a, x/h(y)\}$ and $S\sigma_2 = \{p(a,h(y),h(g(a))), p(a,h(y),h(y))\}$.
(d) $D_2 = \{y, g(a)\}$, $\sigma_3 = \{z/a, x/h(g(a)), y/g(a)\}$ and $S\sigma_3 = \{p(a,h(g(a)),h(g(a)))\}$.
Thus S is unifiable and σ_3 is an mgu.

In step 3 of the unification algorithm, a check is made to see whether v occurs in t. This is called the *occur check*. The next example illustrates the use of the occur check.

Example Let $S = \{p(x,x), p(y,f(y))\}$.

(a) $\sigma_0 = \varepsilon$.
(b) $D_0 = \{x, y\}$, $\sigma_1 = \{x/y\}$ and $S\sigma_1 = \{p(y,y), p(y,f(y))\}$.
(c) $D_1 = \{y, f(y)\}$. Since y occurs in f(y), S is not unifiable.

Next we prove that the unification algorithm does indeed find an mgu of a unifiable set of simple expressions. This result first appeared in [88].

Theorem 4.3 (Unification Theorem)

Let S be a finite set of simple expressions. If S is unifiable, then the unification algorithm terminates and gives an mgu for S. If S is not unifiable, then the unification algorithm terminates and reports this fact.

Proof We have already noted that the unification algorithm always terminates. It suffices to show that if S is unifiable, then the algorithm finds an mgu. In fact, if S is not unifiable, then the algorithm cannot terminate at step 2 and, since it does terminate, it must terminate at step 3. Thus it does report the fact that S is not unifiable.

Assume then that S is unifiable and let θ be any unifier for S. We prove first that, for $k\geq0$, if σ_k is the substitution given in the kth iteration of the algorithm, then there exists a substitution γ_k such that $\theta = \sigma_k\gamma_k$.

Suppose first that $k=0$. Then we can put $\gamma_0 = \theta$, since $\theta = \varepsilon\theta$. Next suppose, for some $k\geq0$, there exists γ_k such that $\theta = \sigma_k\gamma_k$. If $S\sigma_k$ is a singleton, then the algorithm terminates at step 2. Hence we can confine attention to the case when $S\sigma_k$ is not a singleton. We want to show that the algorithm will produce a further substitution σ_{k+1} and that there exists a substitution γ_{k+1} such that $\theta = \sigma_{k+1}\gamma_{k+1}$.

Since $S\sigma_k$ is not a singleton, the algorithm will determine the disagreement set D_k of $S\sigma_k$ and go to step 3. Since $\theta = \sigma_k\gamma_k$ and θ unifies S, it follows that γ_k unifies D_k. Thus D_k must contain a variable, say, v. Let t be any other term in D_k. Then v cannot occur in t because $v\gamma_k = t\gamma_k$. We can suppose that $\{v/t\}$ is indeed the substitution chosen at step 3. Thus $\sigma_{k+1} = \sigma_k\{v/t\}$.

We now define $\gamma_{k+1} = \gamma_k\setminus\{v/v\gamma_k\}$. If γ_k has a binding for v, then

$$\begin{aligned}
\gamma_k &= \{v/v\gamma_k\} \cup \gamma_{k+1} & \\
&= \{v/t\gamma_k\} \cup \gamma_{k+1} & \text{(since } v\gamma_k = t\gamma_k\text{)}\\
&= \{v/t\gamma_{k+1}\} \cup \gamma_{k+1} & \text{(since v does not occur in t)}\\
&= \{v/t\}\gamma_{k+1} & \text{(by the definition of composition).}
\end{aligned}$$

If γ_k does not have a binding for v, then $\gamma_{k+1} = \gamma_k$, each element of D_k is a variable and $\gamma_k = \{v/t\}\gamma_{k+1}$. Thus $\theta = \sigma_k\gamma_k = \sigma_k\{v/t\}\gamma_{k+1} = \sigma_{k+1}\gamma_{k+1}$, as required.

Now we can complete the proof. If S is unifiable, then we have shown that the algorithm must terminate at step 2 and, if it terminates at the kth iteration, then $\theta = \sigma_k\gamma_k$, for some γ_k. Since σ_k is a unifier of S, this equality shows that it is indeed an mgu for S. ∎

The unification algorithm which we have presented can be very inefficient. In the worst case, its running time can be an exponential function of the length of the input. Consider the following example, which is taken from [9]. Let $S = \{p(x_1,\ldots,x_n), p(f(x_0,x_0),\ldots,f(x_{n-1},x_{n-1}))\}$. Then $\sigma_1 = \{x_1/f(x_0,x_0)\}$ and $S\sigma_1 = \{p(f(x_0,x_0),x_2,\ldots,x_n), p(f(x_0,x_0),f(f(x_0,x_0),f(x_0,x_0)),f(x_2,x_2),\ldots,f(x_{n-1},x_{n-1}))\}$. The next substitution is $\sigma_2 = \{x_1/f(x_0,x_0), x_2/f(f(x_0,x_0), f(x_0,x_0))\}$, and so on. Note that the second atom in $S\sigma_n$ has 2^k-1 occurrences of f in its kth argument $(1\leq k\leq n)$. In particular, its last argument has 2^n-1 occurrences of f. Now recall that step 3 of the unification algorithm has the occur check. The performance of this check just for the last substitution will thus require exponential time. In fact, printing σ_n also requires exponential time. This example shows that no unification algorithm which *explicitly* presents the (final) unifier can be linear.

Much more efficient unification algorithms than the one presented above are known. For example, [67] and [80] give linear algorithms (see also [68]). In [80], linearity is achieved by the use of a carefully chosen data structure for representing expressions and avoiding the explicit presentation of the unifier, which is instead presented as a composition of constituent substitutions. Despite its linearity, this algorithm is not employed in PROLOG systems. Instead, most use essentially the unification algorithm presented earlier in this section, but with the expensive occur check omitted! From a theoretical viewpoint, this is a disaster because it destroys the soundness of SLD-resolution. We discuss this matter further in §7.

§5. FIXPOINTS

Associated with every definite program is a monotonic mapping which plays a very important role in the theory. This section introduces the requisite concepts and results concerning monotonic mappings and their fixpoints.

Definition Let S be a set. A *relation* R on S is a subset of $S\times S$.

We usually use infix notation writing $(x,y)\in R$ as xRy.

Definition A relation R on a set S is a *partial order* if the following conditions are satisfied:
(a) xRx, for all $x\in S$.
(b) xRy and yRx imply $x=y$, for all $x,y\in S$.

(c) xRy and yRz imply xRz, for all $x,y,z\in S$.

Example Let S be a set and 2^S be the set of all subsets of S. Then set inclusion, $\subseteq$, is easily seen to be a partial order on 2^S.

We adopt the standard notation and use $\leq$ to denote a partial order. Thus we have (a) $x\leq x$, (b) $x\leq y$ and $y\leq x$ imply $x=y$ and (c) $x\leq y$ and $y\leq z$ imply $x\leq z$, for all $x,y,z\in S$.

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

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

The least upper bound of X is unique, if it exists, and is denoted by lub(X). Similarly, the greatest lower bound of X is unique, if it exists, and is denoted by glb(X).

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

We let $\top$ denote the *top element* lub(L) and $\bot$ denote the *bottom element* glb(L) of the complete lattice L.

Example In the previous example, 2^S under $\subseteq$ is a complete lattice. In fact, 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 $\emptyset$.

Definition Let L be a complete lattice and $T : L\to L$ be a mapping. We say T is *monotonic* if $T(x)\leq T(y)$, whenever $x\leq y$.

Definition Let L be a complete lattice and $X\subseteq L$. We say X is *directed* if every finite subset of X has an upper bound in X.

Definition Let L be a complete lattice and $T : L\to L$ be a mapping. We say T is *continuous* if $T(\mathrm{lub}(X)) = \mathrm{lub}(T(X))$, for every directed subset X of L.

By taking $X = \{x,y\}$, we see that every continuous mapping is monotonic. However, the converse is not true. (See problem 12.)

Our interest in these definitions arises from the fact that for a definite program P, the collection of all Herbrand interpretations forms a complete lattice in a natural way and also because there is a continuous mapping associated with P defined on this lattice. Next we study fixpoints of mappings defined on lattices.

Definition Let L be a complete lattice and $T : L \to L$ be a mapping. We say $a \in L$ is the *least fixpoint* of T if a is a fixpoint (that is, $T(a)=a$) and for all fixpoints b of T, we have $a \leq b$. Similarly, we define *greatest fixpoint.*

The next result is a weak form of a theorem due to Tarski [103], which generalises an earlier result due to Knaster and Tarski. For an interesting account of the history of propositions 5.1, 5.3 and 5.4, see [55].

Proposition 5.1 Let L be a complete lattice and $T : L \to L$ be monotonic. Then T has a least fixpoint, $lfp(T)$, and a greatest fixpoint, $gfp(T)$. Furthermore, $lfp(T) = glb\{x : T(x)=x\} = glb\{x : T(x) \leq x\}$ and $gfp(T) = lub\{x : T(x)=x\} = lub\{x : x \leq T(x)\}$.

Proof Put $G = \{x : T(x) \leq x\}$ and $g = glb(G)$. We show that $g \in G$. Now $g \leq x$, for all $x \in G$, so that by the monotonicity of T, we have $T(g) \leq T(x)$, for all $x \in G$. Thus $T(g) \leq x$, for all $x \in G$, and so $T(g) \leq g$, by the definition of glb. Hence $g \in G$.

Next we show that g is a fixpoint of T. It remains to show that $g \leq T(g)$. Now $T(g) \leq g$ implies $T(T(g)) \leq T(g)$ implies $T(g) \in G$. Hence $g \leq T(g)$, so that g is a fixpoint of T.

Now put $g' = glb\{x : T(x)=x\}$. Since g is a fixpoint, we have $g' \leq g$. On the other hand, $\{x : T(x)=x\} \subseteq \{x : T(x) \leq x\}$ and so $g \leq g'$. Thus we have $g=g'$ and the proof is complete for $lfp(T)$.

The proof for $gfp(T)$ is similar. ■

Proposition 5.2 Let L be a complete lattice and $T : L \to L$ be monotonic. Suppose $a \in L$ and $a \leq T(a)$. Then there exists a fixpoint a' of T such that $a \leq a'$. Similarly, if $b \in L$ and $T(b) \leq b$, then there exists a fixpoint b' of T such that $b' \leq b$.

Proof By proposition 5.1, it suffices to put $a'=gfp(T)$ and $b'=lfp(T)$. ■

We will also require the concept of ordinal powers of T. First we recall some elementary properties of ordinal numbers, which we will refer to more simply as ordinals. Intuitively, the ordinals are what we use to count with. The first ordinal 0

is defined to be $\emptyset$. Then we define $1 = \{\emptyset\} = \{0\}$, $2 = \{\emptyset, \{\emptyset\}\} = \{0, 1\}$, $3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} = \{0, 1, 2\}$, and so on. These are the finite ordinals, the non-negative integers. The first infinite ordinal is $\omega = \{0, 1, 2, ...\}$, the set of all non-negative integers. We adopt the convention of denoting finite ordinals by roman letters n, m,..., while arbitrary ordinals will be denoted by Greek letters α, β,.... We can specify an ordering $<$ on the collection of all ordinals by defining $\alpha < \beta$ if $\alpha \in \beta$. For example, $n < \omega$, for all finite ordinals n. We will normally write $n \in \omega$ rather than $n < \omega$. If α is an ordinal, the *successor* of α is the ordinal $\alpha+1 = \alpha \cup \{\alpha\}$, which is the least ordinal greater than α. $\alpha+1$ is then said to be a *successor ordinal.* For example, $1 = 0+1$, $2 = 1+1$, $3 = 2+1$, and so on. If α is a successor ordinal, say $\alpha = \beta+1$, we denote β by $\alpha-1$. An ordinal α is said to be a *limit ordinal* if it is not the successor of any ordinal. The smallest limit ordinal (apart from 0) is ω. After ω comes $\omega+1 = \omega \cup \{\omega\}$, $\omega+2 = (\omega+1)+1$, $\omega+3$, and so on. The next limit ordinal is $\omega 2$, which is the set consisting of all n, where $n \in \omega$, and all $\omega+n$, where $n \in \omega$. Then come $\omega 2+1$, $\omega 2+2$,...,$\omega 3$, $\omega 3+1$,...,$\omega 4$,...,ωn,... .

We will also require the *principle of transfinite induction*, which is as follows. Let $P(\alpha)$ be a property of ordinals. Assume that for all ordinals β, if $P(\gamma)$ holds for all $\gamma < \beta$, then $P(\beta)$ holds. Then $P(\alpha)$ holds for all ordinals α.

Now we can give the definition of the ordinal powers of T.

Definition Let L be a complete lattice and $T : L \to L$ be monotonic. Then we define

$T\uparrow 0 = \bot$
$T\uparrow\alpha = T(T\uparrow(\alpha-1))$, if α is a successor ordinal
$T\uparrow\alpha = lub\{T\uparrow\beta : \beta < \alpha\}$, if α is a limit ordinal
$T\downarrow 0 = \top$
$T\downarrow\alpha = T(T\downarrow(\alpha-1))$, if α is a successor ordinal
$T\downarrow\alpha = glb\{T\downarrow\beta : \beta < \alpha\}$, if α is a limit ordinal

Next we give a well-known characterisation of $lfp(T)$ and $gfp(T)$ in terms of ordinal powers of T.

Proposition 5.3 Let L be a complete lattice and $T : L \to L$ be monotonic. Then, for any ordinal α, $T\uparrow\alpha \leq lfp(T)$ and $T\downarrow\alpha \geq gfp(T)$. Furthermore, there exist ordinals β_1 and β_2 such that $\gamma_1 \geq \beta_1$ implies $T\uparrow\gamma_1 = lfp(T)$ and $\gamma_2 \geq \beta_2$ implies $T\downarrow\gamma_2 = gfp(T)$.

Proof The proof for $\mathrm{lfp}(T)$ follows from (a) and (e) below. The proofs of (a), (b) and (c) use transfinite induction.

(a) For all α, $T\uparrow\alpha \le \mathrm{lfp}(T)$:
If α is a limit ordinal, then $T\uparrow\alpha = \mathrm{lub}\{T\uparrow\beta : \beta<\alpha\} \le \mathrm{lfp}(T)$, by the induction hypothesis. If α is a successor ordinal, then $T\uparrow\alpha = T(T\uparrow(\alpha-1)) \le T(\mathrm{lfp}(T)) = \mathrm{lfp}(T)$, by the induction hypothesis, the monotonicity of T and the fixpoint property.

(b) For all α, $T\uparrow\alpha \le T\uparrow(\alpha+1)$:
If α is a successor ordinal, then $T\uparrow\alpha = T(T\uparrow(\alpha-1)) \le T(T\uparrow\alpha) = T\uparrow(\alpha+1)$, using the induction hypothesis and the monotonicity of T. If α is a limit ordinal, then $T\uparrow\alpha = \mathrm{lub}\{T\uparrow\beta : \beta<\alpha\} \le \mathrm{lub}\{T\uparrow(\beta+1) : \beta<\alpha\} \le T(\mathrm{lub}\{T\uparrow\beta : \beta<\alpha\}) = T\uparrow(\alpha+1)$, using the induction hypothesis and monotonicity of T.

(c) For all α,β, $\alpha<\beta$ implies $T\uparrow\alpha \le T\uparrow\beta$:
If β is a limit ordinal, then $T\uparrow\alpha \le \mathrm{lub}\{T\uparrow\gamma : \gamma<\beta\} = T\uparrow\beta$. If β is a successor ordinal, then $\alpha \le \beta-1$ and so $T\uparrow\alpha \le T\uparrow(\beta-1) \le T\uparrow\beta$, using the induction hypothesis and (b).

(d) For all α,β, if $\alpha<\beta$ and $T\uparrow\alpha = T\uparrow\beta$, then $T\uparrow\alpha = \mathrm{lfp}(T)$:
Now $T\uparrow\alpha \le T\uparrow(\alpha+1) \le T\uparrow\beta$, by (c). Hence $T\uparrow\alpha = T\uparrow(\alpha+1) = T(T\uparrow\alpha)$ and so $T\uparrow\alpha$ is a fixpoint. Furthermore, $T\uparrow\alpha = \mathrm{lfp}(T)$, by (a).

(e) There exists β such that $\gamma \ge \beta$ implies $T\uparrow\gamma = \mathrm{lfp}(T)$:
Let α be the least ordinal of cardinality greater than the cardinality of L. Suppose that $T\uparrow\delta \ne \mathrm{lfp}(T)$, for all $\delta<\alpha$. Define $h:\alpha\to L$ by $h(\delta) = T\uparrow\delta$. Then, by (d), h is injective, which contradicts the choice of α. Thus $T\uparrow\beta = \mathrm{lfp}(T)$, for some $\beta<\alpha$, and the result follows from (a) and (c).
The proof for $\mathrm{gfp}(T)$ is similar. ■

[illegible]

The least α such that $T\uparrow\alpha = \mathrm{lfp}(T)$ is called the *closure ordinal* of T. The next result, which is usually attributed to Kleene, shows that under the stronger assumption that T is continuous, the closure ordinal of T is $\le \omega$.

Proposition 5.4 Let L be a complete lattice and $T : L\to L$ be continuous. Then $\mathrm{lfp}(T) = T\uparrow\omega$.

Proof By proposition 5.3, it suffices to show that $T\uparrow\omega$ is a fixpoint. Note that $\{T\uparrow n : n\in\omega\}$ is directed, since T is monotonic. Thus $T(T\uparrow\omega) = T(\mathrm{lub}\{T\uparrow n : n\in\omega\}) = \mathrm{lub}\{T(T\uparrow n) : n\in\omega\} = T\uparrow\omega$, using the continuity of T. ■

The analogue of proposition 5.4 for $\mathrm{gfp}(T)$ does not hold, that is, $\mathrm{gfp}(T)$ may not be equal to $T\downarrow\omega$. A counterexample is given in the next section.

PROBLEMS FOR CHAPTER 1

1. Consider the interpretation I:
   Domain is the non-negative integers
   s is assigned the successor function $x \to x+1$
   a is assigned 0
   b is assigned 1
   p is assigned the relation $\{(x,y) : x>y\}$
   q is assigned the relation $\{x : x>0\}$
   r is assigned the relation $\{(x,y) : x \text{ divides } y\}$

For each of the following closed formulas, determine the truth value of the formula wrt I:
(a) $\forall x\,\exists y\, p(x,y)$
(b) $\exists x\,\forall y\, p(x,y)$
(c) $p(s(a),b)$
(d) $\forall x(q(x)\to p(x,a))$
(e) $\forall x\, p(s(x),x)$
(f) $\forall x\,\forall y(r(x,y)\to\sim p(x,y))$
(g) $\forall x(\exists y\, p(x,y) \vee r(s(b),s(x)) \to q(x))$

2. Determine whether the following formulas are valid or not:
(a) $\forall x\,\exists y\, p(x,y) \to \exists y\,\forall x\, p(x,y)$
(b) $\exists y\,\forall x\, p(x,y) \to \forall x\,\exists y\, p(x,y)$

3. Consider the formula
$(\forall x\, p(x,x) \wedge \forall x\forall y\forall z\,[(p(x,y)\wedge p(y,z))\to p(x,z)] \wedge \forall x\forall y\,[p(x,y)\vee p(y,x)]) \to \exists y\forall x\, p(y,x)$
(a) Show that every interpretation with a finite domain is a model.
(b) Find an interpretation which is not a model.

4. Complete the proof of proposition 3.2.

5. Let W be a formula. Suppose that each quantifier in W has a distinct variable

following it and no variable in W is both bound and free. (This can be achieved by renaming bound variables in W, if necessary.) Prove that W can be transformed to a logically equivalent formula in prenex conjunctive normal form (called a *prenex conjunctive normal form of* W) by means of the following transformations:

(a) Replace

all occurrences of $F \leftarrow G$ by $F \vee \sim G$

all occurrences of $F \leftrightarrow G$ by $(F \vee \sim G) \wedge (\sim F \vee G)$.

(b) Replace

$\sim \forall x F$ by $\exists x \sim F$

$\sim \exists x F$ by $\forall x \sim F$

$\sim (F \vee G)$ by $\sim F \wedge \sim G$

$\sim (F \wedge G)$ by $\sim F \vee \sim G$

$\sim\sim F$ by F

until each occurrence of $\sim$ immediately precedes an atom.

(c) Replace

$\exists x F \vee G$ by $\exists x (F \vee G)$

$F \vee \exists x G$ by $\exists x (F \vee G)$

$\forall x F \vee G$ by $\forall x (F \vee G)$

$F \vee \forall x G$ by $\forall x (F \vee G)$

$\exists x F \wedge G$ by $\exists x (F \wedge G)$

$F \wedge \exists x G$ by $\exists x (F \wedge G)$

$\forall x F \wedge G$ by $\forall x (F \wedge G)$

$F \wedge \forall x G$ by $\forall x (F \wedge G)$

until all quantifiers are at the front of the formula.

(d) Replace

$(F \wedge G) \vee H$ by $(F \vee H) \wedge (G \vee H)$

$F \vee (G \wedge H)$ by $(F \vee G) \wedge (F \vee H)$

until the formula is in prenex conjunctive normal form.

6. Let W be a closed formula. Prove that there exists a formula V, which is a conjunction of clauses, such that W is unsatisfiable iff V is unsatisfiable.

7. Suppose θ_1 and θ_2 are substitutions and there exist substitutions σ_1 and σ_2 such that $\theta_1 = \theta_2\sigma_1$ and $\theta_2 = \theta_1\sigma_2$. Show that there exists a variable-pure substitution γ such that $\theta_1 = \theta_2\gamma$.

8. A substitution θ is *idempotent* if $\theta = \theta\theta$. Let $\theta = \{x_1/t_1,\ldots,x_n/t_n\}$ and suppose V is the set of variables occurring in terms in $\{t_1,\ldots,t_n\}$. Show that θ is idempotent iff $\{x_1,\ldots,x_n\} \cap V = \emptyset$.

9. Prove that each mgu produced by the unification algorithm is idempotent.

10. Let θ be a unifier of a finite set S of simple expressions. Prove that θ is an mgu and is idempotent iff, for every unifier σ of S, we have $\sigma = \theta\sigma$.

11. For each of the following sets of simple expressions, determine whether mgu's exist or not and find them when they exist:

(a) $\{p(f(y),w,g(z)), p(u,u,v)\}$

(b) $\{p(f(y),w,g(z)), p(v,u,v)\}$

(c) $\{p(a,x,f(g(y))), p(z,h(z,w),f(w))\}$

12. Find a complete lattice L and a mapping $T : L \rightarrow L$ such that T is monotonic but not continuous.

13. Let L be a complete lattice and $T : L \rightarrow L$ be monotonic.

(a) Suppose $a \in L$ and $a \leq T(a)$. Define

$T^0(a) = a$

$T^\alpha(a) = T(T^{\alpha-1}(a))$, if α is a successor ordinal

$T^\alpha(a) = \text{lub}\{T^\beta(a) : \beta < \alpha\}$, if α is a limit ordinal.

Prove that there exists an ordinal β such that $T^\beta(a)$ is a fixpoint of T and $a \leq T^\beta(a)$.

(b) Suppose $b \in L$ and $T(b) \leq b$. Define

$T^0(b) = b$

$T^\alpha(b) = T(T^{\alpha-1}(b))$, if α is a successor ordinal

$T^\alpha(b) = \text{glb}\{T^\beta(b) : \beta < \alpha\}$, if α is a limit ordinal.

Prove that there exists an ordinal γ such that $T^\gamma(b)$ is a fixpoint of T and $T^\gamma(b) \leq b$.

Chapter 2

DEFINITE PROGRAMS

This chapter is concerned with the declarative and procedural semantics of definite programs. First, we introduce the concept of the least Herbrand model of a definite program and prove various important properties of such models. Next, we define correct answers, which provide a declarative description of the desired output from a program and a goal. The procedural counterpart of a correct answer is a computed answer, which is defined using SLD-resolution. We prove that every computed answer is correct and that every correct answer is an instance of a computed answer. This establishes the soundness and completeness of SLD-resolution, that is, shows that SLD-resolution produces only and all correct answers. Other important results established are the independence of the computation rule and the fact that any computable function can be computed by a definite program. Two pragmatic aspects of PROLOG implementations are also discussed. These are the omission of the occur check from the unification algorithm and the control facility, cut.

§6. DECLARATIVE SEMANTICS

This section introduces the least Herbrand model of a definite program. This particular model plays a central role in the theory. We show that the least Herbrand model is precisely the set of ground atoms which are logical consequences of the definite program. We also obtain an important fixpoint characterisation of the least Herbrand model. Finally, we define the key concept of correct answer.

First, let us recall some definitions given in the previous chapter.

Definition A *definite program clause* is a clause of the form

$$A \leftarrow B_1,\dots,B_n$$

which contains precisely one atom (viz. A) in its consequent. A is called the *head* and $B_1,\dots,B_n$ is called the *body* of the program clause.

Definition A *definite program* is a finite set of definite program clauses.

Definition A *definite goal* is a clause of the form

$$\leftarrow B_1,\dots,B_n$$

that is, a clause which has an empty consequent.

In later chapters, we will consider more general programs, in which the body of a program clause can be a conjunction of literals or even an arbitrary formula. Later we will also consider more general goals. The theory of definite programs is simpler than the theory of these more general classes of programs because definite programs do not allow negations in the body of a clause. This means we can avoid the theoretical and practical difficulties of handling negated subgoals. Definite programs thus provide an excellent starting point for the development of the theory.

Proposition 6.1 (Model Intersection Property)
Let P be a definite program and $\{M_i\}_{i\in I}$ be a non-empty set of Herbrand models for P. Then $\cap_{i\in I}M_i$ is an Herbrand model for P.

Proof Clearly $\cap_{i\in I}M_i$ is an Herbrand interpretation for P. It is straightforward to show that $\cap_{i\in I}M_i$ is a model for P. (See problem 1.) ■

Since every definite program P has B_P as an Herbrand model, the set of all Herbrand models for P is non-empty. Thus the intersection of all Herbrand models for P is again a model, called the *least Herbrand model*, for P. We denote this model by M_P.

The intended interpretation of a definite program P can, of course, be different from M_P. However, there are very strong reasons for regarding M_P as the natural interpretation of a program. Certainly, it is usual for the programmer to have in mind the "free" interpretation of the constants and function symbols in the program given by an Herbrand interpretation. Furthermore, the next theorem shows that the atoms in M_P are precisely those that are logical consequences of the program. This result is due to van Emden and Kowalski [107].

Theorem 6.2 Let P be a definite program. Then $M_P = \{A\in B_P : A \text{ is a logical consequence of } P\}$.

Proof We have that
A is a logical consequence of P
iff $P \cup \{\sim A\}$ is unsatisfiable, by proposition 3.1
iff $P \cup \{\sim A\}$ has no Herbrand models, by proposition 3.3
iff $\sim A$ is false wrt all Herbrand models of P, because P has at least one Herbrand model
iff A is true wrt all Herbrand models of P ~~[illegible]~~
~~([illegible])~~
iff $A\in M_P$. ■

? We wish to obtain a deeper characterisation of M_P using fixpoint concepts. For this we need to associate a complete lattice with every definite program.

Let P be a definite program. Then 2^{B_P}, which is the set of all Herbrand interpretations of P, is a complete lattice under the partial order of set inclusion $\subseteq$. The top element of this lattice is B_P and the bottom element is $\emptyset$. The least upper bound of any set of Herbrand interpretations is the Herbrand interpretation which is the union of all the Herbrand interpretations in the set. The greatest lower bound is the intersection.

Definition Let P be a definite program. The mapping $T_P : 2^{B_P} \to 2^{B_P}$ is defined as follows. Let I be an Herbrand interpretation. Then $T_P(I) = \{A\in B_P : A\leftarrow A_1,\dots,A_n$ is a ground instance of a clause in P and $\{A_1,\dots,A_n\} \subseteq I\}$.

Clearly T_P is monotonic. T_P provides the link between the declarative and procedural semantics of P. This definition was first given in [107].

Example Consider the program P
$p(f(x)) \leftarrow p(x)$ $\{q(a)\ p(a), p(f(a)), \dots\}$
$q(a) \leftarrow p(x)$
Put $I_1 = B_P$, $I_2 = T_P(I_1)$ and $I_3 = \emptyset$. Then $T_P(I_1) = \{q(a)\} \cup \{p(f(t)) : t\in U_P\}$, $T_P(I_2) = \{q(a)\} \cup \{p(f(f(t))) : t\in U_P\}$ and $T_P(I_3) = \emptyset$.

Proposition 6.3 Let P be a definite program. Then the mapping T_P is continuous.

Proof Let X be a directed subset of 2^{B_P}. Note first that $\{A_1,\dots,A_n\} \subseteq \text{lub}(X)$ iff $\{A_1,\dots,A_n\} \subseteq I$, for some $I\in X$. (See problem 3.) In order to show T_P is continuous, we have to show $T_P(\text{lub}(X)) = \text{lub}(T_P(X))$, for each directed subset X.

Now we have that

$A \in T_P(\mathrm{lub}(X))$

iff $A \leftarrow A_1,\dots,A_n$ is a ground instance of a clause in P and $\{A_1,\dots,A_n\} \subseteq \mathrm{lub}(X)$

iff $A \leftarrow A_1,\dots,A_n$ is a ground instance of a clause in P and $\{A_1,\dots,A_n\} \subseteq I$, for some $I \in X$

iff $A \in T_P(I)$, for some $I \in X$

iff $A \in \mathrm{lub}(T_P(X))$. ■

Herbrand interpretations which are models can be characterised in terms of T_P.

Proposition 6.4 Let P be a definite program and I be an Herbrand interpretation of P. Then I is a model for P iff $T_P(I) \subseteq I$.

Proof I is a model for P iff for each ground instance $A \leftarrow A_1,\dots,A_n$ of each clause in P, we have $\{A_1,\dots,A_n\} \subseteq I$ implies $A \in I$ iff $T_P(I) \subseteq I$. ■

Now we come to the first major result of the theory. This theorem, which is due to van Emden and Kowalski [107], provides a fixpoint characterisation of the least Herbrand model of a definite program.

Theorem 6.5 (Fixpoint Characterisation of the Least Herbrand Model)
Let P be a definite program. Then $M_P = \mathrm{lfp}(T_P) = T_P\uparrow\omega$.

Proof $M_P = \mathrm{glb}\{I : I \text{ is an Herbrand model for } P\}$
$= \mathrm{glb}\{I : T_P(I) \subseteq I\}$, by proposition 6.4
$= \mathrm{lfp}(T_P)$, by proposition 5.1
$= T_P\uparrow\omega$, by propositions 5.4 and 6.3. ■

We start from the empty set and iterate T_P till we get M_P

However, it can happen that $\mathrm{gfp}(T_P) \neq T_P\downarrow\omega$.

Example Consider the program P

$p(f(x)) \leftarrow p(x)$

$q(a) \leftarrow p(x)$

Then $T_P\downarrow\omega = \{q(a)\}$, but $\mathrm{gfp}(T_P) = \varnothing$. In fact, $\mathrm{gfp}(T_P) = T_P\downarrow(\omega+1)$.

Let us now turn to the definition of a correct answer. This is a central concept in logic programming and provides much of the focus for the theoretical developments.

Definition Let P be a definite program and G a definite goal. An *answer* for $P \cup \{G\}$ is a substitution for variables of G.

It is understood that the answer does not necessarily contain a binding for every variable in G. In particular, if G has no variables the only possible answer is the identity substitution.

Definition Let P be a definite program, G a definite goal $\leftarrow A_1,\dots,A_k$ and θ an answer for $P \cup \{G\}$. We say that θ is a *correct answer* for $P \cup \{G\}$ if $\forall((A_1 \wedge \dots \wedge A_k)\theta)$ is a logical consequence of P.

Using proposition 3.1, we see that θ is a correct answer iff $P \cup \{\sim\forall((A_1 \wedge \dots \wedge A_k)\theta)\}$ is unsatisfiable. The above definition of correct answer does indeed capture the intuitive meaning of this concept. It provides a declarative description of the desired output from a definite program and goal. Much of this chapter will be concerned with showing the equivalence between this declarative concept and the corresponding procedural one, which is defined by the refutation procedure used by the system.

As well as returning substitutions, a logic programming system may also return the answer "no". We say the answer "no" is *correct* if $P \cup \{G\}$ is satisfiable.

Theorem 6.2 and the definition of correct answer suggest that we may be able to strengthen theorem 6.2 by showing that an answer θ is correct iff $\forall((A_1 \wedge \dots \wedge A_k)\theta)$ is true wrt the least Herbrand model of the program. Unfortunately, the result does not hold in this generality, as the following example shows.

Example Consider the program P

$p(a) \leftarrow$

Let G be the goal $\leftarrow p(x)$ and θ be the identity substitution. Then $M_P = \{p(a)\}$ and so $\forall x\, p(x)\theta$ is true in M_P. However, θ is not a correct answer, since $\forall x\, p(x)\theta$ is not a logical consequence of P.

The reason for the problem here is that $\sim\forall x\, p(x)$ is not a clause and hence we cannot restrict attention to Herbrand interpretations when attempting to establish the unsatisfiability of $\{p(a)\leftarrow\} \cup \{\sim\forall x\, p(x)\}$. However, if we make a restriction on θ, we do obtain a result which generalises theorem 6.2.

Theorem 6.6 Let P be a definite program and G a definite goal $\leftarrow A_1,\ldots,A_k$. Suppose θ is an answer for $P \cup \{G\}$ such that $(A_1 \wedge \ldots \wedge A_k)\theta$ is ground. Then the following are equivalent:
(a) θ is correct.
(b) $(A_1 \wedge \ldots \wedge A_k)\theta$ is true wrt every Herbrand model of P.
(c) $(A_1 \wedge \ldots \wedge A_k)\theta$ is true wrt the least Herbrand model of P.

Proof Obviously, it suffices to show that (c) implies (a). Now
$(A_1 \wedge \ldots \wedge A_k)\theta$ is true wrt the least Herbrand model of P
implies $(A_1 \wedge \ldots \wedge A_k)\theta$ is true wrt all Herbrand models of P
implies $\sim(A_1 \wedge \ldots \wedge A_k)\theta$ is false wrt all Herbrand models of P
implies $P \cup \{\sim(A_1 \wedge \ldots \wedge A_k)\theta\}$ has no Herbrand models
implies $P \cup \{\sim(A_1 \wedge \ldots \wedge A_k)\theta\}$ has no models, by proposition 3.3. ■

§7. SOUNDNESS OF SLD-RESOLUTION

In this section, the procedural semantics of definite programs is introduced. Computed answers are defined and the soundness of SLD-resolution is established. The implications of omitting the occur check from the unification algorithm are also discussed. Although all the requisite results concerning SLD-resolution will be discussed in this and subsequent sections, it would be helpful for the reader to have a wider perspective on automatic theorem proving. We suggest consulting [9], [14], [64] or [66].

There are many refutation procedures based on the resolution inference rule, which are refinements of the original procedure of Robinson [88]. The refutation procedure of interest here was first described by Kowalski [48]. It was called *SLD-resolution* in [4]. (The term *LUSH-resolution* has also been used [46].) SLD-resolution stands for SL-resolution for Definite clauses. SL stands for Linear resolution with Selection function. SL-resolution, which is due to Kowalski and Kuehner [53], is a direct descendant of the model elimination procedure of Loveland [65]. In this and the next two sections, we will be concerned with SLD-refutations. In §10, we will study SLD-refutation procedures.

Definition Let G be $\leftarrow A_1,\ldots,A_m,\ldots,A_k$ and C be $A \leftarrow B_1,\ldots,B_q$. Then G' is *derived* from G and C using mgu θ if the following conditions hold:
(a) A_m is an atom, called the *selected* atom, in G.

(b) θ is an mgu of A_m and A.
(c) G' is the goal $\leftarrow(A_1,\ldots,A_{m-1},B_1,\ldots,B_q,A_{m+1},\ldots,A_k)\theta$.

In resolution terminology, G' is called a *resolvent* of G and C.

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

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 *standardising the variables apart*. It is necessary, otherwise, for example, we would not be able to unify p(x) and p(f(x)) in $\leftarrow$p(x) and p(f(x))$\leftarrow$. Each program clause variant $C_1, C_2,\ldots$ is called an *input clause* of the derivation.

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

Throughout this chapter, a "derivation" will always mean an SLD-derivation and a "refutation" will always mean an SLD-refutation. We can picture SLD-derivations as in Figure 1.

It will be convenient in some of the results to have a slightly more general concept available.

Definition An *unrestricted SLD-refutation* is an SLD-refutation, except that we drop the requirement that the substitutions θ_i be most general unifiers. They are only required to be unifiers.

SLD-derivations may be *finite* or *infinite*. A finite SLD-derivation may be successful or failed. A *successful* SLD-derivation is one that ends in the empty clause. In other words, a successful derivation is just 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. Later we shall see examples of successful, failed and infinite derivations (see Figure 2 and Figure 3).

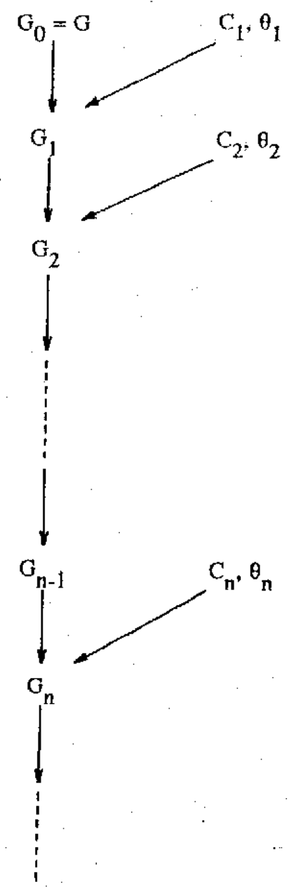

Fig. 1. An SLD-derivation

Definition Let P be a definite program. The *success set* of P is the set of all $A \in B_P$ such that $P \cup \{\leftarrow A\}$ has an SLD-refutation.

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. Similarly, we have the procedural counterpart of a correct answer.

Definition Let P be a definite program and G a definite goal. A *computed answer* θ 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 If P is the slowsort program and G is the goal $\leftarrow$sort(17.22.6.5.nil,y), then {y/5.6.17.22.nil} is a computed answer.

The first soundness result is that computed answers are correct. In the form below, this result is due to Clark [16].

Theorem 7.1 (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 a 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, $\forall((B_1 \wedge \ldots \wedge B_q)\theta_1 \ldots \theta_n)$ 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 7.2 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 7.1, the computed answer θ coming from the refutation is correct. Thus $\forall((A_1 \wedge \ldots \wedge A_k)\theta)$ is a logical consequence of P. It follows that $P \cup \{G\}$ is unsatisfiable. ■

Corollary 7.3 The success set of a definite program is contained in its least Herbrand model.

Proof Let the program be P, let $A \in B_P$ and suppose $P \cup \{\leftarrow A\}$ has a refutation. By theorem 7.1, A is a logical consequence of P. Thus A is in the least Herbrand model of P. ■

It is possible to strengthen corollary 7.3. We can show that if $A \in B_P$ 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].

If A is an atom, we put $[A] = \{A' \in B_P : A' = A\theta$, for some substitution $\theta\}$. Thus $[A]$ is the set of all ground instances of A. Equivalently, $[A]$ is $[A]_J$, where J is the Herbrand pre-interpretation.

Theorem 7.4 Let P be a definite program and G a definite goal $\leftarrow A_1,\dots,A_k$. Suppose that $P \cup \{G\}$ has an SLD-refutation of length n and $\theta_1,\dots,\theta_n$ is the sequence of mgu's of the SLD-refutation. Then we have that $\cup_{j=1}^{k}[A_j\theta_1\dots\theta_n] \subseteq T_P\uparrow n$.

Proof The result is proved by induction on the length of the refutation. Suppose first that n=1. Then G is a goal of the form $\leftarrow A_1$, the program has a unit clause of the form $A \leftarrow$ and $A_1\theta_1 = A\theta_1$. Clearly, $[A] \subseteq T_P\uparrow 1$ and so $[A_1\theta_1] \subseteq T_P\uparrow 1$.

Next suppose the result is true for refutations of length n−1 and consider a refutation of $P \cup \{G\}$ of length n. Let A_j be an atom of G. Suppose first that A_j is not the selected atom of G. Then $A_j\theta_1$ is an atom of G_1, the second goal of the refutation. The induction hypothesis implies that $[A_j\theta_1\theta_2\dots\theta_n] \subseteq T_P\uparrow(n-1)$ and $T_P\uparrow(n-1) \subseteq T_P\uparrow n$, by the monotonicity of T_P.

Now suppose that A_j is the selected atom of G. Let $B \leftarrow B_1,\dots,B_q$ $(q \geq 0)$ be the first input clause. Then $A_j\theta_1$ is an instance of B. If q=0, we have $[B] \subseteq T_P\uparrow 1$. Thus $[A_j\theta_1\dots\theta_n] \subseteq [A_j\theta_1] \subseteq [B] \subseteq T_P\uparrow 1 \subseteq T_P\uparrow n$. If q>0, by the induction hypothesis, $[B_i\theta_1\dots\theta_n] \subseteq T_P\uparrow(n-1)$, for i=1,...,q. By the definition of T_P, we have that $[A_j\theta_1\dots\theta_n] \subseteq T_P\uparrow n$. ■

Next we turn to the problem of the occur check. As we mentioned earlier, the occur check in the unification algorithm is very expensive and most PROLOG

systems leave it out for the pragmatic reason that it is only very rarely required. While this is certainly true, its omission can cause serious difficulties.

Example Consider the program

test ← p(x,x)

p(x,f(x)) ←

Given the goal ←test, a PROLOG system without the occur check will answer "yes" (equivalently, ε is a correct answer)! This answer is quite wrong because test is not a logical consequence of the program. The problem arises because, without the occur check, the unification algorithm of the PROLOG system will mistakenly unify p(x,x) and p(y,f(y)).

Thus we see that the lack of occur check has destroyed one of the principles on which logic programming is based – the soundness of SLD-resolution.

Example Consider the program

test ← p(x,x)

p(x,f(x)) ← p(x,x)

This time a PROLOG system without the occur check will go into an infinite loop in the unification algorithm because it will attempt to use a "circular" binding made in the second step of the computation.

These examples illustrate what can go wrong. We can distinguish two cases. The first case is when a circular binding is constructed in a "unification", but this binding is never used again. The first example illustrates this. The second case happens when an attempt is made to use a previously constructed circular binding in a step of the computation or in printing out an answer. The second example illustrates this. The first case is more insidious because there may be no indication that an error has occurred.

While these examples may appear artificial, it is important to appreciate that we can easily have such behaviour in practical programs. The most commonly encountered situation where this can occur is when programming with difference lists [21]. A difference list is a term of the form x−y, where − is a binary function (written infix). x−y represents the difference between the two lists x and y. For example, 34.56.12.x−x represents the list [34, 56, 12]. Similarly, x−x represents the empty list.

Let us say two difference lists x–y and z–w are compatible if y=z. Then compatible difference lists can be concatenated in constant' time using the following definition which comes from [21]

concat(x–y,y–z,x–z) ←

For example, we can concatenate 12.34.67.45.x–x and 36.89.y–y in one step to obtain 12.34.67.45.36.89.z–z. This is clearly a very useful technique. However, it is also dangerous in the absence of the occur check.

Example Consider the program

test ← concat(u–u,v–v,a.w–w)

concat(x–y,y–z,x–z) ←

Given the goal ←test, a PROLOG system without the occur check will answer "yes". In other words, it thinks that the concatenation of the empty list with the empty list is the list [a]!

Programs which use the difference list technique normally do not have an explicit concat predicate. Instead the concatenation is done implicitly. For example, the following clause is taken from such a version of quicksort [93].

Example Consider the program

qsort(nil,x–x) ←

Given the goal ←qsort(nil,a.y–y), a PROLOG system without the occur check will succeed on the goal (however, it will have a problem printing out its "answer", which contains the circular binding y/a.y).

It is possible to minimise the danger of an occur check problem by using a certain programming methodology. The idea is to "protect" programs which could cause problems by introducing an appropriate top-level predicate to restrict uses of the program to those which are known to be sound. This means that there must be some mechanism for forcing all calls to the program to go through this top-level predicate. However, with this method, the onus is still on the programmer and it thus remains suspect. A better idea [82] is to have a preprocessor which is able to identify which clauses may cause problems and add checking code to these clauses (or perhaps invoke the full unification algorithm when these clauses are used).

§8. COMPLETENESS OF SLD-RESOLUTION

The major result of this section is the completeness of SLD-resolution. We begin with two very useful lemmas.

Lemma 8.1 (Mgu Lemma)

Let P be a definite program and G a definite goal. Suppose that P ∪ {G} has an unrestricted SLD-refutation. Then P ∪ {G} has an SLD-refutation of the same length such that, if $\theta_1,\ldots,\theta_n$ are the unifiers from the unrestricted SLD-refutation and $\theta'_1,\ldots,\theta'_n$ are the mgu's from the SLD-refutation, then there exists a substitution γ such that $\theta_1\ldots\theta_n = \theta'_1\ldots\theta'_n\gamma$.

Proof The proof is by induction on the length of the unrestricted refutation. Suppose first that n=1. Thus P ∪ {G} has an unrestricted refutation G_0=G, $G_1=\square$ with input clause C_1 and unifier θ_1. Suppose θ'_1 is an mgu of the atom in G and the head of the unit clause C_1. Then $\theta_1 = \theta'_1\gamma$, for some γ. Furthermore, P ∪ {G} has a refutation G_0=G, $G_1=\square$ with input clause C_1 and mgu θ'_1.

Now suppose the result holds for n–1. Suppose P ∪ {G} has an unrestricted refutation G_0=G, $G_1,\ldots,G_n=\square$ of length n with input clauses $C_1,\ldots,C_n$ and unifiers $\theta_1,\ldots,\theta_n$. There exists an mgu θ'_1 for the selected atom in G and the head of C_1 such that $\theta_1 = \theta'_1\rho$, for some ρ. Thus P ∪ {G} has an unrestricted refutation G_0=G, G'_1, $G_2,\ldots,G_n=\square$ with input clauses $C_1,\ldots,C_n$ and unifiers θ'_1, $\rho\theta_2$, $\theta_3,\ldots,\theta_n$, where $G_1 = G'_1\rho$. By the induction hypothesis, P ∪ {G'_1} has a refutation G'_1, $G'_2,\ldots,G'_n=\square$ with mgu's $\theta'_2,\ldots,\theta'_n$ such that $\rho\theta_2\ldots\theta_n = \theta'_2\ldots\theta'_n\gamma$, for some γ. Thus P ∪ {G} has a refutation G_0=G, $G'_1,\ldots,G'_n=\square$ with mgu's $\theta'_1,\ldots,\theta'_n$ such that $\theta_1\ldots\theta_n = \theta'_1\rho\theta_2\ldots\theta_n = \theta'_1\ldots\theta'_n\gamma$. ■

Lemma 8.2 (Lifting Lemma)

Let P be a definite program, G a definite goal and θ a substitution. Suppose there exists an SLD-refutation of P ∪ {Gθ}. Then there exists an SLD-refutation of P ∪ {G} of the same length such that, if $\theta_1,\ldots,\theta_n$ are the mgu's from the SLD-refutation of P ∪ {Gθ} and $\theta'_1,\ldots,\theta'_n$ are the mgu's from the SLD-refutation of P ∪ {G}, then there exists a substitution γ such that $\theta\theta_1\ldots\theta_n = \theta'_1\ldots\theta'_n\gamma$.

Proof Suppose the first input clause for the refutation of P ∪ {Gθ} is C_1, the first mgu is θ_1 and G_1 is the goal which results from the first step. We may assume θ does not act on any variables of C_1. Now $\theta\theta_1$ is a unifier for the head of C_1 and the atom in G which corresponds to the selected atom in Gθ. The result of resolving G and C_1 using $\theta\theta_1$ is exactly G_1. Thus we obtain an unrestricted refutation of $P \cup \{G\}$, which looks exactly like the given refutation of $P \cup \{G\theta\}$, except the original goal is different, of course, and the first unifier is $\theta\theta_1$. Now apply the mgu lemma. ■

The first completeness result gives the converse to corollary 7.3. This result is due to Apt and van Emden [4].

Theorem 8.3 The success set of a definite program is equal to its least Herbrand model.

Proof Let the program be P. By corollary 7.3, it suffices to show that the least Herbrand model of P is contained in the success set of P. Suppose A is in the least Herbrand model of P. By theorem 6.5, $A \in T_P\uparrow n$, for some $n \in \omega$. We prove by induction on n that $A \in T_P\uparrow n$ implies that $P \cup \{\leftarrow A\}$ has a refutation and hence A is in the success set.

Suppose first that n=1. Then $A \in T_P\uparrow 1$ means that A is a ground instance of a unit clause of P. Clearly, $P \cup \{\leftarrow A\}$ has a refutation.

Now suppose that the result holds for n−1. Let $A \in T_P\uparrow n$. By the definition of T_P, there exists a ground instance of a clause $B \leftarrow B_1,\dots,B_k$ such that $A = B\theta$ and $\{B_1\theta,\dots,B_k\theta\} \subseteq T_P\uparrow(n-1)$, for some θ. By the induction hypothesis, $P \cup \{\leftarrow B_i\theta\}$ has a refutation, for i=1,...,k. Because each $B_i\theta$ is ground, these refutations can be combined into a refutation of $P \cup \{\leftarrow(B_1,\dots,B_k)\theta\}$. Thus $P \cup \{\leftarrow A\}$ has an unrestricted refutation and we can apply the mgu lemma to obtain a refutation of $P \cup \{\leftarrow A\}$. ■

The next completeness result was first proved by Hill [46]. See also [4].

Theorem 8.4 Let P be a definite program and G a definite goal. Suppose that $P \cup \{G\}$ is unsatisfiable. Then there exists an SLD-refutation of $P \cup \{G\}$.

Proof Let G be the goal $\leftarrow A_1,\dots,A_k$. Since $P \cup \{G\}$ is unsatisfiable, G is false wrt M_P. Hence some ground instance $G\theta$ of G is false wrt M_P. Thus $\{A_1\theta,\dots,A_k\theta\} \subseteq M_P$. By theorem 8.3, there is a refutation for $P \cup \{\leftarrow A_i\theta\}$, for i=1,...,k. Since each $A_i\theta$ is ground, we can combine these refutations into a refutation for $P \cup \{G\theta\}$. Finally, we apply the lifting lemma. ■

Next we turn attention to correct answers. It is not possible to prove the exact converse of theorem 7.1 because computed answers are always "most general".

However, we can prove that every correct answer is an instance of a computed answer.

Lemma 8.5 Let P be a definite program and A an atom. Suppose that $\forall(A)$ is a logical consequence of P. Then there exists an SLD-refutation of $P \cup \{\leftarrow A\}$ with the identity substitution as the computed answer.

Proof Suppose A has variables $x_1,\dots,x_n$. Let $a_1,\dots,a_n$ be distinct constants not appearing in P or A and let θ be the substitution $\{x_1/a_1,\dots,x_n/a_n\}$. Then it is clear that $A\theta$ is a logical consequence of P. Since $A\theta$ is ground, theorem 8.3 shows that $P \cup \{\leftarrow A\theta\}$ has a refutation. Since the a_i do not appear in P or A, by replacing a_i by x_i (i=1,...,n) in this refutation, we obtain a refutation of $P \cup \{\leftarrow A\}$ with the identity substitution as the computed answer. ■

Now we are in a position to prove the major completeness result. This result is due to Clark [16].

Theorem 8.6 (Completeness of SLD-Resolution)

Let P be a definite program and G a definite goal. For every correct answer θ for $P \cup \{G\}$, there exists a computed answer σ for $P \cup \{G\}$ and a substitution γ such that $\theta = \sigma\gamma$.

Proof Suppose G is the goal $\leftarrow A_1,\dots,A_k$. Since θ is correct, $\forall((A_1\wedge\dots\wedge A_k)\theta)$ is a logical consequence of P. By lemma 8.5, there exists a refutation of $P \cup \{\leftarrow A_i\theta\}$ such that the computed answer is the identity, for i=1,...,k. We can combine these refutations into a refutation of $P \cup \{G\theta\}$ such that the computed answer is the identity.

Suppose the sequence of mgu's of the refutation of $P \cup \{G\theta\}$ is $\theta_1,\dots,\theta_n$. Then $G\theta\theta_1\dots\theta_n = G\theta$. By the lifting lemma, there exists a refutation of $P \cup \{G\}$ with mgu's $\theta'_1,\dots,\theta'_n$ such that $\theta\theta_1\dots\theta_n = \theta'_1\dots\theta'_n\gamma'$, for some substitution γ'. Let σ be $\theta'_1\dots\theta'_n$ restricted to the variables in G. Then $\theta = \sigma\gamma$, where γ is an appropriate restriction of γ'. ■

§9. INDEPENDENCE OF THE COMPUTATION RULE

In this section, we introduce the concept of a computation rule, which is used to select atoms in an SLD-derivation. We show that, for any choice of computation rule, if $P \cup \{G\}$ is unsatisfiable, we can always find a refutation