J. W. Lloyd

Foundations of Logic Programming

Second, Extended Edition

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo

CONTENTS

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 $A \leftarrow B_1,...,B_n$ is regarded as a *procedure definition*. If $\leftarrow C_1,...,C_k$ 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 $\leftarrow C_1,...,C_k$, a step in the computation involves unifying some C_j with the head A of a program clause $A \leftarrow B_1,...,B_n$ and thus reducing the current goal to the goal $\leftarrow (C_1,...,C_{j-1},B_1,...,B_n,C_{j+1},...,C_k)\theta$, 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 $\leftarrow B_1,...,B_n$ 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 (~F), (F∧G), (F∨G), (F→G) and (F↔G).
(c) If F is a formula and x is a variable, then (∀x F) and (∃x F) are formulas.

It will often be convenient to write the formula (F→G) as (G←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 (∀x (∃y (p(x,y)→q(x)))), (~(∃x (p(x,a)∧q(f(x))))) and (∀x (p(x,g(x))←(q(x)∧(~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 ∀x∃y (p(x,y)→q(x)), ~∃x (p(x,a)∧q(f(x))) and ∀x (p(x,g(x))←q(x)∧~r(x)). We will simplify formulas in this way wherever possible.

The informal semantics of the quantifiers and connectives is as follows. ~ is negation, ∧ is conjunction (and), ∨ is disjunction (or), → is implication and ↔ is equivalence. Also, ∃ is the existential quantifier, so that "∃x" means "there exists an x", while ∀ is the universal quantifier, so that "∀x" means "for all x". Thus the informal semantics of ∀x (p(x,g(x)) ← q(x)∧~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 ∀x (resp. ∃x) in ∀x F (resp. ∃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 ∃x p(x,y)∧q(x), the first two occurrences of x are bound, while the third occurrence is free, since the scope of ∃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

$$\forall x_1\ldots\forall x_s\,(L_1\vee\ldots\vee L_m)$$

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

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

$$\forall x_1 ... \forall x_s \, (A_1 \vee ... \vee A_k \vee \neg B_1 \vee ... \vee \neg B_n)$$

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

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

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

$$\forall x_1 ... \forall x_s \, (A_1 \vee ... \vee A_k \vee \neg B_1 \vee ... \vee \neg B_n)$$

is equivalent to

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

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

$$A \leftarrow B_1,...,B_n$$

which contains precisely one atom (viz. A) in its consequent. A is called the *head* and $B_1,...,B_n$ 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 $A \leftarrow B_1,...,B_n$ is "for each assignment of each variable, if $B_1,...,B_n$ 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

$$\forall y_1 ... \forall y_r \, (\neg B_1 \vee ... \vee \neg B_n)$$

or, equivalently,

$$\neg \exists y_1 ... \exists y_r \, (B_1 \wedge ... \wedge B_n)$$

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

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

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

$$\leftarrow B_1,...,B_n$$

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

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

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 $y_1,...,y_r$, 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 ←sort(17.22.6.5.nil,y) as a request to prove that ∃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 $t'_1,...,t'_n$ are the term assignments of $t_1,...,t_n$ and f' is the assignment of the n-ary function symbol f, then $f'(t'_1,...,t'_n) \in D$ is the term assignment of $f(t_1,...,t_n)$.

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 $p(t_1,...,t_n)$ and $d_1,...,d_n$ in the domain of J are the term assignments of $t_1,...,t_n$ wrt J and V. We call $A_{J,V} = p(d_1,...,d_n)$ 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 $p(d_1,...,d_n)$ a *J-instance*.

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

(b) If the formula has the form $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$ or $F \leftrightarrow G$, then the truth value of the formula is given by the following table:

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
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 $\exists x$ F, then the truth value of the formula is true if there exists $d \in D$ 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 $\forall x$ F, then the truth value of the formula is true if, for all $d \in D$, 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 $\exists$(W) is true wrt I.

We say W is *valid in* I if $\forall$(W) is true wrt I.

We say W is *unsatisfiable in* I if $\exists$(W) is false wrt I.

We say W is *nonvalid in* I if $\forall$(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 $\forall x \exists y\ p(x,y)$ 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 $S \cup \{\neg F\}$. Thus $S \cup \{\neg F\}$ is unsatisfiable.

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

Example 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.

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 $\leq$. 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 $\varnothing$ (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

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

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 $\tau_1 \times ... \times \tau_n$ and function symbols have types of the form $\tau_1 \times ... \times \tau_n \rightarrow \tau$. If f has type $\tau_1 \times ... \times \tau_n \rightarrow \tau$, 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 $\tau_1 \times ... \times \tau_n \rightarrow \tau$ and t_i is a term of type τ_i (i=1,...,n), then $f(t_1,...,t_n)$ 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 $\tau_1 \times ... \times \tau_n$ and t_i is a term of type τ_i (i=1,...,n), then $p(t_1,...,t_n)$ 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$\rightarrow$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 $\tau_1 \times ... \times \tau_n \rightarrow \tau$ in L, the assignment of a mapping from $D_{\tau_1} \times ... \times D_{\tau_n}$ 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 $E\theta$, the *instance* of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variable v_i in E by the term t_i ($i\approx 1,...,n$). If $E\theta$ is ground, then $E\theta$ is called a *ground instance* of E.

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

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

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

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

by deleting any 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 $\theta = \{x/f(y), y/z\}$ and $\sigma = \{x/a, y/b, z/y\}$. Then $\theta\sigma = \{x/f(b), z/y\}$.

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

We denote the identity substitution by ε. Note that $E\varepsilon = E$, 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) $\theta\varepsilon = \varepsilon\theta = \theta$.

(b) $(E\theta)\sigma = E(\theta\sigma)$, for all expressions E.

(c) $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

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 $\theta_1...\theta_n$ 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 $\{x_1/y_1,...,x_n/y_n\}$ such that $\{x_1,...,x_n\} \subseteq V$, the y_i are distinct and $(V \setminus \{x_1,...,x_n\}) \cap \{y_1,...,y_n\} = \varnothing$.

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

It follows from the definition of an mgu that if θ and σ are both mgu's of $\{E_1,...,E_n\}$, 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}, σ_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}, σ_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)}, σ_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)}, σ_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}, σ_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 $k \geq 0$, 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 $k \geq 0$, 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 $\gamma_{k+1} = \gamma_k \backslash \{v/v\gamma_k\}$. 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 $S \approx \{p(x_1,...,x_n), p(f(x_0,x_0),...,f(x_{n-1},x_{n-1}))\}$. Then $\sigma_1 = \{x_1/f(x_0,x_0)\}$ and $S\sigma_1 = \{p(f(x_0,x_0),x_2,...,x_n), p(f(x_0,x_0),f(f(x_0,x_0),f(x_0,x_0)),f(x_2,x_2),...,f(x_{n-1},x_{n-1}))\}$. 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 ($1 \le k \le n$). 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 $\varnothing$.

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 $T(\text{lub}(X)) = \text{lub}(T(X))$, 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 $\varnothing$. Then we define $1 = \{\varnothing\} = \{0\}$, $2 = \{\varnothing, \{\varnothing\}\} = \{0, 1\}$, $3 = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\} = \{0, 1, 2\}$, 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

$T\uparrow\alpha = \text{lub}\{T\uparrow\beta : \beta<\alpha\}$, if α is a limit ordinal

$T\downarrow 0 = T$

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

$T\downarrow\alpha = \text{glb}\{T\downarrow\beta : \beta<\alpha\}$, 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, $T\uparrow\alpha \le \text{lfp}(T)$ and $T\downarrow\alpha \ge \text{gfp}(T)$. Furthermore, there exist ordinals β_1 and β_2 such that $\gamma_1 \ge \beta_1$ implies $T\uparrow\gamma_1 = \text{lfp}(T)$ and $\gamma_2 \ge \beta_2$ implies $T\downarrow\gamma_2 = \text{gfp}(T)$.

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

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

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

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

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

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

The least α such that $T\uparrow\alpha = lfp(T)$ 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 $\leq \omega$.

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

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 $T(T\uparrow\omega) = T(lub\{T\uparrow n : n\in\omega\}) = lub\{T(T\uparrow n) : n\in\omega\} = T\uparrow\omega$, using the continuity of T. ■

The analogue of proposition 5.4 for gfp(T) does not hold, that is, 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) $\forall x \forall y(r(x,y) \to \neg p(x,y))$

(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 $\theta = \{x_1/t_1,...,x_n/t_n\}$ and suppose V is the set of variables occurring in terms in $\{t_1,...,t_n\}$. Show that θ is idempotent iff $\{x_1,...,x_n\} \cap V = \varnothing$.

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 $T : L \to L$ such that T is monotonic but not continuous.

13 Let L be a complete lattice and $T : L \to L$ 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

$$A \leftarrow B_1,...,B_n$$

which contains precisely one atom (viz. A) in its consequent. A is called the *head* and $B_1,...,B_n$ 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

$$\leftarrow B_1,...,B_n$$

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 $P \cup \{\neg A\}$ is unsatisfiable, by proposition 3.1
iff $P \cup \{\neg A\}$ has no Herbrand models, by proposition 3.3
iff $\neg A$ is false wrt all Herbrand models of P
iff A is true wrt all Herbrand models of P
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 $T_P : 2^{B_P} \rightarrow 2^{B_P}$ is defined as follows. Let I be an Herbrand interpretation. Then $T_P(I) = \{A \in B_P : A \leftarrow A_1,...,A_n \text{ is a ground instance of a clause in P and } \{A_1,...,A_n\} \subseteq I\}$.

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
$p(f(x)) \leftarrow p(x)$
$q(a) \leftarrow p(x)$
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 $\{A_1,...,A_n\} \subseteq lub(X)$ iff $\{A_1,...,A_n\} \subseteq I$, for some $I \in X$. (See problem 3.) In order to show T_P is continuous, we have to show $T_P(lub(X)) = lub(T_P(X))$, for each directed subset X.

Now we have that
$A \in T_P(\text{lub}(X))$
iff $A \leftarrow A_1,...,A_n$ is a ground instance of a clause in P and $\{A_1,...,A_n\} \subseteq \text{lub}(X)$
iff $A \leftarrow A_1,...,A_n$ is a ground instance of a clause in P and $\{A_1,...,A_n\} \subseteq I$, for some $I \in X$
iff $A \in T_P(I)$, for some $I \in X$
iff $A \in \text{lub}(T_P(X))$. ■

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 $A \leftarrow A_1,...,A_n$ of each clause in P, we have $\{A_1,...,A_n\} \subseteq I$ 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 $M_P = \text{lfp}(T_P) = T_P\uparrow\omega$.

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

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

Example Consider the program P
$p(f(x)) \leftarrow p(x)$
$q(a) \leftarrow p(x)$
Then $T_P\downarrow\omega = \{q(a)\}$, but $\text{gfp}(T_P) = \varnothing$. In fact, $\text{gfp}(T_P) = T_P\downarrow(\omega+1)$.

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

Using proposition 3.1, we see that θ is a correct answer iff $P \cup \{\neg\forall((A_1 \wedge ... \wedge A_k)\theta)\}$ 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 $\forall((A_1 \wedge ... \wedge A_k)\theta)$ 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 $\neg\forall x\ p(x)$ is not a clause and hence we cannot restrict attention to Herbrand interpretations when attempting to establish the unsatisfiability of $\{p(a)\leftarrow\} \cup \{\neg\forall x\ p(x)\}$. 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 $\leftarrow A_1,...,A_k$. Suppose θ is an answer for $P \cup \{G\}$ such that $(A_1 \wedge ... \wedge A_k)\theta$ is ground. Then the following are equivalent:
(a) θ is correct.
(b) $(A_1 \wedge ... \wedge A_k)\theta$ is true wrt every Herbrand model of P.
(c) $(A_1 \wedge ... \wedge A_k)\theta$ is true wrt the least Herbrand model of P.

Proof Obviously, it suffices to show that (c) implies (a). Now
$(A_1 \wedge ... \wedge A_k)\theta$ is true wrt the least Herbrand model of P
implies $(A_1 \wedge ... \wedge A_k)\theta$ is true wrt all Herbrand models of P
implies $\neg(A_1 \wedge ... \wedge A_k)\theta$ is false wrt all Herbrand models of P
implies $P \cup \{\neg(A_1 \wedge ... \wedge A_k)\theta\}$ has no Herbrand models
implies $P \cup \{\neg(A_1 \wedge ... \wedge A_k)\theta\}$ 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 $\leftarrow A_1,...,A_m,...,A_k$ and C be $A \leftarrow B_1,...,B_q$. 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 $\leftarrow(A_1,...,A_{m-1},B_1,...,B_q,A_{m+1},...,A_k)\theta$.

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,... of goals, a sequence C_1, C_2,... of variants of program clauses of P and a sequence θ_1, θ_2,... 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,... 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 $\square$ as the last goal in the derivation. If $G_n = \square$, 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 $atom_i$ 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).

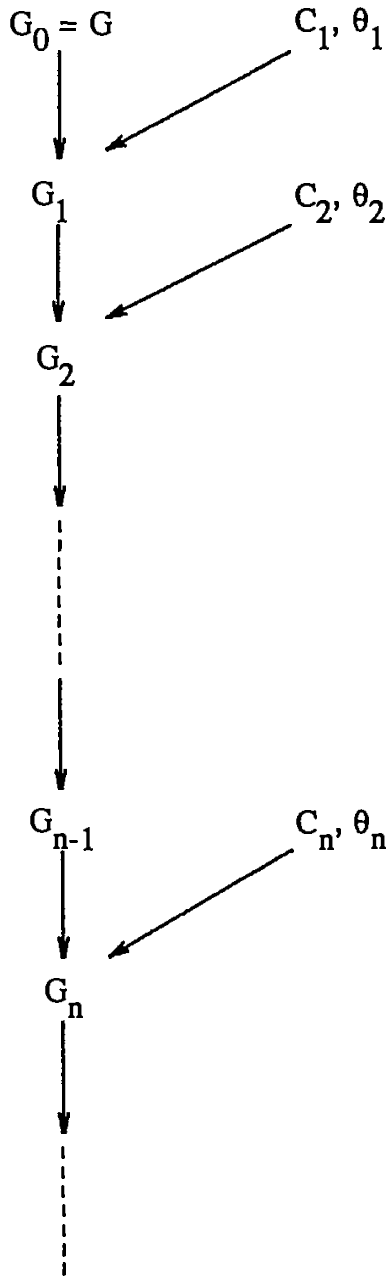

Fig. 1. An SLD-derivation

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 $\theta_1...\theta_n$ to the variables of G, where $\theta_1,...,\theta_n$ 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 ←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 $\leftarrow A_1,...,A_k$ and $\theta_1,...,\theta_n$ be the sequence of mgu's used in a refutation of $P \cup \{G\}$. We have to show that $\forall((A_1 \wedge ... \wedge A_k)\theta_1...\theta_n)$ 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 $\theta_1,...,\theta_n$ is the sequence of mgu's used in a refutation of $P \cup \{G\}$ of length n. Let $A\leftarrow B_1,...,B_q$ $(q\geq 0)$ be the first input clause and A_m the selected atom of G. By the induction hypothesis, $\forall((A_1 \wedge ... \wedge A_{m-1} \wedge B_1 \wedge ... \wedge B_q \wedge A_{m+1} \wedge ... \wedge A_k)\theta_1...\theta_n)$ is a logical consequence of P. Thus, if q>0, $\forall((B_1 \wedge ... \wedge B_q)\theta_1...\theta_n)$ is a logical consequence of P. Consequently, $\forall(A_m\theta_1...\theta_n)$, which is the same as $\forall(A\theta_1...\theta_n)$, is a logical consequence of P. Hence $\forall((A_1 \wedge ... \wedge A_k)\theta_1...\theta_n)$ 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 $\leftarrow A_1,...,A_k$. By theorem 7.1, the computed answer θ coming from the refutation is correct. Thus $\forall((A_1 \wedge ... \wedge A_k)\theta)$ 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 $\leftarrow A_1,...,A_k$. Suppose that $P \cup \{G\}$ has an SLD-refutation of length n and $\theta_1,...,\theta_n$ is the sequence of mgu's of the SLD-refutation. Then we have that $\cup_{j=1}^{k} [A_j\theta_1...\theta_n] \subseteq T_P \uparrow n$.

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 $[A_j\theta_1\theta_2...\theta_n] \subseteq T_P \uparrow (n-1)$ 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 $B\leftarrow B_1,...,B_q$ $(q \geq 0)$ 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 $[A_j\theta_1...\theta_n] \subseteq [A_j\theta_1] \subseteq [B] \subseteq T_P \uparrow 1 \subseteq T_P \uparrow n$. If q>0, by the induction hypothesis, $[B_i\theta_1...\theta_n] \subseteq T_P \uparrow (n-1)$, for i=1,...,q. By the definition of T_P, we have that $[A_j\theta_1...\theta_n] \subseteq T_P \uparrow n$. ■

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

Proof The proof is by induction on the length of the unrestricted refutation. Suppose first that n=1. Thus $P \cup \{G\}$ 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, $P \cup \{G\}$ 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 $P \cup \{G\}$ has an unrestricted refutation $G_0=G, G_1,...,G_n=\square$ of length n with input clauses $C_1,...,C_n$ and unifiers $\theta_1,...,\theta_n$. 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 $P \cup \{G\}$ has an unrestricted refutation $G_0=G, G'_1, G_2,...,G_n=\square$ with input clauses $C_1,...,C_n$ and unifiers $\theta'_1, \rho\theta_2, \theta_3,...,\theta_n$, where $G_1 = G'_1\rho$. By the induction hypothesis, $P \cup \{G'_1\}$ has a refutation $G'_1, G'_2,...,G'_n=\square$ with mgu's $\theta'_2,...,\theta'_n$ such that $\rho\theta_2...\theta_n = \theta'_2...\theta'_n\gamma$, for some γ. Thus $P \cup \{G\}$ has a refutation $G_0=G, G'_1,...,G'_n=\square$ with mgu's $\theta'_1,...,\theta'_n$ such that $\theta_1...\theta_n = \theta'_1\rho\theta_2...\theta_n = \theta'_1...\theta'_n\gamma$. ■

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 $P \cup \{G\theta\}$. Then there exists an SLD-refutation of $P \cup \{G\}$ of the same length such that, if $\theta_1,...,\theta_n$ are the mgu's from the SLD-refutation of $P \cup \{G\theta\}$ and $\theta'_1,...,\theta'_n$ are the mgu's from the SLD-refutation of $P \cup \{G\}$, then there exists a substitution γ such that $\theta\theta_1...\theta_n = \theta'_1...\theta'_n\gamma$.

Proof Suppose the first input clause for the refutation of $P \cup \{G\theta\}$ 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 $G\theta$. 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 $B \leftarrow B_1,\ldots,B_k$ such that $A=B\theta$ and $\{B_1\theta,\ldots,B_k\theta\} \subseteq T_P\uparrow(n-1)$, 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 $P \cup \{\leftarrow (B_1,\ldots,B_k)\theta\}$. 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 $\leftarrow A_1,\ldots,A_k$. 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 $\{A_1\theta,\ldots,A_k\theta\} \subseteq M_P$. 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 $x_1,...,x_n$. Let $a_1,...,a_n$ be distinct constants not appearing in P or A and let θ be the substitution $\{x_1/a_1,...,x_n/a_n\}$. 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 $\leftarrow A_1,...,A_k$. Since θ is correct, $\forall((A_1 \wedge ... \wedge A_k)\theta)$ 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 $\theta_1,...,\theta_n$. Then $G\theta\theta_1...\theta_n = G\theta$. By the lifting lemma, there exists a refutation of $P \cup \{G\}$ with mgu's $\theta'_1,...,\theta'_n$ such that $\theta\theta_1...\theta_n = \theta'_1...\theta'_n\gamma'$, for some substitution γ'. Let σ be $\theta'_1...\theta'_n$ 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

using the given computation rule. This fact is called the "independence" of the computation rule. We also prove that every computable function can be computed by a definite program.

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

Definition Let P be a definite program, G a definite goal and R a computation rule. An *SLD-derivation* of $P \cup \{G\}$ *via R* is an SLD-derivation of $P \cup \{G\}$ in which the computation rule R is used to select atoms.

It is important to realise that using a computation rule to select atoms in an SLD-derivation is actually a restriction, in the sense that, if the same goal occurs in different places, then the computation rule will always select the *same* atom of that goal. In other words, there are SLD-derivations which are not SLD-derivations via R, for any computation rule R.

Definition Let P be a definite program, G a definite goal and R a computation rule. An *SLD-refutation* of $P \cup \{G\}$ *via R* is an SLD-refutation of $P \cup \{G\}$ in which the computation rule R is used to select atoms.

Definition Let P be a definite program, G a definite goal and R a computation rule. An *R-computed answer* for $P \cup \{G\}$ is a computed answer for $P \cup \{G\}$ which has come from an SLD-refutation of $P \cup \{G\}$ via R.

Now we are in a position to consider the independence result. According to theorem 8.4, if $P \cup \{G\}$ is unsatisfiable, then there exists a refutation of $P \cup \{G\}$. In fact, we will show that, for any computation rule R, there is actually a refutation of $P \cup \{G\}$ *via R*. This result means that, in principle, a logic programming system can use any computation rule it finds convenient. We will explore the practical consequences of this result in §10.

The key to the independence result is a technical lemma. For this, it will be convenient to introduce some new notation. If C is a definite program clause, then C^+ denotes the head of the clause and C^- denotes the body.

Lemma 9.1 (Switching Lemma)

Let P be a definite program and G a definite goal. Suppose that $P \cup \{G\}$ has an SLD-refutation $G_0=G, G_1,\ldots,G_{n-1}, G_n, G_{n+1},\ldots,G_n=\square$ with input clauses

$C_1,...,C_n$ and mgu's $\theta_1,...,\theta_n$. Suppose that

G_{q-1} is $\leftarrow A_1,...,A_{i-1},A_i,...,A_{j-1},A_j,...,A_k$

G_q is $\leftarrow (A_1,...,A_{i-1},C_q^-,...,A_{j-1},A_j,...,A_k)\theta_q$

G_{q+1} is $\leftarrow (A_1,...,A_{i-1},C_q^-,...,A_{j-1},C_{q+1}^-,...,A_k)\theta_q\theta_{q+1}$.

Then there exists an SLD-refutation of $P \cup \{G\}$ in which A_j is selected in G_{q-1} instead of A_i and A_i is selected in G_q instead of A_j. Furthermore, if σ is the computed answer for $P \cup \{G\}$ from the given refutation and σ' is the computed answer for $P \cup \{G\}$ from the new refutation, then $G\sigma$ is a variant of $G\sigma'$.

Proof We have $A_j\theta_q\theta_{q+1} = C_{q+1}^+\theta_{q+1} = C_{q+1}^+\theta_q\theta_{q+1}$. Thus we can unify A_j and C_{q+1}^+. Let θ'_q be an mgu of A_j and C_{q+1}^+. Thus $\theta_q\theta_{q+1} = \theta'_q\sigma$, for some substitution σ. Clearly, we can assume that θ'_q does not act on any of the variables of C_q.

Furthermore, $C_q^+\sigma = C_q^+\theta'_q\sigma = C_q^+\theta_q\theta_{q+1} = A_i\theta_q\theta_{q+1} = A_i\theta'_q\sigma$. Hence we can unify C_q^+ and $A_i\theta'_q$. Suppose θ'_{q+1} is an mgu. Thus $\sigma = \theta'_{q+1}\sigma'$, for some σ'. Consequently, $\theta_q\theta_{q+1} = \theta'_q\theta'_{q+1}\sigma'$. We have now shown that A_i and A_j can be selected in the reverse order.

Next, note that $A_i\theta'_q\theta'_{q+1} = C_q^+\theta'_q\theta'_{q+1}$, but that θ_q is an mgu of A_i and C_q^+. Thus $\theta'_q\theta'_{q+1} = \theta_q\gamma$, for some γ. But $A_j\theta_q\gamma = A_j\theta'_q\theta'_{q+1} = C_{q+1}^+\theta'_q\theta'_{q+1} = C_{q+1}^+\theta_q\gamma = C_{q+1}^+\gamma$. Thus γ unifies $A_j\theta_q$ and C_{q+1}^+, and so $\gamma = \theta_{q+1}\sigma''$, for some σ''. Consequently, $\theta'_q\theta'_{q+1} = \theta_q\theta_{q+1}\sigma''$ and so the (q+1)st goal in the new refutation is a variant of G_{q+1}.

The remainder of the new refutation now proceeds in the same way as the given refutation (modulo variants) and the result follows. ■

Theorem 9.2 (Independence of the Computation Rule).

Let P be a definite program and G a definite goal. Suppose there is an SLD-refutation of $P \cup \{G\}$ with computed answer σ. Then, for any computation rule R, there exists an SLD-refutation of $P \cup \{G\}$ via R with R-computed answer σ' such that $G\sigma'$ is a variant of $G\sigma$.

Proof Apply the switching lemma repeatedly. (See problem 15.) ■

We can use theorem 9.2 to strengthen theorems 8.3, 8.4 and 8.6.

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

Theorem 9.3 Let P be a definite program and R a computation rule. Then the R-success set of P is equal to its least Herbrand model.

Proof The theorem follows immediately from theorems 8.3 and 9.2. ■

Theorem 9.4 Let P be a definite program, G a definite goal and R a computation rule. Suppose that $P \cup \{G\}$ is unsatisfiable. Then there exists an SLD-refutation of $P \cup \{G\}$ via R.

Proof The theorem follows immediately from theorems 8.4 and 9.2. ■

Theorem 9.5 (Strong Completeness of SLD-Resolution)
Let P be a definite program, G a definite goal and R a computation rule. Then for every correct answer θ for $P \cup \{G\}$, there exists an R-computed answer σ for $P \cup \{G\}$ and a substitution γ such that $\theta = \sigma\gamma$.

Proof The theorem follows immediately from theorems 8.6 and 9.2. ■

Theorem 9.4 is due to Hill [46]. See also [4]. Theorem 9.5 is due to Clark [16].

We now establish the important result that every computable function can be computed by an appropriate definite program. There are a number of ways of establishing this result, depending on the definition of "computable" chosen. For example, Tarnlund [102] showed that every Turing computable function can be computed by a definite program. Shepherdson established the result using unlimited register machines to define computable functions [96]. Kowalski [52] established the result by showing how to transform a set of recursive equations into a definite program. Andreka and Nemeti [1] and Sonenberg and Topor [100] show the adequacy of definite programs for computation over an Herbrand universe. Here, we follow Sebelik and Stepanek [91] by showing that every partial recursive function can be computed by a definite program. The definition of a

partial recursive function and the basic results of computability are contained in [23], for example. For a survey of these computability results, see [100].

Theorem 9.6 (Computational Adequacy of Definite Programs)

Let f be an n-ary partial recursive function. Then there exists a definite program P_f and an (n+1)-ary predicate symbol p_f such that all computed answers for $P_f \cup \{\leftarrow p_f(s^{k_1}(0),...,s^{k_n}(0),x)\}$ have the form $\{x/s^k(0)\}$ and, for all non-negative integers $k_1,...,k_n$ and k, we have $f(k_1,...,k_n)=k$ iff $\{x/s^k(0)\}$ is a computed answer for $P_f \cup \{\leftarrow p_f(s^{k_1}(0),...,s^{k_n}(0),x)\}$.

Proof In the program P_f, a non-negative integer k is represented by the term $s^k(0)$, where s represents the successor function. By theorem 9.2, we can suppose that all computed answers are R-computed, where R is the computation rule which always selects the leftmost atom. The result is proved by induction on the number q of applications of composition, primitive recursion and minimalisation needed to define f.

Suppose first that q=0. Thus f must be either the zero function, the successor function or a projection function.

Zero function

Suppose that f is the zero function defined by f(x)=0. Define P_f to be the program $p_f(x,0)\leftarrow$.

Successor function

Suppose that f is the successor function defined by f(x)=x+1. Define P_f to be the program $p_f(x,s(x))\leftarrow$.

Projection functions

Suppose that f is the projection function defined by $f(x_1,...,x_n)=x_j$, where $1\leq j\leq n$. Define P_f to be the program $p_f(x_1,...,x_n,x_j)\leftarrow$.

Clearly, for each of the basic functions, the program P_f defined has the desired properties.

Next suppose the partial recursive function f is defined by q (q>0) applications of composition, primitive recursion and minimalisation.

Composition

Suppose that f is defined by $f(x_1,...,x_n) = h(g_1(x_1,...,x_n),...,g_m(x_1,...,x_n))$, where $g_1,...,g_m$ and h are partial recursive functions. By the induction hypothesis, corresponding to each g_i, there is a definite program P_{g_i} and a predicate symbol p_{g_i} satisfying the properties of the theorem. Similarly, corresponding to h, there is a

definite program P_h and a predicate symbol p_h satisfying the properties of the theorem. We can suppose that the programs $P_{g_1},...,P_{g_m}$ and P_h do not have any predicate symbols in common. Define P_f to be the union of these programs together with the clause

$p_f(x_1,...,x_n,z) \leftarrow p_{g_1}(x_1,...,x_n,y_1),...,p_{g_m}(x_1,...,x_n,y_m), p_h(y_1,...,y_m,z)$

Clearly all computed answers for $P_f \cup \{\leftarrow p_f(s^{k_1}(0),...,s^{k_n}(0),z)\}$ have the form $\{z/s^k(0)\}$, using the induction hypothesis.

Now suppose that $f(k_1,...,k_n)=k$. Thus $g_i(k_1,...,k_n)=n_i$, say, for $1 \leq i \leq m$. By the induction hypothesis, $\{y_i/s^{n_i}(0)\}$ is a computed answer for $P_{g_i} \cup \{\leftarrow p_{g_i}(s^{k_1}(0),...,s^{k_n}(0),y_i)\}$. Also, by the induction hypothesis, $\{z/s^k(0)\}$ is a computed answer for $P_h \cup \{\leftarrow p_h(s^{n_1}(0),...,s^{n_m}(0),z)\}$. Hence $\{z/s^k(0)\}$ is a computed answer for $P_f \cup \{\leftarrow p_f(s^{k_1}(0),...,s^{k_n}(0),z)\}$.

Conversely, suppose that $\{z/s^k(0)\}$ is a computed answer for $P_f \cup \{\leftarrow p_f(s^{k_1}(0),...,s^{k_n}(0),z)\}$. From the refutation giving this answer, we can extract computed answers $\{y_i/s^{n_i}(0)\}$ for $P_{g_i} \cup \{\leftarrow p_{g_i}(s^{k_1}(0),...,s^{k_n}(0),y_i)\}$, for $1 \leq i \leq m$, and a computed answer $\{z/s^k(0)\}$ for $P_h \cup \{\leftarrow p_h(s^{n_1}(0),...,s^{n_m}(0),z)\}$. It now follows from the induction hypothesis that $g_i(k_1,...,k_n)=n_i$, for $1 \leq i \leq m$, and that $h(n_1,...,n_m)=k$. Hence $f(k_1,...,k_n)=k$.

Primitive recursion

Suppose that f is defined by

$f(x_1,...,x_n,0) = h(x_1,...,x_n)$

$f(x_1,...,x_n,y+1) = g(x_1,...,x_n,y,f(x_1,...,x_n,y))$,

where h and g are partial recursive functions. By the induction hypothesis, corresponding to h (resp., g), there is a definite program P_h (resp., P_g) and a predicate symbol p_h (resp., p_g) satisfying the properties of the theorem. We can also suppose that P_h and P_g do not have any predicate symbols in common. Define P_f to be the union of P_h and P_g together with the clauses

$p_f(x_1,...,x_n,0,z) \leftarrow p_h(x_1,...,x_n,z)$

$p_f(x_1,...,x_n,s(y),z) \leftarrow p_f(x_1,...,x_n,y,u), p_g(x_1,...,x_n,y,u,z)$.

An argument similar to the one for composition shows that P_f has the desired properties.

Minimalisation

Suppose that f is defined by $f(x_1,...,x_n) = \mu y(g(x_1,...,x_n,y)=0)$, where g is a partial recursive function. That is, $\mu y(g(x_1,...,x_n,y)=0)$ is the least y such that $g(x_1,...,x_n,z)$ is defined for all $z \leq y$ and $g(x_1,...,x_n,y)=0$, if such a y exists;

otherwise, $\mu y(g(x_1,...,x_n,y)=0)$ is undefined. By the induction hypothesis, corresponding to g, there is a definite program P_g and a predicate symbol p_g satisfying the properties of the theorem. Define P_f to be P_g together with the clauses

$p_f(x_1,...,x_n,y) \leftarrow p_g(x_1,...,x_n,0,u), r(x_1,...,x_n,0,u,y)$
$r(x_1,...,x_n,y,0,y) \leftarrow$
$r(x_1,...,x_n,y,s(v),z) \leftarrow p_g(x_1,...,x_n,s(y),u), r(x_1,...,x_n,s(y),u,z).$

An argument similar to the one for composition shows that P_f has the desired properties. ■

§10. SLD-REFUTATION PROCEDURES

In this section, we consider the possible strategies a logic programming system might adopt in its search for a refutation. We show that the use of a depth-first search strategy has serious implications with regard to completeness. We also briefly discuss the automatic generation of control.

The search space is a certain type of tree, called an SLD-tree. The results of §9 show that in building the SLD-tree, the system does not have to consider alternative computation rules. A computation rule can be fixed in advance and an SLD-tree constructed using this computation rule. This dramatically reduces the size of the search space.

Definition Let P be a definite program and G a definite goal. An *SLD-tree* for $P \cup \{G\}$ is a tree satisfying the following:

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

(b) The root node is G.

(c) Let $\leftarrow A_1,...,A_m,...,A_k$ ($k \geq 1$) be a node in the tree and suppose that A_m is the selected atom. Then, for each input clause $A \leftarrow B_1,...,B_q$ such that A_m and A are unifiable with mgu θ, the node has a child

$$\leftarrow (A_1,...,A_{m-1},B_1,...,B_q,A_{m+1},...,A_k)\theta$$

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

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

Definition Let P be a definite program, G a definite goal and R a computation rule. The *SLD-tree* for $P \cup \{G\}$ *via R* is the SLD-tree for $P \cup \{G\}$ in which the atoms selected are those selected by R.

Example Consider the program

1. $p(x,z) \leftarrow q(x,y), p(y,z)$
2. $p(x,x) \leftarrow$
3. $q(a,b) \leftarrow$

and the goal $\leftarrow p(x,b)$. Figures 2 and 3 show two SLD-trees for this program and goal. The SLD-tree in Figure 2 comes from the standard PROLOG computation rule (select the leftmost atom). The SLD-tree in Figure 3 comes from the computation rule which always selects the rightmost atom. The selected atoms are underlined and the success, failure and infinite branches are shown. Note that the first tree is finite, while the second tree is infinite. Each tree has two success branches corresponding to the answers {x/a} and {x/b}.

This example shows that the choice of computation rule has a great bearing on the size and structure of the corresponding SLD-tree. However, no matter what the choice of computation rule, if $P \cup \{G\}$ is unsatisfiable, then the corresponding SLD-tree does have a success branch. This is just a restatement of theorem 9.4.

Theorem 10.1 Let P be a definite program, G a definite goal and R a computation rule. Suppose that $P \cup \{G\}$ is unsatisfiable. Then the SLD-tree for $P \cup \{G\}$ via R has at least one success branch.

Theorem 9.5 can also be restated.

Theorem 10.2 Let P be a definite program, G a definite goal and R a computation rule. Then every correct answer θ for $P \cup \{G\}$ is "displayed" on the SLD-tree for $P \cup \{G\}$ via R.

"Displayed" means that, given θ, there is a success branch such that θ is an instance of the computed answer from the refutation corresponding to this branch.

While any two SLD-trees may have greatly different size and structure, they are essentially the same with respect to success branches.

Theorem 10.3 Let P be a definite program and G a definite goal. Then either every SLD-tree for $P \cup \{G\}$ has infinitely many success branches or every SLD-tree for $P \cup \{G\}$ has the same finite number of success branches.

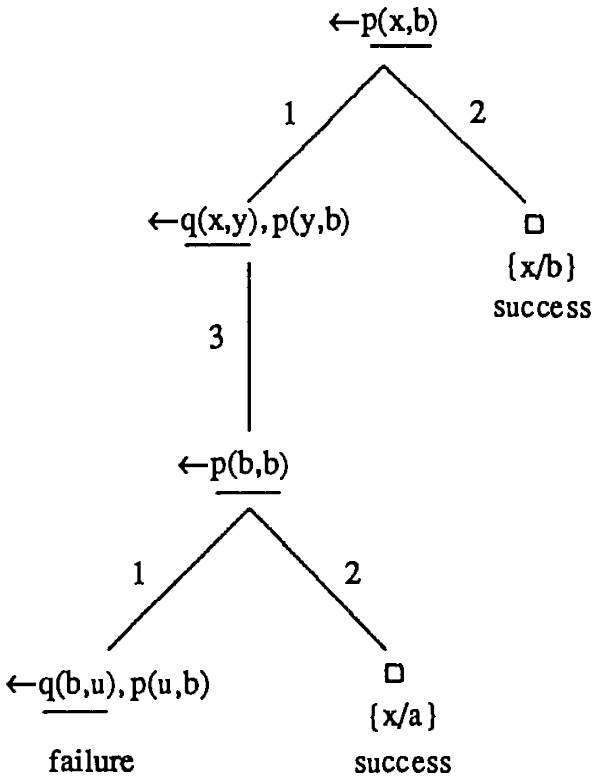

Fig. 2. A finite SLD-tree

Proof Using the switching lemma, we can set up a bijection between the success branches of any pair of SLD-trees. (See problem 17.) ■

For example, in Figures 2 and 3, the respective success branches giving the answer {x/a} can be transformed into one another by using the switching lemma.

Next we turn to the problem of searching SLD-trees to find success branches.

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

Standard PROLOG systems employ the computation rule which always selects the leftmost atom in a goal together with a depth-first search rule. The search rule is implemented by using a stack of goals. An instance of the goal stack represents the branch currently being investigated. The computation essentially becomes an

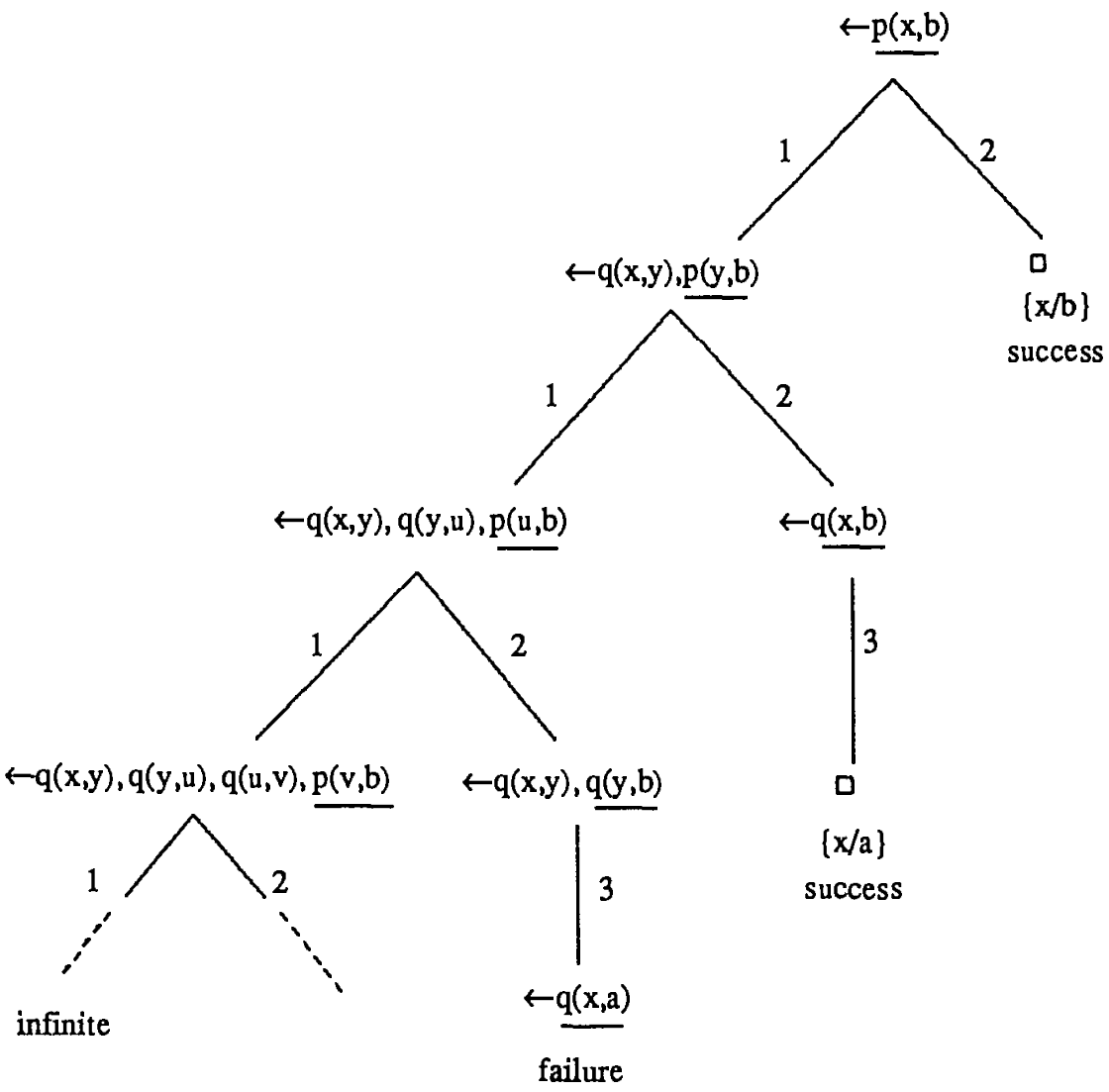

Fig. 3. An infinite SLD-tree

interleaved sequence of pushes and pops on this stack. A push occurs when the selected atom in the goal at the top of the stack is successfully unified with the head of a program clause. The resolvent is pushed onto the stack. A pop occurs when there are no (more) program clauses with head to match the selected atom in the goal at the top of the stack. This goal is then popped and the next choice of matching clause for the new top of stack is investigated. While depth-first search rules have undeniable problems (see below), they can be very efficiently implemented. This approach is entirely consistent with the view, which we share,

that PROLOG is primarily a programming language rather than a theorem prover.

For a system that searches depth-first, the search rule reduces to an *ordering rule*, that is, a rule which specifies the order in which program clauses are to be tried. Standard PROLOG systems use the order of clauses in a program as the fixed order in which they are to be tried. This is very simple and efficient to implement, but has the disadvantage that each call to a definition tries the clauses in the definition in exactly the same order.

Naturally, we would prefer the search rule to be *fair*, that is, to be such that each success branch on the SLD-tree will eventually be found. For infinite SLD-trees, search rules which do not have a breadth-first component will not be fair in general. However, a breadth-first component is less compatible with an efficient implementation.

Let us now consider the "completeness" of logic programming systems that use a depth-first search rule combined with a fixed order for trying clauses given by their ordering in the program. As well as standard PROLOG systems, let us also consider systems, such as IC-PROLOG [19], MU-PROLOG [73], [74] and NU-PROLOG [104], [75], which allow more complex computation rules. According to theorem 10.1, if $P \cup \{G\}$ is unsatisfiable, no matter what the computation rule, the corresponding SLD-tree will always contain a success branch. The question is this: will a logic programming system with a depth-first search rule using a fixed order for trying program clauses and an arbitrary computation rule, guarantee to always find the success branch? Unfortunately, the answer is no. In other words, none of the earlier completeness results is applicable to most current PROLOG systems because efficiency considerations have forced the implementation of unfair search rules!

Let us consider an example to make this clear.

Example Let P be the program

1. $p(a,b) \leftarrow$
2. $p(c,b) \leftarrow$
3. $p(x,z) \leftarrow p(x,y), p(y,z)$
4. $p(x,y) \leftarrow p(y,x)$

and G be the goal $\leftarrow p(a,c)$. It is straightforward to show that $P \cup \{G\}$ has a refutation and, moreover, that if any clause of P is omitted, $P \cup \{G\}$ will no

longer have a refutation.

We claim that no matter how the clauses of P are ordered and no matter what the computation rule, a logic programming system using a depth-first search with the fixed order for trying program clauses, will never find a refutation.

This claim follows immediately from the fact that clauses 3 and 4 have completely general heads. They will therefore always match any subgoal. Thus if clause 3 is before clause 4 in the program, the system will never consider clause 4 and vice versa. However, all the clauses are needed in any refutation. (See problem 18.)

Figure 4 illustrates the situation. There we have given the SLD-tree resulting from the use of the standard computation rule, which selects the leftmost atom, and the order for trying clauses given by the order of the clauses in the above program. As can be seen, the leftmost branch of this SLD-tree is infinite and thus a depth-first search will never find the success branch. In fact, for every computation rule and every fixed order for trying the program clauses, the leftmost branch of the corresponding SLD-tree will be infinite.

Finally, we discuss the importance of using appropriate computation rules. It would clearly be a substantial step towards purely declarative programming if we were able to build systems which would automatically find an appropriate computation rule for each program run on the system. To illustrate what is involved in this, consider once again the 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

Now the first thing to note about slowsort is that it does not run on standard PROLOG systems! Consider the goal ←sort(17.22.6.5.nil,y). A standard PROLOG system goes into an infinite loop because sorted makes longer and longer incorrect guesses for y. Of course, sorted has no business guessing at all. It is purely a test.

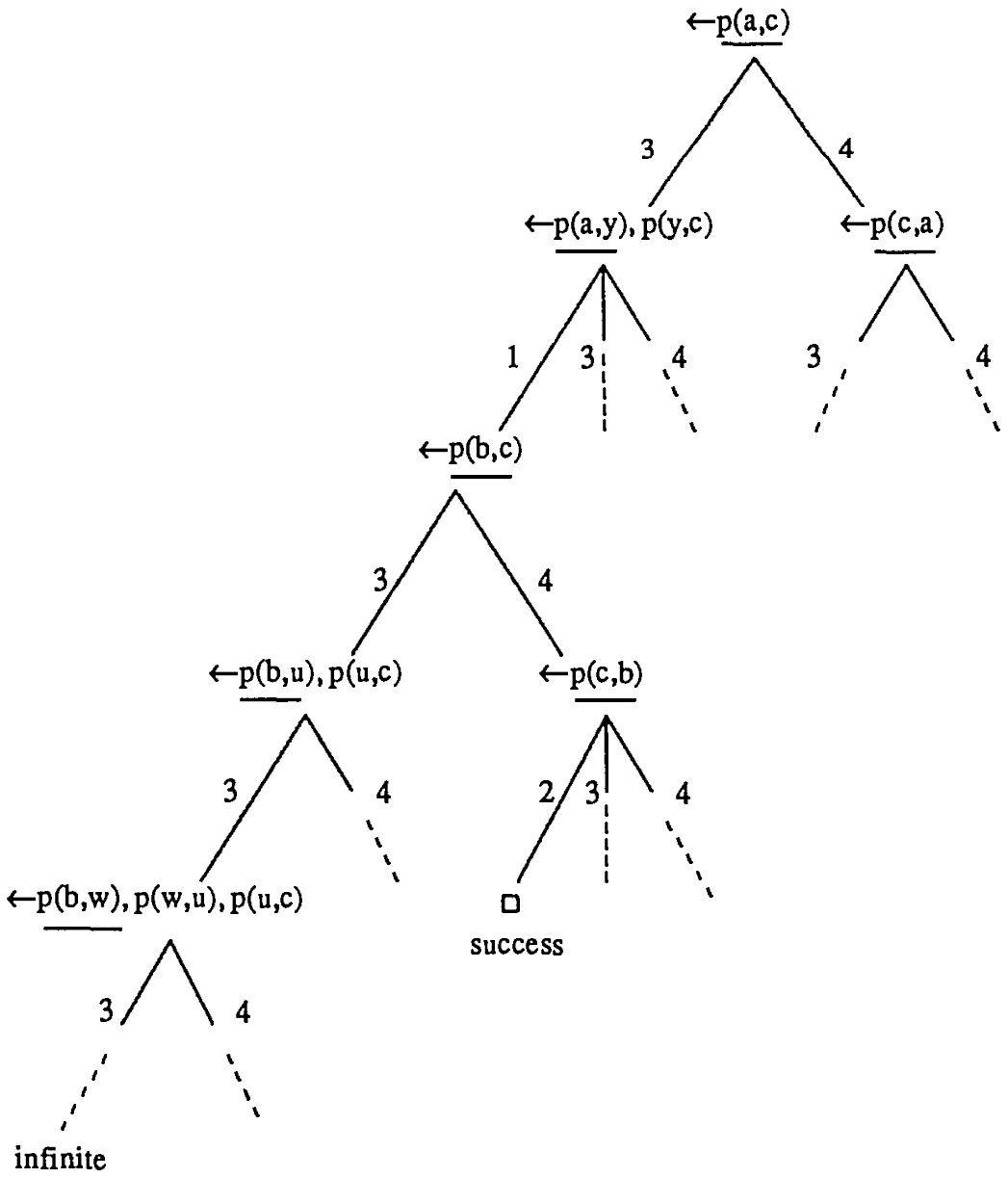

Fig. 4. SLD-tree which illustrates the problem with depth-first search

Thus a way to fix the problem is to change the definition of sort to

sort(x,y) ← perm(x,y), sorted(y)

This at least gives a program which runs, even if it is spectacularly inefficient. It sorts the given list by making random permutations of it and then using sorted to check if the permutations are sorted.

The attraction of the slowsort program is that it does give a very clear logic component for a sorting program. The disadvantage for standard PROLOG systems is that the only way to make it reasonably efficient is to substantially change the logic. To keep the above simple logic what we require is a computation rule which coroutines between perm and sorted. In this case, the list is given to perm which generates a *partial* permutation of it and then checks with sorted to see if the partial permutation is correct so far. If sorted finds that the partial permutation is indeed sorted, perm generates a bit more of the permutation and then checks with sorted again. Otherwise, perm undoes a bit of the partial permutation, generates a slightly different partial permutation and checks with sorted again. Such a program is clearly going to be a great deal more efficient than the one which generates an entire permutation before checking to see if it is sorted.

Thus we can obtain a more efficient sorting program by adding clever control to the simple logic. (Of course, much more efficient sorting programs are known, but this is not the point of the discussion.) There are now a number of PROLOG systems which allow the programmer to specify such control. For example, in NU-PROLOG [104] the programmer could add the *when declarations*

?- sorted(nil) when ever

?- sorted(x.y) when y

to the program. If the argument of the call to sorted either is nil or has the form s.t, where t is a non-variable, then the call proceeds. Thus the calls sorted(nil) and sorted(3.2.x) will proceed. If the argument of the call to sorted does not unify with nil or x.y, then the call proceeds (and then fails). If the argument of the call to sorted has the form y or s.y, then the call to sorted delays. Thus the call sorted(3.y) will delay. When a call sorted(y) or sorted(s.y) is delayed, the variable y is marked. When this variable is bound later, the delayed subgoal is resumed. This simple mechanism achieves the desired behaviour.

In standard PROLOG systems, a "generator" subgoal should come before a "test" subgoal. Thus perm should be put before sorted, if slowsort is to be run on a standard PROLOG system. However, in NU-PROLOG, the "test" should be put before the "generator". This order, together with appropriate when declarations on the "test", ensures proper coroutining between the "test" and the "generator". The coroutining starts by delaying the "test". The "generator" is then run until it creates a binding which causes the "test" to be resumed, and so on.

When declarations would not be of major interest if their addition always required programmer intervention. However, NU-PROLOG has a preprocessor which is able to *automatically* add when declarations to many programs in order to obtain more sensible behaviour. For example, given the slowsort program as input, the preprocessor outputs the above when declarations for sorted. (It also gives when declarations for perm, delete and ≤, but these are not needed for the use we have made of slowsort.) It does this by finding clauses with recursive calls which could cause infinite loops and generating sufficient when declarations to stop the loops. The preprocessor is also able to recognise that sorted is a "test" and should appear before perm in the first clause. It will reorder sorted and perm, if necessary. An account of the automatic generation of control is given in [74]. By relieving programmers of some of the responsibility for providing control in this way, NU-PROLOG is a step towards the ideal of purely declarative programming.

§11. CUTS

In this section, we discuss the cut, which is a widely used and controversial control facility offered by PROLOG systems. It is usually written as "!" in programs, although some systems call it "slash" and write it as "/". There has been considerable discussion of the advantages and disadvantages of cut and, in particular, whether it "affects the semantics" of programs in which it appears. We argue that cut does *not* affect the declarative semantics of definite programs, but it can introduce an undesirable form of incompleteness into the refutation procedure. (In §15, we discuss the effect that cuts can have in a program which has negative literals in the body of a program clause.)

First, we must be precise about what a cut actually does. Throughout this discussion, we restrict attention to systems which always select the leftmost atom in a goal. Cut is simply a non-logical annotation of programs which conveys certain control information to the system. Although it is written like an atom in the body of a clause, it is not an atom and has no logical significance at all. On the other hand, for pedagogical reasons, it is sometimes convenient to regard it as an atom which succeeds immediately on being called. The declarative semantics of a program with cuts is exactly the declarative semantics of the program with the cuts removed. In other words, the cuts do not in any way modify the declarative reading of the program.

What, then, is the nature of the control information conveyed by a cut? First, we need some terminology. Let us call the goal which caused the clause containing the cut to be activated, the *parent* goal. That is, the selected atom in the parent matched the head of the clause whose body contains the cut. Now, when "selected", the cut simply "succeeds" immediately. However, if backtracking later returns to the cut, the system discontinues searching in the subtree which has the parent goal at the root. The cut thus causes the remainder of that subtree to be pruned from the SLD-tree.

To clarify this, consider the following program fragment

A ← B, C
⋮
B ← D, !, E
⋮
D ←
⋮

where A, B, C, D and E are atoms. In Figure 5, we show part of the SLD-tree for a call to this program. The selected atom B in the goal ←B,C causes the cut to be introduced. The atom D is then selected and succeeds. The cut then succeeds, but the subgoal E eventually fails and the system backtracks to the cut. At this point, "deep" backtracking occurs. The system discontinues any further searching in the subtree which has the root ←B,C and, instead, resumes the search with the next choice for the goal ←A. This can be implemented very simply by popping goals from the goal stack until the goal ←A becomes top of the stack.

So a cut "merely" prunes the SLD-tree. Is it possible that a cut can somehow be harmful? The key issue is whether or not there is an answer to the (top level) goal in the part of the SLD-tree pruned by the cut. If there is no answer in the pruned part (that is, if the pruned part does not contain a success branch), then we call such a use of cut *safe*. However, if a success branch gets pruned by the cut, we call such a use of cut *unsafe*. Safe uses of cut are beneficial – they improve efficiency without missing answers. Unsafe uses of cut are harmful to the extent that a correct answer is missed.

Thus the harmful effect of cuts is that they can introduce a form of incompleteness into the SLD-resolution implementation of correct answer. Theorem 9.5 assures us that in the absence of cuts every correct answer can be computed. However, a cut in a program can destroy the completeness guaranteed by this theorem.

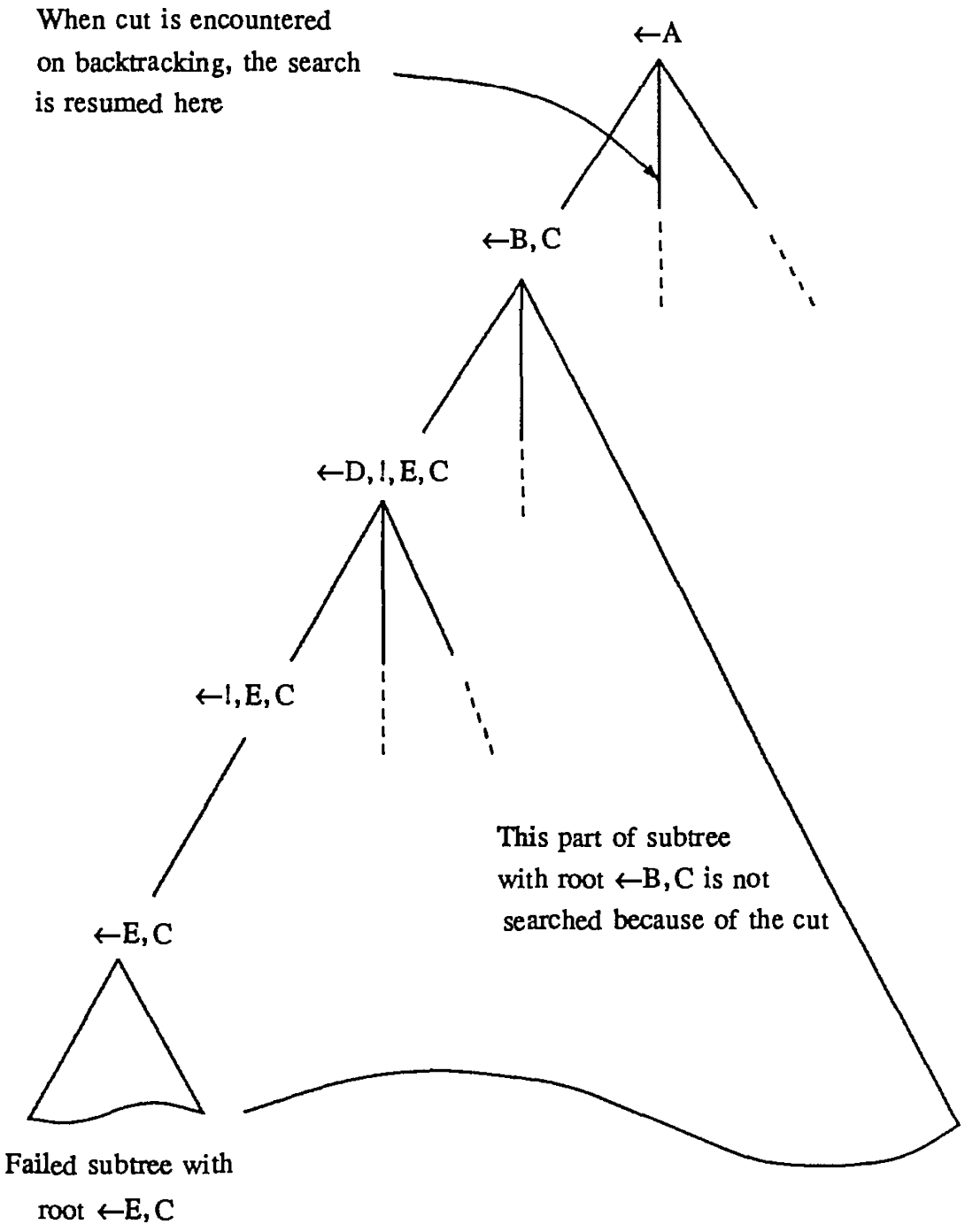

Fig. 5. The effect of cut

Note that this form of incompleteness is of a different nature from the form of incompleteness mentioned in §10, which occurs because a depth-first search can get lost down an infinite branch. A system which allows the search to become lost down an infinite branch does not give any answer at all (only a stack overflow message!). With an unsafe use of cut, a system can answer "no" when it should have answered "yes". However you look at it, the system has given an incorrect answer.

But, there is a further, much more harmful, effect of cuts. This occurs when programmers take advantage of cuts to write programs which are not even declaratively correct. For example, consider the program

$\max(x,y,y) \leftarrow x \leq y, !$

$\max(x,y,x) \leftarrow$

where $\max(x,y,z)$ is intended to be true iff z is the maximum of x and y. Advantage has been taken of the effect of the cut to leave the test $x>y$ out of the second clause. Procedurally, the semantics of the above program is the maximum relation. Declaratively, it is something else entirely. Such programs severely compromise the credibility of logic programming as declarative programming.

Admittedly, there are occasions when efficiency considerations force the use of such aberrations. However, it is far better for programmers, whenever possible, to make use of such higher level facilities as (sound implementations of) if-then-else, negation and not equals, which are not only reasonably efficient, but also lead to programs whose declarative semantics more accurately reflects the relation being computed.

PROBLEMS FOR CHAPTER 2

1. Complete the proof of proposition 6.1.

2. Find a finite set S of clauses and a non-empty set $\{M_i\}_{i\in I}$ of Herbrand models for S such that $\cap_{i\in I}M_i$ is not a model for S.

3. Let X be a directed subset of the lattice of Herbrand interpretations of a definite program. Show that $\{A_1,...,A_n\} \subseteq \text{lub}(X)$ iff $\{A_1,...,A_n\} \subseteq I$, for some $I\in X$.

4. Let P be the program

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

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

$q(b) \leftarrow$

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

Show that $T_P{\downarrow}\omega = \{p(f^n(a)) : n{\in}\omega\} \cup \{q(f^n(b)) : n{\in}\omega\}$ and $gfp(T_P) = T_P{\downarrow}\omega 2 = lfp(T_P) = \{q(f^n(b)) : n{\in}\omega\}$.

5. Let P be the program

$q(b) \leftarrow$

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

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

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

$r(c) \leftarrow r(x), q(x)$

$r(f(x)) \leftarrow r(x)$

Show that $T_P{\uparrow}\omega = \{q(f^n(b)) : n{\in}\omega\}$, $T_P{\downarrow}\omega = \{p(f^n(a)) : n{\in}\omega\} \cup \{q(f^n(b)) : n{\in}\omega\} \cup \{r(f^n(c)) : n{\in}\omega\}$ and $T_P{\downarrow}\omega 2 = \{p(f^n(a)) : n{\in}\omega\} \cup \{q(f^n(b)) : n{\in}\omega\} = gfp(T_P)$.

6. Let P be the program

$p_1(f(x)) \leftarrow p_1(x)$

$p_2(a) \leftarrow p_1(x)$

$p_2(f(x)) \leftarrow p_2(x)$

$p_3(a) \leftarrow p_2(x)$

$p_3(f(x)) \leftarrow p_3(x)$

$p_4(a) \leftarrow p_3(x)$

$p_4(f(x)) \leftarrow p_4(x)$

$p_5(a) \leftarrow p_4(x)$

$p_5(f(x)) \leftarrow p_5(x)$

Show that $T_P{\downarrow}\omega 4 \neq gfp(T_P)$, but $T_P{\downarrow}\omega 5 = \varnothing = gfp(T_P) = lfp(T_P)$.

7. (a) Let P be a definite program which contains no function symbols. Show that $T_P{\downarrow}\omega = gfp(T_P)$.

(b) Let P be a definite program with the property that, for each clause, each variable in the body of the clause also appears in the head. Show that $T_P{\downarrow}\omega = gfp(T_P)$.

8. Let P be a definite program with the following property: for each clause in P, if the clause has variables in the body that do not appear in the head, then the set of variables in the head is disjoint from the set of variables in the body. Prove that $gfp(T_P) = T_P{\downarrow}\omega n$, for some finite n depending on P.

9. Give an example of a correct answer which is not computed.

10. Let P be the slowsort program, G the goal ←sort(1.0.nil,y) and R the computation rule which always selects the leftmost atom. Show directly that $P \cup \{G\}$ has an SLD-refutation via R.

11. Consider the program

leaves(tree(void,v,void),v.x–x) ←

leaves(tree(u,v,w),x–y) ← leaves(u,x–z), leaves(w,z–y)

Find a goal such that a PROLOG system without the occur check will answer the goal incorrectly.

12. Show that if the occur check is omitted from the unification algorithm, one can use SLD-resolution to "prove" that $\forall x \exists y\, p(x,y) \rightarrow \exists y \forall x\, p(x,y)$ is valid.
(Hint: this problem requires the use of Skolem functions [66, p.126]).

13. Find an example to show that $A \in T_P{\uparrow}n$, for some $n \in \omega$, does not necessarily imply that there exists an SLD-refutation of *length* $\leq n$ for $P \cup \{\leftarrow A\}$.

14. Let P be a definite program and A an atom. Determine whether the following statement is correct or not:
$\forall(A)$ is a logical consequence of P iff $[A] \subseteq T_P{\uparrow}n$, for some $n \in \omega$.

15. Complete the details of the proof of theorem 9.2.

16. Let P be the program

p(x) ← q(x), r(x)

q(a) ←

r(x) ← r_1(x)

r_1(a) ←

Let R be the computation rule which always selects the leftmost atom and R' be

the computation rule which always selects the rightmost atom. Use the switching lemma to transform the refutation of $P \cup \{\leftarrow p(x)\}$ via R into one via R'.

17. Complete the details of the proof of theorem 10.3.

18. Let P be the program

p(a,b) ←
p(c,b) ←
p(x,z) ← p(x,y), p(y,z)
p(x,y) ← p(y,x)

and G be the goal ←p(a,c). Show that, if any clause of P is omitted, $P \cup \{G\}$ does not have a refutation (no matter what the computation rule).

19. Find a definite program P and a definite goal G such that each SLD-tree for $P \cup \{G\}$ has two success branches, but no depth-first search will ever find *both* success branches no matter what the computation rule and even if the program clauses can be dynamically reordered for each call to each definition of the program.

20. Let P be the slowsort program and G the goal ←sort(1.0.2.nil,y). Find an SLD-refutation of $P \cup \{G\}$ using a computation rule which suitably delays calls to sorted.

21. What problems arise in a PROLOG system which allows coroutining computation rules and also has the cut facility? How might these problems be solved?

Chapter 3

NORMAL PROGRAMS

In this chapter, we study various forms of negation. Since only positive information can be a logical consequence of a program, special rules are needed to deduce negative information. The most important of these rules are the closed world assumption and the negation as failure rule. This chapter introduces normal programs, which are programs for which the body of a program clause is a conjunction of literals. The major results of this chapter are soundness and completeness theorems for the negation as failure rule and SLDNF-resolution for normal programs.

§12. NEGATIVE INFORMATION

The inference system we have studied so far is very specialised. SLD-resolution applies only to sets of Horn clauses with exactly one goal clause. Using SLD-resolution, we can never deduce negative information. To be precise, let P be a definite program and $A \in B_P$. Then we cannot prove that $\neg A$ is a logical consequence of P. The reason is that $P \cup \{A\}$ is satisfiable, having B_P as a model.

To illustrate this, consider the program

student(joe) ←
student(bill) ←
student(jim) ←
teacher(mary) ←

Now suppose we wish to establish that mary is not a student, that is, ¬student(mary). As we have shown above, ¬student(mary) is not a logical consequence of the program. However, note that student(mary) is also not a logical consequence of the program. What we can do now is invoke a special inference rule: if a ground atom A is not a logical consequence of a program, then

infer ~A. This inference rule, introduced by Reiter [86], is called the *closed world assumption* (CWA). (Because of the approach taken here to the CWA, we would have preferred it to have been called the closed world *rule*.) Under this inference rule, we are entitled to infer ~student(mary) on the grounds that student(mary) is not a logical consequence of the program.

The CWA is often a very natural rule to use in a database context. In relational databases, this rule is usually applied – information not explicitly present in the database is taken to be false. Of course, in logic programs, the situation is complicated by the presence of non-unit clauses. The information content of a program is not determined by mere inspection. It is now the set of all things which can be deduced from the program. Whether or not use of the CWA is justified must be determined for each particular application. While it is often natural to use the CWA, its use may not always be justified.

The CWA is an example of a non-monotonic inference rule. Such rules are currently of great interest in artificial intelligence. (See, for example, [57] and the references therein.) An inference rule is *non-monotonic* if the addition of new axioms can decrease the set of theorems that previously held. As an example, if we add sufficient clauses to the above program so as to be able to deduce student(mary), then we will no longer be able to use the CWA to infer ~student(mary).

Now let us consider a program P for which the CWA is applicable. Let $A \in B_P$ and suppose we wish to infer ~A. In order to use the CWA, we have to show that A is not a logical consequence of P. Unfortunately, because of the undecidability of the validity problem of first order logic, there is no algorithm which will take an arbitrary A as input and respond in a finite amount of time with the answer whether A is or is not a logical consequence of P. If A is not a logical consequence, it may loop forever. Thus, in practice, the application of the CWA is generally restricted to those $A \in B_P$ whose attempted proofs fail finitely. Let us make this idea precise.

For a definite program P, the *SLD finite failure set* of P is the set of all $A \in B_P$ for which there exists a finitely failed SLD-tree for $P \cup \{\leftarrow A\}$, that is, one which is finite and contains no success branches. By proposition 13.4 and corollary 7.2, if A is in the SLD finite failure set of P, then A is not a logical consequence of P and every SLD-tree for $P \cup \{\leftarrow A\}$ contains only infinite or failure branches.

Now let us return to the CWA. In order to show that $A \in B_P$ is not a logical consequence of P, we can try giving $\leftarrow A$ as a goal to the system. Let us assume that A is not, in fact, in the success set of P. Now there are two possibilities: either A is in the SLD finite failure set or it is not. If A is in the SLD finite failure set, then the system can construct a finitely failed SLD-tree and return the answer "no". The CWA then allows us to infer ~A. In the other case, each SLD-tree has at least one infinite branch. Thus, unless the system has a mechanism for detecting infinite branches, it will never be able to complete the task of showing that A is not a logical consequence of P.

These considerations lead us to another non-monotonic inference rule, called the *negation as failure rule*. This rule, first studied in detail by Clark [15], is also used to infer negative information. It states that if A is in the SLD finite failure set of P, then infer ~A. Since the SLD finite failure set is a subset of the complement of the success set, we see that the negation as failure rule is less powerful than the CWA. However, in practice, implementing anything beyond negation as failure is difficult. The possibility of extending negation as failure closer to the CWA by adding mechanisms for detecting infinite branches has hardly been explored.

Negation as failure is easily and efficiently implemented by "reversing" the notions of success and failure. Suppose $A \in B_P$ and we have the goal $\leftarrow$ ~A. The system tries the goal $\leftarrow A$. If $\leftarrow A$ succeeds, then $\leftarrow$ ~A fails, while if it fails finitely, then $\leftarrow$ ~A succeeds.

Next we note that definite programs lack sufficient expressiveness for many situations. The problem is that often a negative condition is needed in the body of a clause. As an example, consider the definition

different(x,y) $\leftarrow$ member(z,x), ~member(z,y)

different(x,y) $\leftarrow$ ~member(z,x), member(z,y)

which defines when two sets are different. Practical PROLOG programs often require such extra expressiveness. Thus it is important to extend the definition of programs to include negative literals in the bodies of clauses. This is done in §14, where normal programs are introduced. These are programs for which the body of a program clause is a conjunction of literals.

However, even though normal programs allow negative literals in the bodies of program clauses, we still cannot deduce negative information from them. As before, the reason is that a normal program only contains the if halves of the

definitions of its predicate symbols, so that its Herbrand base is a model of the program. To deduce negative information from a normal program, we could "complete" the program. This involves adding the only-if halves of the definitions of the predicate symbols, together with an equality theory, to the program. In our previous example, if we add the missing only-if half to the definition of student, we obtain

$$\forall x\ (\text{student}(x) \leftrightarrow (x=\text{joe}) \vee (x=\text{bill}) \vee (x=\text{jim}))$$

Adding appropriate axioms for =, we can now deduce ~student(mary). This process of completion is another way of capturing the idea that information not given by the program is taken to be false. The concept of a correct answer can be extended to this context by defining an answer to be correct if the goal, with the answer applied, is a logical consequence of the *completion* of the program.

Having given the definition of the appropriate declarative concept, it remains to give the definition of a computed answer, which is the procedural counterpart of a correct answer. The mechanism usually chosen to compute answers is to use SLDNF-resolution, which is SLD-resolution augmented by the negation as failure rule. In §15 and §16, we study soundness and completeness results for SLDNF-resolution and the negation as failure rule for normal programs.

For additional discussion of the relationship between the CWA, the negation as failure rule and the completion of a program, we refer the reader to papers by Shepherdson [95], [97] and [98]. In [95], alternatives to the soundness theorems 15.4 and 15.6 below are presented, based on the idea of making explicit the appropriate first order theory underlying the CWA. Problems 26-31 at the end of this chapter are based on results from [95]. [98] contains a detailed discussion of some of the forms of negation used in logic programming, which as well as the approaches to negation based on (classical) first order logic mentioned above, also include the use of 3-valued logic, modal logic and intuitionistic logic. In this book, we concentrate on the approach to negation which is based on the completion of a program and first order logic.

§13. FINITE FAILURE

The main results of this section are several characterisations of the finite failure set of a definite program.

First, we give the definition of the finite failure set of a definite program. This definition was first given by Lassez and Maher [54].

Definition Let P be a definite program. Then F_P^d, the set of atoms in B_P which are *finitely failed by depth d*, is defined as follows:
(a) $A \in F_P^1$ if $A \notin T_P \downarrow 1$.
(b) $A \in F_P^d$, for $d>1$, if for each clause $B \leftarrow B_1,...,B_n$ in P and each substitution θ such that $A=B\theta$ and $B_1\theta,...,B_n\theta$ are ground, there exists k such that $1 \leq k \leq n$ and $B_k\theta \in F_P^{d-1}$.

Definition Let P be a definite program. The *finite failure set* F_P of P is defined by $F_P = \cup_{d \geq 1} F_P^d$.

Note the following simple relationship between F_P and $T_P \downarrow \omega$. (See problem 1.)

Proposition 13.1 Let P be a definite program. Then $F_P = B_P \backslash T_P \downarrow \omega$.

We now give, more formally, the definition of the SLD finite failure set of a definite program [4], [15].

Definition Let P be a definite program and G a definite goal. A *finitely failed* SLD-tree for $P \cup \{G\}$ is one which is finite and contains no success branches.

Definition Let P be a definite program. The *SLD finite failure set* of P is the set of all $A \in B_P$ for which there exists a finitely failed SLD-tree for $P \cup \{\leftarrow A\}$.

Note carefully in this last definition that there is no requirement that all SLD-trees fail finitely, only that there exists at least one.

Our main task is to establish the equivalence of F_P and the SLD finite failure set. We begin with two lemmas, due to Apt and van Emden [4], whose easy proofs are omitted. (See problems 2 and 3.)

Lemma 13.2 Let P be a definite program, G a definite goal and θ a substitution. Suppose that $P \cup \{G\}$ has a finitely failed SLD-tree of depth $\leq k$. Then $P \cup \{G\theta\}$ also has a finitely failed SLD-tree of depth $\leq k$.

Lemma 13.3 Let P be a definite program and $A_i \in B_P$, for i=1,...,m. Suppose that $P \cup \{\leftarrow A_1,...,A_m\}$ has a finitely failed SLD-tree of depth $\leq k$. Then there exists $i \in \{1,...,m\}$ such that $P \cup \{\leftarrow A_i\}$ has a finitely failed SLD-tree of depth $\leq k$.

The next proposition is due to Apt and van Emden [4].

Proposition 13.4 Let P be a definite program and $A \in B_P$. If $P \cup \{\leftarrow A\}$ has a finitely failed SLD-tree of depth $\leq k$, then $A \notin T_P{\downarrow}k$.

Proof Suppose first that $P \cup \{\leftarrow A\}$ has a finitely failed SLD-tree of depth 1. Then $A \notin T_P{\downarrow}1$.

Now assume the result holds for k–1. Suppose that $P \cup \{\leftarrow A\}$ has a finitely failed SLD-tree of depth $\leq k$. Suppose, to obtain a contradiction, that $A \in T_P{\downarrow}k$. Then there exists a clause $B \leftarrow B_1,...,B_n$ in P such that $A=B\theta$ and $\{B_1\theta,...,B_n\theta\} \subseteq T_P{\downarrow}(k-1)$, for some ground substitution θ. Thus there exists an mgu γ such that $A\gamma=B\gamma$ and $\theta=\gamma\sigma$, for some σ. Now $\leftarrow(B_1,...,B_n)\gamma\gamma$ is the root of a finitely failed SLD-tree of depth $\leq k-1$. By lemma 13.2, so also is $\leftarrow(B_1,...,B_n)\theta$. By lemma 13.3, some $\leftarrow B_i\theta$ is the root of a finitely failed SLD-tree of depth $\leq$ k–1. By the induction hypothesis, $B_i\theta \notin T_P{\downarrow}(k-1)$, which gives the contradiction. ■

It is interesting that the (strict) converse of proposition 13.4 does not hold. (See problem 4.) Next we note that SLD finite failure only guarantees the existence of one finitely failed SLD-tree – others may be infinite. It would be helpful to identify exactly those ways of selecting atoms which guarantee to find a finitely failed SLD-tree, if one exists at all. For this purpose, the concept of fairness was introduced by Lassez and Maher [54].

Definition An SLD-derivation is *fair* if it is either failed or, for every atom B in the derivation, (some further instantiated version of) B is selected within a finite number of steps.

Note that there are SLD-derivations via the standard computation rule which are not fair. One can achieve fairness by, for example, selecting the leftmost atom to the right of the (possibly empty set of) atoms introduced at the previous derivation step, if there is such an atom; otherwise, selecting the leftmost atom.

Definition An SLD-tree is *fair* if every branch of the tree is a fair SLD-derivation.

Proposition 13.5 Let P be a definite program and $\leftarrow A_1,...,A_m$ a definite goal. Suppose there is a non-failed fair derivation $\leftarrow A_1,...,A_m=G_0, G_1,...$ with mgu's $\theta_1, \theta_2,...$. Then, given $k \in \omega$, there exists $n \in \omega$ such that $[A_i\theta_1...\theta_n] \subseteq T_P{\downarrow}k$, for i=1,...,m.

Proof Theorem 7.4 shows that we can assume that the derivation is infinite. Clearly it suffices to show that given $i\in\{1,...,m\}$ and $k\in\omega$, there exists $n\in\omega$ such that $[A_i\theta_1...\theta_n] \subseteq T_P\downarrow k$.

Fix $i\in\{1,...,m\}$. The result is clearly true for $k=0$. Assume it is true for $k-1$. Suppose $A_i\theta_1...\theta_{p-1}$ is selected in the goal G_{p-1}. (By fairness, A_i must eventually be selected.) Let G_p be $\leftarrow B_1,...,B_q$, where $q\geq 1$. By the induction hypothesis, there exists $s\in\omega$ such that $\cup_{j=1}^{q}[B_j\theta_{p+1}...\theta_{p+s}] \subseteq T_P\downarrow(k-1)$. Hence we have that $[A_i\theta_1...\theta_{p+s}] \subseteq T_P(\cup_{j=1}^{q}[B_j\theta_{p+1}...\theta_{p+s}]) \subseteq T_P(T_P\downarrow(k-1)) = T_P\downarrow k$. ■

Combining the results of Apt and van Emden [4] and Lassez and Maher [54], we can now obtain the characterisations of the finite failure set.

Theorem 13.6 Let P be a definite program and $A\in B_P$. Then the following are equivalent:

(a) $A\in F_P$.

(b) $A\notin T_P\downarrow\omega$.

(c) A is in the SLD finite failure set.

(d) Every fair SLD-tree for $P \cup \{\leftarrow A\}$ is finitely failed.

Proof (a) is equivalent to (b) by proposition 13.1. That (d) implies (c) is obvious. Also (c) implies (b) by proposition 13.4.

Finally, suppose that (d) does not hold. Then there exists a non-failed fair derivation for $\leftarrow A$. By proposition 13.5, $A\in T_P\downarrow\omega$. Thus (b) does not hold. ■

Theorem 13.6 shows that fair SLD-resolution is a sound and complete implementation of finite failure.

§14. PROGRAMMING WITH THE COMPLETION

In this section, normal programs are introduced. These are programs whose program clauses may contain negative literals in their body. The completion of a normal program is also defined. The completion will play an important part in the soundness and completeness results for the negation as failure rule and SLDNF-resolution. The definition of a correct answer is extended to normal programs.

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

$$A\leftarrow L_1,...,L_n$$

where A is an atom and $L_1,...,L_n$ are literals.

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

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

$$\leftarrow L_1,...,L_n$$

where $L_1,...,L_n$ are literals.

Definition The *definition* of a predicate symbol p in a normal program P is the set of all program clauses in P which have p in their head.

Every definite program is a normal program, but not conversely.

In order to justify the use of the negation as failure rule, Clark [15] introduced the idea of the completion of a normal program. We next give the definition of the completion.

Let $p(t_1,...,t_n)\leftarrow L_1,...,L_m$ be a program clause in a normal program P. We will require a new predicate symbol =, not appearing in P, whose intended interpretation is the identity relation. The first step is to transform the given clause into

$$p(x_1,...,x_n)\leftarrow(x_1=t_1)\wedge...\wedge(x_n=t_n)\wedge L_1\wedge...\wedge L_m$$

where $x_1,...,x_n$ are variables not appearing in the clause. Then, if $y_1,...,y_d$ are the variables of the original clause, we transform this into

$$p(x_1,...,x_n)\leftarrow\exists y_1...\exists y_d\ ((x_1=t_1)\wedge...\wedge(x_n=t_n)\wedge L_1\wedge...\wedge L_m)$$

Now suppose this transformation is made for each clause in the definition of p. Then we obtain $k\geq 1$ transformed formulas of the form

$$p(x_1,...,x_n)\leftarrow E_1$$
$$\vdots$$
$$p(x_1,...,x_n)\leftarrow E_k$$

where each E_i has the general form

$$\exists y_1...\exists y_d\ ((x_1=t_1)\wedge...\wedge(x_n=t_n)\wedge L_1\wedge...\wedge L_m)$$

The *completed definition* of p is then the formula

$$\forall x_1...\forall x_n\ (p(x_1,...,x_n)\leftrightarrow E_1\vee...\vee E_k)$$

Example Let the definition of a predicate symbol p be

$p(y) \leftarrow q(y), \sim r(a,y)$

$p(f(z)) \leftarrow \sim q(z)$

$p(b) \leftarrow$

Then the completed definition of p is

$$\forall x\,(p(x) \leftrightarrow (\exists y((x=y)\wedge q(y)\wedge\neg r(a,y)) \vee \exists z((x=f(z))\wedge\neg q(z)) \vee (x=b)))$$

Example The completed definition of the predicate symbol student from the example in §12 is

$$\forall x\,(\text{student}(x)\leftrightarrow(x=\text{joe})\vee(x=\text{bill})\vee(x=\text{jim}))$$

Some predicate symbols in the program may not appear in the head of any program clause. For each such predicate symbol q, we explicitly add the clause

$$\forall x_1 ... \forall x_n \neg q(x_1,...,x_n)$$

This is the definition of such q given implicitly by the program. We also call this clause the *completed definition* of such q.

It is essential to also include some axioms which constrain =. The following *equality theory* is sufficient for our purpose. In these axioms, we use the standard notation ≠ for not equals.

1. $c\neq d$, for all pairs c,d of distinct constants.
2. $\forall(f(x_1,...,x_n)\neq g(y_1,...,y_m))$, for all pairs f,g of distinct function symbols.
3. $\forall(f(x_1,...,x_n)\neq c)$, for each constant c and function symbol f.
4. $\forall(t[x]\neq x)$, for each term t[x] containing x and different from x.
5. $\forall((x_1\neq y_1)\vee...\vee(x_n\neq y_n)\rightarrow f(x_1,...,x_n)\neq f(y_1,...,y_n))$, for each function symbol f.
6. $\forall(x=x)$.
7. $\forall((x_1=y_1)\wedge...\wedge(x_n=y_n)\rightarrow f(x_1,...,x_n)=f(y_1,...,y_n))$, for each function symbol f.
8. $\forall((x_1=y_1)\wedge...\wedge(x_n=y_n)\rightarrow(p(x_1,...,x_n)\rightarrow p(y_1,...,y_n)))$, for each predicate symbol p (including =).

Definition Let P be a normal program. The *completion* of P, denoted by comp(P), is the collection of completed definitions of predicate symbols in P together with the equality theory.

Axioms 6, 7 and 8 are the usual axioms for first order theories with equality. Note that axioms 6 and 8 together imply that = is an equivalence relation. (See problem 9.) The equality theory places a strong restriction on the possible interpretations of =. This restriction is essential to obtain the desired justification of negation as failure. Roughly speaking, we are forcing = to be interpreted as the identity relation on U_P. (See problem 10.)

Now, as Clark [15] has pointed out, it is appropriate to regard the *completion* of the normal program, not the normal program itself, as the prime object of interest. Even though a programmer only gives a logic programming system the

normal program, the understanding is that, conceptually, the normal program is completed by the system and that the programmer is actually programming with the completion. Corresponding to this notion, we have the concept of a correct answer. The problem then arises of showing that SLD-resolution, augmented by the negation as failure rule, is a sound and complete implementation of the declarative concept of a correct answer. We tackle this problem in §15 and §16.

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

Definition Let P be a normal program, G a normal goal $\leftarrow L_1,...,L_n$, and θ an answer for $P \cup \{G\}$. We say θ is a *correct answer* for $comp(P) \cup \{G\}$ if $\forall((L_1 \wedge ... \wedge L_n)\theta)$ is a logical consequence of comp(P).

It is important to establish that this definition generalises the definition of correct answer given in §6. The first result we need to prove this is the following proposition.

Proposition 14.1 Let P be a normal program. Then P is a logical consequence of comp(P).

Proof Let M be a model for comp(P). We have to show that M is a model for P. Let $p(t_1,...,t_n) \leftarrow L_1,...,L_m$ be a program clause in P and suppose that $L_1,...,L_m$ are true in M, for some assignment of the variables $y_1,...,y_d$ in the clause.

Consider the completed definition of p

$$\forall x_1 ... \forall x_n \ (p(x_1,...,x_n) \leftrightarrow E_1 \vee ... \vee E_k)$$

and suppose E_i is

$$\exists y_1 ... \exists y_d \ ((x_1 = t_1) \wedge ... \wedge (x_n = t_n) \wedge L_1 \wedge ... \wedge L_m)$$

Now let x_j be t_j $(1 \leq j \leq n)$, for the same assignment of the variables $y_1,...,y_d$ as above. Thus E_i is true in M, since $L_1,...,L_m$ are true in M and also since M must satisfy axiom 6. Hence $p(t_1,...,t_n)$ is true in M. ■

We now define a mapping T_P^J from the lattice of interpretations based on some pre-interpretation J to itself.

Definition Let J be a pre-interpretation of a normal program P and I an interpretation based on J. Then $T_P^J(I) = \{ A_{J,V} : A \leftarrow L_1 \wedge ... \wedge L_n \in P$, V is a variable assignment wrt J, and $L_1 \wedge ... \wedge L_n$ is true wrt I and V$\}$.

When J is the Herbrand pre-interpretation of P, we write T_P instead of T_P^J. This convention is consistent with our earlier usage of T_P. Note that T_P^J is generally not monotonic. For example, if P is the program

$p \leftarrow \neg p$

then T_P is not monotonic. However, if P is a definite program, then T_P^J is monotonic. Many other properties of T_P easily extend to T_P^J.

Proposition 14.2 Let P be a normal program, J a pre-interpretation of P, and I an interpretation based on J. Then I is a model for P iff $T_P^J(I) \subseteq I$.

Proof Similar to the proof of proposition 6.4. (See problem 11.) ■

The next result shows that fixpoints of T_P^J give models for comp(P).

Proposition 14.3 Let P be a normal program, J a pre-interpretation of P, and I an interpretation based on J. Suppose that I, together with the identity relation assigned to =, is a model for the equality theory. Then I, together with the identity relation assigned to =, is a model for comp(P) iff $T_P^J(I) = I$.

Proof Suppose first that $T_P^J(I) = I$. Since we have assumed that I, together with the identity relation assigned to =, is a model for the equality theory, it suffices to show that this interpretation is a model for each of the completed definitions of comp(P). Consider a completed definition of the form $\forall x_1 ... \forall x_n \neg q(x_1,...,x_n)$. Since I is a fixpoint, it is clear that the interpretation is a model of this formula. Now consider a completed definition of the form

$$\forall x_1 ... \forall x_n \, (p(x_1,...,x_n) \leftrightarrow E_1 \vee ... \vee E_k)$$

Since $T_P^J(I) \subseteq I$, it follows that the interpretation is a model for the formula

$$\forall x_1 ... \forall x_n \, (p(x_1,...,x_n) \leftarrow E_1 \vee ... \vee E_k)$$

Also, since $T_P^J(I) \supseteq I$, it follows that the interpretation is a model for the formula

$$\forall x_1 ... \forall x_n \, (p(x_1,...,x_n) \rightarrow E_1 \vee ... \vee E_k)$$

Conversely, suppose that I, together with the identity relation assigned to =, is a model for the completion. Then using the fact that the interpretation is a model for formulas of the form

$$\forall x_1 ... \forall x_n \, (p(x_1,...,x_n) \leftarrow E_1 \vee ... \vee E_k)$$

it follows that $T_P^J(I) \subseteq I$. Similarly, using the fact that the interpretation is a model for formulas of the form

$$\forall x_1 ... \forall x_n \, (p(x_1,...,x_n) \rightarrow E_1 \vee ... \vee E_k)$$

it follows that $T_P^J(I) \supseteq I$. ■

Proposition 14.4 Let P be a definite program and $A \in B_P$. Then $A \in gfp(T_P)$ iff comp(P) $\cup$ {A} has an Herbrand model.

Proof Suppose $A \in gfp(T_P)$. Then $gfp(T_P) \cup \{s=s : s \in U_P\}$ is an Herbrand model for comp(P) $\cup$ {A}, by proposition 14.3.

Conversely, suppose comp(P) $\cup$ {A} has an Herbrand model M. By the equality theory, the identity relation on U_P must be assigned to = in the model M. Thus M has the form $I \cup \{s=s : s \in U_P\}$, for some Herbrand interpretation I of P. Hence $I=T_P(I)$, by proposition 14.3, and so $A \in gfp(T_P)$. ■

Proposition 14.5 Let P be a definite program and $A_1,...,A_m$ be atoms. If $\forall(A_1 \wedge ... \wedge A_m)$ is a logical consequence of comp(P), then it is also a logical consequence of P.

Proof Let $x_1,...,x_k$ be the variables in $A_1 \wedge ... \wedge A_m$. We have to show that $\forall x_1 ... \forall x_k (A_1 \wedge ... \wedge A_m)$ is a logical consequence of P, that is, $P \cup \{\neg \forall x_1 ... \forall x_k (A_1 \wedge ... \wedge A_m)\}$ is unsatisfiable or, equivalently, $S = P \cup \{\neg A'_1 \vee ... \vee \neg A'_m\}$ is unsatisfiable, where A'_i is A_i with $x_1,...,x_k$ replaced by appropriate Skolem constants.

Since S is in clause form, we can restrict attention to Herbrand interpretations of S. Let I be an Herbrand interpretation of S. We can also regard I as an interpretation of P. (Note that I is not necessarily an *Herbrand* interpretation of P.) Suppose I is a model for P. Consider the pre-interpretation J obtained from I by ignoring the assignments to the predicate symbols in I. By proposition 14.2, we have that $T_P^J(I) \subseteq I$. Since T_P^J is monotonic, proposition 5.2 shows that there exists a fixpoint $I' \subseteq I$ of T_P^J. Since I', together with the identity relation assigned to =, is obviously a model for the equality theory, proposition 14.3 shows that this interpretation is a model for comp(P). Hence $\neg A'_1 \vee ... \vee \neg A'_m$ is false in this interpretation. Since $I' \subseteq I$, we have that $\neg A'_1 \vee ... \vee \neg A'_m$ is false in I. Thus S is unsatisfiable. ■

Note that by combining propositions 14.1 and 14.5, it follows that the *positive* information which can be deduced from comp(P) is exactly the same as the positive information which can be deduced from P. To be precise, we have the following result.

Theorem 14.6 Let P be a definite program, G a definite goal, and θ an answer for P $\cup$ {G}. Then θ is a correct answer for comp(P) $\cup$ {G} iff θ is a

correct answer for $P \cup \{G\}$.

Theorem 14.6 shows that the definition of correct answer given in this section generalises the definition given in §6.

Every normal program is consistent, but the completion of a normal program may not be consistent. (See problem 8.) We now investigate a weak syntactic condition sufficient to ensure that the completion of a normal program is consistent. The motivation is to limit the use of negation in recursive rules to keep the model theory manageable.

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

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

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

Clearly, every definite program and every hierarchical normal program is stratified. We can assume without loss of generality that the levels of a stratified program are 0,1,...,k, for some k. Stratified normal programs were introduced by Apt, Blair and Walker [3] as a generalisation of a class of databases discussed by Chandra and Harel [13], and later, independently, by Van Gelder [109]. Other papers on stratified programs are contained in [70].

Even though the mapping T_P^J is, in general, not monotonic, it does have an important property similar to monotonicity for stratified normal programs. This result is due to Lloyd, Sonenberg and Topor [60].

Proposition 14.7 Let P be a stratified normal program and J a pre-interpretation for P.
(a) Suppose P has only predicates of level 0. Then P is definite and T_P^J is monotonic over the lattice of interpretations based on J.

(b) Suppose P has maximum predicate level k+1. Let P_k denote the set of program clauses in P with the property that the predicate symbol in the head of the clause has level $\leq$ k. Suppose that M_k is an interpretation based on J for P_k and M_k is a fixpoint of $T^J_{P_k}$. Then $\Lambda = \{M_k \cup S : S \subseteq \{p(d_1,...,d_n) : p$ is a level k+1 predicate symbol and each d_i is in the domain of J$\}$ $\}$ is a complete lattice, under set inclusion. Furthermore, Λ is a sublattice of the lattice of interpretations based on J, and T^J_P, restricted to Λ, is well-defined and monotonic.

Proof Straightforward. (See problem 13.) ■

Corollary 14.8 Let P be a stratified normal program. Then comp(P) has a minimal normal Herbrand model.

Proof (A *normal* model is one for which the identity relation is assigned to =. *Minimal* means that there is no strictly smaller normal Herbrand model.) The proof is by induction on the maximum level, k, of the predicate symbols in P. The case k=0 uses proposition 14.7(a) and proposition 5.1 to obtain the least fixpoint of T_P. Proposition 14.3 yields the model. The induction step uses proposition 5.1, proposition 14.3 and proposition 14.7(b) with M_k as the fixpoint provided by the induction hypothesis. ■

Corollary 14.8 is due to Apt, Blair and Walker [3].

§15. SOUNDNESS OF SLDNF-RESOLUTION

In section §14, we introduced the fundamental concept of a correct answer for comp(P) $\cup$ {G}. Now that we have the appropriate declarative concept, let us see how we can implement it. The basic idea is to use SLD-resolution, augmented by the negation as failure rule (SLDNF-resolution). In this section, we prove the soundness of the negation as failure rule and of SLDNF-resolution. We give conditions which are sufficient for a computation to avoid floundering. We also discuss the effect that cuts in a normal program can have on the soundness results.

Our first task is to give a precise definition of an SLDNF-refutation and a finitely failed SLDNF-tree. For this, we first give the mutually recursive definitions of the concepts of SLDNF-refutation of rank k and finitely failed SLDNF-tree of rank k.

In the definitions which follow, it will be necessary to select literals from normal goals. The choice of which literal is selected is constrained in the following way. There is no restriction on which positive literal can be selected; however, only a *ground* negative literal can be selected. This condition is called the *safeness condition* on the selection of literals. It is used to ensure the soundness of SLDNF-resolution. Later we discuss the possibility of weakening this condition.

Definition Let G be $\leftarrow L_1,...,L_m,...,L_p$ and C be $A\leftarrow M_1,...,M_q$. Then G' is *derived* from G and C using mgu θ if the following conditions hold:
(a) L_m is an atom, called the *selected* atom, in G.
(b) θ is an mgu of L_m and A.
(c) G' is the normal goal $\leftarrow(L_1,...,L_{m-1},M_1,...,M_q,L_{m+1},...,L_p)\theta$.

Definition Let P be a normal program and G a normal goal. An *SLDNF-refutation of rank 0* of $P \cup \{G\}$ consists of a sequence $G_0=G, G_1,..., G_n=\square$ of normal goals, a sequence $C_1,...,C_n$ of variants of program clauses of P and a sequence $\theta_1,...,\theta_n$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

Definition Let P be a normal program and G a normal goal. A *finitely failed SLDNF-tree of rank 0* for $P \cup \{G\}$ is a tree satisfying the following:
(a) The tree is finite and each node of the tree is a non-empty normal goal.
(b) The root node is G.
(c) Only positive literals are selected at nodes in the tree.
(d) Let $\leftarrow L_1,...,L_m,...,L_p$ be a non-leaf node in the tree and suppose that L_m is an atom and it is selected. Then, for each program clause (variant) $A\leftarrow M_1,...,M_q$ such that L_m and A are unifiable with mgu θ, this node has a child $\leftarrow(L_1,...,L_{m-1},M_1,...,M_q,L_{m+1},...,L_p)\theta$.
(e) Let $\leftarrow L_1,...,L_m,...,L_p$ be a leaf node in the tree and suppose that L_m is an atom and it is selected. Then there is no program clause (variant) in P whose head unifies with L_m.

Definition Let P be a normal program and G a normal goal. An *SLDNF-refutation of rank k+1* of $P \cup \{G\}$ consists of a sequence $G_0=G, G_1,..., G_n=\square$ of normal goals, a sequence $C_1,...,C_n$ of variants of program clauses of P or ground negative literals, and a sequence $\theta_1,...,\theta_n$ of substitutions, such that, for each i, either

(i) G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}, or

(ii) G_i is $\leftarrow L_1,...,L_m,...,L_p$, the selected literal L_m in G_i is a ground negative literal $\sim A_m$ and there is a finitely failed SLDNF-tree of rank k for $P \cup \{\leftarrow A_m\}$. In this case, G_{i+1} is $\leftarrow L_1,...,L_{m-1},L_{m+1},...,L_p$, θ_{i+1} is the identity substitution and C_{i+1} is $\sim A_m$.

Definition Let P be a normal program and G a normal goal. A *finitely failed SLDNF-tree of rank k+1* for $P \cup \{G\}$ is a tree satisfying the following:

(a) The tree is finite and each node of the tree is a non-empty normal goal.

(b) The root node is G.

(c) Let $\leftarrow L_1,...,L_m,...,L_p$ be a non-leaf node in the tree and suppose that L_m is selected. Then either

(i) L_m is an atom and, for each program clause (variant) $A \leftarrow M_1,...,M_q$ such that L_m and A are unifiable with mgu θ, the node has a child $\leftarrow (L_1,...,L_{m-1},M_1,...,M_q,L_{m+1},...,L_p)\theta$, or

(ii) L_m is a ground negative literal $\sim A_m$ and there is a finitely failed SLDNF-tree of rank k for $P \cup \{\leftarrow A_m\}$, in which case the only child is $\leftarrow L_1,...,L_{m-1},L_{m+1},...,L_p$.

(d) Let $\leftarrow L_1,...,L_m,...,L_p$ be a leaf node in the tree and suppose that L_m is selected. Then either

(i) L_m is an atom and there is no program clause (variant) in P whose head unifies with L_m, or

(ii) L_m is a ground negative literal $\sim A_m$ and there is an SLDNF-refutation of rank k of $P \cup \{\leftarrow A_m\}$.

Note that an SLDNF-refutation (resp., finitely failed SLDNF-tree) of rank k is also an SLDNF-refutation (resp., finitely failed SLDNF-tree) of rank n, for all $n \geq k$.

Definition Let P be a normal program and G a normal goal. An *SLDNF-refutation* of $P \cup \{G\}$ is an SLDNF-refutation of rank k of $P \cup \{G\}$, for some k.

Definition Let P be a normal program and G a normal goal. A *finitely failed SLDNF-tree* for $P \cup \{G\}$ is a finitely failed SLDNF-tree of rank k for $P \cup \{G\}$, for some k.

Definition Let P be a normal program and G a normal goal. A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1...\theta_n$ to the variables of G, where $\theta_1,...,\theta_n$ is the sequence of substitutions used in an SLDNF-refutation of $P \cup \{G\}$.

Since only ground negative literals are selected, it follows that $L_i\theta$ must be ground, for each negative literal L_i in G. This definition extends the definition of a computed answer given in §7.

Now that we have given the definition of a computed answer, we consider the procedure a logic programming system might use to compute answers. The basic idea is to use SLD-resolution, augmented by the negation as failure rule. When a positive literal is selected, we use essentially SLD-resolution to derive a new goal. However, when a ground negative literal is selected, the goal answering process is entered recursively in order to try to establish the negative subgoal. We can regard these negative subgoals as separate *lemmas*, which must be established to compute the result. Having selected a ground negative literal ~A in some goal, an attempt is made to construct a finitely failed SLDNF-tree with root ←A before continuing with the remainder of the computation. If such a finitely failed tree is constructed, then the subgoal ~A succeeds. Otherwise, if an SLDNF-refutation is found for ←A, then the subgoal ~A fails. Note that bindings are only made by successful calls of positive literals. Negative calls never create bindings; they only succeed or fail. Thus negation as failure is purely a test.

Next we give the definitions of SLDNF-derivation and SLDNF-tree.

Definition Let P be a normal program and G a normal goal. An *SLDNF-derivation* of $P \cup \{G\}$ consists of a (finite or infinite) sequence G_0=G, G_1,... of normal goals, a sequence C_1, C_2,... of variants of program clauses (called *input clauses*) of P or ground negative literals, and a sequence θ_1, θ_2,... of substitutions satisfying the following:

(a) For each i, either

(i) G_{i+1} is derived from G_i and an input clause C_{i+1} using θ_{i+1}, or

(ii) G_i is $\leftarrow L_1,...,L_m,...,L_p$, the selected literal L_m in G_i is a ground negative literal $\sim A_m$ and there is a finitely failed SLDNF-tree for $P \cup \{\leftarrow A_m\}$. In this case, G_{i+1} is $\leftarrow L_1,...,L_{m-1},L_{m+1},...,L_p$, θ_{i+1} is the identity substitution and C_{i+1} is $\sim A_m$.

(b) If the sequence G_0, G_1,... of goals is finite, then either

(i) the last goal is empty, or

(ii) the last goal is $\leftarrow L_1,...,L_m,...,L_p$, L_m is an atom, L_m is selected and there is no program clause (variant) in P whose head unifies with L_m, or

(iii) the last goal is $\leftarrow L_1,...,L_m,...,L_p$, L_m is a ground negative literal $\sim A_m$, L_m is selected and there is an SLDNF-refutation of $P \cup \{\leftarrow A_m\}$.

Definition Let P be a normal program and G a normal goal. An *SLDNF-tree* for $P \cup \{G\}$ is a tree satisfying the following:

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

(b) The root node is G.

(c) Let $\leftarrow L_1,...,L_m,...,L_p$ $(p \geq 1)$ be a non-leaf node in the tree and suppose that L_m is selected. Then either

(i) L_m is an atom and, for each program clause (variant) $A \leftarrow M_1,...,M_q$ such that L_m and A are unifiable with mgu θ, the node has a child $\leftarrow (L_1,...,L_{m-1},M_1,...,M_q,L_{m+1},...,L_p)\theta$, or

(ii) L_m is a ground negative literal $\sim A_m$ and there is a finitely failed SLDNF-tree for $P \cup \{\leftarrow A_m\}$, in which case the only child is $\leftarrow L_1,...,L_{m-1},L_{m+1},...,L_p$.

(d) Let $\leftarrow L_1,...,L_m,...,L_p$ $(p \geq 1)$ be a leaf node in the tree and suppose that L_m is selected. Then either

(i) L_m is an atom and there is no program clause (variant) in P whose head unifies with L_m, or

(ii) L_m is a ground negative literal $\sim A_m$ and there is an SLDNF-refutation of $P \cup \{\leftarrow A_m\}$.

(e) Nodes which are the empty clause have no children.

The concepts of SLDNF-derivation, SLDNF-refutation and SLDNF-tree generalise those of SLD-derivation, SLD-refutation and SLD-tree. An SLDNF-derivation is *finite* if it consists of a finite sequence of goals; otherwise, it is *infinite*. An SLDNF-derivation is *successful* if it is finite and the last goal is the empty goal. An SLDNF-derivation is *failed* if it is finite and the last goal is not the empty goal. Similarly, we define success, infinite and failure branches of an SLDNF-tree. It is clear that a successful SLDNF-derivation is indeed an SLDNF-refutation and an SLDNF-tree, for which every branch is a failure branch, is indeed a finitely failed SLDNF-tree.

If a goal contains only non-ground negative literals, then, because of the safeness condition, no literal is available for selection. Let us formalise this notion. By a *computation* of $P \cup \{G\}$, we mean an attempt to construct an SLDNF-derivation of $P \cup \{G\}$.

Definition Let P be a normal program and G a normal goal. We say a computation of $P \cup \{G\}$ *flounders* if at some point in the computation a goal is reached which contains only non-ground negative literals.

Example If G is $\leftarrow \neg p(x)$ and P is any normal program, then the computation of $P \cup \{G\}$ flounders immediately.

We now give a condition under which we can be sure that SLDNF-resolution never flounders.

Definition Let P be a normal program and G a normal goal.

We say a program clause $A \leftarrow L_1,...,L_n$ in P is *admissible* if every variable that occurs in the clause occurs either in the head A or in a positive literal of the body $L_1,...,L_n$.

We say a program clause $A \leftarrow L_1,...,L_n$ in P is *allowed* if every variable that occurs in the clause occurs in a positive literal of the body $L_1,...,L_n$.

We say G is *allowed* if G is $\leftarrow L_1,...,L_n$ and every variable that occurs in G occurs in a positive literal of the body $L_1,...,L_n$.

We say $P \cup \{G\}$ is *allowed* if the following conditions are satisfied:

(a) Every clause in P is admissible.

(b) Every clause in the definition of a predicate symbol occurring in a positive literal in the body of G or in a positive literal in the body of a clause in P is allowed.

(c) G is allowed.

Note that an allowed unit clause must be ground and every allowed clause is admissible. These definitions generalise Clark's definition [15] of an allowed query and Shepherdson's covering axiom [95]. The next result is due to Lloyd and Topor [63] and Shepherdson [97]. Other results on allowedness are contained in [97].

Proposition 15.1 Let P be a normal program and G a normal goal. Suppose that $P \cup \{G\}$ is allowed. Then we have the following properties.

(a) No computation of $P \cup \{G\}$ flounders.

(b) Every computed answer for $P \cup \{G\}$ is a ground substitution for all variables in G.

Proof (a) Since $P \cup \{G\}$ is allowed, one can prove that every goal in an SLDNF-derivation of $P \cup \{G\}$ (including subsidiary derivations) is allowed. The result then follows as a goal containing only non-ground negative literals is not allowed.

(b) Let G be $\leftarrow L_1,...,L_m$ and let $G_0=G, G_1,...,G_n=\square$ be an SLDNF-refutation

of $P \cup \{G\}$ using substitutions $\theta_1,...,\theta_n$. Note that any input clause whose head is matched against a positive literal in (the top level of) the refutation has the property that each variable which occurs in the head also occurs in the body. It is straightforward to prove by induction on the length n of the refutation that $(L_1 \wedge ... \wedge L_m)\theta_1...\theta_n$ is ground. The result then follows. ■

The next result of this section is the soundness of the negation as failure rule. In preparation for the proof of this result, we establish two lemmas due to Clark [15].

Lemma 15.2 Let $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$ be atoms.
(a) If $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$ are not unifiable, then $\sim\exists((s_1=t_1)\wedge...\wedge(s_n=t_n))$ is a logical consequence of the equality theory.
(b) If $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$ are unifiable with mgu $\theta = \{x_1/r_1,...,x_k/r_k\}$ given by the unification algorithm, then $\forall((s_1=t_1)\wedge...\wedge(s_n=t_n) \leftrightarrow (x_1=r_1)\wedge...\wedge(x_k=r_k))$ is a logical consequence of the equality theory.

Proof Suppose that $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$ are unifiable with mgu $\theta = \{x_1/r_1,...,x_k/r_k\}$. Then it follows from equality axioms 6, 7 and 8 that $\forall((s_1=t_1)\wedge...\wedge(s_n=t_n)\leftarrow(x_1=r_1)\wedge...\wedge(x_k=r_k))$ is a logical consequence of the equality theory. The remainder of the lemma is proved by induction on the number of steps k of an attempt by the unification algorithm to unify $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$.

Suppose first that k=1. If the unification algorithm finds a substitution $\{x_1/r_1\}$, say, which unifies $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$, then equality axiom 5 can be used to show that $\forall((s_1=t_1)\wedge...\wedge(s_n=t_n)\rightarrow(x_1=r_1))$ is a logical consequence of the equality theory. Otherwise, we use equality axiom 5 and one of the equality axioms 1 to 4 to conclude that $\sim\exists((s_1=t_1)\wedge...\wedge(s_n=t_n))$ is a logical consequence of the equality theory.

Suppose now that the result holds for k–1. Let $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$ be such that it takes the unification algorithm k steps to decide whether they are unifiable or not. Suppose that $\theta_1 = \{x_1/r_1'\}$ is the first substitution made by the unification algorithm. Then $p(s_1,...,s_n)\theta_1$ and $p(t_1,...,t_n)\theta_1$ are such that the unification algorithm can discover in k–1 steps whether they are unifiable or not.

Suppose that $p(s_1,...,s_n)\theta_1$ and $p(t_1,...,t_n)\theta_1$ are not unifiable. Then the induction hypothesis gives that $\sim\exists((s_1=t_1)\theta_1\wedge...\wedge(s_n=t_n)\theta_1)$ is a logical consequence of the equality theory. It then follows from this and the fact that θ_1 was the first substitution made by the unification algorithm that

$\neg\exists((s_1=t_1)\wedge\ldots\wedge(s_n=t_n))$ is a logical consequence of the equality theory.

On the other hand, suppose that $p(s_1,\ldots,s_n)\theta_1$ and $p(t_1,\ldots,t_n)\theta_1$ are unifiable. Then the induction hypothesis is used to obtain that $\forall((s_1=t_1)\theta_1\wedge\ldots\wedge(s_n=t_n)\theta_1\rightarrow(x_2=r_2)\wedge\ldots\wedge(x_k=r_k))$ is a logical consequence of the equality theory. It follows from this, the fact that r_1 is $r_1'\gamma$, where $\gamma = \{x_2/r_2,\ldots,x_k/r_k\}$, and equality axioms 5, 6, 7 and 8 that $\forall((s_1=t_1)\wedge\ldots\wedge(s_n=t_n)\rightarrow(x_1=r_1)\wedge\ldots\wedge(x_k=r_k))$ is a logical consequence of the equality theory. ■

Lemma 15.3 Let P be a normal program and G a normal goal. Suppose the selected literal in G is positive.

(a) If there are no derived goals, then G is a logical consequence of comp(P).

(b) If the set $\{G_1,\ldots,G_r\}$ of derived goals is non-empty, then $G\leftrightarrow G_1\wedge\ldots\wedge G_r$ is a logical consequence of comp(P).

Proof Suppose G is the normal goal $\leftarrow M_1,\ldots,M_q$ and the selected positive literal M_j is $p(s_1,\ldots,s_n)$. If the completed definition for p is $\forall(\neg p(x_1,\ldots,x_n))$, then it is clear that G is a logical consequence of comp(P).

Next suppose that the completed definition of p is

$$\forall(p(x_1,\ldots,x_n)\leftrightarrow E_1\vee\ldots\vee E_k)$$

where E_i is

$$\exists y_{i,1}\ldots\exists y_{i,d_i}\ ((x_1=t_{i,1})\wedge\ldots\wedge(x_n=t_{i,n})\wedge L_{i,1}\wedge\ldots\wedge L_{i,m_i})$$

It follows that

$$G\leftrightarrow\wedge_{i=1}^{k}\neg\exists(M_1\wedge\ldots\wedge M_{j-1}\wedge(s_1=t_{i,1})\wedge\ldots\wedge(s_n=t_{i,n})\wedge L_{i,1}\wedge\ldots\wedge L_{i,m_i}\wedge M_{j+1}\wedge\ldots\wedge M_q)$$

is a logical consequence of comp(P). If $p(s_1,\ldots,s_n)$ does not unify with the head of any program clause in the definition of p, then it follows from lemma 15.2(a) that G is a logical consequence of comp(P).

On the other hand, suppose θ is an mgu of $p(s_1,\ldots,s_n)$ and $p(t_{i,1},\ldots,t_{i,n})$. Then we have that

$$\exists(M_1\wedge\ldots\wedge M_{j-1}\wedge(s_1=t_{i,1})\wedge\ldots\wedge(s_n=t_{i,n})\wedge L_{i,1}\wedge\ldots\wedge L_{i,m_i}\wedge M_{j+1}\wedge\ldots\wedge M_q)\leftrightarrow$$
$$\exists((M_1\wedge\ldots\wedge M_{j-1}\wedge L_{i,1}\wedge\ldots\wedge L_{i,m_i}\wedge M_{j+1}\wedge\ldots\wedge M_q)\theta)$$

is a logical consequence of comp(P), using lemma 15.2(b) and the equality axioms 6, 7 and 8. Thus, if $\{G_1,\ldots,G_r\}$ is the set of derived goals, then $G\leftrightarrow G_1\wedge\ldots\wedge G_r$ is a logical consequence of comp(P). ■

The next result is due to Clark [15].

Theorem 15.4 (Soundness of the Negation as Failure Rule)

Let P be a normal program and G a normal goal. If $P \cup \{G\}$ has a finitely failed SLDNF-tree, then G is a logical consequence of comp(P).

Proof The proof is by induction on the rank k of the finitely failed SLDNF-tree for $P \cup \{G\}$. Let G be the goal $\leftarrow L_1,...,L_n$.

Suppose first that k=0. Then the result follows by a straightforward induction on the depth of the tree, using lemma 15.3.

Next suppose the result holds for finitely failed SLDNF-trees of rank k. Consider a finitely failed SLDNF-tree of rank k+1 for $P \cup \{G\}$. We establish the result by a secondary induction on the depth of this tree.

Suppose first that the depth of this tree is 1. Suppose the selected literal in G is positive. Then the result follows from lemma 15.3(a). On the other hand, suppose the selected literal L_i in G is the ground negative literal $\neg A_i$. Since the depth is 1, there is an SLDNF-refutation of rank k of $P \cup \{\leftarrow A_i\}$. Note that for a goal whose selected literal is positive, the derived goal is a logical consequence of the given goal and the input clause. Thus, using proposition 14.1 and applying the induction hypothesis on any finitely failed SLDNF-trees of rank k–1 in this refutation, we obtain that A_i is a logical consequence of comp(P). Hence $\neg\exists(L_1 \wedge ... \wedge L_n)$ is also a logical consequence of comp(P). (This last step uses the fact that A_i is ground.)

Now suppose that the finitely failed SLDNF-tree for $P \cup \{G\}$ has depth d+1. Suppose that the selected literal in G is positive. Then the result follows from lemma 15.3(b) and the secondary induction hypothesis. Suppose the selected literal in G is the ground negative literal L_i. By the secondary induction hypothesis, we obtain that $\neg\exists(L_1 \wedge ... \wedge L_{i-1} \wedge L_{i+1} \wedge ... \wedge L_n)$ is a logical consequence of comp(P). Hence $\neg\exists(L_1 \wedge ... \wedge L_n)$ is also a logical consequence of comp(P). ■

Corollary 15.5 Let P be a definite program. If $A \in F_P$, then $\neg A$ is a logical consequence of comp(P).

Now we come to the soundness of SLDNF-resolution. This result, which generalises theorem 7.1, is essentially due to Clark [15].

Theorem 15.6 (Soundness of SLDNF-Resolution)

Let P be a normal program and G a normal goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $comp(P) \cup \{G\}$.

Proof Let G be the normal goal $\leftarrow L_1,...,L_k$ and $\theta_1,...,\theta_n$ be the sequence of substitutions used in an SLDNF-refutation of $P \cup \{G\}$. We have to show that $\forall((L_1 \wedge ... \wedge L_k)\theta_1...\theta_n)$ is a logical consequence of comp(P). The result is proved by induction on the length of the SLDNF-refutation.

Suppose first that n=1. This means that G has the form $\leftarrow L_1$. We consider two cases.

(a) L_1 is positive.

Then P has a unit clause of the form $A\leftarrow$ and $L_1\theta_1 = A\theta_1$. Since $L_1\theta_1\leftarrow$ is an instance of a unit clause of P, it follows that $\forall(L_1\theta_1)$ is a logical consequence of P and, hence, of comp(P).

(b) L_1 is negative.

In this case, L_1 is ground, θ_1 is the identity substitution and theorem 15.4 shows that L_1 is a logical consequence of comp(P).

Next suppose that the result holds for computed answers which come from SLDNF-refutations of length n–1. Suppose $\theta_1,...,\theta_n$ is the sequence of substitutions used in the SLDNF-refutation of $P \cup \{G\}$ of length n. Let L_m be the selected literal of G. Again we consider two cases.

(a) L_m is positive.

Let $A\leftarrow M_1,...,M_q$ $(q\geq 0)$ be the first input clause. By the induction hypothesis, $\forall((L_1 \wedge ... \wedge L_{m-1} \wedge M_1 \wedge ... \wedge M_q \wedge L_{m+1} \wedge ... \wedge L_k)\theta_1...\theta_n)$ is a logical consequence of comp(P). Therefore, if $q>0$, $\forall((M_1 \wedge ... \wedge M_q)\theta_1...\theta_n)$ is a logical consequence of comp(P). Consequently, $\forall(L_m\theta_1...\theta_n)$, which is the same as $\forall(A\theta_1...\theta_n)$, is a logical consequence of comp(P). Hence we have that $\forall((L_1 \wedge ... \wedge L_k)\theta_1...\theta_n)$ is a logical consequence of comp(P).

(b) L_m is negative.

In this case, L_m is ground, θ_1 is the identity substitution and theorem 15.4 shows that L_m is a logical consequence of comp(P). Using the induction hypothesis, we obtain that $\forall((L_1 \wedge ... \wedge L_k)\theta_1...\theta_n)$ is a logical consequence of comp(P). ■

Finally, we turn to the problem of weakening the safeness condition on the selection of literals. First we show that if the safeness condition is dropped, then theorem 15.4 will no longer hold.

Example Consider the normal program P

$p \leftarrow \neg q(x)$

$q(a) \leftarrow$

If we drop the safeness condition, then the literal ~q(x) can be selected and we obtain a "finitely failed SLDNF-tree" for $P \cup \{\leftarrow p\}$. The subgoal ~q(x) fails because there is a refutation of $\leftarrow q(x)$ in which x is bound to a. However, it is easy to see that ~p is not a logical consequence of comp(P).

It is possible to weaken the safeness condition a little and still obtain the results. Consider the following weaker safeness condition. Non-ground negative subgoals are allowed to proceed. If the negative subgoal succeeds, then we proceed as before. However, if the negative subgoal fails, a check is made to make sure no bindings were made to any variables in the top-level goal of the corresponding refutation. If no such binding was made, then the negative subgoal is allowed to fail and we proceed as before. But, if such a binding *was* made, then a different literal is selected and the negative subgoal is delayed in the hope that more of its variables will be bound later. Alternatively, a control error could be generated and the program halted.

The key point here is that the refutation which causes the negative subgoal to fail must prove something of the form $\forall(A)$ rather than only $\exists(A)$. For this weakened safeness condition, theorems 15.4 and 15.6 continue to hold. The only change to their proofs is in the proof of theorem 15.4 at the place where we remarked that use was made of the fact that A was ground.

The simplest way to implement the safeness condition in a PROLOG system is to delay negative subgoals until any variables appearing in the subgoal have been bound to ground terms. For example, this is the method used by MU-PROLOG [73] and NU-PROLOG [104]. Unfortunately, the majority of PROLOG systems do not have a mechanism for delaying subgoals and so this solution is not available to them. Worse still, most PROLOG systems do not bother to check that negative subgoals are ground when called. This can lead to rather bizarre behaviour.

Example Consider the program

p(a) $\leftarrow$

q(b) $\leftarrow$

and the normal goal $\leftarrow$ ~p(x),q(x). If this program and goal are run on a PROLOG system which uses the standard computation rule and does not bother to check that negative subgoals are ground when called, then it will return the answer "no"! On the other hand, MU-PROLOG and NU-PROLOG will delay the first subgoal, solve the second subgoal and then solve the first subgoal to give the correct answer

{x/b}. Of course, the problem with this particular goal can be fixed for a standard PROLOG system by reordering the subgoals in the goal. However, that is not the point. A problem similar to this could lie undetected deep inside a very large and complex software system.

We now discuss the effect that cuts in a normal program can have on the soundness results. In §11, we showed that the existence of a cut in a definite program does not affect the soundness, but may introduce a form of incompleteness into the SLD-resolution implementation of correct answer. However, for normal programs, it is possible for a cut to affect soundness.

Example Consider the subset program

subset(x,y) ← ~p(x,y)

p(x,y) ← member(z,x), ~member(z,y)

member(x, x.y) ← !

member(x, y.z) ← member(x,z)

in which sets are represented by lists. The goal ←subset([1,2,3], [1]) succeeds for this program! The reason is that the unsafe use of cut in the definition of member causes a finitely failed tree for ←p([1,2,3],[1]) to be incorrectly constructed. Hence the negated subgoal ~p([1,2,3],[1]) incorrectly succeeds.

As before, the best solution to the problems of cut seems to be to replace its use by higher level facilities, such as if-then-else and not equals.

§16. COMPLETENESS OF SLDNF-RESOLUTION

In this section, we prove completeness results for the negation as failure rule for definite programs and SLDNF-resolution for hierarchical programs. We also present a summary of the main results of the chapter for definite programs.

The next result is due to Jaffar, Lassez and Lloyd [47]. The simpler definition of the equivalence relation in the proof, which avoids most of the technical complications of the original proof in [47], is due to Wolfram, Maher and Lassez [112].

Theorem 16.1 (Completeness of the Negation as Failure Rule)

Let P be a definite program and G a definite goal. If G is a logical consequence of comp(P), then every fair SLD-tree for $P \cup \{G\}$ is finitely failed.

Proof Let G be the goal $\leftarrow A_1,...,A_q$. Suppose that $P \cup \{G\}$ has a fair SLD-tree which is not finitely failed. We prove that $comp(P) \cup \{\exists(A_1 \wedge ... \wedge A_q)\}$ has a model.

Let BR be any non-failed branch in the fair SLD-tree for $P \cup \{G\}$. Suppose BR is G_0=G, G_1,... with mgu's θ_1, θ_2,... and input clauses C_1, C_2,... . The first step is to use BR to define a pre-interpretation J for P.

Suppose L is the underlying first order language for P. Naturally, L is assumed to be rich enough to support any standardising apart necessary in BR. We define a relation * on the set of all terms in L as follows. Let s and t be terms in L. Then s*t if there exists $n \geq 1$ such that $s\theta_1...\theta_n = t\theta_1...\theta_n$, that is, $\theta_1...\theta_n$ unifies s and t. It is clear that * is indeed an equivalence relation. We then define the domain D of the pre-interpretation J as the set of all *-equivalence classes of terms in L. If s is a term in L, we denote the equivalence class containing s by [s].

Next we give the assignments to the constants and function symbols in L. If c is a constant in L, we assign [c] to c. If f is an n-ary function symbol in L, we assign the mapping from D^n into D defined by $([s_1],...,[s_n]) \rightarrow [f(s_1,...,s_n)]$ to f. It is clear that the mapping is indeed well-defined. This completes the definition of J.

The next task is to give the assignments to the predicate symbols in order to extend J to an interpretation for $comp(P) \cup \{\exists(A_1 \wedge ... \wedge A_q)\}$. First we define the set I_0 as follows:

$$I_0 = \{p([t_1],...,[t_n]) : p(t_1,...,t_n) \text{ appears in BR}\}.$$

We next show that $I_0 \subseteq T_P^J(I_0)$, where T_P^J is the mapping associated with the pre-interpretation J. Suppose that $p([t_1],...,[t_n]) \in I_0$, where $p(t_1,...,t_n)$ appears in some G_i, $i \in \omega$. Because BR is fair and not failed, there exists $j \in \omega$ such that $p(s_1,...,s_n) \approx p(t_1,...,t_n)\theta_{i+1}...\theta_{i+j}$ appears in goal G_{i+j} and $p(s_1,...,s_n)$ is the selected atom in G_{i+j}. Suppose C_{i+j+1} is $p(r_1,...,r_n) \leftarrow B_1,...,B_m$. By the definition of T_P^J, it follows that $p([r_1\theta_{i+j+1}],...,[r_n\theta_{i+j+1}]) \in T_P^J(I_0)$. Then, using the fact that, for each k, $\theta_1...\theta_k$ can be assumed to be idempotent, we have that

$$\begin{aligned}
&p([t_1],...,[t_n]) \\
&= p([t_1\theta_{i+1}...\theta_{i+j}],...,[t_n\theta_{i+1}...\theta_{i+j}]) \\
&= p([s_1],...,[s_n]) \\
&= p([s_1\theta_{i+j+1}],...,[s_n\theta_{i+j+1}]) \\
&= p([r_1\theta_{i+j+1}],...,[r_n\theta_{i+j+1}]),
\end{aligned}$$

so that $p([t_1],...,[t_n]) \in T_P^J(I_0)$. Thus $I_0 \subseteq T_P^J(I_0)$.

Now, by proposition 5.2, there exists I such that $I_0 \subseteq I$ and $I = T_P^J(I)$. I gives

the assignments to the predicate symbols in L. We assign the identity relation on D to =.

This completes the definition of the interpretation I, together with the identity relation assigned to =, for comp(P) ∪ {∃(A_1∧...∧A_q)}. Note that this interpretation is a model for ∃(A_1∧...∧A_q) because $I_0 \subseteq I$. Note further that this interpretation is clearly a model for the equality theory. Hence, proposition 14.3 gives that I, together with the identity relation assigned to =, is a model for comp(P) ∪ {∃(A_1∧...∧A_q)}. ■

Corollary 16.2 Let P be a definite program and $A \in B_P$. If ~A is a logical consequence of comp(P), then $A \in F_P$.

The model constructed in the proof of theorem 16.1 is not an Herbrand model. In fact, the next example shows that theorem 16.1 simply cannot be proved by restricting attention to Herbrand models (based on the constants and function symbols appearing in the program).

Example Consider the program P

p(f(y)) ← p(y)

q(a) ← p(y)

Note that $q(a) \notin F_P$. Now $gfp(T_P)=\emptyset$ and hence $q(a) \notin gfp(T_P)$. According to proposition 14.4, comp(P) ∪ {q(a)} does not have an Herbrand model.

Problem 34 shows that theorem 16.1 generalises to stratified normal programs. However, this generalisation is not really a completeness result because, as the next example shows, the existence of a (fair) SLDNF-tree is not guaranteed, in contrast to the definite case, where fair SLD-trees always exist. To obtain a completeness result for stratified normal programs, it will thus be necessary to impose further restrictions to ensure the existence of a fair SLDNF-tree.

Example Consider the stratified normal program P

q ← ~r

r ← p

r ← ~p

p ← p

Then it is easy to show that ~q is a logical consequence of comp(P), but that P ∪ {←q} does not have an SLDNF-tree. (See problem 20.)

Next, we turn to the question of completeness of SLDNF-resolution.

Example Consider the program

$p(x) \leftarrow$

$q(a) \leftarrow$

$r(b) \leftarrow$

and the goal $\leftarrow p(x), \neg q(x)$. Clearly, x/b is a correct answer. However, this answer can never be computed, nor can any more general version of it.

This simple example clearly illustrates one of the problems in obtaining a completeness result for SLDNF-resolution. SLD-resolution returns most general answers. In the above example, it will return the identity substitution ε for the subgoal p(x). What we would like is for the negation as failure rule to further instantiate x by the binding x/b and thus compute the correct answer. However, negation as failure is only a test and cannot make any bindings. Unless it is presented with a goal which already is the root of a finitely failed SLD-tree, it has no machinery for further instantiating the goal so as to obtain such a tree. In the above example, $\leftarrow q(x)$ is not the root of a finitely failed SLD-tree and negation as failure has no way to find the appropriate binding x/b.

The next example illustrates another problem in obtaining a completeness result for SLDNF-resolution.

Example Consider the normal program P

$r \leftarrow p$

$r \leftarrow \neg p$

$p \leftarrow p$

Then the identity substitution ε is a correct answer for $comp(P) \cup \{\leftarrow r\}$, but ε cannot be computed. (See problem 21.)

These examples show that to obtain a completeness result, it will be necessary to impose rather strong restrictions. We now show that for hierarchical programs, there is such a completeness result. Sadly, this result is not very useful because the hierarchical condition bans any recursion. For the statement of this result, we need to generalise the concept of a computation rule.

Definition A *safe computation rule* is a function from a set of normal goals, none of which consists entirely of non-ground negative literals, to a set of literals such that the value of the function for such a goal is either a positive literal or a

ground negative literal, called the *selected* literal, in that goal.

Definition Let P be a normal program, G a normal goal, and R a safe computation rule.

An *SLDNF-derivation* of $P \cup \{G\}$ *via R* is an SLDNF-derivation of $P \cup \{G\}$ in which the computation rule R is used to select literals.

An *SLDNF-tree* for $P \cup \{G\}$ *via R* is an SLDNF-tree for $P \cup \{G\}$ in which the computation rule R is used to select literals.

An *SLDNF-refutation* of $P \cup \{G\}$ *via R* is an SLDNF-refutation of $P \cup \{G\}$ in which the computation rule R is used to select literals.

An *R-computed answer* for $P \cup \{G\}$ is a computed answer for $P \cup \{G\}$ which has come from an SLDNF-refutation of $P \cup \{G\}$ via R.

Now we can give the completeness result for hierarchical programs. Versions of this result are due to Clark [15], Shepherdson [97], and Lloyd and Topor [63].

Theorem 16.3 (Completeness of SLDNF-Resolution for Hierarchical Programs)
Let P be a hierarchical normal program, G a normal goal, and R a safe computation rule. Suppose that $P \cup \{G\}$ is allowed. Then the following properties hold.
(a) The SLDNF-tree for $P \cup \{G\}$ via R exists and is finite.
(b) If θ is a correct answer for $comp(P) \cup \{G\}$ and θ is a ground substitution for all variables in G, then θ is an R-computed answer for $P \cup \{G\}$.

Proof (a) By proposition 15.1(a), the computation of $P \cup \{G\}$ via R does not flounder.

To show that there are no infinite derivations, we use multisets. If M and M' are finite multisets of non-negative integers, then we define $M' < M$ if M' can be obtained from M by replacing one or more elements in M by any finite number of non-negative integers, each of which is smaller than one of the replaced elements. It is shown in [28] that the set of all finite multisets of non-negative integers under $<$ is a well-founded set. Now consider the multiset of levels of the predicate symbols in the literals of the body of a goal G' in an SLDNF-derivation via R. Since P is hierarchical, the child of the goal G' has a smaller multiset than G'. Hence there are no infinite derivations.

Moreover, an induction argument on the levels of predicate symbols shows that the SLDNF-tree for $P \cup \{G\}$ via R does indeed exist.

(b) Note that, by corollary 14.8, comp(P) is consistent because P is

hierarchical. Let G be the goal $\leftarrow L_1,...,L_n$. The SLDNF-tree for $P \cup \{G\theta\}$ via R is not finitely failed; otherwise, by theorem 15.4, we would have that $\sim(L_1 \wedge ... \wedge L_n)\theta$ is a logical consequence of comp(P), which contradicts the consistency of comp(P) and the assumption that θ is correct.

Hence there exists an SLDNF-refutation for $P \cup \{G\theta\}$ via R. We now modify the selection of literals in (the top level of) this refutation so that the first part of the refutation contains goals in which the selected literal is positive and the last part contains goals in which the selected literal is negative. We can now apply the argument of lemma 8.2, the fact that θ is a ground substitution for all the variables in G, and the allowedness of $P \cup \{G\}$ to obtain an SLDNF-refutation of $P \cup \{G\}$ in which the computed answer is θ.

We next apply essentially the argument of lemma 9.1 so that the selection of literals in (the top level of) this refutation is made using R. Since any subsidiary finitely failed trees are not modified by these constructions, their literals are still selected using R. Thus θ is an R-computed answer for $P \cup \{G\}$. ■

For further discussion and results on completeness the reader is referred to [95], [97] and [98]. The completeness of the negation as failure rule and SLDNF-resolution are of such importance that finding more general completeness results is an urgent priority. The most interesting completeness results would be for classes of stratified programs, which strictly include the class of hierarchical programs.

Finally, we summarise the main results for definite programs given in this chapter. First we need one more definition. The *Herbrand rule* is as follows: if $comp(P) \cup \{A\}$ has no Herbrand model, then infer $\sim A$.

We now have three possible rules for inferring negative information: the CWA, the Herbrand rule and the negation as failure rule. If P is a definite program, then we have the following results (see Figure 6):

$\{A \in B_P : \sim A \text{ can be inferred under the negation as failure rule}\} = B_P \backslash T_P{\downarrow}\omega$
$\{A \in B_P : \sim A \text{ can be inferred under the Herbrand rule}\} = B_P \backslash gfp(T_P)$
$\{A \in B_P : \sim A \text{ can be inferred under the CWA}\} = B_P \backslash T_P{\uparrow}\omega$

Since $T_P{\uparrow}\omega \subseteq gfp(T_P) \subseteq T_P{\downarrow}\omega$, it follows that the CWA is the most powerful rule, followed by the Herbrand rule, followed by the negation as failure rule. Since $T_P{\uparrow}\omega$, $gfp(T_P)$ and $T_P{\downarrow}\omega$ are generally distinct (see problem 5, chapter 2), it follows that the rules are distinct.

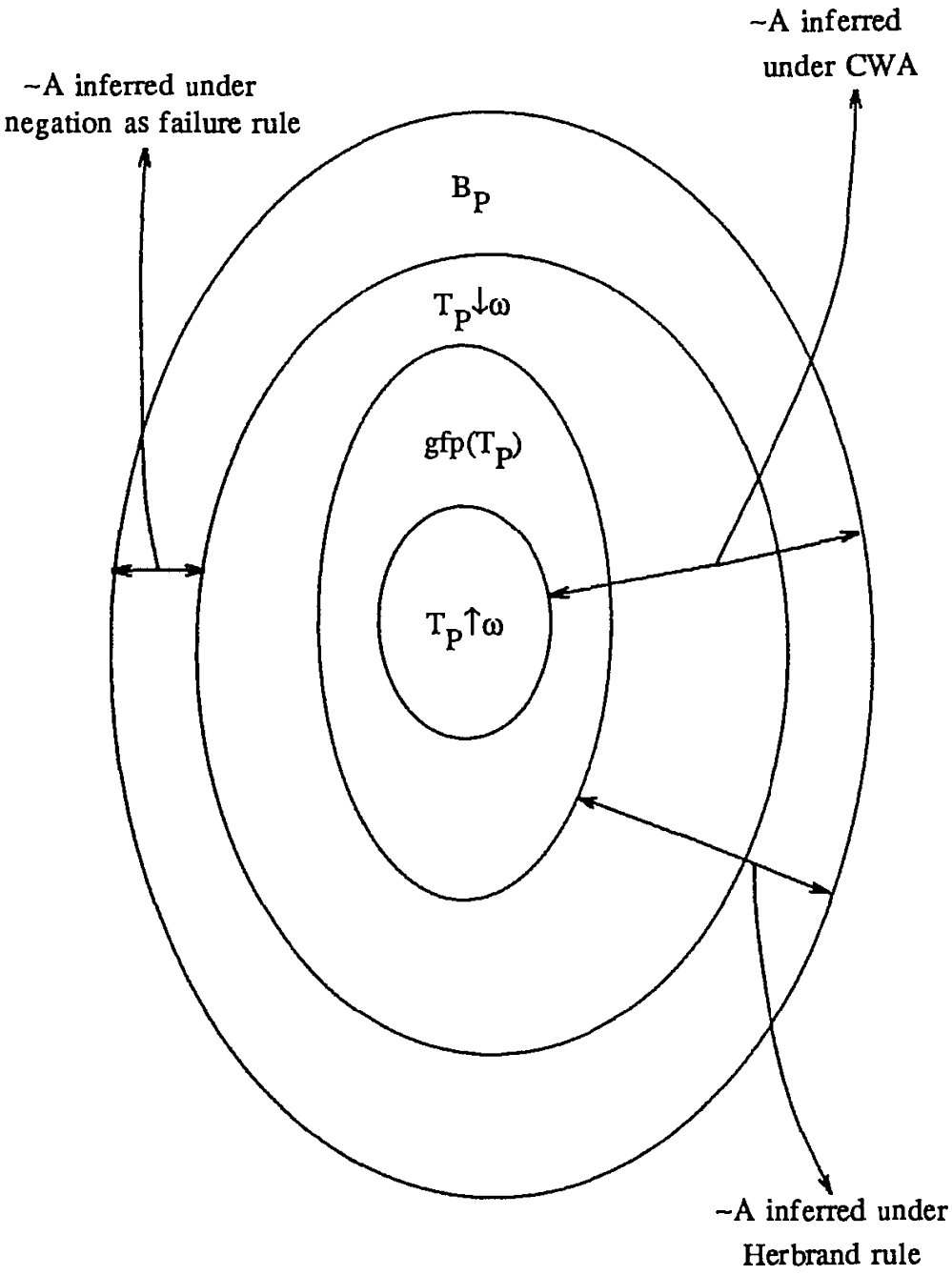

Fig. 6. Relationship between the various rules

We can combine theorem 13.6 with corollaries 15.5 and 16.2.

Theorem 16.4 Let P be a definite program and $A \in B_P$. Then the following are equivalent:
(a) $A \in F_P$.
(b) $A \notin T_P \downarrow \omega$.
(c) A is in the SLD finite failure set.
(d) Every fair SLD-tree for $P \cup \{\leftarrow A\}$ is finitely failed.
(e) ~A is a logical consequence of comp(P).

We can also combine theorems 15.4 and 16.1.

Theorem 16.5 Let P be a definite program and G a definite goal. Then G is a logical consequence of comp(P) iff $P \cup \{G\}$ has a finitely failed SLD-tree.

It is also worth emphasising the following facts, which highlight the difference between (arbitrary) models and Herbrand models for comp(P) and between $T_P \downarrow \omega$ and $gfp(T_P)$. Let $A \in B_P$. Then we have the following properties:
(a) $A \in gfp(T_P)$ iff $comp(P) \cup \{A\}$ has an Herbrand model.
(b) $A \in T_P \downarrow \omega$ iff $comp(P) \cup \{A\}$ has a model.

PROBLEMS FOR CHAPTER 3

1. Let P be a definite program. Show that $F_P^d = B_P \setminus T_P \downarrow d$, for $d \geq 1$.

2. Prove lemma 13.2.

3. Prove lemma 13.3.

4. Show that the converse of proposition 13.4 does not hold. In fact, show that, given k, there exists a definite program P and $A \in B_P$ such that $A \notin T_P \downarrow 2$ and yet the depth of every SLD-tree for $P \cup \{\leftarrow A\}$ is at least k.

5. Let P be a definite program and G a definite goal. Then G is called *infinite* (with respect to P) if every SLD-tree for $P \cup \{G\}$ is infinite. Show that there exists a program P and $A \in B_P$ such that $\leftarrow A$ is infinite and yet A is in the success set of P.

6. Let P be a definite program, $A \in B_P$ and A not be in the success set of P. Show that $\leftarrow A$ is infinite iff A is not in the SLD finite failure set.

7. Consider the program P

$p(x) \leftarrow q(y), r(y)$

$q(h(y)) \leftarrow q(y)$

$r(g(y)) \leftarrow$

Find two SLD-trees for $P \cup \{\leftarrow p(a)\}$, one of which is infinite and the other finitely failed.

8. Give an example of a normal program P such that comp(P) is not consistent.

9. Use equality axioms 6 and 8 to show that, in any model of the equality theory, the relation assigned to = is an equivalence relation.

10. Let P be a normal program and $s,t \in U_P$. Prove the following:
(a) s=s is a logical consequence of the equality theory.
(b) If s and t are syntactically different, then $s \neq t$ is a logical consequence of the equality theory.
(c) The domain of every model for comp(P) contains an isomorphic copy of U_P and the relation assigned to =, when restricted to U_P, is the identity relation.

11. Prove proposition 14.2.

12. Show that proposition 14.5 does not hold for normal programs.

13. Prove proposition 14.7.

14. Show that lemma 15.2 (b) does not hold if we drop the phrase "given by the unification algorithm" from its statement.

15. Show that corollary 15.5 no longer holds if we drop any one of the equality axioms 1 to 5 from the definition of comp(P).

16. Show that the safeness condition cannot be dropped from theorem 15.6.

17. Consider the normal program P

$p \leftarrow \neg q(x)$

$q(a) \leftarrow$

Show that $\neg p$ is not a logical consequence of comp(P).

18. Consider the normal program P

$p \leftarrow \neg r$

$r \leftarrow q(x)$

$q(a) \leftarrow$

Show that $P \cup \{\leftarrow p\}$ has a finitely failed SLDNF-tree and that $\neg p$ is a logical consequence of comp(P). This program looks equivalent to the one in problem 17. Explain the difference.

19. Consider the definite program P

$p(f(y)) \leftarrow p(y)$

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

and let A be q(a). What is the model for comp(P) $\cup$ {A} given by the construction in theorem 16.1 for this program? Show that the domain of this model is isomorphic to $U_P \cup Z$, where Z is the integers.

20. Consider the normal program P

$q \leftarrow \neg r$

$r \leftarrow p$

$r \leftarrow \neg p$

$p \leftarrow p$

Show that $\neg q$ is a logical consequence of comp(P), but that $P \cup \{\leftarrow q\}$ does not have an SLDNF-tree.

21. Consider the normal program P

$r \leftarrow p$

$r \leftarrow \neg p$

$p \leftarrow p$

Show that the identity substitution ε is a correct answer for comp(P) $\cup$ $\{\leftarrow r\}$, but that ε cannot be computed.

22. Give an example of a normal program whose completion has a model, but no

Herbrand model (based on the constants and function symbols appearing in the program).

23. Give an example of a normal program P and goal G such that the computation of $P \cup \{G\}$ produces an infinite nested sequence of negated calls, but the computation never flounders (in the sense of §15) and never produces an infinite branch. Prove that, if P is stratified, there can never be an infinite nested sequence of negated calls.

24. Let P be a normal program and G a normal goal. Suppose that $P \cup \{G\}$ has a finitely failed SLDNF-tree. Prove that there exists a safe computation rule R such that $P \cup \{G\}$ has a finitely failed SLDNF-tree via R.

25. Let P be a normal program and G a normal goal. Suppose that $P \cup \{G\}$ has a computed answer θ. Prove that there exists a safe computation rule R and an R-computed answer ϕ for $P \cup \{G\}$ such that $G\phi$ is a variant of $G\theta$.

26. Let P be a normal program and G a normal goal. Suppose that $P \cup \{G\}$ has a computed answer θ. Let γ be a substitution. Prove that $P \cup \{G\theta\gamma\}$ has the identity substitution as a computed answer.

27. Let P be a normal program and G a normal goal. Suppose that $P \cup \{G\}$ has a finitely failed SLDNF-tree. Let γ be a substitution. Prove that $P \cup \{G\gamma\}$ has a finitely failed SLDNF-tree.

28. Let P be a normal program and G a ground normal goal $\leftarrow L_1,...,L_n$. Suppose that $P \cup \{G\}$ has a finitely failed SLDNF-tree. Prove that there exists $i \in \{1,...,n\}$ such that $P \cup \{\leftarrow L_i\}$ has a finitely failed SLDNF-tree.

29. Let P be a normal program and G a ground normal goal $\leftarrow L_1,...,L_n$. Suppose that $P \cup \{G\}$ has an SLDNF-refutation. Prove that $P \cup \{\leftarrow L_i\}$ has an SLDNF-refutation, for all $i \in \{1,...,n\}$.

30. Let P be a normal program and G a normal goal. Suppose that $P \cup \{G\}$ has an SLDNF-refutation. Prove that $P \cup \{G\}$ does not have a finitely failed SLDNF-tree.

31. Let P be a normal program. Define $M = \{A \in B_P : P \cup \{\leftarrow A\}$ does not have a finitely failed SLDNF-tree$\}$. Prove that M is a model for P.

32. Let P be a normal program and G a normal goal. Put $P^* = P \cup \{\neg A : A \in B_P$ and $P \cup \{\leftarrow A\}$ has a finitely failed SLDNF-tree$\}$. Determine whether the following statements are correct or not:
(a) If $P \cup \{G\}$ has a finitely failed SLDNF-tree, then $P \cup \{G\}$ is consistent.
(b) If $P \cup \{G\}$ has a finitely failed SLDNF-tree, then G is a logical consequence of P^*.

33. Let P be a definite program and G an allowed normal goal. Determine whether the following statement is correct or not:
If $comp(P) \cup \{G\}$ is unsatisfiable, then there is a correct answer for $comp(P) \cup \{G\}$.

34. Let P be a normal program and G a normal goal. An SLDNF-derivation for $P \cup \{G\}$ is *fair* if it is either failed or, for every literal L in (the top level of) the derivation, (some further instantiated version of) L is selected within a finite number of steps. An SLDNF-tree for $P \cup \{G\}$ is *fair* if every (top level) branch of the tree is a fair SLDNF-derivation. Prove the following generalisation of theorem 16.1:
Let P be a stratified normal program and G a normal goal. If G is a logical consequence of comp(P), then every fair SLDNF-tree for $P \cup \{G\}$ is finitely failed.

35. Let P be a stratified normal program and A a ground atom. Suppose that A is a logical consequence of comp(P). Let P^* be the definite program obtained from P by deleting all negative literals appearing in the bodies of program clauses in P. Prove that A is a logical consequence of P^*.

36. Give an example of an infinite SLDNF-derivation which has subsidiary finitely failed trees of unbounded rank. (In other words, the derivation does not have rank k, for any k.)

37. Let R be any computation rule. Prove that there exists an SLD-derivation via R which is not fair.

NOTATION

$\cap$ intersection

$\cup$ union

$\in$ membership

$\subseteq$ improper subset

$\supseteq$ improper superset

$\subset$ subset

$\supset$ superset

$\leftarrow, \rightarrow$ implication

$\leftrightarrow$ equivalence

$\wedge$ conjunction

$\vee$ disjunction

$\sim$ negation

$\forall$ universal quantifier

$\exists$ existential quantifier

$\forall(F)$ universal closure of F

$\exists(F)$ existential closure of F

$\forall_\tau$ universal quantifier of type τ

$\exists_\tau$ existential quantifier of type τ

$\varnothing$ empty set

∞ infinity

$|X|$ cardinality of X

$X \backslash Y$ set difference

$X \times Y$ cartesian product

$\square$ empty clause

$=$ equality predicate

$=_\tau$ equality predicate of type τ

$\blacksquare$ end of proof

$\top$ top element

$\bot$ bottom element

ω non-negative integers

2^S set of all subsets of S

gfp(T) greatest fixpoint of T

lfp(T) least fixpoint of T

glb(X) greatest lower bound of X

lub(X) least upper bound of X

P program

G goal

D database

Q query

comp(P) completion of a program P

comp(D) completion of a database D