ARTIFICIAL INTELLIGENCE

J.W. Lloyd

Foundations of Logic Programming

Second, Extended Edition

Springer-Verlag

J. W. Lloyd

Foundations of Logic Programming

Second, Extended Edition

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo

John Wylie Lloyd

Department of Computer Science
University of Bristol
Queen's Building, University Walk
Bristol BS8 1TR, UK

First Corrected Printing 1993

ISBN 3-540-18199-7 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-18199-7 Springer-Verlag New York Berlin Heidelberg

ISBN 3-540-13299-6 1. Auflage Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-13299-6 1st edition Springer-Verlag New York Berlin Heidelberg Tokyo

Library of Congress Cataloging in Publication Data. Lloyd, J. W. (John Wylie), 1947-. Foundations of logic programming. (Symbolic computation. Artificial intelligence) Bibliography: p. Includes index. 1. Logic programming. 2. Programming languages (Electronic computers)–Semantics. I. Title. II. Series. QA76.6.L583 1987 005.1 87-20753
ISBN 0-387-18199-7 (U.S.)

Printing: Druckhaus Beltz, 6944 Hemsbach/Bergstr.
Bookbinding: J. Schäffer GmbH & Co. KG, 6718 Grünstadt
45/3140-543210

To
Susan, Simon and Patrick

PREFACE TO THE SECOND EDITION

In the two and a half years since the first edition of this book was published, the field of logic programming has grown rapidly. Consequently, it seemed advisable to try to expand the subject matter covered in the first edition. The new material in the second edition has a strong database flavour, which reflects my own research interests over the last three years. However, despite the fact that the second edition has about 70% more material than the first edition, many worthwhile topics are still missing. I can only plead that the field is now too big to expect one author to cover everything.

In the second edition, I discuss a larger class of programs than that discussed in the first edition. Related to this, I have also taken the opportunity to try to improve some of the earlier terminology. Firstly, I introduce "program statements", which are formulas of the form $A \leftarrow W$, where the head A is an atom and the body W is an arbitrary formula. A "program" is a finite set of program statements. There are various restrictions of this class. "Normal" programs are ones where the body of each program statement is a conjunction of literals. (The terminology "general", used in the first edition, is obviously now inappropriate). This terminology is new and I risk causing some confusion. However, there is no widely used terminology for such programs and "normal" does have the right connotation. "Definite" programs are ones where the body of each program statement is a conjunction of atoms. This terminology is more standard.

The material in chapters 1 and 2 of the first edition has been reorganised so that the first chapter now contains all the preliminary results and the second chapter now contains both the declarative and procedural semantics of definite programs. In addition, chapter 2 now contains a proof of the result that every computable function can be computed by a definite program.

Further material on negation has been added to chapter 3, which is now entitled "Normal Programs". This material includes discussions of the consistency of the completion of a normal program, the floundering of SLDNF-resolution, and the completeness of SLDNF-resolution for hierarchical programs.

The fourth chapter is a new one on (unrestricted) programs. There is no good practical or theoretical reason for restricting the bodies of program statements or goals to be conjunctions of literals. Once a single negation is allowed, one should go all the way and allow arbitrary formulas. This chapter contains a discussion of SLDNF-resolution for programs, the main results being the soundness of the negation as failure rule and SLDNF-resolution. There is also a discussion of error diagnosis in logic programming, including proofs of the soundness and completeness of a declarative error diagnoser.

The fifth chapter builds on the fourth by giving a theoretical foundation for deductive database systems. The main results of the chapter are soundness and completeness results for the query evaluation process and a simplification theorem for integrity constraint checking. This chapter should also prove useful to those in the "conventional" database community who want to understand the impact logic programming is having on the database field.

The last chapter of the second edition is the same as the last chapter of the first edition on perpetual processes. This chapter is still the most speculative and hence has been left to the end. It can be read directly after chapter 2, since it does not depend on the material in chapters 3, 4 and 5.

This second edition owes much to Rodney Topor, who collaborated with me on four of the papers reported here. Various people made helpful suggestions for improvements of the first edition and drafts of the second edition. These include David Billington, Torkel Franzen, Bob Kowalski, Jean-Louis Lassez, Donald Loveland, Gabor Markus, Jack Minker, Ken Ross, John Shepherdson, Harald Sondergaard, Liz Sonenberg, Rodney Topor, Akihiro Yamamoto and Songyuan Yan. John Crossley read the entire manuscript and found many improvements. John Shepherd showed me how to use ditroff to produce an index. He also introduced me to the delights of cip, which I used to draw the figures. Rodney Topor helped with the automation of the references.

April 1987 JWL

PREFACE TO THE FIRST EDITION

This book gives an account of the mathematical foundations of logic programming. I have attempted to make the book self-contained by including proofs of almost all the results needed. The only prerequisites are some familiarity with a logic programming language, such as PROLOG, and a certain mathematical maturity. For example, the reader should be familiar with induction arguments and be comfortable manipulating logical expressions. Also the last chapter assumes some acquaintance with the elementary aspects of metric spaces, especially properties of continuous mappings and compact spaces.

Chapter 1 presents the declarative aspects of logic programming. This chapter contains the basic material from first order logic and fixpoint theory which will be required. The main concepts discussed here are those of a logic program, model, correct answer substitution and fixpoint. Also the unification algorithm is discussed in some detail.

Chapter 2 is concerned with the procedural semantics of logic programs. The declarative concepts are implemented by means of a specialised form of resolution, called SLD-resolution. The main results of this chapter concern the soundness and completeness of SLD-resolution and the independence of the computation rule. We also discuss the implications of omitting the occur check from PROLOG implementations.

Chapter 3 discusses negation. Current PROLOG systems implement a form of negation by means of the negation as failure rule. The main results of this chapter are the soundness and completeness of the negation as failure rule.

Chapter 4 is concerned with the semantics of perpetual processes. With the advent of PROLOG systems for concurrent applications, this has become an area of great theoretical importance.

The material of chapters 1 to 3, which is now very well established, could be described as "what every PROLOG programmer should know". In chapter 4, I have allowed myself the luxury of some speculation. I believe the material presented there will eventually be incorporated into a much more extensive theoretical foundation for concurrent PROLOGs. However, this chapter is incomplete insofar as I have confined attention to a single perpetual process. Problems of concurrency and communication, which are not very well understood at the moment, have been ignored.

My view of logic programming has been greatly enriched by discussions with many people over the last three years. In this regard, I would particularly like to thank Keith Clark, Maarten van Emden, Jean-Louis Lassez, Frank McCabe and Lee Naish. Also various people have made suggestions for improvements of earlier drafts of this book. These include Alan Bundy, Hervé Gallaire, Joxan Jaffar, Donald Loveland, Jeffrey Schultz, Marek Sergot and Rodney Topor. To all these people and to others who have contributed in any way at all, may I say thank you.

July 1984 JWL

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 $A \leftarrow B_1,\ldots,B_n$ is regarded as a *procedure definition*. If $\leftarrow C_1,\ldots,C_k$ is a goal, then each C_j is regarded as a *procedure call*. A program is run by giving it an initial goal. If the current goal is $\leftarrow C_1,\ldots,C_k$, a step in the computation involves unifying some C_j with the head A of a program clause $A \leftarrow B_1,\ldots,B_n$ and thus reducing the current goal to the goal $\leftarrow (C_1,\ldots,C_{j-1},B_1,\ldots,B_n,C_{j+1},\ldots,C_k)\theta$, where θ is the unifying substitution. Unification thus becomes a uniform mechanism for parameter passing, data selection and data construction. The computation terminates when the empty goal is produced.

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

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

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

For ''system'' languages, the emphasis is on AND-parallelism, don't-care non-determinism and definite programs (that is, no negation). In these languages, according to the *process interpretation* of logic, a goal $\leftarrow B_1,...,B_n$ 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:

$$\begin{array}{c} \sim, \forall, \exists \\ \vee \\ \wedge \\ \rightarrow, \leftrightarrow \end{array}$$

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

Definition A *term* is defined inductively as follows:

(a) A variable is a term.

(b) A constant is a term.

(c) If f is an n-ary function symbol and $t_1,...,t_n$ are terms, then $f(t_1,...,t_n)$ is a term.

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

(a) If p is an n-ary predicate symbol and $t_1,...,t_n$ are terms, then $p(t_1,...,t_n)$ is a formula (called an *atomic formula* or, more simply, an *atom*).

(b) If F and G are formulas, then so are $(\sim F)$, $(F \land G)$, $(F \lor G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$.

(c) If F is a formula and x is a variable, then $(\forall x\ F)$ and $(\exists x\ F)$ are formulas.

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

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

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

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

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

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

$\exists x\ (p(x,y) \wedge q(x))$, all occurrences of x are bound, since the scope of $\exists x$ is $p(x,y) \wedge q(x)$.

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

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

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

Example If F is $p(x,y) \wedge q(x)$, then $\forall(F)$ is $\forall x \forall y\ (p(x,y) \wedge q(x))$, while $\exists(F)$ is $\exists x \exists y\ (p(x,y) \wedge q(x))$.

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

Definition An atom A *occurs positively* in A.

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

If atom A occurs positively (resp., negatively) in a formula W, then A *occurs negatively* (resp., *positively*) in $\sim W$ and $V \leftarrow W$.

Next we introduce an important class of formulas called clauses.

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

Definition A *clause* is a formula of the form

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

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

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

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

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

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

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

is equivalent to

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

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

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

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

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

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

$$A \leftarrow$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

then this clausal notation is shorthand for

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

or, equivalently,

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

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

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

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

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

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

§3. INTERPRETATIONS AND MODELS

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

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

First order logic provides methods for deducing the theorems of a theory. These can be characterised (by Gödel's completeness theorem [69], [99]) as the formulas which are logical consequences of the axioms of the theory, that is, they are true in every interpretation which is a model of each of the axioms of the theory. In particular, each theorem is true in the intended interpretation of the theory. The logic programming systems in which we are interested use the resolution rule as the only inference rule.

Suppose we want to prove that the formula

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

is a logical consequence of a program P. Now resolution theorem provers are refutation systems. That is, the negation of the formula to be proved is added to the axioms and a contradiction is derived. If we negate the formula we want to prove, we obtain the goal

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

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

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

is indeed a logical consequence of P.

From a theorem proving point of view, the only interest is to demonstrate logical consequence. However, from a programming point of view, we are much more interested in the bindings that are made for the variables $y_1, \ldots, y_r$, because these give us the *output* from the running of the program. In fact, the ideal view of a logic programming system is that it is a black box for computing bindings and our only interest is in its input-output behaviour. The internal workings of the system should be invisible to the programmer. Unfortunately, this situation is not true, to various extents, with current PROLOG systems. Many programs can only be understood in a procedural (i.e. operational) manner, because of the way they use cuts and other non-logical features.

Returning to the slowsort program, from a theorem proving point of view, we can regard the goal ←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 $t'_1,\ldots,t'_n$ are the term assignments of $t_1,\ldots,t_n$ and f' is the assignment of the n-ary function symbol f, then $f'(t'_1,\ldots,t'_n) \in D$ is the term assignment of $f(t_1,\ldots,t_n)$.

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 $p(t_1,\ldots,t_n)$ and $d_1,\ldots,d_n$ in the domain of J are the term assignments of $t_1,\ldots,t_n$ wrt J and V. We call $A_{J,V} = p(d_1,\ldots,d_n)$ 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 $p(d_1,\ldots,d_n)$ 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 $p(t_1,\ldots,t_n)$, then the truth value is obtained by calculating the value of $p'(t'_1,\ldots,t'_n)$, where p' is the mapping assigned to p by I and $t'_1,\ldots,t'_n$ are the term assignments of $t_1,\ldots,t_n$ wrt I and V.

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

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

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

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

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

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

We say W is *satisfiable in* I if ∃(W) is true wrt I.

We say W is *valid in* I if ∀(W) is true wrt I.

We say W is *unsatisfiable in* I if ∃(W) is false wrt I.

We say W is *nonvalid in* I if ∀(W) is false wrt I.

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

Example Consider the formula ∀x∃y p(x,y) and the following interpretation I. Let the domain D be the non-negative integers and let p be assigned the relation <. Then I is a model of the formula, as is easily seen. In I, the formula expresses the true statement that "for every non-negative integer, there exists a non-negative integer which is strictly larger than it''. On the other hand, I is not a model of the formula $\exists y \forall x\ p(x,y)$.

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

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

If T has a model, we say T is *consistent*.

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

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

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

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

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

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

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

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

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

Conversely, suppose $S \cup \{\sim F\}$ is unsatisfiable. Let I be any interpretation of L. Suppose I is a model for S. Since $S \cup \{\sim F\}$ is unsatisfiable, I cannot be a model for ~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 $\leftarrow B_1,\ldots,B_n$ with variables $y_1,\ldots,y_r$, then proposition 3.1 states that showing $P \cup \{G\}$ unsatisfiable is exactly the same as showing that $\exists y_1\ldots\exists y_r\ (B_1 \wedge\ldots\wedge B_n)$ 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 $(t_1,...,t_n) \to f(t_1,...,t_n)$ is assigned to f.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$Qx_1 \ldots Qx_k \; ((L_{11} \lor \ldots \lor L_{1m_1}) \land \ldots \land (L_{n1} \lor \ldots \lor L_{nm_n}))$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We say I is *based on* J.

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

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

§4. UNIFICATION

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

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

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

Definition Let $\theta = \{v_1/t_1,\ldots,v_n/t_n\}$ be a substitution and E be an expression. Then $E\theta$, the *instance* of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variable v_i in E by the term t_i $(i=1,\ldots,n)$. 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 $S = \{E_1,\ldots,E_n\}$ is a finite set of expressions and θ is a substitution, then $S\theta$ denotes the set $\{E_1\theta,\ldots,E_n\theta\}$.

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

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

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 $v_j\in\{u_1,\ldots,u_m\}$.

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 $\theta = \{u_1/s_1,\ldots,u_m/s_m\}$ and $\sigma = \{v_1/t_1,\ldots,v_n/t_n\}$. If $x\notin\{u_1,\ldots,u_m\} \cup \{v_1,\ldots,v_n\}$, then $(x\theta)\sigma = x = x(\theta\sigma)$. If $x\in\{u_1,\ldots,u_m\}$, say $x=u_i$, then $(x\theta)\sigma = s_i\sigma = x(\theta\sigma)$. If $x\in\{v_1,\ldots,v_n\}\backslash\{u_1,\ldots,u_m\}$, say $x=v_j$, then $(x\theta)\sigma = t_j = x(\theta\sigma)$.

(c) Clearly it suffices to show that if x is a variable, then $x((\theta\sigma)\gamma) = x(\theta(\sigma\gamma))$. In fact, $x((\theta\sigma)\gamma) = (x(\theta\sigma))\gamma = ((x\theta)\sigma)\gamma = (x\theta)(\sigma\gamma) = x(\theta(\sigma\gamma))$, by (b). ∎

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

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

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

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

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

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

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

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

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

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

Example {p(f(x),z), p(y,a)} is unifiable, since 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 $\gamma_{k+1} = \gamma_k \setminus \{v/v\gamma_k\}$. If γ_k has a binding for v, then

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

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

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

The unification algorithm which we have presented can be very inefficient. In the worst case, its running time can be an exponential function of the length of the input. Consider the following example, which is taken from [9]. Let $S = \{p(x_1,...,x_n), p(f(x_0,x_0),...,f(x_{n-1},x_{n-1}))\}$. 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×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 $\text{lub}(X)$. Similarly, the greatest lower bound of X is unique, if it exists, and is denoted by $\text{glb}(X)$.

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

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

Example In the previous example, 2^S under $\subseteq$ is a complete lattice. In fact, the least upper bound of a collection of subsets of S is their union and the greatest lower bound is their intersection. The top element is S and the bottom element is $\emptyset$.

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

Proof Put $G = \{x : T(x) \leq x\}$ and $g = \text{glb}(G)$. 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 $g' = \text{glb}\{x : T(x)=x\}$. 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 $T : L \rightarrow L$ 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 $a'=\text{gfp}(T)$ and $b'=\text{lfp}(T)$. ∎

We will also require the concept of ordinal powers of T. First we recall some elementary properties of ordinal numbers, which we will refer to more simply as ordinals. Intuitively, the ordinals are what we use to count with. The first ordinal 0 is defined to be $\emptyset$. Then we define $1 = \{\emptyset\} = \{0\}$, $2 = \{\emptyset, \{\emptyset\}\} = \{0, 1\}$, $3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} = \{0, 1, 2\}$, and so on. These are the finite ordinals, the non-negative integers. The first infinite ordinal is $\omega = \{0, 1, 2,...\}$, the set of all non-negative integers. We adopt the convention of denoting finite ordinals by roman letters n, m,..., while arbitrary ordinals will be denoted by Greek letters α, β,.... We can specify an ordering < on the collection of all ordinals by defining $\alpha<\beta$ if $\alpha\in\beta$. For example, $n<\omega$, for all finite ordinals n. We will normally write $n\in\omega$ rather than $n<\omega$. If α is an ordinal, the *successor* of α is the ordinal $\alpha+1 = \alpha \cup \{\alpha\}$, which is the least ordinal greater than α. $\alpha+1$ is then said to be a *successor ordinal*. For example, $1 = 0+1$, $2 = 1+1$, $3 = 2+1$, and so on. If α is a successor ordinal, say $\alpha = \beta+1$, we denote β by $\alpha-1$. An ordinal α is said to be a *limit ordinal* if it is not the successor of any ordinal. The smallest limit ordinal (apart from 0) is ω. After ω comes $\omega+1 = \omega \cup \{\omega\}$, $\omega+2 = (\omega+1)+1$, $\omega+3$, and so on. The next limit ordinal is $\omega2$, which is the set consisting of all n, where $n\in\omega$, and all $\omega+n$, where $n\in\omega$. Then come $\omega2+1$, $\omega2+2$,...,$\omega3$, $\omega3+1$,...,$\omega4$,...,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 $T : L\rightarrow L$ be monotonic. Then we define

$T\uparrow 0 = \bot$
$T\uparrow\alpha = T(T\uparrow(\alpha-1))$, if α is a successor ordinal
$T\uparrow\alpha = \mathrm{lub}\{T\uparrow\beta : \beta<\alpha\}$, if α is a limit ordinal
$T\downarrow 0 = \top$
$T\downarrow\alpha = T(T\downarrow(\alpha-1))$, if α is a successor ordinal
$T\downarrow\alpha = \mathrm{glb}\{T\downarrow\beta : \beta<\alpha\}$, 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 $T : L\rightarrow L$ be monotonic. Then, for any ordinal α, $T\uparrow\alpha \le \mathrm{lfp}(T)$ and $T\downarrow\alpha \ge \mathrm{gfp}(T)$. Furthermore, there exist ordinals β_1 and β_2 such that $\gamma_1 \ge \beta_1$ implies $T\uparrow\gamma_1 = \mathrm{lfp}(T)$ and $\gamma_2 \ge \beta_2$ implies $T\downarrow\gamma_2 = \mathrm{gfp}(T)$.

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, $T\uparrow\alpha \leq \text{lfp}(T)$:

If α is a limit ordinal, then $T\uparrow\alpha = \text{lub}\{T\uparrow\beta : \beta<\alpha\} \leq \text{lfp}(T)$, by the induction hypothesis. If α is a successor ordinal, then $T\uparrow\alpha = T(T\uparrow(\alpha-1)) \leq T(\text{lfp}(T)) = \text{lfp}(T)$, 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 $T\uparrow\alpha = \text{lub}\{T\uparrow\beta : \beta<\alpha\} \leq \text{lub}\{T\uparrow(\beta+1) : \beta<\alpha\} \leq T(\text{lub}\{T\uparrow\beta : \beta<\alpha\}) = T\uparrow(\alpha+1)$, 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 $T\uparrow\alpha \leq \text{lub}\{T\uparrow\gamma : \gamma<\beta\} = T\uparrow\beta$. 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 $T\uparrow\alpha = \text{lfp}(T)$:

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, $T\uparrow\alpha = \text{lfp}(T)$, by (a).

(e) There exists β such that $\gamma \geq \beta$ implies $T\uparrow\gamma = \text{lfp}(T)$:

Let α be the least ordinal of cardinality greater than the cardinality of L. Suppose that $T\uparrow\delta \neq \text{lfp}(T)$, 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 $T\uparrow\beta = \text{lfp}(T)$, for some $\beta<\alpha$, and the result follows from (a) and (c).

The proof for gfp(T) is similar. ∎

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

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 $T(T\uparrow\omega) = T(\text{lub}\{T\uparrow n : n\in\omega\}) = \text{lub}\{T(T\uparrow n) : n\in\omega\} = T\uparrow\omega$, 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 $x \rightarrow x+1$

   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) $\forall x(q(x) \rightarrow p(x,a))$

(e) $\forall x\, p(s(x),x)$

(f) $\forall x\, \forall y(r(x,y) \rightarrow \sim p(x,y))$

(g) $\forall x(\exists y\, p(x,y) \vee r(s(b),s(x)) \rightarrow q(x))$

2. Determine whether the following formulas are valid or not:

(a) $\forall x\, \exists y\, p(x,y) \rightarrow \exists y\, \forall x\, p(x,y)$

(b) $\exists y\, \forall x\, p(x,y) \rightarrow \forall x\, \exists y\, p(x,y)$

3. Consider the formula

$(\forall x\, p(x,x) \wedge \forall x \forall y \forall z\, [(p(x,y) \wedge p(y,z)) \rightarrow p(x,z)] \wedge \forall x \forall y\, [p(x,y) \vee p(y,x)]) \rightarrow \exists y \forall x\, p(y,x)$

(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

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

$\sim \exists xF$ 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 ~ immediately precedes an atom.

(c) Replace

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

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

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

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

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

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

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

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

until all quantifiers are at the front of the formula.

(d) Replace

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

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

until the formula is in prenex conjunctive normal form.

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

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

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

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

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

11. For each of the following sets of simple expressions, determine whether mgu's exist or not and find them when they exist:
(a) {p(f(y),w,g(z)), p(u,u,v)}
(b) {p(f(y),w,g(z)), p(v,u,v)}
(c) {p(a,x,f(g(y))), p(z,h(z,w),f(w))}

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

13. Let L be a complete lattice and $T : L \rightarrow L$ be monotonic.
(a) Suppose $a \in L$ and $a \leq T(a)$. Define

$T^0(a) = a$
$T^\alpha(a) = T(T^{\alpha-1}(a))$, if α is a successor ordinal
$T^\alpha(a) = \text{lub}\{T^\beta(a) : \beta < \alpha\}$, if α is a limit ordinal.

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

$T^0(b) = b$
$T^\alpha(b) = T(T^{\alpha-1}(b))$, if α is a successor ordinal
$T^\alpha(b) = \text{glb}\{T^\beta(b) : \beta < \alpha\}$, if α is a limit ordinal.

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

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

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

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

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

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.

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 $M_P = \{A \in B_P : A$ is a logical consequence of P$\}$.

Proof We have that
A is a logical consequence of P
iff $P \cup \{\sim A\}$ is unsatisfiable, by proposition 3.1
iff $P \cup \{\sim A\}$ has no Herbrand models, by proposition 3.3
iff $\sim A$ is false wrt all Herbrand models of P
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 $T_P : 2^{B_P} \to 2^{B_P}$ is defined as follows. Let I be an Herbrand interpretation. Then $T_P(I) = \{A \in B_P : A \leftarrow A_1,\ldots,A_n$ is a ground instance of a clause in P and $\{A_1,\ldots,A_n\} \subseteq I\}$.

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

Now we have that

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

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

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

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

iff $A \in \mathrm{lub}(T_P(X))$. ■

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

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

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

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

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

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

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

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

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

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

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

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

Using proposition 3.1, we see that θ is a correct answer iff $P \cup \{\sim\forall((A_1 \wedge \ldots \wedge A_k)\theta)\}$ 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 $\forall((A_1 \wedge \ldots \wedge A_k)\theta)$ is true wrt the least Herbrand model of the program. Unfortunately, the result does not hold in this generality, as the following example shows.

Example Consider the program P

$p(a) \leftarrow$

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

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

Theorem 6.6 Let P be a definite program and G a definite goal $\leftarrow A_1,\dots,A_k$. Suppose θ is an answer for $P \cup \{G\}$ such that $(A_1 \wedge \dots \wedge A_k)\theta$ is ground. Then the following are equivalent:

(a) θ is correct.

(b) $(A_1 \wedge \dots \wedge A_k)\theta$ is true wrt every Herbrand model of P.

(c) $(A_1 \wedge \dots \wedge A_k)\theta$ is true wrt the least Herbrand model of P.

Proof Obviously, it suffices to show that (c) implies (a). Now
$(A_1 \wedge \dots \wedge A_k)\theta$ is true wrt the least Herbrand model of P
implies $(A_1 \wedge \dots \wedge A_k)\theta$ is true wrt all Herbrand models of P
implies $\sim(A_1 \wedge \dots \wedge A_k)\theta$ is false wrt all Herbrand models of P
implies $P \cup \{\sim(A_1 \wedge \dots \wedge A_k)\theta\}$ has no Herbrand models
implies $P \cup \{\sim(A_1 \wedge \dots \wedge A_k)\theta\}$ 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 $\leftarrow A_1,\dots,A_m,\dots,A_k$ and C be $A \leftarrow B_1,\dots,B_q$. Then G' is *derived* from G and C using mgu θ if the following conditions hold:

(a) A_m is an atom, called the *selected* atom, in G.

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

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

Definition Let P be a definite program and G a definite goal. An *SLD-derivation* of $P \cup \{G\}$ consists of a (finite or infinite) sequence $G_0=G$, $G_1,\ldots$ of goals, a sequence C_1, $C_2,\ldots$ of variants of program clauses of P and a sequence θ_1, $\theta_2,\ldots$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

Each C_i is a suitable variant of the corresponding program clause so that C_i does not have any variables which already appear in the derivation up to G_{i-1}. This can be achieved, for example, by subscripting variables in G by 0 and variables in C_i by i. This process of renaming variables is called *standardising* the variables *apart*. It is necessary, otherwise, for example, we would not be able to unify p(x) and p(f(x)) in $\leftarrow$p(x) and p(f(x))$\leftarrow$. Each program clause variant C_1, $C_2,\ldots$ is called an *input clause* of the derivation.

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

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

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

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

SLD-derivations may be *finite* or *infinite*. A finite SLD-derivation may be successful or failed. A *successful* SLD-derivation is one that ends in the empty clause. In other words, a successful derivation is just a refutation. A *failed* SLD-derivation is one that ends in a non-empty goal with the property that the selected atom in this goal does not unify with the head of any program clause. Later we shall see examples of successful, failed and infinite derivations (see Figure 2 and Figure 3).

$$\begin{tikzcd}
G_0 = G \arrow[d] & C_1, \theta_1 \arrow[dl] \\
G_1 \arrow[d] & C_2, \theta_2 \arrow[dl] \\
G_2 \arrow[d] & \\
{} \arrow[d, dashed, no head] & \\
{} \arrow[d] & \\
G_{n-1} \arrow[d] & C_n, \theta_n \arrow[dl] \\
G_n \arrow[d] & \\
{} \arrow[d, dashed, no head] & \\
{} &
\end{tikzcd}$$

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

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

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 $\theta_1,\ldots,\theta_n$ are the unifiers from the unrestricted SLD-refutation and $\theta'_1,\ldots,\theta'_n$ are the mgu's from the SLD-refutation, then there exists a substitution γ such that $\theta_1\ldots\theta_n = \theta'_1\ldots\theta'_n\gamma$.

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

Lemma 8.2 (Lifting Lemma)

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

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. 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 a (properly standardised apart) unrestricted refutation of $P \cup \{G\}$, which looks exactly like the given refutation of $P \cup \{G\theta\}$, except the original goal is different, of course, and the first unifier is $\theta\theta_1$. Now apply the mgu lemma. ■

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

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

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

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

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

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

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

Proof Let G be the goal $\leftarrow A_1, \dots, A_k$. Since $P \cup \{G\}$ is unsatisfiable, G is false wrt M_P. Hence some ground instance $G\theta$ of G is false wrt M_P. Thus $\{A_1\theta, \dots, A_k\theta\} \subseteq M_P$. By theorem 8.3, there is a refutation for $P \cup \{\leftarrow A_i\theta\}$, for $i=1,\dots,k$. 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 θ and $\sigma\gamma$ have the same effect on all variables in G.

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 θ and $\sigma\gamma$ have the same effect on all variables in G. ∎

§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 ∪ {G} *via R* is an SLD-derivation of P ∪ {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 ∪ {G} *via R* is an SLD-refutation of P ∪ {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 ∪ {G} is a computed answer for P ∪ {G} which has come from an SLD-refutation of P ∪ {G} via R.

Now we are in a position to consider the independence result. According to theorem 8.4, if P ∪ {G} is unsatisfiable, then there exists a refutation of P ∪ {G}. In fact, we will show that, for any computation rule R, there is actually a refutation of P ∪ {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 ∪ {G} has an SLD-refutation G_0=G, G_1,...,G_{q-1}, G_q, G_{q+1},...,G_n= □ with input clauses

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

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

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

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

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 θ and $\sigma\gamma$ have the same effect on all variables in G.

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 $P_f \cup \{\leftarrow p_f(s^{k_1}(0),\ldots,s^{k_n}(0),x)\}$ have the form $\{x/s^k(0)\}$ and, for all non-negative integers $k_1,\ldots,k_n$ and k, we have $f(k_1,\ldots,k_n)=k$ iff $\{x/s^k(0)\}$ is a computed answer for $P_f \cup \{\leftarrow p_f(s^{k_1}(0),\ldots,s^{k_n}(0),x)\}$.

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 $f(x_1,\ldots,x_n)=x_j$, where $1\leq j\leq n$. Define P_f to be the program $p_f(x_1,\ldots,x_n,x_j)\leftarrow$.

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 $f(x_1,\ldots,x_n) = h(g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n))$, where $g_1,\ldots,g_m$ 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 $P_{g_1},\dots,P_{g_m}$ 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

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

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

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

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

Primitive recursion

Suppose that f is defined by

$f(x_1,\dots,x_n,0) = h(x_1,\dots,x_n)$

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

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

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

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

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

Minimalisation

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

otherwise, $\mu y(g(x_1,\ldots,x_n,y)=0)$ 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

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

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 $\leftarrow A_1,\ldots,A_m,\ldots,A_k$ $(k\geq1)$ be a node in the tree and suppose that A_m is the selected atom. Then, for each input clause $A\leftarrow B_1,\ldots,B_q$ such that A_m and A are unifiable with mgu θ, the node has a child

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

(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 ∪ {G} *via R* is the SLD-tree for P ∪ {G} in which the atoms selected are those selected by R.

Example Consider the program

1. p(x,z) ← q(x,y), p(y,z)
2. p(x,x) ←
3. q(a,b) ←

and the goal ←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 ∪ {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 ∪ {G} is unsatisfiable. Then the SLD-tree for P ∪ {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 ∪ {G} is "displayed" on the SLD-tree for P ∪ {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 ∪ {G} has infinitely many success branches or every SLD-tree for P ∪ {G} has the same finite number of success branches.

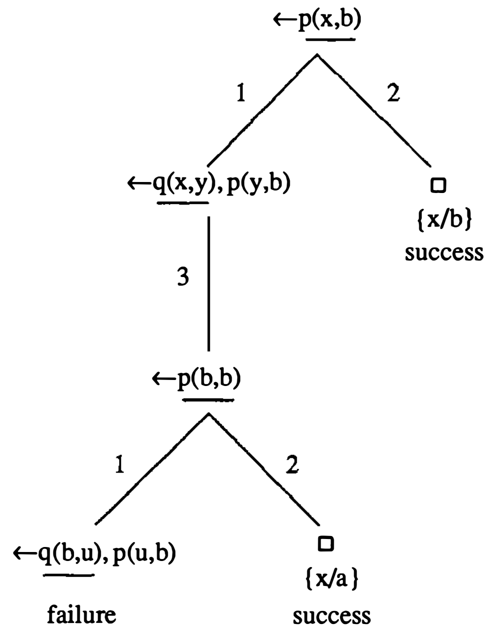

Fig. 2. A finite SLD-tree

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

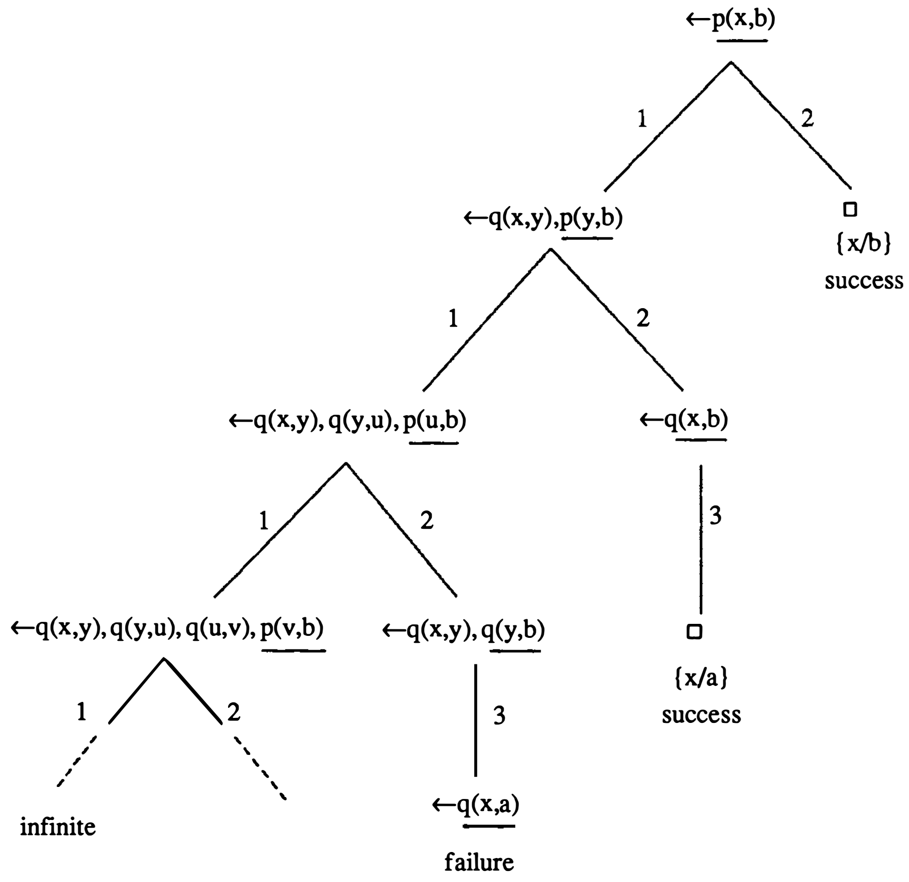

Fig. 3. An infinite SLD-tree

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.

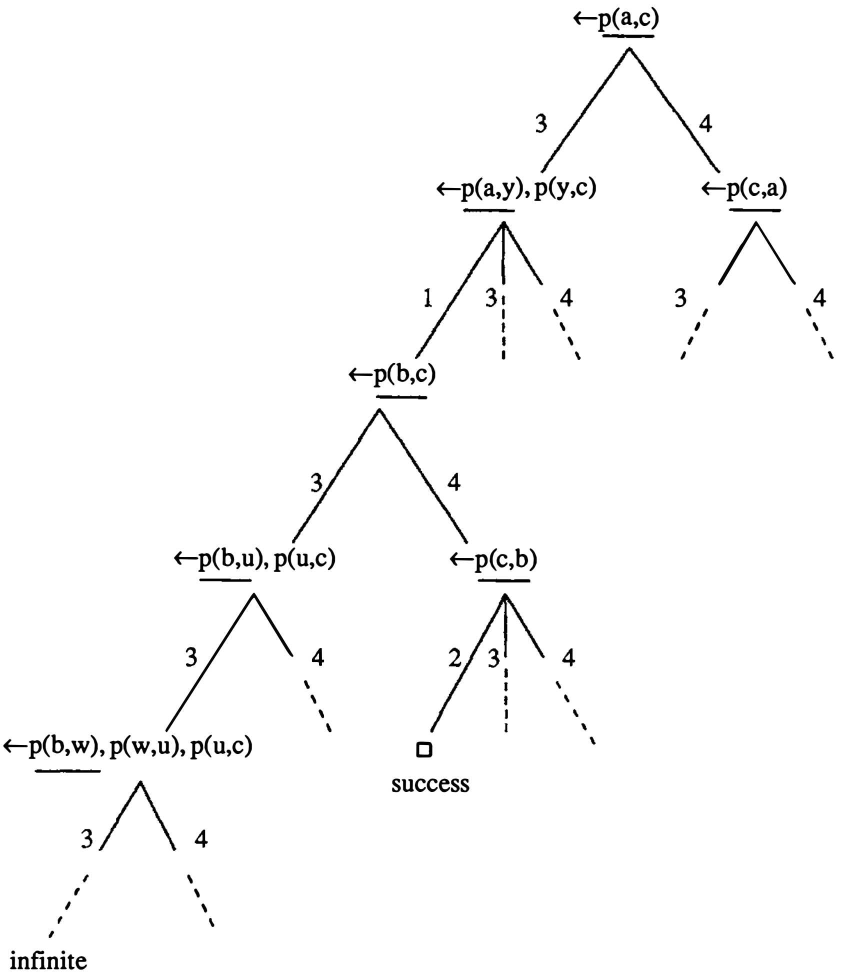

Fig. 4. SLD-tree which illustrates the problem with depth-first search

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.

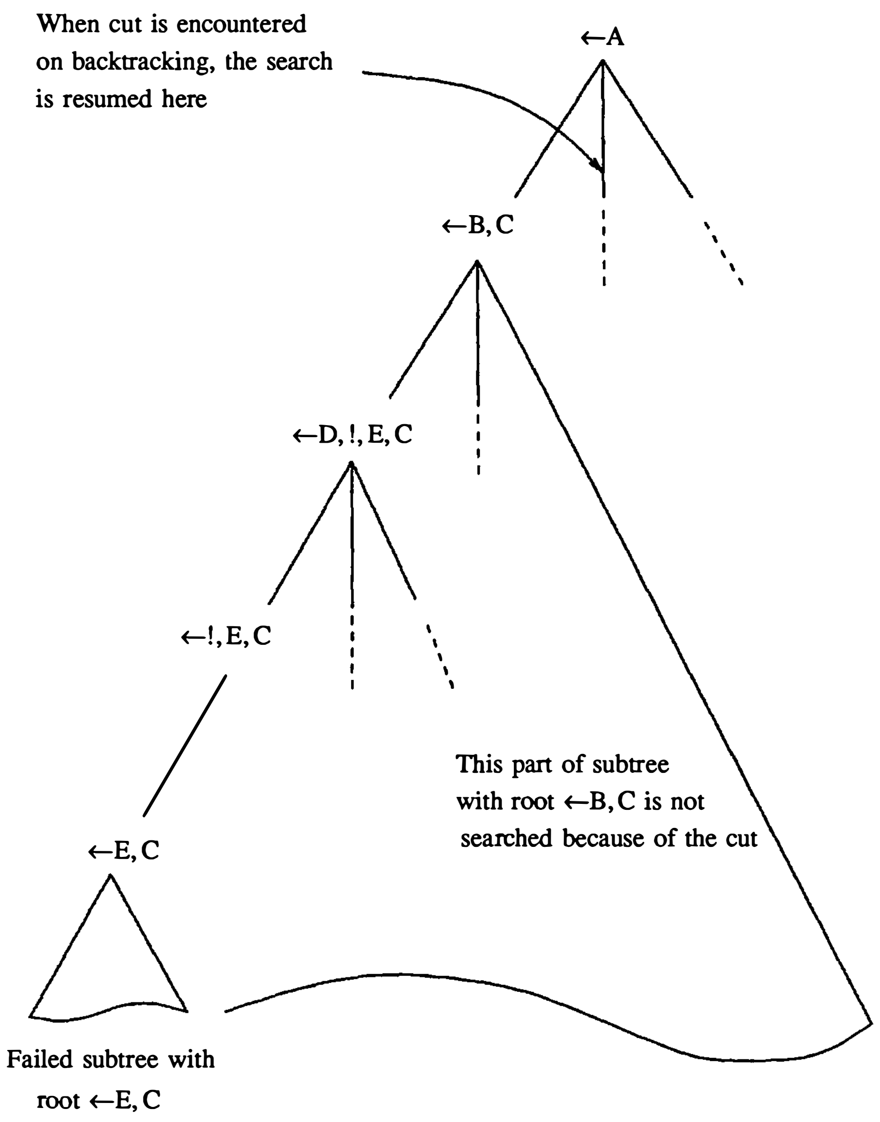
When cut is encountered on backtracking, the search is resumed here
←A
←B, C
←D, !, E, C
←!, E, C
←E, C
This part of subtree with root ←B, C is not searched because of the cut
Failed subtree with root ←E, C

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) ← x≤y, !
max(x,y,x) ←
```

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 $\leftarrow$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) $\leftarrow$

leaves(tree(u,v,w),x–y) $\leftarrow$ 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) $\leftarrow$ q(x), r(x)

q(a) $\leftarrow$

r(x) $\leftarrow$ r_1(x)

r_1(a) $\leftarrow$

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 ∪ {←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 ∪ {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 ∪ {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 ∪ {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?

22. Show that the condition in the lifting lemma that the variables in the input clauses be distinct from the variables in θ and G cannot be dropped.

23. Give an example of a definite program P, a definite goal G, and a correct answer θ for P ∪ {G} such that there does not exist a computed answer σ for P ∪ {G} and a substitution γ for which $\theta = \sigma\gamma$.

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 ~A is a logical consequence of P. The reason is that P ∪ {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 $1 \le k \le n$ 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 $F_P = \cup_{d \ge 1} F_P^d$.

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 $F_P = B_P \setminus T_P \downarrow \omega$.

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 $\le$ k. Then $P \cup \{G\theta\}$ also has a finitely failed SLD-tree of depth $\le$ 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 $\le$ k. Then there exists $i \in \{1,...,m\}$ such that $P \cup \{\leftarrow A_i\}$ has a finitely failed SLD-tree of depth $\le$ 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 $\leftarrow (B_1,...,B_n)\gamma$ 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

$$\leftarrow L_1,\dots,L_n$$

where $L_1,\dots,L_n$ 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 $p(t_1,\dots,t_n)\leftarrow L_1,\dots,L_m$ 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

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

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

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

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

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

where each E_i has the general form

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

The *completed definition* of p is then the formula

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

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

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

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

$$\forall x_1\ldots\forall x_n\ \sim q(x_1,\ldots,x_n)$$

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. $\forall(f(x_1,\ldots,x_n){\neq}g(y_1,\ldots,y_m))$, for all pairs f,g of distinct function symbols.
3. $\forall(f(x_1,\ldots,x_n){\neq}c)$, 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. $\forall((x_1{\neq}y_1)\vee\ldots\vee(x_n{\neq}y_n)\rightarrow f(x_1,\ldots,x_n){\neq}f(y_1,\ldots,y_n))$, for each function symbol f.
6. $\forall(x{=}x)$.
7. $\forall((x_1{=}y_1)\wedge\ldots\wedge(x_n{=}y_n)\rightarrow f(x_1,\ldots,x_n){=}f(y_1,\ldots,y_n))$, for each function symbol f.
8. $\forall((x_1{=}y_1)\wedge\ldots\wedge(x_n{=}y_n)\rightarrow(p(x_1,\ldots,x_n)\rightarrow p(y_1,\ldots,y_n)))$, 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 $\leftarrow L_1,\dots,L_n$, and θ an answer for $P \cup \{G\}$. We say θ is a *correct answer* for $\text{comp}(P) \cup \{G\}$ if $\forall((L_1 \wedge \dots \wedge L_n)\theta)$ 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 $p(t_1,\dots,t_n) \leftarrow L_1,\dots,L_m$ be a program clause in P and suppose that $L_1,\dots,L_m$ are true in M, for some assignment of the variables $y_1,\dots,y_d$ in the clause.

Consider the completed definition of p

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

and suppose E_i is

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

Now let x_j be t_j $(1 \le j \le n)$, for the same assignment of the variables $y_1,\dots,y_d$ as above. Thus E_i is true in M, since $L_1,\dots,L_m$ are true in M and also since M must satisfy axiom 6. Hence $p(t_1,\dots,t_n)$ 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 $T_P^J(I) = \{ A_{J,V} : A \leftarrow L_1 \wedge \dots \wedge L_n \in P$, V is a variable assignment wrt J, and $L_1 \wedge \dots \wedge L_n$ 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

$p \leftarrow \sim p$

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 $\forall x_1 \ldots \forall x_n \sim q(x_1,\ldots,x_n)$. 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

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

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

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

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

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

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

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

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

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

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

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

Proof Suppose $A \in \text{gfp}(T_P)$. Then $\text{gfp}(T_P) \cup \{s=s : s \in U_P\}$ 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 $A \in \text{gfp}(T_P)$. ∎

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

Proof Let $x_1,\ldots,x_k$ be the variables in $A_1 \wedge \ldots \wedge A_m$. We have to show that $\forall x_1 \ldots \forall x_k (A_1 \wedge \ldots \wedge A_m)$ is a logical consequence of P, that is, $P \cup \{\sim\forall x_1 \ldots \forall x_k (A_1 \wedge \ldots \wedge A_m)\}$ is unsatisfiable or, equivalently, $S = P \cup \{\sim A'_1 \vee \ldots \vee \sim A'_m\}$ is unsatisfiable, where A'_i is A_i with $x_1,\ldots,x_k$ 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 $T^J_P(I) \subseteq I$. Since T^J_P is monotonic, proposition 5.2 shows that there exists a fixpoint $I' \subseteq I$ of T^J_P. 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 $\sim A'_1 \vee \ldots \vee \sim A'_m$ is false in this interpretation. Since $I' \subseteq I$, we have that $\sim A'_1 \vee \ldots \vee \sim A'_m$ 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 ≤ 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 $\Lambda = \{M_k \cup S : S \subseteq \{p(d_1,...,d_n) : p \text{ is a level } k+1 \text{ predicate symbol and each } d_i \text{ is in the domain of } J\}\}$ 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) ∪ {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 $\leftarrow L_1,\dots,L_m,\dots,L_p$, 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 $\leftarrow L_1,\dots,L_{m-1},L_{m+1},\dots,L_p$, 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 $\leftarrow L_1,\dots,L_m,\dots,L_p$ 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) $A \leftarrow M_1,\dots,M_q$ such that L_m and A are unifiable with mgu θ, the node has a child $\leftarrow (L_1,\dots,L_{m-1},M_1,\dots,M_q,L_{m+1},\dots,L_p)\theta$, 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 $\leftarrow L_1,\dots,L_{m-1},L_{m+1},\dots,L_p$.

(d) Let $\leftarrow L_1,\dots,L_m,\dots,L_p$ 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 $\theta_1\dots\theta_n$ to the variables of G, where $\theta_1,\dots,\theta_n$ 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 $G_0=G, G_1,\ldots$ of normal goals, a sequence $C_1, C_2,\ldots$ of variants of program clauses (called *input clauses*) of P or ground negative literals, and a sequence $\theta_1, \theta_2,\ldots$ 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 $\leftarrow L_1,\ldots,L_m,\ldots,L_p$, 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 $\leftarrow L_1,\ldots,L_{m-1},L_{m+1},\ldots,L_p$, θ_{i+1} is the identity substitution and C_{i+1} is $\sim A_m$.

(b) If the sequence $G_0, G_1,\ldots$ of goals is finite, then either

(i) the last goal is empty, or

(ii) the last goal is $\leftarrow L_1,\ldots,L_m,\ldots,L_p$, 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 $\leftarrow L_1,\ldots,L_m,\ldots,L_p$, 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 $\leftarrow \sim p(x)$ 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, $G_1,...,G_n = \Box$ be an SLDNF-refutation of $P \cup \{G\}$ using substitutions $\theta_1,\ldots,\theta_n$. 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 $(L_1 \wedge \ldots \wedge L_m)\theta_1 \ldots \theta_n$ 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 $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$ be atoms.
(a) If $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$ are not unifiable, then $\sim\exists((s_1=t_1)\wedge\ldots\wedge(s_n=t_n))$ is a logical consequence of the equality theory.
(b) If $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$ are unifiable with mgu $\theta = \{x_1/r_1,\ldots,x_k/r_k\}$ given by the unification algorithm, then $\forall((s_1=t_1)\wedge\ldots\wedge(s_n=t_n) \leftrightarrow (x_1=r_1)\wedge\ldots\wedge(x_k=r_k))$ is a logical consequence of the equality theory.

Proof Suppose that $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$ are unifiable with mgu $\theta = \{x_1/r_1,\ldots,x_k/r_k\}$. Then it follows from equality axioms 6, 7 and 8 that $\forall((s_1=t_1)\wedge\ldots\wedge(s_n=t_n)\leftarrow(x_1=r_1)\wedge\ldots\wedge(x_k=r_k))$ 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 $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$.

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

Suppose now that the result holds for k–1. Let $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$ 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 $p(s_1,\ldots,s_n)\theta_1$ and $p(t_1,\ldots,t_n)\theta_1$ are such that the unification algorithm can discover in k–1 steps whether they are unifiable or not.

Suppose that $p(s_1,\ldots,s_n)\theta_1$ and $p(t_1,\ldots,t_n)\theta_1$ are not unifiable. Then the induction hypothesis gives that $\sim\exists((s_1=t_1)\theta_1\wedge\ldots\wedge(s_n=t_n)\theta_1)$ 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

$\sim\exists((s_1=t_1)\wedge\ldots\wedge(s_n=t_n))$ 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 $\forall(\sim p(x_1,\ldots,x_n))$, 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

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

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 $\sim A_i$. 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 $\sim\exists(L_1 \wedge ... \wedge L_n)$ 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 $\sim\exists(L_1 \wedge ... \wedge L_{i-1} \wedge L_{i+1} \wedge ... \wedge L_n)$ is a logical consequence of comp(P). Hence $\sim\exists(L_1 \wedge ... \wedge L_n)$ is also a logical consequence of comp(P). ∎

Corollary 15.5 Let P be a definite program. If $A \in F_P$, then $\sim A$ 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 $\text{comp}(P) \cup \{G\}$.

Proof Let G be the normal goal $\leftarrow L_1,\ldots,L_k$ and $\theta_1,\ldots,\theta_n$ be the sequence of substitutions used in an SLDNF-refutation of $P \cup \{G\}$. We have to show that $\forall((L_1\wedge\ldots\wedge L_k)\theta_1\ldots\theta_n)$ 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 $\theta_1,\ldots,\theta_n$ 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 $A\leftarrow M_1,\ldots,M_q$ $(q\geq 0)$ be the first input clause. By the induction hypothesis, $\forall((L_1\wedge\ldots\wedge L_{m-1}\wedge M_1\wedge\ldots\wedge M_q\wedge L_{m+1}\wedge\ldots\wedge L_k)\theta_1\ldots\theta_n)$ is a logical consequence of comp(P). Therefore, if q>0, $\forall((M_1\wedge\ldots\wedge M_q)\theta_1\ldots\theta_n)$ is a logical consequence of comp(P). Consequently, $\forall(L_m\theta_1\ldots\theta_n)$, which is the same as $\forall(A\theta_1\ldots\theta_n)$, is a logical consequence of comp(P). Hence we have that $\forall((L_1\wedge\ldots\wedge L_k)\theta_1\ldots\theta_n)$ 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 $\forall((L_1\wedge\ldots\wedge L_k)\theta_1\ldots\theta_n)$ 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

$p \leftarrow \sim q(x)$

$q(a) \leftarrow$

If we drop the safeness condition, then the literal ~q(x) can be selected and we obtain a "finitely failed SLDNF-tree" for P ∪ {←p}. The subgoal ~q(x) fails because there is a refutation of ←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 ∀(A) rather than only ∃(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) ←

q(b) ←

and the normal goal ← ~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 ∪ {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 $p(s_1,...,s_n) = p(t_1,...,t_n)\theta_{i+1}...\theta_{i+j}$ 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) $\cup$ $\{\exists(A_1\wedge\ldots\wedge A_q)\}$. Note that this interpretation is a model for $\exists(A_1\wedge\ldots\wedge A_q)$ 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) $\cup$ $\{\exists(A_1\wedge\ldots\wedge A_q)\}$. ∎

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) $\cup$ {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 $\cup$ {←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) ←
q(a) ←
r(b) ←

and the goal ←p(x),~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, ←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 ← p
r ← ~p
p ← p

Then the identity substitution ε is a correct answer for comp(P) ∪ {←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 ∪ {G} *via R* is an SLDNF-derivation of P ∪ {G} in which the computation rule R is used to select literals.

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

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

An *R-computed answer* for P ∪ {G} is a computed answer for P ∪ {G} which has come from an SLDNF-refutation of P ∪ {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 ∪ {G} is allowed. Then the following properties hold.

(a) The SLDNF-tree for P ∪ {G} via R exists and is finite.

(b) If θ is a correct answer for comp(P) ∪ {G} and θ is a ground substitution for all variables in G, then θ is an R-computed answer for P ∪ {G}.

Proof (a) By proposition 15.1(a), the computation of P ∪ {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 ∪ {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 $\leftarrow L_1,\ldots,L_n$. The SLDNF-tree for $P \cup \{G\theta\}$ via R is not finitely failed; otherwise, by theorem 15.4, we would have that $\sim(L_1\wedge\ldots\wedge L_n)\theta$ 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 $\text{comp}(P) \cup \{A\}$ 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):

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

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

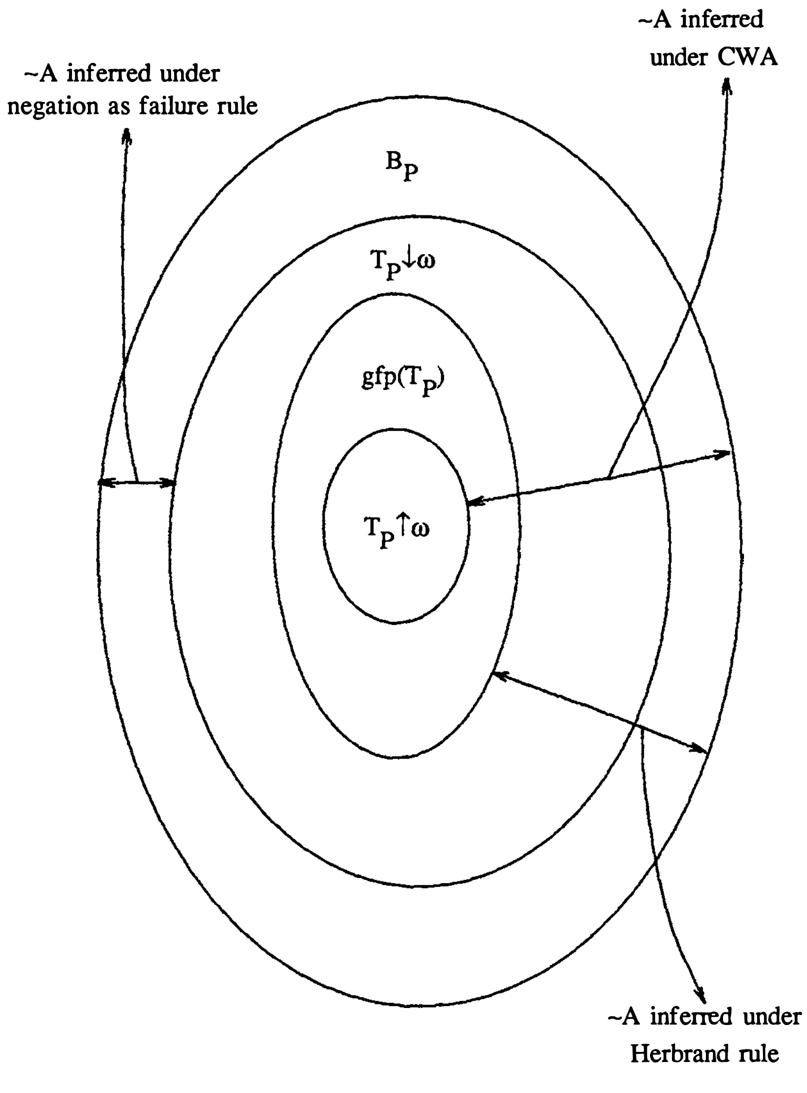

Fig. 6. Relationship between the various rules

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) $\sim A$ 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

    $p \leftarrow \sim q(x)$

    $q(a) \leftarrow$

Show that ~p is not a logical consequence of comp(P).

18. Consider the normal program P

    $p \leftarrow \sim r$

    $r \leftarrow q(x)$

    $q(a) \leftarrow$

Show that $P \cup \{\leftarrow p\}$ has a finitely failed SLDNF-tree and that ~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 $\text{comp}(P) \cup \{A\}$ 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

    $q \leftarrow \sim r$

    $r \leftarrow p$

    $r \leftarrow \sim p$

    $p \leftarrow p$

Show that ~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

    $r \leftarrow p$

    $r \leftarrow \sim p$

    $p \leftarrow p$

Show that the identity substitution ε is a correct answer for $\text{comp}(P) \cup \{\leftarrow r\}$, 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 $\leftarrow L_1,\ldots,L_n$. Suppose that $P \cup \{G\}$ has a finitely failed SLDNF-tree. Prove that there exists $i \in \{1,\ldots,n\}$ 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 $\leftarrow L_1,\ldots,L_n$. Suppose that $P \cup \{G\}$ has an SLDNF-refutation. Prove that $P \cup \{\leftarrow L_i\}$ has an SLDNF-refutation, for all $i \in \{1,\ldots,n\}$.

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 $P^* = P \cup \{\sim A : A \in B_P$ 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.

Chapter 4

PROGRAMS

In this chapter, we study programs and goals. A program is a finite set of program statements, each of which has the form $A \leftarrow W$, where the head A is an atom and the body W is an arbitrary first order formula. Similarly, a goal has the form $\leftarrow W$, where the body W is an arbitrary first order formula. We prove the soundness of the negation as failure rule and SLDNF-resolution for programs and goals. We also study an error diagnoser, which is declarative in the sense that the programmer need only know the intended interpretation of an incorrect program to use the diagnoser.

§17. INTRODUCTION TO PROGRAMS

This section introduces programs and goals. A program is a finite set of program statements, each of which has the form $A \leftarrow W$, where the head A is an atom and the body W is an arbitrary first order formula. Similarly, a goal has the form $\leftarrow W$, where the body W is an arbitrary first order formula. We argue that PROLOG systems should allow the increased expressiveness of programs and goals as a standard feature. The only requirement for implementing such a feature is a sound form of the negation as failure rule. Programs and goals were introduced by Lloyd and Topor [61]. Special cases of them were studied earlier by Clark [15] and Kowalski [49].

Definition A *program statement* is a first order formula of the form

$A \leftarrow W$

where A is an atom and W is a (not necessarily closed) first order formula. The formula W may be absent. Any variables in A and any free variables in W are assumed to be universally quantified at the front of the program statement. A is called the *head* of the statement and W is called the *body* of the statement.

Note that a program clause is a program statement for which the body is a conjunction of literals. Throughout, we make the assumption, as we may, that in each formula each quantifier is followed by a distinct variable and no variable is both bound and free.

Definition A *program* is a finite set of program statements.

Definition A *goal* is a first order formula of the form

$\leftarrow W$

where W is a (not necessarily closed) first order formula. Any free variables in W are assumed to be universally quantified at the front of the goal.

Example Consider the program statement

$A \leftarrow \forall x_1 ... \forall x_n (\exists y_1 ... \exists y_k W \leftarrow W_1 \wedge ... \wedge W_m)$

Often program statements have this form. Typically, W, $W_1,...,W_m$ are atoms and the y_i are absent. For example, the well-ordered predicate can be defined as follows.

well_ordered(x) $\leftarrow \forall z$ (hasleastelt(z) $\leftarrow$ set(z) $\wedge$ $z \subseteq x$ $\wedge$ nonempty(z))

nonempty(z) $\leftarrow \exists u\, u \in z$

hasleastelt(z) $\leftarrow \exists u\, (u \in z \wedge \forall v\, (u \leq v \leftarrow v \in z))$

$x \subseteq y \leftarrow \forall z\, (z \in y \leftarrow z \in x)$

The increased expressiveness of programs and goals is useful for expert systems, deductive database systems, and general purpose programming. In expert systems, it allows the statement of the rules in the knowledge base in a form closer to a natural language statement, such as would be provided by a human expert. This makes it easier to understand the knowledge base. This increased expressiveness also has an application to deductive database systems, by providing first order logic (known as domain relational calculus in database terminology [25]) as a query language in a straightforward manner. (See chapter 5.) In general purpose programming, applications like the example above occur often. If this increased expressiveness is not available, it is only possible to express such statements rather obscurely.

Furthermore, from a theoretical point of view, it makes no sense to stop at normal programs and normal goals. As we will show in the next section, by means of simple transformations it is possible to transform any program to an "equivalent" normal program. By means of this technique, we can extend the theory of normal programs to arbitrary programs in a straightforward way. This transformation technique also provides a straightforward implementation of programs and goals in any PROLOG system which has a safe implementation of the negation as failure rule. NU-PROLOG [75], [104] provides this increased expressiveness as a standard feature.

Next we define the completion of a program P. Throughout, we assume that = does not appear in P.

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

Definition Suppose the definition of an n-ary predicate symbol p in a program is

$A_1 \leftarrow W_1$

$\dots$

$A_k \leftarrow W_k$

Then the *completed definition* of p is the formula

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

where E_i is $\exists y_1 \dots \exists y_d\ ((x_1=t_1) \wedge \dots \wedge (x_n=t_n) \wedge W_i)$, A_i is $p(t_1,\dots,t_n)$, $y_1,\dots,y_d$ are the variables in A_i and the free variables in W_i, and $x_1,\dots,x_n$ are variables not appearing anywhere in the definition of p.

Example Let the definition of p be

$p(y) \leftarrow q(y) \wedge \forall z(r(y,z) \leftarrow q(z))$

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

Then the completed definition of p is

$$\forall x(p(x) \leftrightarrow (\exists y((x=y) \wedge q(y) \wedge \forall z(r(y,z) \leftarrow q(z))) \vee \exists z((x=f(z)) \wedge \sim q(z)))$$

Definition Suppose the n-ary predicate symbol p appears in a program P, but not in the head of any program statement in P. Then the *completed definition* of p is the formula

$$\forall x_1 \dots \forall x_n \sim p(x_1,\dots,x_n)$$

We will also require the *equality theory* given §14.

Definition Let P be a 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.

Next we introduce the declarative concept of a correct answer for a program and goal. In this definition, if W is a formula and θ is a substitution for some of the free variables in W, then $W\theta$ is the formula obtained by simultaneously replacing each such variable by its binding in θ. For example, if W is $\forall x \exists y(p(z,f(x)) \leftarrow q(y))$ and θ is $\{z/g(w)\}$, then $W\theta$ is $\forall x \exists y(p(g(w),f(x)) \leftarrow q(y))$. Note that it may be necessary to rename some bound variables in W before applying θ to avoid clashes with the variables in the terms of the bindings of θ.

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

Definition Let P be a program and G a goal $\leftarrow W$. A *correct answer* for $comp(P) \cup \{G\}$ is an answer θ such that $\forall(W\theta)$ is a logical consequence of comp(P).

This definition, which generalises the previous definition of correct answer (see §14), provides the appropriate declarative description of the output from a program and goal.

We now investigate under what conditions the completion of a program will be consistent. In a way similar to that in chapter 3, the concept of a stratified program gives a satisfactory answer to this question.

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

Definition A program is *stratified* if it has a level mapping such that, in every program statement $p(t_1,...,t_n) \leftarrow W$, the level of the predicate symbol of every atom occurring positively in W is less than or equal to the level of p, and the level of the predicate symbol of every atom occurring negatively in W is less than the level of p.

This definition generalises the definition of stratified normal programs given in §14. We can assume without loss of generality that the levels of a stratified program are 0,1,...,k, for some k. Note that, at level 0, all atoms in the bodies of program statements must occur positively, but that these program statements need not be definite program clauses.

Next we extend the definition of the mapping T_P^J to arbitrary programs.

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

Proposition 17.1 Let P be a 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. ■

Proposition 17.2 Let P be a 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 Similar to the proof of proposition 14.3. ■

Proposition 17.3 Let P be a stratified program and J a pre-interpretation for P.
(a) Suppose P has only predicates of level 0. Then 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 statements in P with the property that the predicate symbol in the head of the statement has level $\leq$ k. Suppose that M_k is an interpretation based on J for P_k and M_k is a fixpoint of $T_{P_k}^J$. 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_P^J, restricted to Λ, is well-defined and monotonic.

Proof Straightforward. (See problem 1.) ■

Corollary 17.4 Let P be a stratified program. Then comp(P) has a minimal normal Herbrand model.

Proof Similar to the proof of corollary 14.8. ■

The results of this section are due to Lloyd, Sonenberg and Topor [60].

§18. SLDNF-RESOLUTION FOR PROGRAMS

In this section, we prove the soundness of the negation as failure rule and SLDNF-resolution for programs and goals. We also give a completeness result for hierarchical programs. The soundness results are proved by first transforming a program and goal into a normal program and normal goal. We then use the fact that the negation as failure rule and SLDNF-resolution are known to be sound in this case (theorems 15.4 and 15.6). This transformation technique can be used to give a straightforward implementation of programs and goals.

The first lemma justifies the transformation of a goal to a normal goal. Suppose P is a program and G is a goal. Let G have the form ←W, where W has free variables $x_1,...,x_n$. Suppose answer is an n-ary predicate symbol not appearing in P or G. The transformation replaces G by the normal goal

$\leftarrow \text{answer}(x_1,...,x_n)$

and adds the program statement

$\text{answer}(x_1,...,x_n) \leftarrow W$

to the program P.

Lemma 18.1 Let P be a program, G a goal, and θ an answer. Assume G has the form ←W, where W has free variables $x_1,...,x_n$ and answer is an n-ary predicate symbol not appearing in P or G. Then we have the following properties.
(a) G is a logical consequence of comp(P) iff $\leftarrow \text{answer}(x_1,...,x_n)$ is a logical consequence of comp(P'), where P' is $P \cup \{\text{answer}(x_1,...,x_n) \leftarrow W\}$.
(b) $\forall(W\theta)$ is a logical consequence of comp(P) iff $\forall(\text{answer}(x_1,...,x_n)\theta)$ is a logical consequence of comp(P').

Proof Note that in the presence of equality axioms 6, 7, and 8

$$\forall z_1 ... \forall z_n\ (\text{answer}(z_1,...,z_n) \leftrightarrow \exists x_1 ... \exists x_n((z_1=x_1) \wedge ... \wedge (z_n=x_n) \wedge W))$$

is logically equivalent to

$$\forall x_1 ... \forall x_n\ (\text{answer}(x_1,...,x_n) \leftrightarrow W)$$

Hence we can assume that comp(P') is simply comp(P) together with the latter formula (and an equality axiom 8 for the predicate symbol answer). Both parts of the lemma now follow easily from this. ∎

The next step is to transform a program P into a normal program P', called a *normal form* of P, by means of the following transformations.

(a) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim(V \wedge W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim V \wedge W_{i+1} \wedge \ldots \wedge W_m$
and $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(b) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \forall x_1 \ldots \forall x_n W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim \exists x_1 \ldots \exists x_n \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(c) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim \forall x_1 \ldots \forall x_n W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \exists x_1 \ldots \exists x_n \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(d) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \leftarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge \ldots \wedge W_m$
and $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(e) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim(V \leftarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge \sim V \wedge W_{i+1} \wedge \ldots \wedge W_m$

(f) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \vee W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge \ldots \wedge W_m$
and $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(g) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim(V \vee W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim V \wedge \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(h) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim\sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(i) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \exists x_1 \ldots \exists x_n W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_m$

(j) Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim \exists x_1 \ldots \exists x_n W \wedge W_{i+1} \wedge \ldots \wedge W_m$
by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim p(y_1, \ldots, y_k) \wedge W_{i+1} \wedge \ldots \wedge W_m$
and $p(y_1, \ldots, y_k) \leftarrow \exists x_1 \ldots \exists x_n W$
where $y_1, \ldots, y_k$ are the free variables in $\exists x_1 \ldots \exists x_n W$ and p is a new predicate symbol not already appearing in the program.

Note that, from a logical viewpoint, the various transformations for negation could be replaced by a single all-encompassing transformation for negation similar to (j). However, the transformations for negation have been presented as above to try to overcome the limitations of the negation as failure rule. For example, without (h), a subgoal of the form $\sim\sim A$ can flounder if A contains any variables. This problem disappears once the subgoal is transformed to A. Similar problems are overcome by (a), (c), (e), and (g).

Example Consider the program statement

$A \leftarrow \forall x_1 \ldots \forall x_n (\exists y_1 \ldots \exists y_k W \leftarrow W_1 \wedge \ldots \wedge W_m)$

If $u_1,\ldots,u_s$ are the free variables in the body and $w_1,\ldots,w_d$ are the free variables in $\exists y_1 \ldots \exists y_k W$, then the above program statement can be transformed to

$A \leftarrow \sim p(u_1,\ldots,u_s)$

$p(u_1,\ldots,u_s) \leftarrow W_1 \wedge \ldots \wedge W_m \wedge \sim q(w_1,\ldots,w_d)$

$q(w_1,\ldots,w_d) \leftarrow W$

Example The subset predicate ($\subseteq$) can be defined by the program statement

$x \subseteq y \leftarrow \forall u (u \in y \leftarrow u \in x)$

A normal form of this program statement is

$x \subseteq y \leftarrow \sim p(x,y)$

$p(x,y) \leftarrow \sim(u \in y) \wedge u \in x$

We apply transformations (a),...,(j) until no more such transformations are possible. The proposition below shows that this process terminates after a finite number of steps and that the resulting normal form of the original program is indeed a normal program. Of course, the normal form is not unique.

Proposition 18.2 Let P be a program. Then the process of continually applying transformations (a),...,(j) to P terminates after a finite number of steps and results in a normal program (called a *normal form* of P).

Proof If M and M' are finite multisets of non-negative integers, then we define $M' < M$ as in the proof of theorem 16.3. The basic idea of the proof is to define a termination function μ from programs into the well-founded set of all finite multisets of non-negative integers under $<$.

Inductively define the mapping μ as follows:

$\mu(\text{atom}) = 1$

$\mu(V \wedge W) = \mu(V) + \mu(W)$

$\mu(\sim W) = \mu(\exists x W) = \mu(W) + 1$

$\mu(V \leftarrow W) = \mu(V) + \mu(W) + 1$

$\mu(V \vee W) = \mu(V) + \mu(W) + 2$

$\mu(\forall x W) = \mu(W) + 4$

$\mu(\text{program } P) = \{\mu(W) : A \leftarrow W \text{ is a statement in } P\}$,

where {...} denotes a multiset. It now suffices to remark that if Q' is obtained from a program Q by a single transformation (a) or ... or (j), then $\mu(Q') < \mu(Q)$, so the process terminates. Furthermore, the resulting program is a normal program since, otherwise, some further transformation would be possible. ■

Lemma 18.3 Let P be a program and let Q be the program which results from a single transformation (a) or ... or (i). Then P and Q are logically equivalent and also comp(P) and comp(Q) are logically equivalent.

Proof Straightforward. (See problem 3.) ■

The corresponding result for transformation (j) is more complicated, as the following lemma shows.

Lemma 18.4 Let P be a program and P' a normal form of P. If U is a closed formula which is a logical consequence of comp(P') and U only contains predicate symbols which appear in P, then U is a logical consequence of comp(P).

Proof It follows from lemma 18.3 that we only have to prove the lemma for a single application of transformation (j). Suppose that P contains the program statement

$A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

and we apply transformation (j) to obtain

$A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim p(x_1,\ldots,x_n) \wedge W_{i+1} \wedge \ldots \wedge W_m$

$p(x_1,\ldots,x_n) \leftarrow W$

where $x_1,\ldots,x_n$ are the free variables of W and W has the form $\exists y_1 \ldots \exists y_k V$. Let Q be the program obtained from P by replacing the statement to which the transformation was applied by these two statements.

Now comp(Q) contains the formula

$$\forall z_1 \ldots \forall z_n\ (p(z_1,\ldots,z_n) \leftrightarrow \exists x_1 \ldots \exists x_n((z_1=x_1) \wedge \ldots \wedge (z_n=x_n) \wedge W))$$

As in the proof of lemma 18.1, we can assume that the latter formula is replaced in comp(Q) by the formula

$$\forall x_1 \ldots \forall x_n (p(x_1,\ldots,x_n) \leftrightarrow W)$$

It follows easily from this that if U is a closed formula which is a logical consequence of comp(Q) and U contains only predicate symbols which appear in P, then U is a logical consequence of comp(P). ∎

Now we are in a position to define computed answers for programs and goals, and to show that computed answers are correct.

Definition Let P be a program and G a goal $\leftarrow$W, where W has free variables $x_1,...,x_n$. A *normal form* of P $\cup$ {G} is a normal program and goal P' $\cup$ {G'}, where G' is $\leftarrow$answer($x_1,...,x_n$) and P' is a normal form of P $\cup$ {answer($x_1,...,x_n$)$\leftarrow$W}.

Definition Let P be a program and G a goal.

An *SLDNF-derivation* of P $\cup$ {G} is an SLDNF-derivation of P' $\cup$ {G'}, where P' $\cup$ {G'} is a normal form of P $\cup$ {G}.

An *SLDNF-refutation* of P $\cup$ {G} is an SLDNF-refutation of P' $\cup$ {G'}, where P' $\cup$ {G'} is a normal form of P $\cup$ {G}.

A *computed answer* for P $\cup$ {G} is a computed answer for P' $\cup$ {G'}, where P' $\cup$ {G'} is a normal form of P $\cup$ {G}.

An *SLDNF-tree* for P $\cup$ {G} is an SLDNF-tree for P' $\cup$ {G'}, where P' $\cup$ {G'} is a normal form of P $\cup$ {G}.

A *finitely failed SLDNF-tree* for P $\cup$ {G} is a finitely failed SLDNF-tree for P' $\cup$ {G'}, where P' $\cup$ {G'} is a normal form of P $\cup$ {G}.

It is straightforward to show that the above definitions essentially extend those given in chapter 3 for normal programs and normal goals. (See problem 4.)

We now consider the problem of computations floundering. Let P be a program and G a goal. By a *computation* of P $\cup$ {G}, we mean an attempt to construct an SLDNF-derivation of P' $\cup$ {G'}, where P' $\cup$ {G'} is a normal form of P $\cup$ {G}.

Definition Let P be a program and G a 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.

Definition Let P be a program and G a goal. We say that P $\cup$ {G} is *allowed* if some normal form of P $\cup$ {G} is allowed.

It is straightforward to show that if one normal form of P ∪ {G} is allowed, then every normal form of P ∪ {G} is allowed. (See problem 5.)

Proposition 18.5 Let P be a program and G a goal ←W. Suppose that P ∪ {G} is allowed. Then we have the following properties.
(a) No computation of P ∪ {G} flounders.
(b) Every computed answer for P ∪ {G} is a ground substitution for all free variables in W.

Proof The proposition follows immediately from proposition 15.1. ■

We now prove the soundness of the negation as failure rule and SLDNF-resolution.

Theorem 18.6 (Soundness of the Negation as Failure Rule)
Let P be a program and G a goal. If P ∪ {G} has a finitely failed SLDNF-tree, then G is a logical consequence of comp(P).

Proof Note first that the result is known to hold when P is a normal program and G is a normal goal (theorem 15.4). Suppose G is the goal ←W, where W has free variables $x_1,...,x_n$. Let P" be P ∪ {answer($x_1,...,x_n$)←W}. Suppose P ∪ {G} has a finitely failed SLDNF-tree. By definition, P' ∪ {G'} has a finitely failed SLDNF-tree, where G' is ←answer($x_1,...,x_n$) and P' is a normal form of P". Thus, G' is a logical consequence of comp(P'). By lemma 18.4, G' is a logical consequence of comp(P"). Thus, by lemma 18.1(a), G is a logical consequence of comp(P). ■

Theorem 18.7 (Soundness of SLDNF-Resolution)
Let P be a program and G a goal. Then every computed answer for P ∪ {G} is a correct answer for comp(P) ∪ {G}.

Proof Note first that the result is known to hold when P is a normal program and G is a normal goal (theorem 15.6). Suppose G is the goal ←W, where W has free variables $x_1,...,x_n$. Let P" be P ∪ {answer($x_1,...,x_n$)←W} and θ be a computed answer for P ∪ {G}. By definition, θ is a computed answer for P' ∪ {G'}, where G' is ←answer($x_1,...,x_n$) and P' is a normal form of P". Hence, θ is a correct answer for comp(P') ∪ {G'}. By lemma 18.4, ∀(answer($x_1,...,x_n$)θ) is a logical consequence of comp(P"). Thus, by lemma 18.1(b), ∀(Wθ) is a logical consequence of comp(P). That is, θ is a correct answer for comp(P) ∪ {G}. ■

Theorems 18.6 and 18.7 are due to Lloyd and Topor [61].

Next, we shall prove a completeness result for hierarchical programs, which extends theorem 16.3.

Lemma 18.8 Let P be a program and P' a normal form of P. Then comp(P) is a logical consequence of comp(P').

Proof By lemma 18.3, we only have to prove the lemma when P' is a program obtained from P by a single application of transformation (j). Suppose that P contains the program statement

$A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim W \wedge W_{i+1} \wedge \ldots \wedge W_m$

and we apply transformation (j) to obtain

$A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \sim p(x_1,\ldots,x_n) \wedge W_{i+1} \wedge \ldots \wedge W_m$

$p(x_1,\ldots,x_n) \leftarrow W$

where $x_1,\ldots,x_n$ are the free variables in W and W has the form $\exists y_1 \ldots \exists y_k V$. Let P' be the program obtained from P by replacing the statement to which the transformation was applied by these two statements.

Now comp(P') contains the formula

$$\forall z_1 \ldots \forall z_n \ (p(z_1,\ldots,z_n) \leftrightarrow \exists x_1 \ldots \exists x_n((z_1=x_1) \wedge \ldots \wedge (z_n=x_n) \wedge W))$$

Using equality axioms 6, 7 and 8, we can assume that the latter formula is replaced in comp(P') by the formula

$$\forall x_1 \ldots \forall x_n(p(x_1,\ldots,x_n) \leftrightarrow W)$$

It follows easily from this that comp(P) is a logical consequence of comp(P'). ∎

If P is a program and P' is a normal form of P, then it follows from lemmas 18.4 and 18.8 that comp(P') is a conservative extension [99] of comp(P).

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

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

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

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

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

Theorem 18.9 (Completeness of SLDNF-Resolution for Hierarchical Programs)

Let P be a hierarchical program, G a goal $\leftarrow$W, and R a safe computation rule. Suppose that P $\cup$ {G} is allowed. Then the following properties hold.

(a) For every normal form of P $\cup$ {G}, the corresponding 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 free variables in W, then θ is an R-computed answer for P $\cup$ {G}.

Proof (a) Let P' $\cup$ {G'} be a normal form of P $\cup$ {G}. Then P' is hierarchical (see problem 8) and part (a) follows from theorem 16.3(a).

(b) Since θ is a correct answer for comp(P) $\cup$ {G} that is a ground substitution for all free variables in W, we have that Wθ is a logical consequence of comp(P). By lemma 18.1(b), answer(x_1,...,x_n)θ is a logical consequence of comp(P $\cup$ {answer(x_1,...,x_n)$\leftarrow$W}). By lemma 18.8, answer(x_1,...,x_n)θ is a logical consequence of comp(P'). The result now follows from theorem 16.3(b). ■

§19. DECLARATIVE ERROR DIAGNOSIS

This section presents an error diagnoser which finds errors in programs that use advanced control facilities and the increased expressiveness of program statements. The diagnoser is declarative, in the sense that the programmer need only know the intended interpretation of an incorrect program to use the diagnoser. In particular, the programmer needs no understanding whatever of the underlying computational behaviour of the PROLOG system which runs the program. It is argued that declarative error diagnosers will be indispensable components of advanced logic programming systems, which are currently under development. The results of this section are due to Lloyd [59].

One of the greatest strengths of logic programming is its declarative nature. To a large extent, programmers need only concern themselves with a declarative understanding of their programs, leaving much of the procedural aspect to the logic programming system itself.

However, the ideal of *purely* declarative programming is still far from being achieved. Current research aimed at attaining this ideal is proceeding on a number of fronts. For example, some PROLOG systems have advanced control facilities to overcome the severe limitations of the standard left to right computation rule (e.g.,

[73], [74]). Improved forms of negation are being introduced (e.g., [75], [104]). There has been work on program transformation, which allows programmers to write programs in a form closer to their specification (e.g., [101] and the references therein).

The advanced logic programming systems, which will become available in the near future, will be compiler systems exploiting all the above techniques. *Source* programs for these systems will be written in a subset of first order logic. This subset will include at least the class of programs defined in this chapter. In the first stage of compilation, source programs will be transformed into *assembly* programs by the automatic addition of control information and the application of various transformation techniques. These assembly language programs will be similar to PROLOG programs as they are currently written for a coroutining system. In the second stage of compilation, the assembly program will be further compiled into a *machine* program, which can then be run on a coroutining version of Warren's abstract PROLOG machine [110]. This second compilation stage is now well understood. Note that, according to the above view, current versions of PROLOG, which are now regarded as high level languages, will eventually be regarded as low level machine languages.

Such systems will allow programmers to write in a more declarative style than is currently possible and should ensure a great decrease in programmer effort. However, there is a catch. The compiled program could be so different from the source program and the control could be so complicated that debugging such programs by conventional tracing techniques is likely to be extraordinarily difficult. In other words, the programmer may only require an understanding of the intended interpretation to *write* the program, but will need to know everything about the computational behaviour of the system to *debug* the program! In fact, this problem in a less extreme form also plagues current PROLOG systems.

For this reason, we argue that an indispensable component of future logic programming systems will be a declarative debugging system, that is, one that can be used without the need to understand the computational behaviour of the system. The main purpose of this section is to present a declarative error diagnoser which finds errors in programs that use advanced control facilities and the increased expressiveness of program statements. Attention is confined to errors which lead to a wrong or missing solution. In particular, errors which lead to infinite loops are not discussed here.

Declarative error diagnosis was introduced into logic programming, under the name *algorithmic* debugging, by Shapiro [92]. As well as an error diagnoser, he also presented an error corrector (regarded as a kind of inductive program synthesiser). Shapiro was mainly concerned with definite programs using the standard computation rule. Av-Ron [6] studied top-down diagnosers for definite programs. Under the name *rational* debugging, Pereira [81] presented a diagnoser for arbitrary PROLOG programs, including the non-declarative features of PROLOG, such as cut. More recently, Ferrand [34] gave a mathematical analysis of an error diagnoser for definite programs. Other work on debugging (not necessarily declarative) is contained in [12], [30], [31], [32], [83] and the references therein.

We now give the definitions of the concepts necessary for a foundation for error diagnosis.

Definition Let P be a program. An *intended interpretation* for P is a normal Herbrand interpretation for comp(P).

The restriction to Herbrand interpretations is not essential. However, in practice, intended interpretations are usually Herbrand and the analysis is a little easier in this case. The foremost aim of a programmer is to write programs which have their intended interpretations as models. This leads to the following definition.

Definition Let P be a program and I an intended interpretation for P. We say P is *correct* wrt I if I is a model for comp(P); otherwise, we say that P is *incorrect* wrt I.

Of course, the reason we want P to be correct wrt I is so that all answers computed by P will be true wrt I.

Proposition 19.1 Let P be a program, G a goal $\leftarrow W$, and θ a computed answer for $P \cup \{G\}$. Let I be an intended interpretation for P and suppose that P is correct wrt I. Then $W\theta$ is valid in I.

Proof The result follows immediately from the soundness of SLDNF-resolution (theorem 18.7), since I is a model for comp(P). ■

However, even if P is correct wrt I, we cannot guarantee that P will compute *everything* in I.

Example Suppose that P is a definite program such that $\text{lfp}(T_P) \neq \text{gfp}(T_P)$. Then P is correct wrt $\text{gfp}(T_P)$, together with the identity relation assigned to =, but P does not compute all atoms in $\text{gfp}(T_P)$.

In other words, even if P is correct wrt I, P may still have a bug in the sense that it is incomplete. This kind of bug is not detectable by the error diagnoser. What it *can* detect is when P is incorrect wrt I.

An error in a program usually shows up because the program gives a wrong answer or misses an answer (more precisely, finitely fails when it should succeed). The next proposition formalises this.

Proposition 19.2 Let P be a program, G a goal $\leftarrow W$, and I an intended interpretation for P.
(a) If θ is a computed answer for $P \cup \{G\}$ and $W\theta$ is not valid in I, then P is incorrect wrt I.
(b) If $P \cup \{G\}$ has a finitely failed SLDNF-tree and W is satisfiable in I, then P is incorrect wrt I.

Proof Part (a) follows directly from the soundness of SLDNF-resolution (theorem 18.7) and part (b) follows directly from the soundness of the negation as failure rule (theorem 18.6). ■

Now we define the two kinds of errors which the diagnoser can detect.

Definition Let P be a program and I an intended interpretation for P. Let A be an atom with predicate symbol p. We say that A is an *uncovered atom* for P wrt I if A is valid in I and, for every program statement $A' \leftarrow W$ in the definition of p such that A and A' unify with mgu θ, say, we have that $W\theta$ is unsatisfiable in I.

Definition Let P be a program and I an intended interpretation for P. We say an instance $A \leftarrow W$ of a program statement in P is an *incorrect statement instance* for P wrt I if A is unsatisfiable in I and W is valid in I. In case the program statement is a program clause, we call the incorrect statement instance an *incorrect clause instance.*

Note that every instance of an uncovered atom is uncovered and every instance of an incorrect statement instance is incorrect.

The next result gives the connection between the concepts of incorrect program, uncovered atom, and incorrect statement instance.

Proposition 19.3 Let P be a program and I an intended interpretation for P. Then P is incorrect wrt I iff there is an uncovered atom for P wrt I or there is an incorrect statement instance for P wrt I.

Proof Suppose that there is an incorrect statement instance (in the definition of p) for P wrt I. It is easy to see that I does not satisfy the if part

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

of the completed definition of p and hence that P is incorrect wrt I. Next suppose that there is an uncovered atom $p(s_1,\ldots,s_n)$ for P wrt I. If there is no definition for p, then it follows immediately that P is incorrect wrt I. Otherwise, I does not satisfy the only if part

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

of the completed definition of p and hence P is incorrect wrt I.

Now suppose that P is incorrect wrt I. Note that any normal Herbrand interpretation for comp(P) is a model for the equality theory of comp(P) and thus I can not be a model for the remainder of comp(P). If I does not satisfy a completed definition of the form

$$\forall x_1 \ldots \forall x_n\ \sim p(x_1,\ldots,x_n)$$

then there is an uncovered atom. If I does not satisfy the only if part

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

of a completed definition, then there is an uncovered atom. Finally, if I does not satisfy the if part

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

of a completed definition, then there is an incorrect statement instance. ■

Propositions 19.2 and 19.3 together show that if a program gives a wrong answer or misses an answer, then there is an uncovered atom or an incorrect statement instance. We now present a diagnoser which detects these errors.

The definitions below are those of the main predicates, wrong and missing. The definitions of the predicates valid, unsatisfiable and clause need to be added. If W is a formula, we let W' denote its image, which is a *ground* term, under the representation scheme used by the diagnoser. This scheme uses "and" for conjunction, "or" for disjunction, "not" for negation, "if" for implication, "all(x',W')" for $\forall x W$, and "some(x',W')" for $\exists x W$.

Declarative Error Diagnoser

wrong(all(v, w), x) $\leftarrow$ wrong(w, x)
wrong(some(v, w), x) $\leftarrow$ wrong(w, x)
wrong(v if w, x) $\leftarrow$ wrong(v, x)
wrong(v if w, x) $\leftarrow$ missing(w, x)
wrong(v or w, x) $\leftarrow$ wrong(v, x)
wrong(v or w, x) $\leftarrow$ wrong(w, x)
wrong(not w, x) $\leftarrow$ missing(w, x)
wrong(v and w, x) $\leftarrow$ wrong(v, x)
wrong(v and w, x) $\leftarrow$ wrong(w, x)
wrong(x, z) $\leftarrow$ clause(x, x_1 if y) $\wedge$ wrong(y, z)
wrong(x, x_1 if y) $\leftarrow$ unsatisfiable(x, x_1) $\wedge$ clause(x, x_1 if y) $\wedge$ valid(y, y)

missing(all(v, w), x) $\leftarrow$ missing(w, x)
missing(some(v, w), x) $\leftarrow$ missing(w, x)
missing(v if w, x) $\leftarrow$ missing(v, x)
missing(v if w, x) $\leftarrow$ wrong(w, x)
missing(v or w, x) $\leftarrow$ missing(v, x)
missing(v or w, x) $\leftarrow$ missing(w, x)
missing(not w, x) $\leftarrow$ wrong(w, x)
missing(v and w, x) $\leftarrow$ missing(v, x)
missing(v and w, x) $\leftarrow$ missing(w, x)
missing(x, z) $\leftarrow$ clause(x, x_1 if y) $\wedge$ missing(y, z)
missing(x, x_1) $\leftarrow$ valid(x, x_1) $\wedge$ $\forall y(\exists x_2$clause(x_1, x_2 if y) $\rightarrow$ unsatisfiable(y, y))

The first argument of wrong is a goal (body). The second argument is an uncovered atom or incorrect statement instance returned by the diagnoser. An incorrect statement instance is actually found using the last statement of the definition of wrong. The first argument of missing is a goal (body). Similarly, the second argument is an uncovered atom or incorrect statement instance returned by the diagnoser. An uncovered atom is actually found using the last statement of the definition of missing.

The definition of clause contains all facts of the form clause(A', B' if W')$\leftarrow$, where A is an atom, B is an instance of A and B$\leftarrow$W is an instance of a program statement. The definition of valid contains all facts of the form valid(W', V')$\leftarrow$, where W is a formula and V is an instance of W valid in I. The definition of unsatisfiable contains all facts of the form unsatisfiable(W', V')←, where W is a formula and V is an instance of W unsatisfiable in I.

What we have presented above is the purely declarative part of the diagnoser. It is important to isolate this declarative component, as we have done, for two reasons. First, it clarifies the theoretical developments. One can prove the soundness and completeness of the diagnoser without the complication of coping with some particular control component. Second, it makes the challenge of building practical error diagnosers clearer. This challenge is to find a sufficiently clever control component to add to the above declarative component. Later we show one way of adding this control.

The last four statements in the definition of wrong and the last four statements in the definition of missing could be used together as a diagnoser for definite programs. This diagnoser can be compared directly with the diagnosers of Shapiro [92], Av-Ron [6] and Ferrand [34] for definite programs. Later we compare Shapiro's single-stepping and divide-and-query algorithms for diagnosing incorrect answers with a top-down version of the diagnoser. The main difference between the diagnoser and Ferrand's is that we have dispensed with the statements in his diagnoser which are concerned with returning the result that the error is undefined.

The seventh statement in the definition of wrong and the seventh statement in the definition of missing together handle negated calls. These statements come from [92], where they are attributed to McCabe. Their motivation is as follows. If the negation of a goal has incorrectly succeeded (resp., incorrectly failed), then the goal must have incorrectly failed (resp., incorrectly succeeded). The remainder of the statements in the definitions handle the other connectives and quantifiers. As an example, we give the motivation for the statements for implication in the definition of wrong: if the goal v if w has returned a wrong answer, then either v has returned a wrong answer or w has missed an answer.

We now show a method for adding control information to obtain a more practical declarative error diagnoser. The idea is to ensure that the following conditions are satisfied. In every call to wrong, the first argument is unsatisfiable. Similarly, in every call to missing, the first argument is valid. For this purpose, we make sure that a top level call to wrong has its first argument unsatisfiable and a top level call to missing has its first argument valid. Furthermore, we add calls to valid and unsatisfiable to ensure that subsequent calls to wrong and missing satisfy the above conditions. (See problem 13.)

We also add calls to succeed and fail. The definition of succeed contains all facts of the form succeed(W', (Wθ)')←, where W is a formula and θ is a computed answer for P ∪ {←W}. The definition of fail contains all facts of the form fail(W')←, where W is a formula and P ∪ {←W} has a finitely failed SLDNF-tree. These additional calls are used as heuristics to guide the search for an error. We call this the *top-down* version of the diagnoser. For definite programs, the top-down diagnoser for wrong answers was given by Av-Ron [6]. A different top-down diagnoser for missing answers for definite programs was also given in [6].

Top-Down Version of the Declarative Error Diagnoser

wrong(all(v, w), x) ← unsatisfiable(w, w_1) ∧ wrong(w_1, x)
wrong(some(v, w), x) ← wrong(w, x)
wrong(v if w, x) ← succeed(v, v_1) ∧ wrong(v_1, x)
wrong(v if w, x) ← fail(w) ∧ missing(w, x)
wrong(v or w, x) ← succeed(v, v_1) ∧ wrong(v_1, x)
wrong(v or w, x) ← succeed(w, w_1) ∧ wrong(w_1, x)
wrong(not w, x) ← missing(w, x)
wrong(v and w, x) ← unsatisfiable(v, v_1) ∧ wrong(v_1, x)
wrong(v and w, x) ← unsatisfiable(w, w_1) ∧ wrong(w_1, x)
wrong(x, z) ← clause(x, x_1 if y) ∧ succeed(y, y) ∧ unsatisfiable(y, y) ∧ wrong(y, z)
wrong(x, x_1 if y) ← unsatisfiable(x, x_1) ∧ clause(x, x_1 if y) ∧ valid(y, y)

missing(all(v, w), x) ← missing(w, x)
missing(some(v, w), x) ← valid(w, w_1) ∧ missing(w_1, x)
missing(v if w, x) ← valid(v, v_1) ∧ missing(v_1, x)
missing(v if w, x) ← unsatisfiable(w, w_1) ∧ wrong(w_1, x)
missing(v or w, x) ← valid(v, v_1) ∧ missing(v_1, x)
missing(v or w, x) ← valid(w, w_1) ∧ missing(w_1, x)
missing(not w, x) ← wrong(w, x)
missing(v and w, x) ← fail(v) ∧ missing(v, x)
missing(v and w, x) ← fail(w) ∧ missing(w, x)
missing(x, z) ← clause(x, x_1 if y) ∧ fail(y) ∧ valid(y, y) ∧ missing(y, z)
missing(x, x_1) ← valid(x, x_1) ∧ ∀y(∃$x_2$clause(x_1, x_2 if y) → unsatisfiable(y, y))

Example Consider the following (incorrect) subset program in which sets are represented by lists.

subset(x,y) ← ∀z (member(z,y) ← member(z,x))

member(x,x.y) ←

member(x,y.z) ← member(y,z)

The goal ←subset(2.nil, 1.2.nil) incorrectly fails. The top-down algorithm produces the following computation (in which some intermediate goals are not shown).

← missing(subset(2.nil, 1.2.nil), x)

← missing(all(z', member(z', 1.2.nil) if member(z', 2.nil)), x)

← missing(member(z', 1.2.nil) if member(z', 2.nil), x)

← missing(member(2, 1.2.nil), x)

□

The computed answer is x/member(2, 1.2.nil), that is, member(2, 1.2.nil) is an uncovered atom.

The implementation of the top-down algorithm in MU-PROLOG and examples of its use are given in [59]. (For examples of the use of various other declarative error diagnosers, the reader should consult [6], [34], [81] and [92].) The implementations of valid and unsatisfiable rely on an *oracle* to answer questions about the intended interpretation. In practice, the oracle is usually the programmer. Answers from the oracle are recorded so that the oracle is never asked the same question twice. Also complex formulas are broken down so that the oracle is only ever questioned about the validity of atoms.

Example For the previous example, the implementation in [59] of the top-down algorithm produces the following sequence of oracle queries.

```
subset(2.nil, 1.2.nil) valid?
member(z, 2.nil) valid?
z=2.
member(2, 1.2.nil) valid?
The following atoms are known to be valid:
member(2, 2.nil)
member(z, 2.nil) valid for other values? n
member(1, 2.nil) valid? n
```

at which point the uncovered atom member(2, 1.2.nil) is printed. A return after the ? indicates yes, while an n followed by a return indicates no. The value z=2 was given by the oracle after a prompt with the variable name.

Note that, by means of the metacalls, succeed and fail, the top-down algorithm has decoupled the diagnosis of the program from whatever transformation, compilation or advanced control was applied to the program. In other words, the top-down algorithm is essentially independent of the underlying computational behaviour of the logic programming system, which could therefore be changed or improved without affecting the diagnoser.

We have tried to minimise the number of oracle calls made by the top-down algorithm, without being too concerned about its computational complexity. Nevertheless, this algorithm makes rather extravagant use of metacalls and hence could be prohibitively expensive for some programs. It would be possible to reduce this cost by building the erroneous refutation (or finitely failed tree) *once* at the beginning of the diagnosis and then searching this refutation (or tree) for the error. Wrong could be easily adapted to this approach, but missing would seem to require more extensive changes, along the lines of [6].

The top-down algorithm for diagnosing missing answers for definite programs is similar to Shapiro's algorithm for missing answers [92, p.55]. We now briefly compare the top-down algorithm for diagnosing incorrect answers for definite programs with the single-stepping and divide-and-query algorithms of Shapiro [92]. For this comparison, it is convenient to assume that, for all three algorithms, the final computation tree of the erroneous computation is first constructed and the algorithms search this tree for the incorrect clause instance. The final computation tree is the AND-tree corresponding to the refutation obtained by applying all the mgu's used in the refutation to all the nodes in the tree. For simplicity, we also assume that the goal (body) is a single atom. Thus some instance of this atom is the root of the final computation tree and its children are instances of atoms in the body of the input clause invoked by the goal.

The single-stepping algorithm finds the error by doing a post-order traversal of the final computation tree. Suppose the algorithm has just queried the oracle about all the children of some node and found them to be valid. It then queries the oracle about the node itself. If this node is not valid, then an incorrect clause instance has been found. If this node is valid, then the algorithm continues the post-order traversal. This algorithm is essentially a bottom-up algorithm. It has the disadvantage that its worst case query complexity is equal to the number of nodes in the tree. A version of the single-stepping algorithm is as follows.

wrong(v and w, x) $\leftarrow$ wrong(v, x)
wrong(v and w, x) $\leftarrow$ wrong(w, x)
wrong(x, z) $\leftarrow$ clause(x, x_1 if y) $\wedge$ succeed(y, y) $\wedge$ wrong(y, z)
wrong(x, x_1 if y) $\leftarrow$ unsatisfiable(x, x_1) $\wedge$ clause(x, x_1 if y) $\wedge$ valid(y, y)

The divide-and-query algorithm is an improvement in that its query complexity is optimal to within a constant factor. The idea of this algorithm is as follows. It finds a node in the tree such that the weight of the subtree rooted at that node is as close as possible to half the weight of the entire tree. It then queries the oracle about this node. If this node is not valid, then the algorithm recursively enters the subtree rooted at this node. If not, the algorithm calculates a new "middle" node for the entire tree with this subtree deleted. It is shown in [92] that this algorithm has logarithmic query complexity. Unfortunately, it is rarely possible to divide the tree in half. Usually, we must settle for a "middle" node which is the root of a subtree with somewhat smaller weight. This detracts from the performance of the divide-and-query algorithm. If the tree has n nodes and branching factor b, then the worst case query complexity is blog n (not log n, as a superficial analogy with the binary search algorithm might suggest).

The top-down algorithm searches the final computation tree as follows. First, the oracle is queried about the root node, which is presumably not valid. It then queries each child of the root node in turn. If they are all valid, then an incorrect clause instance has been found. Otherwise, it enters the subtree rooted at the leftmost child which it finds to be not valid and continues the search in the same way in this subtree. The top-down algorithm does indeed search the tree in a top-down fashion. Note that it would be easy to add the flexibility of querying the children in some preferred order. If the final computation tree has branching factor b and height h, then the worst case query complexity of the top-down algorithm is bh.

We now compare in more detail the query complexity of the top-down and divide-and-query algorithms. First, the top-down algorithm can perform worse than the divide-and-query algorithm. Suppose the tree is linear and the error is right at the bottom of the tree. The top-down algorithm queries all nodes in the tree, while the divide-and-query algorithm only queries the logarithm of this number. On the other hand, suppose the tree has two subtrees, the one on the right being very much greater than the one on the left, and the only error is in the left subtree. The top-down algorithm will quickly find the error by immediately searching the left subtree, while the divide-and-query algorithm will fruitlessly search the right subtree before finally searching the left subtree. Thus the top-down algorithm can perform better than the divide-and-query algorithm.

Suppose the final computation tree is perfectly balanced (that is, every internal node has b children and all leaf nodes are at the same level) with height h and branching factor b (>1). In this case, the "middle" node will be the leftmost child of the root node. If this node is valid and b>2, the next "middle" node will be the second from left child of the root node. Assuming the rightmost child is the only child which is not valid, the divide-and-query algorithm will query all the other children before searching the subtree rooted at the rightmost node. Thus, for a perfectly balanced tree, the top-down and divide-and-query algorithms search the tree in a very similar manner. They both have worst case query complexity bh, approximately.

The advantage of the divide-and-query algorithm is its logarithmic worst case query complexity for *any* computation tree. However, its method of deciding which node to query next is relatively inflexible and is dependent on a syntactic criterion unrelated to the error. In this regard, the top-down algorithm is more flexible, as it would be easy to add heuristics to suggest an order in which to query the children of a node. It would be interesting to compare these two algorithms on a large variety of incorrect programs and also to see the effectiveness of various heuristics.

§20. SOUNDNESS AND COMPLETENESS OF THE DIAGNOSER

Let us now turn to the soundness and completeness of the (first version on page 124 of the) diagnoser. In the following theorems, it is assumed that valid, unsatisfiable and clause have the sound and complete definitions indicated above. The results of this section are due to Lloyd [59].

Theorem 20.1 (Soundness of the Error Diagnoser)

Let P be a program, ←W a goal, and I an intended interpretation for P.

(a) If ←wrong(W', x) (resp., ←missing(W', x)) returns the answer x = A' if V', then A←V is an incorrect statement instance for P wrt I.

(b) If ←wrong(W', x) (resp., ←missing(W', x)) returns the answer x = A', then A is an uncovered atom for P wrt I.

In either case, P is incorrect wrt I.

Proof Parts (a) and (b) of the theorem are proved by induction on the total number of calls to wrong and missing on the refutation produced by the diagnoser. If there is only one such call, then either the last statement in the definition of wrong or the (transformed version of the) last statement in the definition of missing must be the single input clause used from either of these definitions. In the first case, it is clear that $A \leftarrow V$ is an incorrect statement instance. In the second case, it is clear that A is an uncovered atom.

Now suppose that parts (a) and (b) of the theorem are true when the total number of calls to wrong and missing is n. Consider a refutation which has n+1 such calls. An examination of the definitions of wrong and missing shows that the first such call can use any statement as an input clause, except the last statement in either definition. Thus the first call merely returns the result given by the derivation starting from the second call to missing or wrong, which produces a correct result, by the induction hypothesis. Parts (a) and (b) of the theorem follow from this.

The last part of the theorem now follows from proposition 19.3. ■

Next we study the completeness of the diagnoser. For this, it is convenient to define (inductively) the concept of a formula and an atom being connected wrt a program.

Definition Let W be a formula, A an atom, and P a program.

We say A is *connected positively* (resp., *negatively*) *to* W *in 0 steps wrt* P if A occurs positively (resp., negatively) in W.

We say A is *connected positively* (resp., *negatively*) *to* W *in n steps wrt* P ($n>0$) if *either* there exists an atom B occurring positively in W and a statement $C \leftarrow V$ in P such that B and C are unifiable with mgu θ, say, and A is connected positively (resp., negatively) to $V\theta$ in $n-1$ steps wrt P *or* there exists an atom B occurring negatively in W and a statement $C \leftarrow V$ in P such that B and C are unifiable with mgu θ, say, and A is connected negatively (resp., positively) to $V\theta$ in $n-1$ steps wrt P.

Definition Let W be a formula, A an atom, and P a program. We say that A is *connected positively* (resp., *negatively*) *to* W *wrt* P if A is connected positively (resp., negatively) to W in n steps wrt P, for some $n \geq 0$.

Lemma 20.2 Let P be a program, $\leftarrow W$ a goal, A an atom, and I an intended interpretation for P. Let A be connected positively (resp., negatively) to W wrt P.

(a) If an instance of A is the head of an incorrect statement instance for P wrt I, then there exists a computed answer for $\leftarrow$wrong(W', x) (resp., $\leftarrow$missing(W', x)) in which x is bound to the representation of this incorrect statement instance.
(b) If an instance of A is an uncovered atom for P wrt I, then there exists a computed answer for $\leftarrow$missing(W', x) (resp., $\leftarrow$wrong(W', x)) in which x is bound to the representation of this uncovered atom.

Proof The proof is a straightforward induction argument on the number of steps needed to connect W and A. (See problem 14.) ■

Lemma 20.3 Let P be a normal program, G a normal goal $\leftarrow$W, and I an intended interpretation for P.
(a) If θ is a computed answer for $P \cup \{G\}$ and Wθ is not valid in I, then *either* there exists an atom A connected positively to W wrt P such that an instance of A is the head of an incorrect clause instance for P wrt I *or* there exists an atom A connected negatively to W wrt P such that an instance of A is an uncovered atom for P wrt I.
(b) If $P \cup \{G\}$ has a finitely failed SLDNF-tree and W is satisfiable in I, then *either* there exists an atom A connected positively to W wrt P such that an instance of A is an uncovered atom for P wrt I *or* there exists an atom A connected negatively to W wrt P such that an instance of A is the head of an incorrect clause instance for P wrt I.

Proof Let W be $L_1 \wedge \ldots \wedge L_n$. Parts (a) and (b) are proved together by induction on the number of calls k (including calls in subsidiary refutations and trees) in the SLDNF-refutation for (a) and in the SLDNF-tree for (b), respectively. When k=1, the result is obvious. Now suppose that (a) and (b) hold when there are at most k–1 calls.

(a) Suppose θ is a computed answer for $P \cup \{G\}$, Wθ is not valid in I and the SLDNF-refutation has k calls. We can assume that θ is actually the composition of the substitutions used in the SLDNF-refutation. Let L_i be the selected literal in G. We consider two cases.

L_i is a negative literal

Suppose L_i is ~B. If B is satisfiable in I, then $P \cup \{\leftarrow B\}$ has a finitely failed SLDNF-tree with < k calls and the result follows by the induction hypothesis. Otherwise, B is unsatisfiable in I and hence L_i is valid in I. Thus θ is a computed answer for $P \cup \{\leftarrow L_1 \wedge \ldots \wedge L_{i-1} \wedge L_{i+1} \wedge \ldots \wedge L_n\}$ and $(L_1 \wedge \ldots \wedge L_{i-1} \wedge L_{i+1} \wedge \ldots \wedge L_n)\theta$ is not valid in I. Hence the result follows from the induction hypothesis.

L_i is a positive literal

Let $B \leftarrow V$ be the first input clause. Suppose that $(L_1 \wedge ... \wedge L_{i-1} \wedge V \wedge L_{i+1} \wedge ... \wedge L_n)\theta$ is not valid in I. Then the result follows from the induction hypothesis. Otherwise, $L_i\theta$ is not valid in I. Hence $B\theta \leftarrow V\theta$ has an incorrect clause instance and the result follows.

(b) Suppose $P \cup \{G\}$ has a finitely failed SLDNF-tree, W is satisfiable in I and the SLDNF-tree has k calls. Let L_i be the selected literal in G. We consider two cases.

L_i is a negative literal

Suppose L_i is $\sim B$. Suppose first that L_i fails. Then the identity substitution is a computed answer for $P \cup \{\leftarrow B\}$ and B is not valid in I. The result follows by applying the induction hypothesis. Otherwise, L_i succeeds. Then $P \cup \{\leftarrow L_1 \wedge ... \wedge L_{i-1} \wedge L_{i+1} \wedge ... \wedge L_n\}$ has a finitely failed SLDNF-tree and $L_1 \wedge ... \wedge L_{i-1} \wedge L_{i+1} \wedge ... \wedge L_n$ is satisfiable in I. Again, the result follows from the induction hypothesis.

L_i is a positive literal

Suppose there exists an input clause $B \leftarrow V$ with mgu θ_1, say, such that $(L_1 \wedge ... \wedge L_{i-1} \wedge V \wedge L_{i+1} \wedge ... \wedge L_n)\theta_1$ is satisfiable in I. Then the result follows by applying the induction hypothesis. Otherwise, an instance of L_i is an uncovered atom and the result follows. ■

Next we generalise lemma 20.3 to arbitrary programs and goals.

Lemma 20.4 Let P be a program, G a goal $\leftarrow W$, and I an intended interpretation for P.

(a) If θ is a computed answer for $P \cup \{G\}$ and $W\theta$ is not valid in I, then *either* there exists an atom A connected positively to W wrt P such that an instance of A is the head of an incorrect statement instance for P wrt I *or* there exists an atom A connected negatively to W wrt P such that an instance of A is an uncovered atom for P wrt I.

(b) If $P \cup \{G\}$ has a finitely failed SLDNF-tree and W is satisfiable in I, then *either* there exists an atom A connected positively to W wrt P such that an instance of A is an uncovered atom for P wrt I *or* there exists an atom A connected negatively to W wrt P such that an instance of A is the head of an incorrect statement instance for P wrt I.

Proof (a) First, we show that we can reduce the lemma to the case that W is an atom. Suppose that W has free variables $x_1,...,x_n$. Let answer be a new n-ary predicate symbol. Let G' be $\leftarrow$answer($x_1,...,x_n$) and P' be P $\cup$ {answer($x_1,...,x_n$)$\leftarrow$W}. Extend I to an interpretation I' for P' by defining answer($t_1,...,t_n$) to be true in I' if W{$x_1/t_1,...,x_n/t_n$} is true in I, where $t_1,...,t_n$ are ground terms. If θ is a computed answer for P $\cup$ {G} and Wθ is not valid in I, then it is clear that θ is a computed answer for P' $\cup$ {G'} and answer($x_1,...,x_n$)θ is not valid in I'. Note also that no instance of the statement for answer is incorrect for P' wrt I' and no instance of answer($x_1,...,x_n$) is uncovered for P' wrt I'. Assuming the result is true for the case when the goal (body) is an atom, either there exists an atom A connected positively to answer($x_1,...,x_n$) wrt P' such that an instance of A is the head of an incorrect statement instance for P' wrt I' or there exists an atom A connected negatively to answer($x_1,...,x_n$) wrt P' such that an instance of A is an uncovered atom for P' wrt I'. Part (a) of the lemma follows easily from this.

Let us now assume that W is an atom. We prove the result by induction on the number of transformation steps k required to transform P into a normal form of P. When k=0, P is already a normal program and the result follows from lemma 20.3.

Next suppose that the result holds for programs which require at most k–1 transformation steps. Let P be a program which requires k such steps. Suppose P' is the program obtained from P by applying the first such transformation step. Note that if θ is a computed answer for P $\cup$ {G}, then θ is a computed answer for P' $\cup$ {G}

Suppose that the first transformation used is one of the first nine transformations, (a) to (i), given in §18. In this case, if B is an uncovered atom for P' wrt I, then B is also an uncovered atom for P wrt I. Similarly, if B$\leftarrow$V is an incorrect statement instance for P' wrt I, then either B$\leftarrow$V is an incorrect statement instance for P wrt I or the statement in P, which gave rise via the transformation to the clause in P' whose instance is B$\leftarrow$V, has a corresponding incorrect statement instance. We can now obtain the result by applying the induction hypothesis to P'.

Finally, suppose that the first such transformation used is the last transformation (j) given in §18, that is,

Replace $B \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \sim\exists x_1 ... \exists x_n V \wedge W_{i+1} \wedge ... \wedge W_m$
by $B \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \sim p(y_1,...,y_k) \wedge W_{i+1} \wedge ... \wedge W_m$
and $p(y_1,...,y_k) \leftarrow \exists x_1 ... \exists x_n V$

where $y_1,...,y_k$ are the free variables in $\exists x_1...\exists x_n V$ and p is a new predicate symbol not already appearing in P. We extend I to I' for P' by defining p($t_1,...,t_k$) to be true in I' if ($\exists x_1...\exists x_n$V){$y_1/t_1,...,y_k/t_k$} is true in I, where $t_1,...,t_k$ are ground terms. Note that no instance of the statement for p is incorrect for P' wrt I' and no instance of $p(x_1,...,x_n)$ is uncovered for P' wrt I'. Note also that if an instance of $B \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \sim p(y_1,...,y_k) \wedge W_{i+1} \wedge ... \wedge W_m$ is incorrect for P' wrt I', then a corresponding instance of $B \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \sim \exists x_1 ... \exists x_n V \wedge W_{i+1} \wedge ... \wedge W_m$ is incorrect for P wrt I. Furthermore, if q is the predicate symbol of B and some atom C with predicate symbol q is uncovered for P' wrt I', then C is also uncovered for P wrt I. The result now follows by applying the induction hypothesis to P'.

(b) The proof of part (b) is similar. ■

Theorem 20.5 (Completeness of the Error Diagnoser)

Let P be a program, G a goal ←W, and I an intended interpretation for P.

(a) If θ is a computed answer for $P \cup \{G\}$ and $W\theta$ is not valid in I, then there exists a computed answer for ←wrong(W', x) in which x is bound to the representation of either an incorrect statement instance or an uncovered atom.

(b) If $P \cup \{G\}$ has a finitely failed SLDNF-tree and W is satisfiable in I, then there exists a computed answer for ←missing(W', x) in which x is bound to the representation of either an incorrect statement instance or an uncovered atom.

Proof The theorem follows immediately from lemmas 20.2 and 20.4. ■

The main advantages of the approach taken in this chapter to error diagnosis are that the diagnoser itself has a simple and elegant semantics, that the programmer only needs to know the intended interpretation of the incorrect program to debug it, and that the diagnoser can handle programs which use advanced control facilities and the increased expressiveness of program statements.

However, a disadvantage of the approach is that it does not cope with the non-declarative features of PROLOG, such as cut, assert and retract. At first sight, this would appear to invalidate the approach, since practically every non-trivial PROLOG program makes some use of these non-declarative features! However, the outlook is more promising than that.

The first point to note in this regard is that well-written PROLOG programs usually consist of a small number of definitions using non-declarative features together with the remainder of the definitions which are purely declarative (except possibly for safe uses of cut, which are only for efficiency and can be ignored for the purposes of debugging). This means that the programmer *can* use a diagnoser like the one above for debugging the major part of the program which is purely declarative. Second, as we pointed out earlier, there is a strong effort being put towards making the new generation of PROLOG systems more declarative. Advanced control facilities and better forms of negation allow the programmers to write their programs in a more declarative style. In fact, it may even be possible to avoid the overt use of cut entirely. All these advances in the design of PROLOG systems make the job of debugging much easier. They will also make the declarative diagnoser more practically useful, since the proportion of programs to which the pure approach above applies will increase.

Leaving aside the problem of the non-declarative features of PROLOG, we now look at other ways in which the diagnoser could be improved. A useful way of thinking about error diagnosers is that they are *expert systems* and a number of recent papers (e.g. [31], [32]) have taken this approach. One can imagine the diagnoser being augmented with expert knowledge about typical program errors and all kinds of heuristics for quickly locating them. Another interesting possibility would be the incorporation of the intelligent backtracking ideas of [81]. This has been investigated in some detail for definite programs in [6]. These ideas need to be extended to (arbitrary) programs.

The diagnoser also needs some method of locating errors which lead to infinite loops [92]. The analysis of a looping program is complicated by the fact that it may actually be correct wrt the intended interpretation, but get into an infinite loop because of the deficiencies of the standard PROLOG computation rule. The employment of advanced control facilities, which are more likely to avoid infinite loops [73], will help here.

Much more research needs to be done before we will be able to build truly practical declarative error diagnosers. We hope the results of this chapter will provide a useful foundation for this research.

PROBLEMS FOR CHAPTER 4

1. Prove proposition 17.3.

2. Consider the following program

grandparent(x,y) ← parent(x,z), parent(z,y)

parent(x,y) ← mother(x,y)
parent(x,y) ← father(x,y)

ancestor(x,y) ← parent(z,y), ancestor(x,z)
ancestor(x,y) ← parent(x,y)

father(Fred, Mary)
father(George, James)
father(John, Fred)
father(Albert, Jane)

mother(Sue, Mary)
mother(Jane, Sue)
mother(Liz, Fred)
mother(Sue, James)

(a) Write the following queries as goals.
(i) Who is the father of Jane?
(ii) Who has Sue as mother and John as grandfather?
(iii) Who are the ancestors of Mary?
(iv) Does every person with a mother also have a father?
(v) Are all Sue's children childless?
(vi) Find everyone who has a grandparent in common with Mary.
(vii) Find every mother who has no father.
(viii) Is it true that everyone who has a grandparent in common with George has an ancestor in common with Mary?
(b) For the above program and each of the goals in part (a), show a normal program and normal goal which result from the transformation process.

3. Prove lemma 18.3.

4. Let P be a normal program and G a normal goal.
(a) Prove that θ is a computed answer for $P \cup \{G\}$ in the sense of §18 iff θ is a computed answer for $P \cup \{G\}$ in the sense of §15.
(b) Prove that $P \cup \{G\}$ has a finitely failed SLDNF-tree in the sense of §18 iff $P \cup \{G\}$ has a finitely failed SLDNF-tree in the sense of §15.

(c) What is the relationship between SLDNF-derivations, SLDNF-refutations, and SLDNF-trees in the sense of §18 and in the sense of §15?

5. Let P be a program and G a goal. Prove that if one normal form of $P \cup \{G\}$ is allowed, then every normal form of $P \cup \{G\}$ is allowed.

6. Let P be a program and P' and P" normal forms of P. Let U be a closed formula containing only predicate symbols which appear in P. Prove that U is a logical consequence of comp(P') iff U is a logical consequence of comp(P").

7. Give an example of a program P with a normal form P' such that P is not a logical consequence of P'.

8. Let P be a hierarchical program, G a goal and $P' \cup \{G'\}$ a normal form of $P \cup \{G\}$. Prove that P' is hierarchical.

9. Let P be a program and W a closed formula.
(a) Prove that $P \cup \{\leftarrow W\}$ has a finitely failed SLDNF-tree iff $P \cup \{\leftarrow \sim W\}$ has an SLDNF-refutation.
(b) Prove that $P \cup \{\leftarrow W\}$ has an SLDNF-refutation iff $P \cup \{\leftarrow \sim W\}$ has a finitely failed SLDNF-tree.
What happens if W is not closed?

10. Let P be a program, G_1 a goal $\leftarrow W_1$, and G_2 a goal $\leftarrow W_2$. Suppose that W_1 and W_2 are logically equivalent. Determine whether the following statements are correct or not:
(a) θ is a computed answer for $P \cup \{G_1\}$ iff θ is a computed answer for $P \cup \{G_2\}$.
(b) $P \cup \{G_1\}$ has a finitely failed SLDNF-tree iff $P \cup \{G_2\}$ has a finitely failed SLDNF-tree.

11. Let P be the program
$p(a) \leftarrow$
and G the goal $\leftarrow \forall x\, p(x)$. Show that, if the safeness condition is dropped, the identity substitution is a "computed answer", but that $\forall x\, p(x)$ is not a logical consequence of comp(P).

12. Let P be the program

```
p(a,a) ←
q(b,y) ←
r(a) ← ∀y(q(x,y)←p(x,y))
```

and G the goal ←r(a). Show that r(a) is a logical consequence of comp(P), but that, if the safeness condition is dropped, P ∪ {G} has a "finitely failed SLDNF-tree".

13. Consider the top-down version of the error diagnoser. Assume that a top level call to wrong has its first argument unsatisfiable and a top level call to missing has its first argument valid. Prove that the top-down version of the error diagnoser has the property that any subsequent call to wrong has its first argument unsatisfiable and any subsequent call to missing has its first argument valid.

14. Prove lemma 20.2.

15. Consider the following (incorrect) program for the Sieve of Eratosthenes.

```
primes(x,y) ← integers(2,x,z), sift(z,y)

integers(x,y,x.z) ← x≤y, plus(x,1,w), integers(w,y,z)
integers(x,y,nil) ← x>y

sift(nil,nil)
sift(x.u,x.y) ← remove(x,u,z), sift(z,y)

remove(x,nil,nil)
remove(x,y.u,z) ← ~(x div y), remove(x,u,z)
remove(x,y.u,y.z) ← x div y, remove(x,u,z)
```

The goal ←primes(10, x) returns the incorrect answer x/2.4.8.nil.

(a) Show the oracle queries which would be asked by the single-stepping diagnoser for the goal

← wrong(primes(10, 2.4.8.nil), x)

and hence determine an incorrect clause instance in the program.

(b) Repeat part (a) for the top-down diagnoser.

(c) Repeat part (a) for the divide-and-query diagnoser.

[Note that x div y is true if x divides y. Also plus(x,y,z) is true if x+y=z. You may assume the system predicates >, ≤, plus and div all work correctly. Thus oracle queries for these predicates can be avoided by simply calling them.]

16. Consider the following (incorrect) subset program

$\quad$subset(x,y) $\leftarrow$ $\forall$z (member(z,y) $\leftarrow$ member(z,x))

$\quad$member(x,y.z) $\leftarrow$ member(x,z)

and the goal $\leftarrow$subset(1.2.3.nil, 1.2.nil), which incorrectly succeeds. For the top-down diagnoser, show the computation and oracle queries that result from the goal

$\quad$$\leftarrow$wrong(subset(1.2.3.nil, 1.2.nil), x)

Hence calculate the incorrect statement instance or uncovered atom.

Chapter 5

DEDUCTIVE DATABASES

This chapter provides a theoretical basis for deductive database systems. A deductive database consists of a finite number of database statements, which have the form A←W, where A is an atom and W is a typed first order formula. A query has the form ←W, where W is a typed first order formula. An integrity constraint is a closed, typed first order formula. Function symbols are allowed to appear in formulas. Such a deductive database system can be implemented using a PROLOG system. The main results of this chapter are the soundness and completeness of the query evaluation process, the soundness of the implementation of integrity constraints, and a simplification theorem for implementing integrity constraints.

§21. INTRODUCTION TO DEDUCTIVE DATABASES

In this section, we introduce the important concepts of deductive database systems, such as database, query, correct answer, and integrity constraint. We also introduce several classes of databases, such as hierarchical and stratified databases.

In recent years, there has been a growing interest in deductive database systems [24], [35] to [38], [51], [58], [60] to [63], [70], [87], [105], [111]. Such systems have first order logic as their theoretical foundation. This approach has several desirable properties.

First, it provides an expressive environment for data modelling, since the use of database statements allows a single general statement to replace many explicit facts.

Second, it allows a *single* language to be used for expressing databases, queries, integrity constraints, views and programs. In particular, there is no need for separate query and host programming languages as are commonly used in relational database systems.

Third, logic itself has a well-understood and well-developed theory which already provides much of the theoretical foundation required for database systems.

Fourth, logic allows the *declarative* expression of databases, queries, integrity constraints and, especially, the key concept of a correct answer. The advantage to the user of only having to deal with declarative concepts is obvious.

Finally, and this is most important, the approach encourages a clear separation of the declarative and procedural concepts. For example, we can distinguish the declarative concept of a correct answer from the query evaluation process used to compute the answer. This contrasts with the standard relational database approach in which the declarative concept is commonly either ignored or identified with the implementation. The existence of a declarative definition provides an important yardstick against which the correctness of an implementation can be measured. Without it, we would not be able to even state the soundness and completeness theorems.

As the collection of papers in [70] shows, there is currently a great deal of research into the theoretical aspects of deductive database systems. There is even more interest in the implementation of deductive database systems, especially in the crucial area of query optimisation. Most efforts have been put into finding efficient ways of answering definite queries to (recursive) definite databases without functions. For a recent survey of the techniques for this problem found so far, the reader is referred to [7]. Unfortunately, little attention has so far been paid to optimising normal queries, much less arbitrary queries. However, given the great interest in the implementation problems, there is every chance that commercially competitive deductive database systems will become available in the next couple of years. Certainly, ten years from now, deductive database systems will be the standard database systems in the same way as relational database systems are standard now.

Underlying the theoretical developments of this chapter is a typed first order theory. (See §3 for a discussion of typed theories.) The reason for using a *typed* theory is that types provide a natural way of expressing the domain concept of relational databases. The requirement that formulas be correctly typed ensures that important kinds of semantic integrity constraints are maintained. In this chapter, we assume that the alphabet of the theory contains only finitely many constants, function symbols and predicate symbols. Also we assume that, for each type τ, there is a ground term of type τ.

Next we turn to the definitions of the main concepts. The particular formulation of these concepts presented in this chapter is due to Lloyd and Topor [61], [62], [63].

Definition A *database statement* is a typed first order formula of the form

$$A \leftarrow W$$

where A is an atom and W is a typed first order formula. The formula W may be absent. Any variables in A and any free variables in W are assumed to be universally quantified at the front of the statement. A is called the *head* and W the *body* of the statement.

Definition A *database* is a finite set of database statements.

Definition A *query* is a typed first order formula of the form

$$\leftarrow W$$

where W is a typed first order formula and any free variables of W are assumed to be universally quantified at the front of the query.

Example Consider a supplier-part-job database, whose predicate symbols have types associated with them as follows:

supplier has type sno × sname × city
local_supplier has type sno
major_supplier has type sno
part has type pno × pname × colour × weight
job has type jno × jname × city
spj has type sno × pno × jno × quantity

In a typical state, the database may contain the following statements:

supplier(S1, Smith, Adelaide) ←
supplier(S2, Jones, Sydney) ←
supplier(S3, James, Perth) ←
local_supplier(S1) ←

local_supplier(s) $\leftarrow$ supplier(s,_,Melbourne)

major_supplier(s) $\leftarrow$ $\forall$j/jno $\exists$q/quantity (spj(s,_,j,q) $\wedge$ q$\geq$100)

part(P1, Screw, White, 10) $\leftarrow$

part(P2, Nut, Black, 20) $\leftarrow$

job(J1, Build, Melbourne) $\leftarrow$

job(J2, Repair, Sydney) $\leftarrow$

spj(S1, P1, J1, 100) $\leftarrow$

spj(S2, P2, J3, 200) $\leftarrow$

In these database statements and in subsequent queries and integrity constraints, each underscore ("_") in an argument position represents a unique variable existentially quantified immediately before the atom containing it. Constants are denoted by names beginning with an upper case letter. Some possible queries that may be asked of this database are the following:

(1) Find suppliers who supply the same part to all jobs in Perth:

$\leftarrow$ $\exists$p/pno $\forall$j/jno (spj(s,p,j,_) $\leftarrow$ job(j,_,Perth))

(2) Find parts supplied by all suppliers who supply some red part:

$\leftarrow$ $\forall$s/sno (spj(s,p,_,_) $\leftarrow$ $\exists$p'/pno (spj(s,p',_,_)$\wedge$part(p',_,Red,_)))

(3) Find major suppliers such that if S1 supplies some part to some job then the major supplier supplies either the part or the job:

$\leftarrow$ major_supplier(s) $\wedge$ $\forall$p/pno $\forall$j/jno (spj(s,p,_,_) $\vee$ spj(s,_,j,_) $\leftarrow$ spj(S1,p,j,_))

Definition Let D be a database and Q a query $\leftarrow$W, where W has free variables $x_1,\ldots,x_n$. An *answer* for D $\cup$ {Q} is a substitution for some or all of the variables $x_1,\ldots,x_n$.

It is understood that substitutions are correctly typed in that each variable is bound to a term of the same type as the variable.

Definition An *integrity constraint* is a closed typed first order formula.

Example Some integrity constraints that may be imposed on the above database are the following:

(1) No local supplier supplies part P2:

$\forall$s/sno (~spj(s,P2,_,_) $\leftarrow$ local_supplier(s))

(2) Supplier S2 supplies every job in Sydney:

$\forall$j/jno (spj(S2,_,j,_) $\leftarrow$ job(j,_,Sydney))

(3) Supplier S3 only supplies jobs in Adelaide or Perth:

$\forall$j/jno (job(j,_,Adelaide) $\vee$ job(j,_,Perth) $\leftarrow$ spj(S3,_,j,_)) Next we give the definition of the completion of a database. This definition requires the introduction of a typed equality predicate symbol $=_\tau$ of type $\tau\times\tau$, for each type τ. These predicate symbols are assumed not to appear in the original language. In particular, no database, query or integrity constraint contains any $=_\tau$.

Definition The *definition* of a predicate symbol p appearing in a database D is the set of all database statements in D which have p in their head.

Definition Suppose the definition of a predicate symbol p of type $\tau_1\times\ldots\times\tau_n$ in a database is

$A_1 \leftarrow W_1$

$\cdot\ \cdot\ \cdot$

$A_k \leftarrow W_k$

Then the *completed definition* of p is the formula

$$\forall x_1/\tau_1\ldots\forall x_n/\tau_n\ (p(x_1,\ldots,x_n) \leftrightarrow E_1\vee\ldots\vee E_k)$$

where E_i is $\exists y_1/\sigma_1\ldots\exists y_d/\sigma_d\ ((x_1=_{\tau_1}t_1)\wedge\ldots\wedge(x_n=_{\tau_n}t_n)\wedge W_i)$, A_i is $p(t_1,\ldots,t_n)$, $y_1,\ldots,y_d$ are the variables in A_i and the free variables in W_i, and $x_1,\ldots,x_n$ are variables not appearing anywhere in the definition of p.

Example Let the definition of p be

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

$p(b) \leftarrow$

where x has type τ and y has type σ. Then the completed definition for p is

$$\forall z/\tau\ (p(z) \leftrightarrow (\exists x/\tau\ \exists y/\sigma\ ((z=_\tau x)\wedge q(x,y)) \vee (z=_\tau b)))$$

Definition Let D be a database and p a predicate symbol of type $\tau_1\times\ldots\times\tau_n$ occurring in D. Suppose there is no database statement in D with predicate symbol p in its head. Then the *completed definition* of p is the formula

$$\forall x_1/\tau_1\ldots\forall x_n/\tau_n\ \sim p(x_1,\ldots,x_n)$$

The *equality theory* for a database consists of all axioms of the following form:

1. $c\neq_\tau d$, where c and d are distinct constants of type τ.

2. $\forall(f(x_1,\ldots,x_n)\neq_\tau g(y_1,\ldots,y_m))$, where f and g are distinct function symbols of range type τ.

3. $\forall(f(x_1,\ldots,x_n)\neq_\tau c)$, where c is a constant of type τ and f is a function symbol of range type τ.

4. $\forall(t[x]\neq_\tau x)$, where t[x] is a term of type τ containing x and different from x.

5. $\forall((x_1\neq_{\tau_1} y_1) \vee \ldots \vee (x_n\neq_{\tau_n} y_n) \rightarrow f(x_1,\ldots,x_n)\neq_\tau f(y_1,\ldots,y_n))$, where f is a function symbol of type $\tau_1\times\ldots\times\tau_n\rightarrow\tau$.

6. $\forall x/\tau\ (x=_\tau x)$.

7. $\forall((x_1=_{\tau_1} y_1) \wedge \ldots \wedge (x_n=_{\tau_n} y_n) \rightarrow f(x_1,\ldots,x_n)=_\tau f(y_1,\ldots,y_n))$, where f is a function symbol of type $\tau_1\times\ldots\times\tau_n\rightarrow\tau$.

8. $\forall((x_1=_{\tau_1} y_1) \wedge \ldots \wedge (x_n=_{\tau_n} y_n) \rightarrow (p(x_1,\ldots,x_n) \rightarrow p(y_1,\ldots,y_n)))$, where p (including every $=_\tau$) is a predicate symbol of type $\tau_1\times\ldots\times\tau_n$.

9. $\forall x/\tau\ ((x=_\tau a_1) \vee \ldots \vee (x=_\tau a_k) \vee (\exists x_1/\tau_1\ldots\exists x_n/\tau_n(x=_\tau f_1(x_1,\ldots,x_n))) \vee$
$\ldots \vee (\exists y_1/\sigma_1\ldots\exists y_m/\sigma_m(x=_\tau f_r(y_1,\ldots,y_m))))$,
where $a_1,\ldots,a_k$ are all the constants of type τ and $f_1,\ldots,f_r$ are all the function symbols of range type τ.

Axioms 1 to 8 are the typed versions of the usual equality axioms for a program. (See §14.) The axioms 9 are the *domain closure axioms*, which were introduced in the function-free case by Reiter [85].

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

Definition Let D be a database, Q a query $\leftarrow$W, and θ an answer for $D \cup \{Q\}$. We say θ is a *correct answer* for comp(D) $\cup$ {Q} if $\forall(W\theta)$ is a logical consequence of comp(D).

The concept of a correct answer gives a declarative description of the desired output from a query to a database. Next we give the definition of a database satisfying or violating an integrity constraint.

Definition Let D be a database such that comp(D) is consistent and let W be an integrity constraint. We say D *satisfies* W if W is a logical consequence of comp(D); otherwise, we say D *violates* W.

This definition is due to Reiter [87]. Intuitively, an integrity constraint should be an invariant of the database.

There are two common views of databases, at least relational databases, which have been called the model-theoretic view and the proof-theoretic view [51], [79], [87].

In the model-theoretic view, a database is a model of its integrity constraints. Furthermore, an answer to a query should make the query true in the model given by the database. This view is essentially that provided by conventional relational database theory [25].

In the proof-theoretic view, the database is a first order theory and its integrity constraints should be an invariant of the theory. Furthermore, answering a query involves proving the query to be a logical consequence of the database. This chapter takes a proof-theoretic view of databases.

The proof-theoretic view has a number of advantages over the model-theoretic view, which are mainly concerned with the extension from relational databases to more general databases. For example, the model-theoretic view only works in a natural way for relational databases because the facts in the database can equally well be regarded as constituting an Herbrand interpretation. Once we move beyond having just ground facts in the database, there is no natural way of regarding the database as an interpretation any more. The other advantages are related to the fact that, if the database is regarded as a first order theory, then we have available more powerful data modelling capabilities for the treatment of incomplete information and null values, and the incorporation of more real world semantics. We refer the interested reader to [51] and [87] for a detailed discussion of these matters.

Next, we give the definitions of several important classes of queries and databases.

Definition A *normal query* is a query of the form $\leftarrow L_1 \wedge \ldots \wedge L_n$, where $L_1, \ldots, L_n$ are literals.

Definition A *definite query* is a query of the form $\leftarrow A_1 \wedge \ldots \wedge A_n$, where $A_1, \ldots, A_n$ are atoms.

Definition A *database clause* is a database statement that has the form $A \leftarrow L_1 \wedge ... \wedge L_n$, where $L_1,...,L_n$ are literals. A *normal database* is a database that consists of database clauses only.

Definition A *definite database clause* is a database clause that has the form $A \leftarrow A_1 \wedge ... \wedge A_n$, where $A_1,...,A_n$ are atoms. A *definite database* is a database that consists of definite database clauses only.

Definition A *level mapping* of a database 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 database is *hierarchical* if it has a level mapping such that, in every database statement $p(t_1,...,t_n) \leftarrow W$, the level of every predicate symbol in W is less than the level of p.

Definition A database is *stratified* if it has a level mapping such that, in every database statement $p(t_1,...,t_n) \leftarrow W$, the level of the predicate symbol of every atom occurring positively in W is less than or equal to the level of p, and the level of the predicate symbol of every atom occurring negatively in W is less than the level of p.

Clearly, every hierarchical database is stratified and also every definite database is stratified.

We can assume without loss of generality that the levels of a stratified database are 0,1,...,k, for some k, and we will normally assume this without comment in what follows. However, whenever we deal with stratified databases D and D' such that $D \subseteq D'$, it will be convenient to assume that D inherits the stratification induced by D'. This implies that for the smaller database D, there may not be predicate symbols of all levels 0,1,...,k. Note that, at level 0, all atoms in the bodies of database statements must occur positively, but that these database statements need not be definite database clauses.

Since every formula can be transformed into a logically equivalent formula in prenex conjunctive normal form (see proposition 3.4), we can transform the body of each statement in a database into this form. The transformed database is logically equivalent to the original one, and the completion of the transformed database is logically equivalent to the completion of the original one. Also the mapping T (defined below) associated with the transformed database is equal to the mapping associated with the original one. Furthermore, if W' is a prenex conjunctive normal form of W, then an atom occurs positively (resp., negatively) in W iff it occurs positively (resp., negatively) in W'. (See problem 1.) Thus the transformed database is stratified iff the original database is stratified. Also the transformed database is hierarchical iff the original database is hierarchical.

To simplify the proofs in this chapter, we assume without loss of generality that the body of each statement in a database is in prenex conjunctive normal form. In this case, it is easy to identify positive and negative occurrences of atoms. An atom occurring in the body of a statement occurs positively if it appears in a positive literal; otherwise, it occurs negatively.

We now define a mapping T_D^J from the lattice of interpretations based on J to itself.

Definition Let J be a pre-interpretation of a database D and I an interpretation based on J. Then $T_D^J(I) = \{ A_{J,V} : A \leftarrow W \in D$, V is a variable assignment wrt J, and W is true wrt I and V$\}$.

It will be convenient to suppress the J and denote this mapping by T_D. Let E be $\cup_\tau [=_\tau(x,x)]_J$. Subsequent use of E ensures that all models considered are normal, that is, assign an identity relation to each equality predicate.

The following propositions and corollary are the database versions of propositions 17.1 to 17.3 and corollary 17.4, and have the same proofs.

Proposition 21.1 Let D be a database, J a pre-interpretation of D, and I an interpretation based on J. Then I is a model for D iff $T_D(I) \subseteq I$.

Proposition 21.2 Let D be a database, J a pre-interpretation of D, and I an interpretation based on J. Suppose that $I \cup E$ is a model for the equality theory. Then $I \cup E$ is a model for comp(D) iff $T_D(I) = I$.

Proposition 21.3 Let D be a stratified database and J a pre-interpretation for D.

(a) Suppose D has only predicates of level 0. Then T_D is monotonic over the lattice of interpretations based on J.

(b) Suppose D has maximum predicate level k+1. Let D_k denote the set of database statements in D with the property that the predicate symbol in the head of the statement has level $\leq$ k. Suppose that M_k is an interpretation based on J for D_k and M_k is a fixpoint of T_{D_k}. Then $\Lambda = \{M_k \cup S : S \subseteq \{p(d_1,\ldots,d_n) : p$ 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_D, restricted to Λ, is well-defined and monotonic.

Corollary 21.4 Let D be a stratified database. Then comp(D) has a minimal normal Herbrand model.

The results of this section are due to Lloyd, Sonenberg and Topor [60].

§22. SOUNDNESS OF QUERY EVALUATION

In this section, we present the query evaluation process, and prove that it is sound and never flounders. These results are due to Lloyd and Topor [61], [62], [63]. The first step of the query evaluation process transforms typed first order formulas into corresponding type-free first order formulas. For this, we use a standard transformation [33].

Definition Let W be a typed first order formula. For each type τ, we associate a new unary *type predicate symbol* also denoted by τ. Then the *type-free form* W* of W is the first order formula obtained from W by applying the following transformations to all subformulas of W of the form $\forall x/\tau\, V$ and $\exists x/\tau\, V$:
(a) Replace $\forall x/\tau\, V$ by $\forall x(V \leftarrow \tau(x))$.
(b) Replace $\exists x/\tau\, V$ by $\exists x(V \wedge \tau(x))$.

Example Let W be the database statement
$p(x) \leftarrow \exists y/\sigma\ q(x,y)$
where x has type τ. Then W* is the program statement
$p(x) \leftarrow \exists y(q(x,y) \wedge \sigma(y)) \wedge \tau(x)$
If Q is the query
$\leftarrow \forall x/\tau\ q(x,y)$
then Q* is the goal
$\leftarrow \forall x(q(x,y) \leftarrow \tau(x)) \wedge \sigma(y)$
More generally, if Q is the query $\leftarrow$W, where W has free variables $x_1,\ldots,x_n$ and x_i has type τ_i (i=1,...,n), then Q* is the goal
$\leftarrow W^* \wedge \tau_1(x_1) \wedge \ldots \wedge \tau_n(x_n)$

We will also require the usual type theory [33].

Definition The *type theory* Φ consists of all axioms of the following form:
1. $\tau(a)\leftarrow$, where a is a constant of type τ.
2. $\forall x_1 \ldots \forall x_n\ (\tau(f(x_1,\ldots,x_n)) \leftarrow \tau_1(x_1) \wedge \ldots \wedge \tau_n(x_n))$, where f is a function symbol of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$.

Since we are allowing functions, a query can have infinitely many answers. However, under a reasonable restriction on the type theory Φ, we can ensure that each query can have at most finitely many answers. If Φ is hierarchical, then there are only finitely many ground terms of each type. (See problem 2.) Consequently, each query can have at most finitely many answers. We emphasise that it is not so much the presence of functions which causes queries to have infinitely many answers, but rather the presence of a ''recursive'' type theory.

Now we are in a position to give the definitions of the appropriate procedural concepts.

Definition Let D be a database, Φ its type theory, Q a query and R a safe computation rule. Let D* and Q* be the type-free forms of D and Q. (That is, D* is the set of type-free forms of each of its database statements.)

An *SLDNF-derivation* of $D \cup \{Q\}$ (*via R*) is an SLDNF-derivation of $D^* \cup \Phi \cup \{Q^*\}$ (via R).

An *SLDNF-refutation* of $D \cup \{Q\}$ (*via R*) is an SLDNF-refutation of $D^* \cup \Phi \cup \{Q^*\}$ (via R).

An *(R-)computed answer* for $D \cup \{Q\}$ is an (R-)computed answer for $D^* \cup \Phi \cup \{Q^*\}$.

An *SLDNF-tree* for $D \cup \{Q\}$ (*via R*) is an SLDNF-tree for $D^* \cup \Phi \cup \{Q^*\}$ (via R).

A *finitely failed SLDNF-tree* for $D \cup \{Q\}$ (*via R*) is a finitely failed SLDNF-tree for $D^* \cup \Phi \cup \{Q^*\}$ (via R).

Thus, to answer a query Q to a database D, we first transform D and Q to their type-free forms and then apply the techniques of §18 to the goal Q* and program $D^* \cup \Phi$. Note that, due to the presence of the type predicate symbols, every computed answer is a ground substitution for all the free variables in the body of the query. (See problem 3.) Also every computed answer is correctly typed. The next theorem shows that this implementation is sound.

Lemma 22.1 Let D be a database, Φ its type theory, and W a closed typed first order formula. Let D* and W* be the type-free forms of D and W. If W* is a logical consequence of comp(D* $\cup$ Φ), then W is a logical consequence of comp(D).

Proof The proof is rather long and requires some preparation. Given a model M for comp(D), we construct a model M* for comp(D* $\cup$ Φ). The complexity of the construction of M* which we use is needed to ensure that the equality axioms are satisfied.

Let M be a model for comp(D). Using (the typed version of) [69, p.83], we can assume without loss of generality that M is normal, that is, the identity relation on the domain C_τ is assigned to $=_\tau$, for each type τ. We can also assume that the C_τ's are disjoint. Put $C = \cup_\tau C_\tau$.

The underlying language L* for the interpretation M* includes all the constants, function symbols and (non-equality) predicate symbols of the underlying language L for M. L* differs from L in that all type information is suppressed, the various typed equality predicate symbols $=_\tau$ are replaced by a single equality predicate symbol = and there is a unary predicate symbol τ for each type τ.

Let F' be the set of mappings on the C_τ assigned by M to the function symbols in L. Let T be the set of all (free) terms that can be formed using elements of C as primitive terms and elements of F' as function symbols. (Note that the type restrictions are ignored in forming these terms.) The domain of M* will be the set of equivalence classes of a particular equivalence relation Δ on T.

To define Δ, we introduce a reduction operation on T. We write $f'(d_1,...,d_n) \rightarrow d$, if f has type $\tau_1 \times ... \times \tau_n \rightarrow \tau$, f' is the mapping assigned to f by M, $d_i \in C_{\tau_i}$, $d \in C_\tau$, and $f'(d_1,...,d_n)=d$. For $s,t \in T$, we write $s \Rightarrow t$ if t is the result of replacing some (not necessarily proper) subterm $f'(d_1,...,d_n)$ of s by d, where $f'(d_1,...,d_n) \rightarrow d$. We say that $s \in T$ is *irreducible* if there is no $t \in T$ such that $s \Rightarrow t$. Finally, for $s,t \in T$, we say that s *reduces to* t if there exist $r_0,r_1,...,r_n \in T$ such that $s=r_0 \Rightarrow r_1 \Rightarrow ... \Rightarrow r_n=t$.

Now we can define the equivalence relation Δ on T. Let $s,t \in T$. Then $s \Delta t$ if there exists $u \in T$ such that s reduces to u and t reduces to u. To prove that Δ is an equivalence relation, we use the following lemma.

Lemma 22.2 Let $s \in T$. Then there exists a unique irreducible $t \in T$ such that s reduces to t. (We say that t is the *irreducible form* of s.)

Proof of lemma 22.2 Clearly there exists an irreducible form of each $s \in T$, since, in each reduction $u \Rightarrow v$, v has fewer subterms than u.

To prove that irreducible forms are unique, first note that if $f'(s_1,...,s_n)$ reduces to $g'(t_1,...,t_m)$, then $f'=g'$, and that the last step in any reduction of $f'(s_1,...,s_n)$ to an element $d \in C$ therefore has the form $f'(d_1,...,d_n) \Rightarrow d$. We then use induction on the structure of s and a case analysis to show that if u and v are irreducible forms of s, then $u = v$. ■

Lemma 22.3 Δ is an equivalence relation.

Proof of lemma 22.3 Clearly, Δ is reflexive and symmetric. That Δ is transitive follows immediately from lemma 22.2. ■

We now define the domain of the model M* to be T/Δ, the set of Δ-equivalence classes in T. If $t \in T$, we let [t] denote the Δ-equivalence class containing t. Note that T/Δ contains a copy of C via the injective mapping $d \rightarrow [d]$. Thus, in essence, we have simply enlarged C in a particular way to obtain a domain for M*.

If c is a constant in L* and M assigns $c' \in C$ to c, then M* assigns [c'] in T/Δ to c. Let $f \in L^*$ be an n-ary function symbol. Suppose M assigns the mapping f' to f. Then M* assigns the mapping from $(T/\Delta)^n$ into T/Δ defined by $([t_1],...,[t_n]) \rightarrow [f'(t_1,...,t_n)]$ to f. It is easy to see that this mapping is well-defined. Note that this mapping is an extension of f'.

Suppose p is an n-ary predicate symbol in L*. If M assigns the relation p' to p, then M* assigns the relation $\{([d_1],...,[d_n]) : (d_1,...,d_n) \in p'\}$ on $(T/\Delta)^n$ to p. To a type predicate symbol τ, M* assigns the unary relation $\{[d] : d \in C_\tau\}$. Finally, M* assigns the identity relation on T/Δ to =.

This completes the definition of the interpretation M* for comp($D^* \cup \Phi$). We now check that M* is a model for comp($D^* \cup \Phi$). Much of the verification is routine and we take the liberty of omitting some details.

We first check that M* is a model for the equality theory of comp($D^* \cup \Phi$). The eight axioms of the equality theory are given in §14. Apart from axiom 4, these axioms are easily seen to be satisfied. Axiom 4 is

$\forall(t[x]\neq x)$, where t[x] is a term containing x and different from x.
That this axiom is satisfied follows immediately from the next lemma.

Lemma 22.4 Let $r,s\in T$. If r is a proper subterm of s, then $r\not\Delta s$.

Proof of lemma 22.4 Suppose $r\Delta s$. Then there exists an irreducible $t\in T$ such that r reduces to t and s reduces to t. Let $u\in T$ be the result of replacing the occurrence of r in s by t. Then t is a proper subterm of u and u reduces to t. If $t\in C$, then we obtain a contradiction using axiom 4 of the equality theory for D. Otherwise, t has the form $f'(t_1,...,t_n)$, in which case we again have a contradiction since it is impossible for u to reduce to t. ■

The remainder of the verification that M* is a model for comp(D* ∪ Φ) depends on another lemma. For this we need a definition. A *variable assignment* V *wrt* M is an assignment to each variable x in L of an element $d\in C_\tau$, where τ is the type of x. Corresponding to V, there is a variable assignment V* wrt M* which assigns [d] to x.

Lemma 22.5 Let W be a (not necessarily closed) typed first order formula, V a variable assignment wrt M, and V* the corresponding variable assignment wrt M*. Then W is true wrt M and V iff W* is true wrt M* and V*.

Proof of lemma 22.5 The proof is a straightforward induction argument on the structure of W. (See problem 5.) ■

Using lemma 22.5, it can now be checked that M* is a model for the remainder of comp(D* ∪ Φ). The domain closure axioms for comp(D) are used to show that M* is a model for the only-if halves of the completed definitions of the type predicate symbols.

We have now finally shown that M* is a model for comp(D* ∪ Φ). Since W* is a logical consequence of comp(D* ∪ Φ), we have that M* is a model for W*. Using lemma 22.5 again, we obtain that M is a model for W. Thus W is a logical consequence of comp(D). This completes the proof of lemma 22.1. ■

Theorem 22.6 (Soundness of Query Evaluation)

Let D be a database and Q a query. Then every computed answer for D ∪ {Q} is a correct answer for comp(D) ∪ {Q}.

Proof Let θ be a computed answer for $D \cup \{Q\}$, where Q is $\leftarrow W$, W has free variables $x_1,\ldots,x_n$ and x_i has type τ_i $(i=1,\ldots,n)$. By theorem 18.7, $(W^* \wedge \tau_1(x_1) \wedge \ldots \wedge \tau_n(x_n))\theta$ is a logical consequence of $comp(D^* \cup \Phi)$, where Φ is the type theory of D. Thus $(W\theta)^*$ is a logical consequence of $comp(D^* \cup \Phi)$. By lemma 22.1, $W\theta$ is a logical consequence of comp(D). That is, θ is a correct answer for $comp(D) \cup \{Q\}$. ■

As the following example shows, theorem 22.6 no longer holds if we omit the domain closure axioms from the definition of comp(D).

Example Let D be the database

$p(a) \leftarrow$

and Q be the query $\leftarrow \forall x/\tau\ p(x)$. Suppose that the type theory is just $\tau(a) \leftarrow$. Then the identity substitution is a computed answer, but $\forall x/\tau\ p(x)$ is not a logical consequence of comp(D) if the domain closure axiom $\forall x/\tau\ (x=a)$ is omitted from comp(D).

Theorem 22.6 is the fundamental result which guarantees the soundness of the query evaluation process. The implementation of the query evaluation process is, at least in principle, quite straightforward. The main part of the implementation concerns the 10 transformations given in §18. These can be implemented in a PROLOG program which contains one clause for each transformation plus a short procedure for locating free variables. Also, it is easy to avoid the explicit introduction of new predicate symbols which is formally required. A direct implementation of types would also be easy. However, such an implementation would be inefficient and hence some optimisations would be required.

Next we show that the query evaluation process never flounders. Let D be a database, Φ its type theory, and Q a query. By a *computation* of $D \cup \{Q\}$, we mean a computation of $D^* \cup \Phi \cup \{Q^*\}$.

Definition Let D be a database, Φ its type theory, and Q a query. We say a computation of $D \cup \{Q\}$ *flounders* if at some point in the computation a goal is reached which contains only non-ground negative literals.

Lemma 22.7 Let D be a database, Φ its type theory, and Q a query. Then $D^* \cup \Phi \cup \{Q^*\}$ is allowed.

Proof The form of the 10 transformations in §18 and the presence of the type predicate symbols ensures that every normal form of $D^* \cup \Phi \cup \{Q^*\}$ is allowed. (See problem 8.) ■

Note that not every clause in a normal form of D^* need be allowed.

Example Let D be

$p(x) \leftarrow \forall y/\sigma\ q(x,y)$

where x is of type τ. Then a normal form of D^* is

$p(x) \leftarrow \sim r(x) \wedge \tau(x)$

$r(x) \leftarrow \sim q(x,y) \wedge \sigma(y)$

where r is a new predicate symbol. The second clause is admissible, but not allowed.

Proposition 22.8 Let D be a database and Q a query. Then no computation of $D \cup \{Q\}$ flounders.

Proof The result follows immediately from lemma 22.7 and proposition 18.5(a). ■

§23. COMPLETENESS OF QUERY EVALUATION

In §22, we proved that every computed answer for $D \cup \{Q\}$ is a correct answer for $\text{comp}(D) \cup \{Q\}$. We would like to obtain the converse of this result. Unfortunately, there is no hope of this because there is no general completeness result even for normal programs. However, we can prove that query evaluation is complete for the special cases that the database is definite or hierarchical. These results are due to Lloyd and Topor [63]. We start by proving the converse of lemma 22.1.

Lemma 23.1 Let D be a database, Φ its type theory, and W a closed typed first order formula. Let D^* and W^* be the type-free forms of D and W. If W is a logical consequence of comp(D), then W^* is a logical consequence of $\text{comp}(D^* \cup \Phi)$.

Proof Let M^* be a normal model for $\text{comp}(D^* \cup \Phi)$. We construct a normal model M for comp(D). Suppose M^* has domain C. We define $C_\tau = \{c \in C : c \text{ is in the relation assigned to } \tau\}$. M assigns to a constant the same element of C as M^*

does. Note that a constant of type τ is thus assigned an element of C_τ, since M* satisfies Φ. If f is a function symbol of type $\tau_1 \times ... \times \tau_n \rightarrow \tau$ and M* assigns f' to f, then M assigns $f'|(C_{\tau_1} \times ... \times C_{\tau_n})$ to f. Note that the range of $f'|(C_{\tau_1} \times ... \times C_{\tau_n})$ is contained in C_τ, since M* satisfies Φ. Let p be a predicate symbol different from = and τ, for each type τ. If p is of type $\tau_1 \times ... \times \tau_n$ and M* assigns p to p', then M assigns $p' \cap (C_{\tau_1} \times ... \times C_{\tau_n})$ to p. Finally, M assigns the identity relation on C_τ to $=_\tau$, for each type τ.

We now show that M is a model for comp(D). It is easy to see that M is a model for the equality axioms. For the remainder of the proof, we require the following lemma, whose proof is a straightforward induction argument on the structure of W. (See problem 9.)

Lemma 23.2 Let W be a (not necessarily closed) typed first order formula, V a variable assignment wrt M, and V* the corresponding variable assignment wrt M*. Then W is true wrt M and V iff W* is true wrt M* and V*.

Using lemma 23.2, one can establish that M is indeed a model for comp(D). Hence M is a model for W and, using lemma 23.2 again, M* is a model for W*. Thus W* is a logical consequence of comp(D* $\cup$ Φ). This completes the proof of lemma 23.1. ■

Lemma 23.3 Let D be a database, Φ its type theory, and Q a query $\leftarrow$W, where $x_1,...,x_n$ are the free variables in W and x_i has type τ_i (i=1,...,n). Let θ be a correct answer for comp(D) $\cup$ {Q} that is a ground substitution for $x_1,...,x_n$. Then θ is a correct answer for comp(D* $\cup$ Φ) $\cup$ {Q*}.

Proof Since θ is a correct answer for comp(D) $\cup$ {Q} and since θ is a ground substitution for the free variables $x_1,...,x_n$ in W, it follows that $W\theta$ is a logical consequence of comp(D). By lemma 23.1, $W^*\theta$ is a logical consequence of comp(D* $\cup$ Φ). Hence $(W^* \wedge \tau_1(x_1) \wedge ... \wedge \tau_n(x_n))\theta$ is a logical consequence of comp(D* $\cup$ Φ). That is, θ is a correct answer for comp(D* $\cup$ Φ) $\cup$ {Q*}. ■

The next theorem is a database version of theorem 9.5.

Theorem 23.4 (Completeness of Query Evaluation for Definite Databases)

Let D be a definite database, Q a definite query $\leftarrow$W, and R a computation rule. Let θ be a correct answer for comp(D) $\cup$ {Q} that is a ground substitution for all variables in W. Then θ is an R-computed answer for D $\cup$ {Q}.

Proof Let D have type theory Φ. By lemma 23.3, θ is a correct answer for comp($D^* \cup \Phi$) $\cup$ {Q^*}. By theorem 14.6, θ is a correct answer for $D^* \cup \Phi \cup \{Q^*\}$. By theorem 9.5, there exists an R-computed answer σ for $D^* \cup \Phi \cup \{Q^*\}$ and a substitution γ such that $\theta = \sigma\gamma$. Since σ is a ground substitution for all the variables in W, it follows that $\theta = \sigma$. That is, θ is an R-computed answer for D $\cup$ {Q}. ∎

The requirement in theorem 23.4 that θ be a ground substitution for all variables in W cannot be omitted, since every computed answer for D $\cup$ {Q} has this property. From a database viewpoint, theorem 23.4 is a rather weak completeness result. It would be preferable to have conditions under which a query had only finitely many answers and the query evaluation process was guaranteed to find all these answers and then terminate. One rather strong condition, which ensures these properties hold, is that the database be hierarchical. We now present this completeness result for hierarchical databases, which is the database version of theorem 18.9.

Theorem 23.5 (Completeness of Query Evaluation for Hierarchical Databases)
Let D be a database, Φ its type theory, Q a query $\leftarrow$W, and R a safe computation rule. Suppose that both D and Φ are hierarchical. Then the following properties hold.
(a) Each SLDNF-tree for D $\cup$ {Q} via R exists and is finite.
(b) If θ is a correct answer for comp(D) $\cup$ {Q} and θ is a ground substitution for all free variables in W, then θ is an R-computed answer for D $\cup$ {Q}.

Proof By lemma 22.7, $D^* \cup \Phi \cup \{Q^*\}$ is allowed. Also $D^* \cup \Phi$ is hierarchical. By lemma 23.3, θ is a correct answer for comp($D^* \cup \Phi$) $\cup$ {Q^*}. Hence the result follows from theorem 18.9. ∎

§24. INTEGRITY CONSTRAINTS

In this section, we study integrity constraints in deductive database systems and prove the correctness of a simplification method for checking integrity constraints.

A number of proofs in this section use typed versions of results from earlier chapters. In each case, it will be clear from the context that the reference to the earlier result is actually a reference to the appropriate typed version of the result.

The standard method of determining whether a database satisfies or violates an integrity constraint W is by evaluating the query ←W. The following two theorems, due to Lloyd and Topor [61], [62], show that this method is sound.

Theorem 24.1 Let D be a database and W an integrity constraint. Suppose that comp(D) is consistent. If there exists an SLDNF-refutation of D ∪ {←W}, then D satisfies W.

Proof The theorem follows immediately from theorem 22.6. ■

Theorem 24.2 Let D be a database and W an integrity constraint. Suppose that comp(D) is consistent. If D ∪ {←W} has a finitely failed SLDNF-tree, then D violates W.

Proof The theorem follows easily from theorem 18.6 and lemma 22.1. ■

Now we turn to the simplification theorem for integrity constraint checking. From a theoretical viewpoint, it is highly desirable for a database to satisfy its integrity constraints at all times. However, from a practical viewpoint, there are serious difficulties in finding efficient ways of checking the integrity constraints after each update. The problem is especially difficult for deductive databases, since the addition of a single fact can have a substantial impact on the logical consequences of the database because of the presence of rules.

In spite of these difficulties, it is possible to reduce the amount of computation if advantage is taken of the fact that, before the update was made, the database was known to satisfy its integrity constraints. The simplification theorem shows that it is only necessary to check certain *instances* of each integrity constraint. For a very large database, this can lead to a dramatic reduction in the amount of computation required. This idea is originally due to Nicolas [78] in the context of relational database systems. A method related to the one given in this chapter was presented by Decker [27]. An alternative "theorem proving" approach was given by Sadri and Kowalski [90].

To cover the most general situation by a single theorem, we use the concept of a transaction. A *transaction* is a finite sequence of additions of statements to a database and deletions of statements from a database. If D is a database and t is a transaction, then the application of t to D produces a new database D', which is obtained by applying each of the deletions and additions in t in turn. We assume that, in any transaction, we do not have the addition and deletion of the same statement. As the deletions and additions in a transaction can then be performed in any order, we assume that all the deletions are performed before the additions. With respect to integrity constraint checking, we regard a transaction as indivisible, so we need only check the constraints at the end of the transaction. Note that we can use a single transaction to pass from any database D to any other database D'.

Suppose L is the typed language underlying the database D. We make the assumption throughout that, whatever changes D may undergo, L remains fixed. Thus, for example, adding a new statement to D does not introduce new constants into the language.

Implementing the simplification method involves computing four sets of atoms, computing two sets of substitutions by unifying atoms in the sets with atoms in an integrity constraint, and evaluating corresponding instances of the integrity constraint. We begin with the definitions of the appropriate sets of atoms.

Definition Let D and D' be databases such that $D \subseteq D'$. We define the sets $pos_{D,D'}$ and $neg_{D,D'}$ inductively as follows:

$$pos^0_{D,D'} = \{ A : A \leftarrow W \in D' \setminus D \}$$

$$neg^0_{D,D'} = \{ \}$$

$$\begin{aligned} pos^{n+1}_{D,D'} = \{ & A\theta : A \leftarrow W \in D, \text{ B occurs positively in W, } C \in pos^n_{D,D'}, \\ & \text{and } \theta \text{ is an mgu of B and C} \} \\ \cup \{ & A\theta : A \leftarrow W \in D, \text{ B occurs negatively in W, } C \in neg^n_{D,D'}, \\ & \text{and } \theta \text{ is an mgu of B and C} \} \end{aligned}$$

$$\begin{aligned} neg^{n+1}_{D,D'} = \{ & A\theta : A \leftarrow W \in D, \text{ B occurs positively in W, } C \in neg^n_{D,D'}, \\ & \text{and } \theta \text{ is an mgu of B and C} \} \\ \cup \{ & A\theta : A \leftarrow W \in D, \text{ B occurs negatively in W, } C \in pos^n_{D,D'}, \\ & \text{and } \theta \text{ is an mgu of B and C} \} \end{aligned}$$

$$pos_{D,D'} = \cup_{n \geq 0} pos^n_{D,D'}$$

$$neg_{D,D'} = \cup_{n \geq 0} neg^n_{D,D'}$$

To motivate the above definitions, consider the case when we add a fact $A\leftarrow$ to a database D to obtain a database D'. An important task of the simplification method is to capture the difference between a model for comp(D') and a model for comp(D). In the case that D is a relational database, we see that $pos_{D,D'}$ is $\{A\}$, which is precisely the difference between a model for comp(D) and a model for comp(D'). (In this case, the models are essentially unique.) For a deductive database, the presence of rules means that the difference between the models could be larger. However, as we shall see, for stratified databases, $pos_{D,D'}$ and $neg_{D,D'}$ can still be used to capture the differences between (suitably related) models of comp(D) and comp(D'). Intuitively, $pos_{D,D'}$ captures the part that is added to the model for comp(D) when passing from D to D' and $neg_{D,D'}$ captures the part that is lost. (See lemma 24.4 below.) In the context of normal databases, $pos_{D,D'}$ and $neg_{D,D'}$ have been discussed by Topor et al [105].

Definition Let D and D' be databases such that $D \subseteq D'$ and J a pre-interpretation of D. We define

$$posinst_{D,D',J} = \cup_{A\in pos_{D,D'}}[A]_J$$

$$neginst_{D,D',J} = \cup_{A\in neg_{D,D'}}[A]_J.$$

Lemma 24.3 Let D and D' be databases such that $D \subseteq D'$. Let J be a pre-interpretation of D and V be a variable assignment wrt J. Suppose there exists an interpretation I based on J such that $I \cup E$ is a model for the equality theory.
(a) If $A\leftarrow W$ is in D, B occurs positively in W, and $B_{J,V} \in neginst_{D,D',J}$, then $A_{J,V} \in neginst_{D,D',J}$.
(b) If $A\leftarrow W$ is in D, B occurs positively in W, and $B_{J,V} \in posinst_{D,D',J}$, then $A_{J,V} \in posinst_{D,D',J}$.
(c) If $A\leftarrow W$ is in D, B occurs negatively in W, and $B_{J,V} \in posinst_{D,D',J}$, then $A_{J,V} \in neginst_{D,D',J}$.
(d) If $A\leftarrow W$ is in D, B occurs negatively in W, and $B_{J,V} \in neginst_{D,D',J}$, then $A_{J,V} \in posinst_{D,D',J}$.

Proof (a) Recall that $B_{J,V}$ denotes the J-instance of atom B wrt V. Since $B_{J,V} \in neginst_{D,D',J}$, we have that $B_{J,V}$ is also a J-instance of some $C \in neg_{D,D'}$. By lemma 15.2 (a), B and C are unifiable with mgu $\theta = \{x_1/r_1,\dots,x_m/r_m\}$, say. Since $C \in neg_{D,D'}$ and $B\theta = C\theta$, we have that $A\theta \in neg_{D,D'}$. By lemma 15.2 (b), the variable assignment, which we can suppose without loss of generality to be V, that maps B and C to $B_{J,V}$ also maps

x_j and r_j to the same domain element, for each j. Hence $A_{J,V}$ is also a J-instance of $A\theta$ and so $A_{J,V} \in neginst_{D,D',J}$.

The proofs of the other parts are similar. ■

Lemma 24.4 Let D and D' be stratified databases such that $D \subseteq D'$ and let J be a pre-interpretation of D.

(a) Let M' be an interpretation based on J for D' such that $M' \cup E$ is a model for comp(D'). Then there exists an interpretation M based on J such that $M \cup E$ is a model for comp(D), $M' \setminus M \subseteq posinst_{D,D',J}$, and $M \setminus M' \subseteq neginst_{D,D',J}$.

(b) Let M be an interpretation based on J for D such that $M \cup E$ is a model for comp(D). Then there exists an interpretation M' based on J such that $M' \cup E$ is a model for comp(D'), $M' \setminus M \subseteq posinst_{D,D',J}$, and $M \setminus M' \subseteq neginst_{D,D',J}$.

Proof (a) The proof is by induction on the maximum level, k, of D'.

Base step, k=0.

By proposition 21.2, M' is a fixpoint of $T_{D'}$ and hence $T_D(M') \subseteq M'$. By proposition 21.3(a), T_D is monotonic and so $T_D^{\alpha}(M')$ is defined, for every ordinal α. (See problem 13 of chapter 1.) We prove by transfinite induction that $M' \setminus T_D^{\alpha}(M') \subseteq posinst_{D,D',J}$, for every ordinal α.

α is a limit ordinal.

The case $\alpha = 0$ is trivial. Otherwise, $M' \setminus T_D^{\alpha}(M') = M' \setminus \cap_{\beta<\alpha} T_D^{\beta}(M') = \cup_{\beta<\alpha}(M' \setminus T_D^{\beta}(M')) \subseteq posinst_{D,D',J}$, by the induction hypothesis.

α is a successor ordinal.

The case $\alpha = 1$ is immediate from the definition of $posinst_{D,D',J}$. Otherwise, note that $M' \setminus T_D^{\alpha}(M') = (M' \setminus T_D(M')) \cup (T_D(M') \setminus T_D^{\alpha}(M'))$. Suppose that $B \in T_D(M') \setminus T_D^{\alpha}(M')$. Then one can prove that there exists a statement $A \leftarrow W$ in D such that, for some variable assignment V wrt J and for some atom C in W, B is $A_{J,V}$ and $C_{J,V} \in M' \setminus T_D^{\alpha-1}(M')$. Thus, by the induction hypothesis, $C_{J,V} \in posinst_{D,D',J}$. By lemma 24.3, we have that $B \in posinst_{D,D',J}$. This completes the proof that $M' \setminus T_D^{\alpha}(M') \subseteq posinst_{D,D',J}$, for every ordinal α.

Since T_D is monotonic, there exists an ordinal γ such that $T_D^{\gamma}(M')$ is a fixpoint of T_D. (See problem 13 of chapter 1.) Put $M = T_D^{\gamma}(M')$. By proposition 21.2, $M \cup E$ is a model for comp(D). Finally, note that $M \setminus M' = \emptyset = neginst_{D,D',J}$.

Induction step.

Suppose the result holds for stratified databases of maximum level k and D' has maximum level k+1. Let D'_k (resp., D_k) be the set of database statements in D' (resp., D) with the property that the predicate symbol in the head of the statement has level $\leq$ k. Let M'_k be the set of all $p(d_1,...,d_n)$ in M' such that p has level $\leq$ k. Then $M'_k \cup E$ is a model for $comp(D'_k)$. By the induction hypothesis, there exists an interpretation M_k based on J such that $M_k \cup E$ is a model for $comp(D_k)$, $M'_k \setminus M_k \subseteq posinst_{D_k,D'_k,J}$, and $M_k \setminus M'_k \subseteq neginst_{D_k,D'_k,J}$.

Put $N = M_k \cup (M' \setminus M'_k) \cup neginst_{D,D',J}|(k+1)$, where $neginst_{D,D',J}|(k+1)$ is the set of all $p(d_1,...,d_n)$ in $neginst_{D,D',J}$ such that p has level k+1. Then one can prove that $T_D(N) \subseteq N$, using the fact that M_k is a fixpoint of T_{D_k}, the definition of $neginst_{D,D',J}$, lemma 24.3, and the induction hypothesis.

We now consider transfinite iterations of T_D on N in the lattice Λ defined in proposition 21.3(b). We claim the following properties hold:

(i) $T_D^{\alpha}(N) \setminus M' \subseteq neginst_{D,D',J}$, for every ordinal α.

(ii) $M' \setminus T_D^{\alpha}(N) \subseteq posinst_{D,D',J}$, for every ordinal α.

For (i), note that, for all α, we have

$$T_D^{\alpha}(N) \setminus M' \subseteq N \setminus M' \subseteq (M_k \setminus M'_k) \cup neginst_{D,D',J}|(k+1) \subseteq neginst_{D,D',J},$$

using the induction hypothesis on $M_k \setminus M'_k$, and the definition of $neginst_{D,D',J}$.

We prove (ii) by transfinite induction.

α is a limit ordinal.

Suppose α=0. Then we have

$$M' \setminus N \subseteq M'_k \setminus M_k \subseteq posinst_{D_k,D'_k,J} \subseteq posinst_{D,D',J}.$$

Now suppose α>0. Then we have

$$M' \setminus T_D^{\alpha}(N) = M' \setminus \cap_{\beta<\alpha} T_D^{\beta}(N) = \cup_{\beta<\alpha}(M' \setminus T_D^{\beta}(N)) \subseteq posinst_{D,D',J}.$$

α is a successor ordinal.

Suppose that $B \in M' \setminus T_D^{\alpha}(N)$. Then, as M' is a fixpoint of $T_{D'}$, there exists a statement $A \leftarrow W$ in D' such that, for some variable assignment V wrt J, B is $A_{J,V}$ and W is true wrt M' and V. If the statement is in $D' \setminus D$, then $A \in pos^0_{D,D'}$ and so $B \in posinst_{D,D',J}$ immediately. Now suppose that the statement is in D. Since $B \notin T_D^{\alpha}(N)$, one can prove that there exists a variable assignment V* and an atom C in W such that $A_{J,V} = A_{J,V*}$ and either C occurs positively in W and $C_{J,V*} \in M' \setminus T_D^{\alpha-1}(N)$ or C occurs negatively in W and $C_{J,V*} \in T_D^{\alpha-1}(N) \setminus M'$.

In the first case, by the induction hypothesis, $C_{J,V*} \in posinst_{D,D',J}$. By lemma 24.3, we have that $B \in posinst_{D,D',J}$. In the second case, by (i), $C_{J,V*} \in neginst_{D,D',J}$. By lemma 24.3, we have that $B \in posinst_{D,D',J}$. This completes the proof of (ii).

By proposition 21.3(b) and problem 13 of chapter 1, there exists an ordinal γ

such that $T^{\gamma}_{D}(N)$ is a fixpoint of T_D restricted to Λ. Put $M = T^{\gamma}_{D}(N)$. Since M is a fixpoint of T_D, by proposition 21.2, we have that $M \cup E$ is a model for comp(D). This completes the proof of part (a).

(b) The proof is similar to part (a). We use a construction based on the set $N' = M'_k \cup [(M \setminus M_k) \setminus neginst_{D,D',J}|(k+1)]$, for which it can be shown that $T_{D'}(N') \supseteq N'$. (See problem 12.) ∎

Now we are in a position to state and prove the simplification theorem. This theorem is due to Lloyd, Sonenberg and Topor [60], [62].

Theorem 24.5 (Simplification Theorem for Integrity Constraint Checking)

Let D and D' be stratified databases and t a transaction whose application to D produces D'. Suppose t consists of a sequence of deletions followed by a sequence of additions and that the application of the sequence of deletions to D produces the intermediate database D". Let W be an integrity constraint $\forall x_1 ... \forall x_n W'$ in prenex conjunctive normal form. Suppose D satisfies W. Let $\Theta = \{ \theta : \theta$ is the restriction to $x_1,...,x_n$ of either an mgu of an atom occurring negatively in W and an atom in $pos_{D'',D'}$ or an mgu of an atom occurring positively in W and an atom in $neg_{D'',D'}$ } and $\Psi = \{ \psi : \psi$ is the restriction to $x_1,...,x_n$ of either an mgu of an atom occurring positively in W and an atom in $pos_{D'',D}$ or an mgu of an atom occurring negatively in W and an atom in $neg_{D'',D}$ }. Then the following properties hold.

(a) D' satisfies W iff D' satisfies $\forall(W'\phi)$ for all $\phi \in \Theta \cup \Psi$.

(b) If $D' \cup \{\leftarrow \forall(W'\phi)\}$ has an SLDNF-refutation for all $\phi \in \Theta \cup \Psi$, then D' satisfies W.

(c) If $D' \cup \{\leftarrow \forall(W'\phi)\}$ has a finitely failed SLDNF-tree for some $\phi \in \Theta \cup \Psi$, then D' violates W.

Proof (a) Suppose D' satisfies $\forall(W'\phi)$, for all $\phi \in \Theta \cup \Psi$. Note that the formula W' is not necessarily quantifier free. Let M' be an interpretation for D' based on J such that $M' \cup E$ is a model for comp(D'). By lemma 24.4(a), there exists an interpretation M" based on J such that $M'' \cup E$ is a model for comp(D"), $M' \setminus M'' \subseteq posinst_{D'',D',J}$, and $M'' \setminus M' \subseteq neginst_{D'',D',J}$. Similarly, by lemma 24.4(b), there exists an interpretation M based on J such that $M \cup E$ is a model for comp(D), $M \setminus M'' \subseteq posinst_{D'',D,J}$, and $M'' \setminus M \subseteq neginst_{D'',D,J}$.

By supposition, W is true wrt $M \cup E$. Let V be a variable assignment wrt J. We have to prove that W' is true wrt $M' \cup E$ and V. If V* is a variable assignment that agrees with V on $x_1,...,x_n$, then we say V* is *compatible* with V.

We consider the following two cases.

Case 1: For every atom A occurring negatively in W and for every V* compatible with V, the J-instance $A_{J,V*}$ of A wrt V* is not in M' \ M, and for every atom B occurring positively in W and for every V* compatible with V, the J-instance $B_{J,V*}$ of B wrt V* is not in M \ M'.

Let A be an atom occurring negatively in W and suppose that, for some V* compatible with V, we have that $A_{J,V*} \notin M$. By the condition of case 1, we have that $A_{J,V*} \notin M' \setminus M$. Hence $A_{J,V*} \notin M'$.

Let B be an atom occurring positively in W and suppose that, for some V* compatible with V, we have that $B_{J,V*} \in M$. By the condition of case 1, we have that $B_{J,V*} \notin M \setminus M'$. Hence $B_{J,V*} \in M'$.

It follows from this that W' is true wrt M' ∪ E and V.

Case 2: Either (a) there exists an atom A occurring negatively in W and a V* compatible with V such that the J-instance $A_{J,V*}$ of A wrt V* is in M' \ M or (b) there exists an atom B occurring positively in W and a V* compatible with V such that the J-instance $B_{J,V*}$ of B wrt V* is in M \ M'.

Case 2(a): Then $A_{J,V*} \in (M' \setminus M'') \cup (M'' \setminus M)$ and, hence, either $A_{J,V*} \in posinst_{D'',D',J}$ or $A_{J,V*} \in neginst_{D'',D,J}$. In the first case, $A_{J,V*}$ is also a J-instance of an atom $F \in pos_{D'',D'}$. By lemma 15.2 (a), A and F are unifiable with mgu θ', say. Let θ be the restriction of θ' to $x_1,...,x_n$. By supposition, ∀(W'θ) is true wrt M' ∪ E. It then follows from lemma 15.2 (b) that W' is true wrt M' ∪ E and V. Similarly, in the second case, using Ψ, we obtain that W' is true wrt M' ∪ E and V.

Case 2(b): Then $B_{J,V*} \in (M \setminus M'') \cup (M'' \setminus M')$ and, hence, either $B_{J,V*} \in posinst_{D'',D,J}$ or $B_{J,V*} \in neginst_{D'',D',J}$. In the first case, $B_{J,V*}$ is also a J-instance of an atom $G \in pos_{D'',D}$. By lemma 15.2 (a), B and G are unifiable with mgu ψ', say. Let ψ be the restriction of ψ' to $x_1,...,x_n$. By supposition, ∀(W'ψ) is true wrt M' ∪ E. It then follows from lemma 15.2 (b) that W' is true wrt M' ∪ E and V. Similarly, in the second case, using Θ, we obtain that W' is true wrt M' ∪ E and V.

(b) This part follows immediately from theorem 22.6 and part (a).

(c) Suppose D' ∪ {←∀(W'φ)} has a finitely failed SLDNF-tree, for some φ ∈ Θ ∪ Ψ. By theorem 18.6 and lemma 22.1 ~∀(W'φ) is a logical consequence of comp(D'). By the consistency of comp(D'), W is not a logical consequence of comp(D') and so D' violates W. ■

The theorem has an immediate corollary for the case when the transaction consists of a single addition.

Corollary 24.6 Let D be a stratified database, C a database statement, and $D' = D \cup \{C\}$ a stratified database. Let W be an integrity constraint $\forall x_1 ... \forall x_n W'$ in prenex conjunctive normal form. Suppose D satisfies W. Let $\Theta = \{ \theta : \theta$ is the restriction to $x_1,...,x_n$ of either an mgu of an atom occurring negatively in W and an atom in $pos_{D,D'}$ or an mgu of an atom occurring positively in W and an atom in $neg_{D,D'}$ $\}$. Then the following properties hold.
(a) D' satisfies W iff D' satisfies $\forall(W'\theta)$ for all $\theta \in \Theta$.
(b) If $D' \cup \{\leftarrow \forall(W'\theta)\}$ has an SLDNF-refutation for all $\theta \in \Theta$, then D' satisfies W.
(c) If $D' \cup \{\leftarrow \forall(W'\theta)\}$ has a finitely failed SLDNF-tree for some $\theta \in \Theta$, then D' violates W.

Similarly, the theorem has a corollary for the case when the transaction consists of a single deletion.

Corollary 24.7 Let D be a stratified database, C a database statement in D, and $D' = D \setminus \{C\}$ a stratified database. Let W be an integrity constraint $\forall x_1 ... \forall x_n W'$ in prenex conjunctive normal form. Suppose D satisfies W. Let $\Psi = \{ \psi : \psi$ is the restriction to $x_1,...,x_n$ of either an mgu of an atom occurring positively in W and an atom in $pos_{D',D}$ or an mgu of an atom occurring negatively in W and an atom in $neg_{D',D}$ $\}$. Then the following properties hold.
(a) D' satisfies W iff D' satisfies $\forall(W'\psi)$ for all $\psi \in \Psi$.
(b) If $D' \cup \{\leftarrow \forall(W'\psi)\}$ has an SLDNF-refutation for all $\psi \in \Psi$, then D' satisfies W.
(c) If $D' \cup \{\leftarrow \forall(W'\psi)\}$ has a finitely failed SLDNF-tree for some $\psi \in \Psi$, then D' violates W.

Next we briefly discuss some implementation issues related to the simplification theorem. The theorem shows that the implementation of the simplification method involves calculating four atom sets $pos_{D'',D'}$, $neg_{D'',D'}$, $pos_{D'',D}$, and $neg_{D'',D}$, computing Θ and Ψ, and then evaluating each query $\leftarrow \forall(W'\phi)$, where $\phi \in \Theta \cup \Psi$. Note that the method is independent of the level mappings used to show that the databases are stratified.

Some special cases of the theorem are of interest. If $\Theta \cup \Psi$ is empty, then the corresponding integrity constraint W can be eliminated from further consideration, since the theorem shows that D' satisfies W. If $\Theta \cup \Psi$ contains the identity substitution, then no simplification of W is possible. Nicolas [78] also studied various refinements of the basic idea which could lead to optimisations of the implementation. We do not discuss these optimisations here except to note that all of them are equally applicable to stratified databases.

The key to an efficient implementation of the simplification theorem is to find an efficient way to calculate $pos_{D,D'}$ and $neg_{D,D'}$, for $D \subseteq D'$. We emphasise that this calculation only involves the rules and not the facts in D. This is an important point because, even for a large deductive database, the number of rules is likely to be very much smaller than the number of facts. In particular, the rules are likely to be kept in main memory, so that access to the disk during the calculation of these sets is obviated.

We now briefly consider some aspects of the computation of the atom sets. In principle, this computation involves the calculation of infinitely many sets $pos^n_{D,D'}$ and $neg^n_{D,D'}$, for $n \geq 0$. However, in practice, we can often use a stopping rule to terminate the computation after only finitely many steps. Application of one such stopping rule involves computing sets of atoms P^n and N^n rather than the sets $pos^n_{D,D'}$ and $neg^n_{D,D'}$. P^n and N^n are defined and used in much the same way as $pos^n_{D,D'}$ and $neg^n_{D,D'}$, except for the following additional (simplifying) step. We omit any atom from P^n (resp., N^n) which is an instance of another atom in P^k (resp., N^k), for $0 \leq k \leq n$.

The *stopping rule* is then as follows. If after deletions in this manner, some P^n and N^n both become empty, then terminate the computation and use the unions, P and N, of the respective sets of atoms computed thus far in place of $pos_{D,D'}$ and $neg_{D,D'}$. The proof of the simplification theorem is valid for the sets P and N used in place of $pos_{D,D'}$ and $neg_{D,D'}$. A further refinement is to delete from P (resp., N) any atom which is an instance of another atom in P (resp., N). The example below illustrates the application of this stopping rule.

Example Let D be the database

no_male_descendant(x) $\leftarrow \forall y$ (female(y) $\leftarrow$ ancestor(x,y))

ancestor(x,y) $\leftarrow$ parent(x,z) $\wedge$ ancestor(z,y)

ancestor(x,y) $\leftarrow$ parent(x,y)

parent(x,y) ← mother(x,y)

parent(x,y) ← father(x,y)

together with facts for the predicate symbols mother, father, male and female. If we give no_male_descendant level 1 and all other predicate symbols level 0, then we see that D is a stratified database. Let C be the clause

mother(Mary, Bill) ←

and let D' = D ∪ {C}. Then we obtain

$pos^0_{D,D'}$ = {mother(Mary, Bill)} = P^0

$neg^0_{D,D'}$ = {} = N^0

$pos^1_{D,D'}$ = {parent(Mary, Bill)} = P^1

$neg^1_{D,D'}$ = {} = N^1

$pos^2_{D,D'}$ = {ancestor(Mary, Bill), ancestor(Mary, y)}

P^2 = {ancestor(Mary, y)}

$neg^2_{D,D'}$ = {} = N^2

$pos^3_{D,D'}$ = {ancestor(x, Bill), ancestor(x, y)}

P^3 = {ancestor(x, y)}

$neg^3_{D,D'}$ = {no_male_descendant(Mary)} = N^3

$pos^4_{D,D'}$ = {ancestor(x, Bill), ancestor(x, y)}

P^4 = {}

$neg^4_{D,D'}$ = {no_male_descendant(x)} = N^4

P^5 = {}

N^5 = {}

At this point, we can apply the stopping rule. Thus, when applying the simplification theorem, in place of $pos_{D,D'}$, we can use the set P = {mother(Mary, Bill), parent(Mary, Bill), ancestor(x, y)} and, in place of $neg_{D,D'}$, we can use the set N = {no_male_descendant(x)}.

Another possibility in the computation of $pos_{D,D'}$ and $neg_{D,D'}$ is that one or both of them may contain infinitely many ''independent'' atoms, in which case the simplification method may require checking infinitely many instances of an integrity constraint. For example, let D be the database

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

C the clause $p(a,b) \leftarrow$, and $D' = D \cup \{C\}$. Then $pos_{D,D'}$ is the infinite set $\{p(a,b), p(f(a),b), p(f(f(a)),b), \ldots\}$. In this case, the previous stopping rule is not applicable. However, we can add the instance, $p(f(x),b)$, of the head of the offending clause in D to $pos^1_{D,D'}$ instead of $p(f(a),b)$. If we do this, we can use $\{p(a,b), p(f(x),b)\}$ in place of $pos_{D,D'}$. This example suggests the existence of another stopping rule, which replaces an infinite set of atoms by a single more general instance of a statement head.

The simplification method appears to be an essential ingredient of any efficient method of checking integrity constraints. The main issues which require further research are finding more powerful stopping rules and investigating the various techniques which will be required for a really practical implementation.

PROBLEMS FOR CHAPTER 5

1. Let W be a formula and W' a prenex conjunctive normal form of W obtained by applying the transformations of problem 5 of chapter 1. Prove that an atom occurs positively (resp., negatively) in W iff it occurs positively (resp., negatively) in W'.

2. Let Φ be a hierarchical type theory. Prove that there are only finitely many ground terms of each type.

3. Let D be a database and Q a query $\leftarrow W$. Prove that every computed answer for $D \cup \{Q\}$ is a ground substitution for all the free variables in W.

4. Give an example to show that lemma 22.1 no longer holds if we omit the domain closure axioms from the equality theory.

5. Prove lemma 22.5

6. (a) Consider the database D

   $p(a) \leftarrow$

   $q(a) \leftarrow$

   $q(b) \leftarrow$

   $r(a) \leftarrow$

and the query Q

$\leftarrow \forall x/\tau\ (q(x) \leftarrow p(x)) \wedge \sim r(y)$

where p, q and r have type τ and the constants of type τ are a and b. Show the result of transforming D and Q into a normal program and goal, which is required by the query evaluation process. Hence compute the answer(s), if any, to the query Q.

(b) Repeat (a) for the query

$\leftarrow \forall y/\tau\ (p(y) \leftarrow \forall x/\tau\ r(x))$

7. Consider the supplier-part-job database of §21.

(a) The following query is ambiguous:

Is it true that each red part is supplied by a supplier located in Perth?

Find two possible meanings for the query and for each of these meanings write down the corresponding (first order logic) query $\leftarrow W$.

(b) For each of the (first order logic) queries of part (a) show the normal program and goal which results from the query transformation process.

(c) Write each of the (first order logic) queries of part (a) in SQL [25].

(d) Compare first order logic and SQL as query languages with regard to expressiveness, semantic clarity, conciseness and simplicity.

8. Prove lemma 22.7.

9. Prove lemma 23.2.

10. Let D be a database and Q a query. Suppose that $D \cup \{Q\}$ has a finitely failed SLDNF-tree. Prove that Q is a logical consequence of comp(D).

11. Let D be a definite database and Q a definite query. Suppose that Q is a logical consequence of comp(D). Prove that every fair SLD-tree for $D \cup \{Q\}$ is finitely failed.

12. Complete the details of the proof of lemma 24.4(b).

13. Let D be the database

no_male_descendant(x) $\leftarrow \forall y\ (\text{female}(y) \leftarrow \text{ancestor}(x,y))$

ancestor(x,y) $\leftarrow$ parent(x,z) $\wedge$ ancestor(z,y) ancestor(x,y) $\leftarrow$ parent(x,y)

parent(x,y) $\leftarrow$ mother(x,y)

parent(x,y) $\leftarrow$ father(x,y)

together with facts for the predicate symbols mother, father, male and female. Let D' be the database obtained by adding to D the facts

father(John, Fred)$\leftarrow$

mother(Jane, Fred)$\leftarrow$

(a) Calculate $pos_{D,D'}$, $neg_{D,D'}$, P, and N.

(b) For each of the integrity constraints below, state which instances of them will need to be checked when the database changes from D to D', assuming D satisfies the integrity constraints.

(i) $\forall$x (male(x) $\leftarrow$ $\exists$y father(x,y))

(ii) $\forall$x (~$\exists$y mother(x,y) $\vee$ ~$\exists$z father(x,z))

(iii) $\forall$x $\forall$y (no_male_descendant(y) $\leftarrow$ ancestor(x,y) $\wedge$ no_male_descendant(x))

(iv) $\forall$x $\forall$y (~parent(x,y) $\vee$ ~parent(y,x))

Chapter 6

PERPETUAL PROCESSES

A perpetual process is a definite program which does not terminate and yet is doing useful computation, in some sense. With the advent of PROLOG systems for concurrent applications [18], [93], [106], especially operating systems, more and more programs will be of this type. Unfortunately, the semantics for definite programs developed in chapter 2 do not apply to perpetual processes, simply because they do not terminate. In this chapter, starting from the pioneering work of Andreka, van Emden, Nemeti and Tiuryn [2], we discuss the basic results of a semantics for perpetual processes.

§25. COMPLETE HERBRAND INTERPRETATIONS

In this section, we introduce complete Herbrand interpretations. We define the complete Herbrand universe and base and prove that they are compact metric spaces under a suitable metric. Some elementary notions from metric space topology, all of which can be found in [29], for example, will be required.

The complete Herbrand universe for a definite program is the collection of all (possibly infinite) terms which can be constructed from the constants and function symbols in the program. Thus our first task is to give a precise definition of a (possibly infinite) term, which extends the definition given in §2 of a (finite) term.

Let ω^* denote the set of all finite lists of non-negative integers. Lists are denoted by $[i_1,\ldots,i_k]$, where $i_1,\ldots,i_k \in \omega$. If $m,n \in \omega^*$, then $[m,n]$ denotes the list which is the concatenation of m and n. If $n \in \omega^*$ and $i \in \omega$, then $[n,i]$ denotes the list $[n,[i]]$. We let $|X|$ denote the cardinality of the set X. Similarly, if $n \in \omega^*$, then $|n|$ denotes the number of elements in n.

Definition We say T ⊆ ω* is a *tree* if the following conditions are satisfied:
(a) For all n∈ω* and for all i,j∈ω, if [n,i]∈T and j<i, then n∈T and [n,j]∈T.
(b) |{i : [n,i]∈T}| is finite, for all n∈T.

Definition A tree T is *finite* if T is a finite subset of ω*. Otherwise, T is *infinite.*

Example The finite tree {[], [0], [1], [2], [1,0], [1,1], [2,0], [2,1], [2,2]} can be pictured as in Figure 7.

The infinite tree {[], [0], [1], [1,0], [1,1], [1,1,0], [1,1,1], [1,1,1,0], [1,1,1,1],...} can be pictured as in Figure 8.

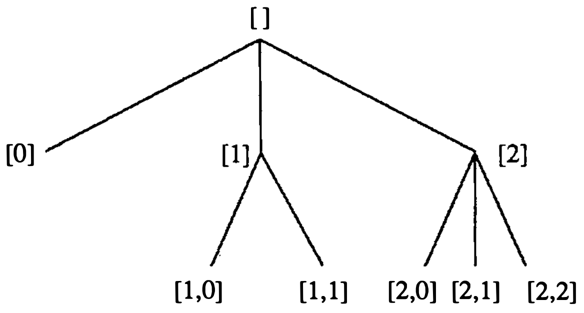

Fig. 7. A finite tree

Intuitively, each n∈T is a node of the tree T. Condition (b) in the definition of tree states that each node has bounded degree.

We let S be a set of *symbols* and ar be a mapping from S into ω, which determines the *arity* of each symbol in S.

Definition A *term (over S)* is a function t : dom(t) → S such that
(a) The domain of t, dom(t), is a non-empty tree.
(b) For all n∈dom(t), ar(t(n)) = |{i : [n,i]∈dom(t)}|.

We say the tree dom(t) *underlies* t. We let Term_S denote the set of all terms over S.

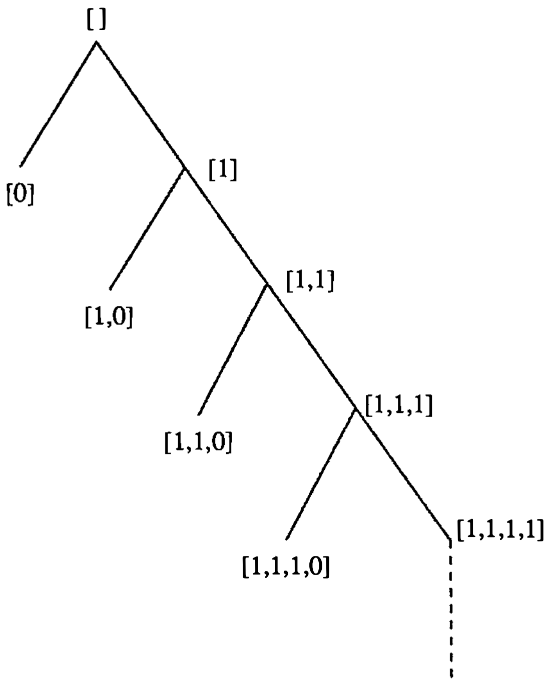

Fig. 8. An infinite tree

Intuitively, a term is a (possibly infinite) tree, whose nodes are labelled by symbols in such a way that the arity of the label of each node is equal to the degree of that node.

Definition The term t is *finite* if dom(t) is finite. Otherwise, t is *infinite.*

Definition Let t be a term. The *depth*, dp(t), of t is defined as follows:
(a) If t is infinite, then dp(t) = ∞.
(b) If t is finite, then dp(t) = 1 + max{|n| : n$\in$dom(t)}.

It will be convenient to have available the concept of the *truncation at depth n* ($n\in\omega$) of a term t, denoted by $\alpha_n(t)$. For this purpose, we introduce a new symbol Ω of arity 0, which will be used to indicate that a branch of the term t has been cut off in the truncation. Thus α_n is a mapping from Term_S into $\text{Term}_S \cup \{\Omega\}$ defined as follows:

(a) $\mathrm{dom}(\alpha_n(t)) = \{m \in \mathrm{dom}(t) : |m| \le n\}$.
(b) $\alpha_n(t) : \mathrm{dom}(\alpha_n(t)) \rightarrow S \cup \{\Omega\}$ is defined by

$$\begin{aligned} \alpha_n(t)(m) &= t(m), && \text{if } |m| < n \\ &= \Omega, && \text{if } |m| = n. \end{aligned}$$

Clearly, $\alpha_n(t)$ is a finite term with $\mathrm{dp}(\alpha_n(t)) \le n+1$.

Term_S can be made into a metric space in a natural way. First, we recall the definition of a metric space [29].

Definition Let X be a set. A mapping $d : X \times X \rightarrow$ non-negative reals is a *metric* for X if
(a) $d(x,y) = 0$ iff $x=y$, for all $x,y \in X$.
(b) $d(x,y) = d(y,x)$, for all $x,y \in X$.
(c) $d(x,z) \le d(x,y) + d(y,z)$, for all $x,y,z \in X$.
d is an *ultrametric* [5] if
(d) $d(x,z) \le \max\{d(x,y), d(y,z)\}$, for all $x,y,z \in X$.

Definition (X,d) is a *metric space*, if d is a metric on X. If d is an ultrametric, then (X,d) is an *ultrametric space.*

Ultrametric spaces have topological properties rather similar to discrete metric spaces [5].

Now let $s,t \in \mathrm{Term}_S$. If $s \ne t$, then it is clear that $\alpha_n(s) \ne \alpha_n(t)$, for some $n>0$. Consequently, if $s \ne t$, then $\{n : \alpha_n(s) \ne \alpha_n(t)\}$ is not empty. We define $\alpha(s,t) = \min\{n : \alpha_n(s) \ne \alpha_n(t)\}$. Thus $\alpha(s,t)$ is the least depth at which s and t differ.

Proposition 25.1 (Term_S, d) is an ultrametric space, where d is defined by

$$\begin{aligned} d(s,t) &= 0, && \text{if } s=t \\ &= 2^{-\alpha(s,t)}, && \text{otherwise.} \end{aligned}$$

Proof Straightforward. (See problem 1.) ■

Convergence in the topology induced by d is denoted by $\rightarrow$. Thus $t_n \rightarrow t$ means that the sequence $\{t_n\}_{n \in \omega}$ converges to t in this topology. The closure of a set A in this topology is denoted by $\bar{A}$.

Definition A metric space (X,d) is *compact* if every sequence in X has a subsequence which converges to a point in X.

A crucial fact about Term_S is given by the following proposition [72].

Proposition 25.2 (Term_S, d) is compact iff S is finite.

Proof Suppose first that S is infinite. Let $\{t_s : s \in S\}$ be any collection of terms with the property that $t_s([])=s$ (that is, the root is labelled by s). If $s_1 \neq s_2$, then $d(t_{s_1}, t_{s_2}) = 1/2$. Thus Term_S is not compact.

Conversely, suppose that S is finite. Let $\{t_k\}_{k\in\omega}$ be a sequence in Term_S. We consider two cases.

(a) *There exists $m \in \omega$ and $p \in \omega$ such that, for all $n \geq p$, we have $dp(t_n) \leq m$.*

Since S is finite, there are only a finite number of terms over S of depth $\leq$ m. Hence $\{t_k\}_{k\in\omega}$ must have a constant and, hence, convergent subsequence.

(b) *Given $m \in \omega$ and $p \in \omega$, there exists $n \geq p$ such that $dp(t_n) > m$.*

In this case, we can suppose without loss of generality that the sequence $\{t_k\}_{k\in\omega}$ is such that $dp(t_k)>k$, for $k \in \omega$. Note that every subsequence of $\{t_k\}_{k\in\omega}$ has the property that the depths of the terms in the subsequence are unbounded.

We define by induction an infinite term $t \in \mathrm{Term}_S$ such that, for each $n \geq 1$, there exists a subsequence $\{t_{k_m}\}_{m\in\omega}$ of $\{t_k\}_{k\in\omega}$ with $\alpha_n(t_{k_m}) = \alpha_n(t)$, for $m \in \omega$.

Suppose first that n=1. Since S is finite, a subsequence $\{t_{k_m}\}_{m\in\omega}$ of $\{t_k\}_{k\in\omega}$ must have the same symbol, say s, labelling their root nodes. We define $t([]) = s$.

Next suppose that t is defined up to depth n. Thus there exists a subsequence $\{t_{k_m}\}_{m\in\omega}$ of $\{t_k\}_{k\in\omega}$ such that $\alpha_n(t_{k_m}) = \alpha_n(t)$, for $m \in \omega$. Since S is finite, there exists a subsequence $\{t_{k_{m_p}}\}_{p\in\omega}$ of $\{t_{k_m}\}_{m\in\omega}$ such that the $\alpha_{n+1}(t_{k_{m_p}})$ are all equal, for $p \in \omega$. Define the nodes at depth n+1 for t in the same way as each of the $t_{k_{m_p}}$. This completes the inductive definition.

Since it is clear that t is an accumulation point of $\{t_k\}_{k\in\omega}$, we have shown that Term_S is compact. ∎

Now we are in a position to define the complete Herbrand universe. Let P be a definite program and F be the finite set of constants and function symbols in P. We regard constants as function symbols of arity 0.

Definition The *complete Herbrand universe* U'_P for P is Term_F. The elements of U'_P are called *ground terms*.

Thus U'_P is the set of all ground (possibly infinite) terms which can be formed out of the constants and function symbols appearing in P. It is straightforward to show that "ground term", as defined in §3, can be identified with "finite ground term", as just defined. (See problem 2.) This identification is taken for granted throughout this chapter. Thus we have $U_P \subseteq U'_P$. As long as P contains at least one function symbol, it is clear that U_P is a proper subset of U'_P.

We adopt the convention throughout this chapter that "term", without qualification, will always mean a possibly infinite term. If a term is finite, this will always be explicitly stated.

Despite the fact that we have given a rather formal definition of term, in the material which follows we will rarely make direct reference to this definition, relying instead on the reader's intuitive understanding of a term. All the arguments presented could easily be formalised, if desired. We will also find it convenient to use a more informal notation for terms. In particular, for finite terms we will continue to use the old notation.

Example fff... is the infinite term pictured in Figure 9.
f(a,f(a,f(a,...))) is the infinite term pictured in Figure 10.

Proposition 25.3 Let P be a definite program. Then U'_P is a compact metric space, under the metric d introduced earlier.

Proof The result follows from proposition 25.2, since the set of constants and function symbols in P is finite. ∎

The proof of the next result is straightforward. (See problem 3.)

Proposition 25.4 Let P be a definite program. Then U_P is dense in U'_P, under the topology induced by d.

U'_P is called "complete" because it is the completion [29] of the metric space U_P. We will also require the concept of a (possibly infinite) atom. Let P be a definite program, F be the set of constants and function symbols in P, R be the set of predicate symbols in P and V be the set of variables in P (more precisely, the first order language underlying P). All variables have arity 0.

Definition An *atom* A is an element of $\text{Term}_{V \cup F \cup R}$ such that $A(n) \in R$ iff $n=[]$, for all $n \in \text{dom}(A)$.

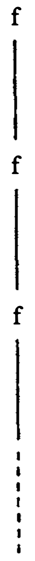

Fig. 9. The infinite term fff...

Thus an atom is a term with the root node (only) labelled by a predicate symbol. Just as we did for terms, we can identify "finite atom", as just defined, with "atom", as defined in §2. Whenever an atom is finite, this will always be explicitly stated in this chapter.

Definition The *complete Herbrand base* B'_P for a definite program P is the set of all terms A in $Term_{F \cup R}$ for which $A(n) \in R$ iff $n=[]$, for all $n \in dom(A)$. The elements of B'_P are called *ground atoms.*

Thus B'_P is the set of all ground (possibly infinite) atoms which can be formed out of the finite set of constants, function symbols and predicate symbols appearing in P. Note that $B_P \subseteq B'_P$.

Proposition 25.5 Let P be a definite program. Then B'_P is a compact metric space, under the metric d introduced earlier.

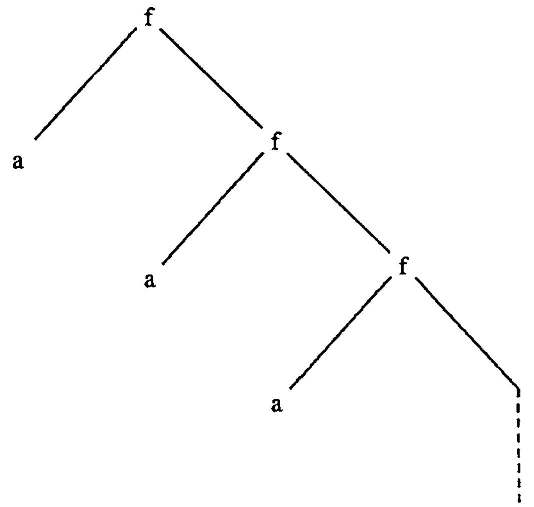

Fig. 10. The infinite term f(a,f(a,f(a,...)))

Proof $\text{Term}_{F \cup R}$ is compact, by proposition 25.2. It is easy to show that B'_P is a closed and, therefore, compact subspace of $\text{Term}_{F \cup R}$. ■

Proposition 25.6 Let P be a definite program. Then B_P is dense in B'_P, under the topology induced by d.

Proof Straightforward. ■

The concept of a substitution applied to an atom in §4 can be easily generalised to the present more general definition of atom and term. We restrict attention to ground substitutions applied to finite atoms, which is all that is needed in this chapter.

Definition A *ground substitution* θ is a finite set of the form $\{v_1/t_1,...,v_k/t_k\}$, where each v_i is a variable, the variables are distinct and $t_i \in U'_P$, for i=1,...,k.

Definition Let A be a finite atom with variables $\{v_1,...,v_k\}$ and $\theta = \{v_1/t_1,...,v_k/t_k\}$ be a ground substitution. Then Aθ is the ground atom defined as follows:

(a) $\mathrm{dom}(A\theta) = \mathrm{dom}(A) \cup \{[m,n] : m \in \mathrm{dom}(A),\ A(m)=v_i \text{ and } n \in \mathrm{dom}(t_i), \text{ for some } i \in \{1,\ldots,k\}\}$.

(b) $A\theta : \mathrm{dom}(A\theta) \rightarrow F \cup R$ is defined by

$A\theta(m) \quad = A(m)$, if $m \in \mathrm{dom}(A)$ and $A(m) \notin \{v_1,\ldots,v_k\}$

$A\theta([m,n]) = t_i(n)$, if $m \in \mathrm{dom}(A)$, $A(m)=v_i$ and $n \in \mathrm{dom}(t_i)$, for some $i \in \{1,\ldots,k\}$.

We say $A\theta$ is a *ground instance* of A. The collection of all ground instances of the finite atom A is denoted by [[A]]. Note that $[A] \subseteq [[A]] \subseteq B'_P$.

Proposition 25.7 Let P be a definite program and $C = \{A_1,\ldots,A_m\}$ be a set of finite atoms with variables $x_1,\ldots,x_n$. Consider the mapping

$$S_C : (U'_P)^n \rightarrow (B'_P)^m$$

defined by

$$S_C(t_1,\ldots,t_n) = (A_1\theta,\ldots,A_m\theta),$$

where $\theta = \{x_1/t_1,\ldots,x_n/t_n\}$. Then S_C is continuous, where $(U'_P)^n$ and $(B'_P)^m$ are each given the product topology.

Proof Suppose that $\{(t_{1,k},\ldots,t_{n,k})\}_{k\in\omega}$ converges to $(t_1,\ldots,t_n)$ in the product topology on $(U'_P)^n$. Put $\theta_k = \{x_1/t_{1,k},\ldots,x_n/t_{n,k}\}$, for $k \in \omega$. Clearly $A_i\theta_k \rightarrow A_i\theta$, for $i=1,\ldots,m$, and hence S_C is continuous. ■

Proposition 25.8 Let A be a finite atom. Then [[A]] is a closed subset of B'_P.

Proof Put $C = \{A\}$. If A has n variables, then $[[A]] = S_C((U'_P)^n)$. Since S_C is continuous and U'_P is compact, $S_C((U'_P)^n)$ is a compact and, therefore, closed subset of B'_P. ■

Proposition 25.9 Let A be a finite atom. Then $\overline{[A]} = [[A]]$.

Proof Since $[A] \subseteq [[A]]$ and [[A]] is closed, $\overline{[A]} \subseteq [[A]]$. On the other hand, if $C=\{A\}$ and A has n variables, then $[[A]] = S_C((U'_P)^n) = S_C((\bar{U}_P)^n) \subseteq \overline{S_C((U_P)^n)} = \overline{[A]}$, by propositions 25.4 and 25.7. ■

We conclude this section with the definition of a complete Herbrand interpretation and the mapping T'_P.

Definition Let P be a definite program. An interpretation for P is a *complete Herbrand interpretation* if the following conditions are satisfied:

(a) The domain of the interpretation is the complete Herbrand universe U'_P.
(b) Constants in P are assigned themselves in U'_P.
(c) If f is an n-ary function symbol in P, then the mapping from $(U'_P)^n$ into U'_P defined by $(t_1,...,t_n) \rightarrow f(t_1,...,t_n)$ is assigned to f.

We make no restrictions on the assignment to the predicate symbols in P, so that different complete Herbrand interpretations arise by taking different such assignments. In an analogous way to that in §3, we identify a complete Herbrand interpretation with a subset of B'_P. The set of all complete Herbrand interpretations for P is a complete lattice under the partial order of set inclusion.

Definition Let P be a definite program. A *complete Herbrand model* for P is a complete Herbrand interpretation which is a model for P.

We also define a mapping T'_P from the lattice of complete Herbrand interpretations to itself as follows. Let I be a complete Herbrand interpretation. Then $T'_P(I) = \{A \in B'_P : A \leftarrow B_1,...,B_n$ is a ground instance of a clause in P and $\{B_1,...,B_n\} \subseteq I\}$.

Note that T'_P is T^J_P for the pre-interpretation J consisting of the domain U'_P and the above assignments to constants and function symbols. It turns out that because of the compactness of U'_P and B'_P, T'_P has an even richer set of properties than T_P. We explore these properties in the next section.

§26. PROPERTIES OF T'_P

In this section we establish various important properties of T'_P, notably that $gfp(T'_P) = T'_P \downarrow \omega$.

We begin with four results, which are the analogues for T'_P of propositions 6.1, 6.3 and 6.4 and theorem 6.5. The proofs of these results are essentially the same as the earlier ones.

Proposition 26.1 (Model Intersection Property)
Let P be a definite program and $\{M_i\}_{i \in I}$ be a non-empty set of complete Herbrand models for P. Then $\cap_{i \in I} M_i$ is a complete Herbrand model for P.

We let M'_P denote the least complete Herbrand model for P. Thus M'_P is the intersection of all complete Herbrand models for P.

Proposition 26.2 Let P be a definite program. Then the mapping T'_P is continuous (in the lattice-theoretic sense of §5).

Proposition 26.3 Let P be a definite program and I be a complete Herbrand interpretation for P. Then I is a model for P iff $T'_P(I) \subseteq I$.

Theorem 26.4 Let P be a definite program. Then $M'_P = \text{lfp}(T'_P) = T'_P\uparrow\omega$.

The next result is due to Andreka, van Emden, Nemeti and Tiuryn [2].

Theorem 26.5 (Closedness of T'_P)

Let P be a definite program and I be a closed subset of B'_P. Then $T'_P(I)$ is a closed subset of B'_P. Furthermore, $\overline{T'_P(J)} \subseteq T'_P(\bar{J})$, for $J \subseteq B'_P$.

Proof Let I be a closed subset of B'_P. We show $T'_P(I)$ is closed. It is sufficient to consider the case when P consists of a single clause, say, $A \leftarrow A_1,\ldots,A_m$. Suppose the clause has n variables. Put $C=\{A, A_1,\ldots,A_m\}$ and let S_C be the associated mapping defined in §25. Since S_C is continuous and U'_P is compact, we have that $S_C((U'_P)^n)$ is a closed subset of $(B'_P)^{m+1}$. Let π denote the projection from $(B'_P)^{m+1}$ onto its first component. Then $T'_P(I) = \pi(S_C((U'_P)^n) \cap (B'_P \times I^m))$ and thus $T'_P(I)$ is closed.

For the last part, it is straightforward to show that T'_P maps closed sets to closed sets iff $\overline{T'_P(J)} \subseteq T'_P(\bar{J})$, for $J \subseteq B'_P$. ∎

Corollary 26.6 $T'_P\downarrow k$ is closed, for $k\in\omega$. Furthermore, $T'_P\downarrow\omega$ is closed.

Note carefully that we do not necessarily have the opposite inclusion $\overline{T'_P(J)} \supseteq T'_P(\bar{J})$, for $J \subseteq B'_P$.

Example Let P be the program

q(a) ← p(f(x),f(x))

Let $J = \{p(t,f(t)) : t\in U_P\}$. Then $T'_P(\bar{J})=\{q(a)\}$, but $\overline{T'_P(J)}=\emptyset$.

Next we establish an important weak continuity result for T'_P. For this we need the concept of the limit superior of a sequence of subsets of a metric space [5].

Definition Let (X,d) be a metric space and $\{Y_n\}_{n\in\omega}$ be a sequence of subsets of X. Then we define $LS_{n\in\omega}(Y_n) = \{x\in X$: for every neighbourhood V of x and for every $m\in\omega$, there exists $k\geq m$ such that $V \cap Y_k \neq \emptyset\}$.

If $\{Y_n\}_{n\in\omega}$ is a decreasing sequence of closed sets, it is easy to show that $LS_{n\in\omega}(Y_n)=\cap_{n\in\omega}Y_n$.

Theorem 26.7 (Weak Continuity of T'_P)

Let P be a definite program and $\{I_k\}_{k\in\omega}$ be a sequence of sets in B'_P. Then $LS_{k\in\omega}(T'_P(I_k)) \subseteq T'_P(LS_{k\in\omega}(I_k))$.

Proof Suppose $A\in LS_{k\in\omega}(T'_P(I_k))$. Then, for every neighbourhood V of A, there exist infinitely many k such that $V \cap T'_P(I_k)\neq\varnothing$. Since P is finite, there exist a clause $A_0\leftarrow A_1,\ldots,A_m$ in P, a subsequence $\{I_{k_p}\}_{p\in\omega}$ of $\{I_k\}_{k\in\omega}$ and a sequence $\{\theta_p\}_{p\in\omega}$ of ground substitutions for the variables $x_1,\ldots,x_n$ of the clause such that $A_0\theta_p\rightarrow A$ and $A_j\theta_p\in I_{k_p}$, for $j=1,\ldots,m$ and $p\in\omega$.

Suppose θ_p is $\{x_1/t_{1,p},\ldots,x_n/t_{n,p}\}$. Since U'_P is compact, we can assume without loss of generality that $(t_{1,p},\ldots,t_{n,p})\rightarrow(t_1,\ldots,t_n)$, say. Put $\theta = \{x_1/t_1,\ldots,x_n/t_n\}$. By proposition 25.7, we have that $(A_0\theta_p,\ldots,A_m\theta_p)\rightarrow(A_0\theta,\ldots,A_m\theta)$. Since $A_0\theta_p\rightarrow A$, we have that $A_0\theta=A$. Furthermore, since $A_j\theta_p\rightarrow A_j\theta$, we have that $A_j\theta\in LS_{k\in\omega}(I_k)$, for $j=1,\ldots,m$. Hence $A\in T'_P(LS_{k\in\omega}(I_k))$. ■

Note that we do not generally have $LS_{k\in\omega}(T'_P(I_k)) = T'_P(LS_{k\in\omega}(I_k))$.

Example Consider the program

q(a) ← p(f(x),f(x))

Put $I_k=\{p(f^k(a),f^{k+1}(a))\}$, for $k\in\omega$. Then $LS_{k\in\omega}(I_k)=\{p(fff\ldots,fff\ldots)\}$. Thus $T'_P(LS_{k\in\omega}(I_k))=\{q(a)\}$, but $LS_{k\in\omega}(T'_P(I_k))=\varnothing$.

Corollary 26.8 (Intersection Property for T'_P)

Let P be a definite program and $\{I_k\}_{k\in\omega}$ be a decreasing sequence of closed sets in B'_P. Then $T'_P(\cap_{k\in\omega}I_k) = \cap_{k\in\omega}T'_P(I_k)$.

Proof We have that

$T'_P(\cap_{k\in\omega}I_k)$

$= T'_P(LS_{k\in\omega}(I_k))$, since the I_k are closed and decreasing

$\supseteq LS_{k\in\omega}(T'_P(I_k))$, by theorem 26.7

$= \cap_{k\in\omega}T'_P(I_k)$, since the $T'_P(I_k)$ are closed and decreasing.

Furthermore, since T'_P is monotonic, we have $T'_P(\cap_{k\in\omega}I_k) \subseteq \cap_{k\in\omega}T'_P(I_k)$. ■

We cannot drop the requirement that each I_k be closed in corollary 26.8.

Example Consider the program

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

Let I_k be $\{p(f^n(a)) : n\geq k\}$, for $k\in\omega$. Then $\{I_k\}_{k\in\omega}$ is a decreasing sequence. Furthermore, $\cap_{k\in\omega}I_k=\varnothing$, so that $T'_P(\cap_{k\in\omega}I_k)=\varnothing$. However, $T'_P(I_k) = \{q(a)\}$, for $k\in\omega$. Thus $\cap_{k\in\omega}T'_P(I_k)=\{q(a)\}$.

Part (a) of the next theorem is due to Andreka, van Emden, Nemeti and Tiuryn [2]. Recall that it can happen that $gfp(T_P)\neq T_P\downarrow\omega$.

Theorem 26.9 Let P be a definite program. Then we have

(a) $gfp(T'_P) = T'_P\downarrow\omega$.

(b) $T'_P(\cap_{k\in\omega}\overline{T_P\downarrow k}) \supseteq \cap_{k\in\omega}\overline{T_P\downarrow k}$.

Proof (a) It suffices to show that $T'_P(T'_P\downarrow\omega)=T'_P\downarrow\omega$. Now we have

$T'_P(T'_P\downarrow\omega)$

$= T'_P(\cap_{k\in\omega}T'_P\downarrow k)$

$= \cap_{k\in\omega}T'_P(T'_P\downarrow k)$, by corollaries 26.6 and 26.8

$= T'_P\downarrow\omega$.

(b) We have

$T'_P(\cap_{k\in\omega}\overline{T_P\downarrow k})$

$= \cap_{k\in\omega}T'_P(\overline{T_P\downarrow k})$, by corollary 26.8

$\supseteq \cap_{k\in\omega}\overline{T'_P(T_P\downarrow k)}$, by theorem 26.5

$\supseteq \cap_{k\in\omega}\overline{T_P\downarrow k}$. ■

It is apparent that the essential reason that $gfp(T'_P)=T'_P\downarrow\omega$ is because U'_P is compact. We generally have $gfp(T_P)\neq T_P\downarrow\omega$ precisely because limits of sequences of finite terms are missing from U_P. In many respects, T'_P, U'_P and B'_P give a more appropriate setting for the foundations of logic programming than T_P, U_P and B_P.

Note that $\cap_{k\in\omega}\overline{T_P{\downarrow}k}$ may not be a fixpoint of T'_P.

Example Let P be the program
q(a) ← p(x,f(x))
p(f(x),f(x)) ← p(x,x)
Then $\cap_{k\in\omega}\overline{T_P{\downarrow}k} = \{p(fff...,fff...)\}$, but $T'_P(\cap_{k\in\omega}\overline{T_P{\downarrow}k}) = \{q(a), p(fff...,fff...)\}$.

Proposition 26.10 Let P be a definite program. Then we have
(a) $\overline{T_P{\downarrow}k} = T'_P{\downarrow}k$, for k=0, 1.
(b) $\overline{T_P{\downarrow}k} \subseteq T'_P{\downarrow}k$, for k≥2.

Proof By corollary 26.6, $T'_P{\downarrow}k$ is closed, for $k\in\omega$. Also it is easy to show by induction that $T_P{\downarrow}k \subseteq T'_P{\downarrow}k$, for $k\in\omega$. Thus we have $\overline{T_P{\downarrow}k} \subseteq T'_P{\downarrow}k$, for $k\in\omega$. Furthermore, $\overline{T_P{\downarrow}0} = \bar{B}_P = B'_P = T'_P{\downarrow}0$. Finally, we leave the proof that $\overline{T_P{\downarrow}1} = T'_P{\downarrow}1$ to problem 9. ∎

Note that $\overline{T_P{\downarrow}k}$ may be a proper subset of $T'_P{\downarrow}k$, for k≥2. (See problem 10.)

Proposition 26.11 Let P be a definite program. Then $\overline{T_P{\downarrow}\omega} \subseteq \cap_{k\in\omega}\overline{T_P{\downarrow}k} \subseteq T'_P{\downarrow}\omega$.

Proof We have
$\overline{T_P{\downarrow}\omega}$
$= \overline{\cap_{k\in\omega}T_P{\downarrow}k}$
$\subseteq \cap_{k\in\omega}\overline{T_P{\downarrow}k}$
$\subseteq \cap_{k\in\omega}T'_P{\downarrow}k$, by proposition 26.10
$= T'_P{\downarrow}\omega$. ∎

Note that both of the inclusions in proposition 26.11 may be proper. (See problem 11.)

Next, we prove a useful characterisation of $\cap_{k\in\omega}\overline{T_P{\downarrow}k}$.

Theorem 26.12 Let P be a definite program and $A\in B'_P$. Then the following are equivalent:
(a) $A\in \cap_{k\in\omega}\overline{T_P{\downarrow}k}$.
(b) There exists a sequence $\{A_k\}_{k\in\omega}$ such that $A_k\in T_P{\downarrow}k$, for $k\in\omega$, and $A_k\rightarrow A$.

(c) There exists a finite atom B and a non-failed fair derivation $\leftarrow B=G_0, G_1,\ldots$ with mgu's $\theta_1, \theta_2,\ldots$ such that $A\in\cap_{k\in\omega}[[B\theta_1\ldots\theta_k]]$. (If the derivation is successful, then the intersection is over the finite set of non-negative integers which index the goals of the derivation).

Proof The equivalence of (a) and (b) is left to problem 12.

(c) implies (a). Suppose (c) holds. By proposition 25.9, we have that $A\in\cap_{n\in\omega}\overline{[B\theta_1\ldots\theta_n]}$. By proposition 13.5, given $k\in\omega$, there exists $n\in\omega$ such that $[B\theta_1\ldots\theta_n]\subseteq T_P{\downarrow}k$. Hence $A\in\cap_{k\in\omega}T_P{\downarrow}k$.

(b) implies (c). For this proof, we ensure fairness in all derivations by always selecting atoms as follows. We select 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, we select the leftmost atom.

Let $\{A_k\}_{k\in\omega}$ be a sequence such that $A_k\in T_P{\downarrow}k$, for $k\in\omega$, and $A_k\to A$. Since $A_k\in T_P{\downarrow}k$, proposition 13.4 shows that there is a derivation D_k beginning with $\leftarrow A_k$, which is either successful (that is, D_k is a refutation of $P\cup\{\leftarrow A_k\}$) or has length > k. We consider two cases.

(1) *Given $m\in\omega$ and $p\in\omega$, there exists $n\geq p$ such that D_n has length > m.*

In this case, by passing to an appropriate subsequence, we can assume without loss of generality that the sequence $\{A_k\}_{k\in\omega}$ is such that $A_k\in T_P{\downarrow}k$, for $k\in\omega$, $A_k\to A$ and D_k has length > k.

We now prove by induction that there exists a finite atom B and an infinite fair derivation $\leftarrow B=G_0, G_1,\ldots$ with input clauses $C_1, C_2,\ldots$ such that, for each $n\in\omega$, there exists a subsequence $\{A_{k_m}\}_{m\in\omega}$ of $\{A_k\}_{k\in\omega}$, where $C_1,\ldots,C_{n+1}$ are the same (up to variants) as the first n+1 input clauses of each of the D_{k_m} and G_{n+1} is more general than the (n+1)th goal in D_{k_m}, for $m\in\omega$.

Suppose first that n=0. Since P contains only finitely many clauses, a subsequence $\{A_{k_m}\}_{m\in\omega}$ of $\{A_k\}_{k\in\omega}$ must use the same program clause, say E, in the first step of D_{k_m}. We let B be the head of E and let C_1 be a suitable variant of E.

Next suppose the result holds for n–1. Thus there exists a finite atom B and a fair derivation $\leftarrow B=G_0, G_1,\ldots,G_n$ via R with input clauses $C_1,\ldots,C_n$ such that there exists a subsequence $\{A_{k_m}\}_{m\in\omega}$ of $\{A_k\}_{k\in\omega}$, where $C_1,\ldots,C_n$ are the same (up to variants) as the first n input clauses of each of the D_{k_m} and G_n is more general than the nth goal in D_{k_m}, for $m\in\omega$. Note that as the lengths of the D_{k_m} are unbounded, the nth goal in each D_{k_m} is not empty. Furthermore, the same atom is selected in the nth goal of each D_{k_m}. Since P contains only finitely many clauses, a subsequence $\{A_{k_{m_p}}\}_{p\in\omega}$ of $\{A_{k_m}\}_{m\in\omega}$ must use the same program clause, say F, as the (n+1)th input clause of the derivation $D_{k_{m_p}}$. It is clear that (a suitable variant of) F can be used as C_{n+1}. This completes the induction argument.

To finish off case (1), we have only to show that if θ_1, θ_2,... are the mgu's of the derivation just constructed, then $A\in[[B\theta_1...\theta_n]]$, for $n\in\omega$. However, this follows from proposition 25.9, since, given $n\in\omega$, there exists a subsequence $\{A_{k_m}\}_{m\in\omega}$ such that $A_{k_m}\rightarrow A$ and $A_{k_m}\in[B\theta_1...\theta_n]$. Thus A satisfies condition (c).

(2) *There exists $m\in\omega$ and $p\in\omega$ such that, for all $n\geq p$, D_n has length $\leq m$.*

In this case, since each D_k is either successful or has length > k, we may assume without loss of generality that there exists $m\in\omega$ such that the sequence $\{A_k\}_{k\in\omega}$ has the properties that $A_k\rightarrow A$ and each D_k is successful with length $\leq$ m. Because P is finite, there exists a subsequence $\{A_{k_m}\}_{m\in\omega}$ such that all the D_{k_m} have exactly the same sequence of input clauses (up to variants). Suppose E is the program clause used first in each of the D_{k_m}. We let B be the head of E and construct a refutation of $P\cup\{\leftarrow B\}$ of length $\leq$ m using the same sequence of input clauses as each of the D_{k_m}. In a similar way to case (1), we can show that A satisfies condition (c). ■

§27. SEMANTICS OF PERPETUAL PROCESSES

As we stated above, a perpetual process is a definite program which does not terminate and yet is doing useful computation, in some sense. The problem is to find the appropriate sense of an infinite computation being "useful". We solve this problem by introducing the concept of an infinite atom in B'_P being "computable at infinity". The set of all such atoms plays the role for perpetual processes that the success set plays for programs which terminate. The major result of this section is that the set of all atoms computable at infinity is a subset of $gfp(T'_P)$. Related results have been obtained by Nait Abdallah and van Emden [76], [77], [108].

We begin with the key definition.

Definition Let P be a definite program and $A \in B'_P \setminus B_P$. We say A is *computable at infinity* if there is a finite atom B and an infinite fair derivation $\leftarrow B = G_0, G_1, \ldots$ with mgu's $\theta_1, \theta_2, \ldots$ such that $d(A, B\theta_1 \ldots \theta_k) \to 0$, as $k \to \infty$.

We put $C_P = \{A \in B'_P \setminus B_P : A \text{ is computable at infinity}\}$.

Example Let P be the program

```
p(f(x)) ← p(x)
```

Since $lfp(T_P) = \emptyset$, this program does not compute anything in the sense of chapter 2. However, given the goal ←p(x), the atom p(fff...) can be "computed at infinity". In fact, it is clear that $C_P = \{p(fff\ldots)\}$.

Example Let P be the program

```
fib(x) ← fib1(0.1.x)
fib1(x.y.z.w) ← plus(x,y,z), fib1(y.z.w)
plus(0,x,x) ←
plus(f(x),y,f(z)) ← plus(x,y,z)
```

(Recall the convention that n stands for $f^n(0)$). Clearly fib(1.2.3.5.8.13....)$\in C_P$, where the argument of fib is the Fibonacci sequence. We simply let B be fib(x) and we obtain the approximating sequence fib(1.x_1), fib(1.2.x_2), fib(1.2.3.x_3),.... .

Example We consider Hamming's problem, which is to construct the sorted sequence t of positive integers containing no prime factors other than 2, 3 or 5. Thus the initial part of the sequence t is 2.3.4.5.6.8.9.10.12.15.... . The following program P to solve this problem appeared in [17] and [41].

```
hamming(x) ← seqprod(1.x,2,u), seqprod(1.x,3,v), seqprod(1.x,5,w),
                  merge(u,v,z), merge(z,w,x)
merge(x.u,y.v,x.w) ← y>x, merge(u,y.v,w)
merge(x.u,y.v,y.w) ← x>y, merge(x.u,v,w)
merge(x.u,x.v,x.w) ← merge(u,v,w)
seqprod(x.u,y,z.v) ← prod(x,y,z), seqprod(u,y,v)
f(x)>f(y) ← x>y
f(x)>0 ←
prod(x,0,0) ←
prod(x,f(y),z) ← prod(x,y,w), plus(w,x,z)
plus(0,x,x) ←
plus(f(x),y,f(z)) ← plus(x,y,z)
```

Then it is clear that hamming(t)$\in C_P$.

The next proposition gives a characterisation of C_P independent of the metric d.

Proposition 27.1 Let P be a definite program and $A \in B'_P \backslash B_P$. Then $A \in C_P$ iff there is a finite atom B and an infinite fair derivation $\leftarrow B = G_0, G_1, \ldots$ with mgu's $\theta_1, \theta_2, \ldots$ such that $\cap_{k \in \omega}[[B\theta_1 \ldots \theta_k]] = \{A\}$.

Proof We have to show that $d(A, B\theta_1 \ldots \theta_k) \to 0$, as $k \to \infty$, iff $\cap_{k \in \omega}[[B\theta_1 \ldots \theta_k]] = \{A\}$.

We first suppose that $\cap_{k \in \omega}[[B\theta_1 \ldots \theta_k]] = \{A\}$. Assume that there exists $n \in \omega$ such that, for all $k \in \omega$, we have $\alpha_n(A) \neq \alpha_n(B\theta_1 \ldots \theta_k)$. Then, for each $k \in \omega$, $\alpha_n(B\theta_1 \ldots \theta_k)$ must have at least one node labelled by a variable. Since $\alpha_n(A)$ is finite, it is clear that there exist a node in $\alpha_n(A)$ and $m \in \omega$ such that, for $k \geq m$, the corresponding node in $B\theta_1 \ldots \theta_k$ is labelled by a variable. (The variable may depend on k.) Consequently, $\cap_{k \in \omega}[[B\theta_1 \ldots \theta_k]]$ contains not just A, but infinitely many ground infinite atoms. Thus our original assumption is incorrect and hence, given $n \in \omega$, there exists $k \in \omega$ such that $\alpha_n(A) = \alpha_n(B\theta_1 \ldots \theta_k)$. Then $d(A, B\theta_1 \ldots \theta_k) \to 0$, as $k \to \infty$.

Conversely, let us suppose that $d(A, B\theta_1 \ldots \theta_k) \to 0$, as $k \to \infty$. Since each $[[B\theta_1 \ldots \theta_k]]$ is closed and $\{[[B\theta_1 \ldots \theta_k]]\}_{k \in \omega}$ is decreasing, it is clear that $A \in \cap_{k \in \omega}[[B\theta_1 \ldots \theta_k]]$. Next suppose $A' \in \cap_{k \in \omega}[[B\theta_1 \ldots \theta_k]]$. Let $\varepsilon > 0$ be given. Choose m such that $d(A, B\theta_1 \ldots \theta_m) < \varepsilon$. Suppose $A' = B\theta_1 \ldots \theta_m \theta$, for some θ. Thus $d(A, A') = d(A, B\theta_1 \ldots \theta_m \theta) < \varepsilon$. Since ε was arbitrary, we have that $d(A, A') = 0$ and hence $A = A'$. Thus $\cap_{k \in \omega}[[B\theta_1 \ldots \theta_k]] = \{A\}$. ∎

We could have adopted a weaker definition of C_P in which we simply demand that $A \in \cap_{k \in \omega}[[B\theta_1 \ldots \theta_k]]$. However, the following example shows that this weaker definition doesn't properly capture the notion of "computable at infinity".

Example Let P be the program

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

Under the weaker definition, we would have $p(fff\ldots) \in C_P$.

Now we can give the main result of this chapter.

Theorem 27.2 (Soundness of SLD-Resolution for Perpetual Processes)
Let P be a definite program. Then $C_P \subseteq \text{gfp}(T'_P)$.

Proof We have

C_P
$\subseteq \overline{\cap_{k\in\omega} T_P{\downarrow}k}$, by theorem 26.12 and proposition 27.1
$\subseteq T'_P{\downarrow}\omega$, by proposition 26.11
$= \mathrm{gfp}(T'_P)$, by theorem 26.9. ■

Theorem 27.2 is the analogue for perpetual processes of theorem 8.3, which states that the success set is equal to $\mathrm{lfp}(T_P)$. Since C_P contains only infinite atoms, it follows from theorem 27.2 that $C_P \subseteq \mathrm{gfp}(T'_P)\backslash B_P$. It would be pleasant if $C_P=\mathrm{gfp}(T'_P)\backslash B_P$. However, as the following examples show, this cannot be achieved without some restrictions on P or modifications to the definitions of C_P and T'_P or both.

Example Let P be the program

p(f(x)) ←

Then $p(fff...)\in \mathrm{gfp}(T'_P)\backslash B_P$, but $p(fff...)\notin C_P$.

Example Let P be the program

p(f(x)) ← p(f(x))

Then $p(fff...)\in \mathrm{gfp}(T'_P)\backslash B_P$, but $p(fff...)\notin C_P$.

Example Let P be the program

p(x,f(x)) ← p(x,x)

Then $p(fff...,fff...)\in \mathrm{gfp}(T'_P)\backslash B_P$, but $p(fff...,fff...)\notin C_P$. The problem here is that no matter what we choose for B in the definition of C_P, the computation will fail. Note that $p(fff...,fff...)\in \mathrm{gfp}(T'_P)$, because T'_P does not respect the occur check.

In view of these developments, we propose the following setting for perpetual processes. The intended interpretation of a perpetual process P is $\mathrm{gfp}(T'_P)$. This is indeed a model for P. $\mathrm{gfp}(T'_P)$ is the analogue of the intended interpretation $\mathrm{lfp}(T_P)$ for (ordinary) definite programs. C_P is then the analogue of the success set for programs. For (ordinary) definite programs, we get soundness and completeness, since $\mathrm{lfp}(T_P)$ = success set. For perpetual processes, we only have the soundness result $C_P \subseteq \mathrm{gfp}(T'_P)$. As we have seen, completeness cannot be achieved without further restrictions.

Taking a complete Herbrand model as the intended interpretation seems to be the simplest and most natural way of providing a semantics for perpetual processes. The results of this chapter suggest that $\mathrm{gfp}(T'_P)$ should be the intended interpretation. However, $gfp(T'_P)$ generally contains infinite atoms which are not intuitively computable at infinity and thus we do not get completeness. For another approach to this topic, we suggest the reader consult the paper by Levi and Palamidessi [56].

This chapter leaves many questions unanswered. Finding a satisfactory semantics for perpetual processes and for communication and synchronisation between concurrent processes is a current research problem. We believe that the appropriate setting in which to discuss such problems is the setting of U'_P, B'_P and T'_P and that the basic results presented in this chapter will play a central role in any satisfactory semantics.

PROBLEMS FOR CHAPTER 6

1. Prove proposition 25.1.

2. Prove that "finite ground term" as defined in §25 can be identified with "ground term" as defined in §3.

3. Prove that U_P is dense in U'_P.

4. Suppose $I \subseteq B'_P$ and $A \in B_P$. Prove that $A \in \bar{I}$ iff $A \in I$.

5. Find a definite program P and a complete Herbrand model I for P such that $\bar{I}$ is not a model for P.

6. Show that we cannot drop the requirement that the sequence $\{I_k\}_{k \in \omega}$ be decreasing in corollary 26.8.

7. The set of all non-empty closed subsets of B'_P can be made into a metric space using the Hausdorff metric ρ defined by $\rho(C,D)=\max\{h(C,D), h(D,C)\}$, where C and D are non-empty closed subsets of B'_P and $h(C,D)=\sup\{d(x,D) : x \in C\}$. (See [29].)
(a) Show that, if $A,B \in B'_P$, then $\rho(\{A\},\{B\}) = d(A,B)$.
(b) Show that, if $\{C_n\}_{n \in \omega}$ is a decreasing sequence of closed subsets of B'_P, then $\{C_n\}_{n \in \omega}$ is convergent in the topology induced by ρ and its limit is $\cap_{n \in \omega} C_n$.

(c) Restrict further attention to P such that $T'_P(\emptyset)\neq\emptyset$. This restriction and the fact that T'_P is closed imply that T'_P is a well-defined mapping from the metric space of non-empty closed subsets of B'_P into itself. Part (b) suggests that corollary 26.8 can be extended by proving that T'_P is continuous in the topology induced by ρ. Show that this conjecture is false.

8. Show that $\mathrm{gfp}(T'_P)$ may no longer be equal to $T'_P{\downarrow}\omega$ if the definite program P is allowed to consist of an infinite number of clauses with an infinite number of constants.

9. Prove that $\overline{T_P{\downarrow}1} = T'_P{\downarrow}1$.

10. Find a definite program P such that $\overline{T_P{\downarrow}2} \subset T'_P{\downarrow}2$.

11. Find a definite program P such that $\overline{T_P{\downarrow}\omega} \subset \cap_{k\in\omega}\overline{T_P{\downarrow}k} \subset T'_P{\downarrow}\omega$.

12. Prove that $A\in\cap_{k\in\omega}\overline{T_P{\downarrow}k}$ iff there is a sequence $\{A_k\}_{k\in\omega}$ such that $A_k\in T_P{\downarrow}k$, for $k\in\omega$, and $A_k\to A$.

13. Illustrate theorem 26.12 with the program

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

and with $A = p(fff...)$.

REFERENCES

1. Andreka, H. and I. Nemeti, ''The Generalized Completeness of Horn Predicate Logic as a Programming Language'', *Acta Cybernetica* **4**, 1 (1978), 3-10.

2. Andreka, H., M. H. van Emden, I. Nemeti and J. Tiuryn, ''Infinite-Term Semantics for Logic Programs'', draft manuscript, 1983.

3. Apt, K. R., H. A. Blair and A. Walker, ''Towards a Theory of Declarative Knowledge'', in *Foundations of Deductive Databases and Logic Programming*, Minker, J. (ed.), Morgan Kaufmann, Los Altos, 1987.

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

5. Arnold, A. and M. Nivat, ''The Metric Space of Infinite Trees: Algebraic and Topological Properties'', *Fundamenta Informatica* **3**, 4 (1980), 445-476.

6. Av-Ron, E., ''Top-Down Diagnosis of Prolog Programs'', M.Sc. Thesis, Weizmann Institute of Science, 1984.

7. Bancilhon, F. and R. Ramakrishnan, ''An Amateur's Introduction to Recursive Query Processing Strategies'', *Proc. ACM Int. Conf. on Management of Data*, Washington, D.C., 1986, 16-52.

8. Battani, G. and H. Meloni, ''Interpreteur du Language de Programmation PROLOG'', Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, 1973.

9. Bibel, W., *Automated Theorem Proving*, Vieweg, Braunschweig, 1982.

10. Bowen, D. L., L. Byrd, D. Ferguson and W. Kornfeld, *Quintus Prolog Reference Manual*, Quintus Computer Systems, Inc., May, 1985.

11. Bowen, K. A., "Programming with Full First-Order Logic", in *Machine Intelligence 10*, Ellis Horwood, Chichester, 1982, 421-440.

12. Byrd, L., "PROLOG Debugging Facilities", Working Paper, Department of Artificial Intelligence, University of Edinburgh, 1980.

13. Chandra, A. K. and D. Harel, "Horn Clause Queries and Generalizations", *J. Logic Programming* **2**, 1 (1985), 1-15.

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

15. Clark, K. L., "Negation as Failure", in *Logic and Data Bases*, Gallaire, H. and J. Minker (eds), Plenum Press, New York, 1978, 293-322.

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

17. Clark, K. L. and S. Gregory, "A Relational Language for Parallel Programming", *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, Portsmouth, N.H., 1981, 171-178.

18. Clark, K. L. and S. Gregory, "PARLOG: A Parallel Logic Programming Language", *ACM Trans. on Prog. Lang. and Systems* **8**, 1 (Jan. 1986), 1-49.

19. Clark, K. L. and F. G. McCabe, "The Control Facilities of IC-PROLOG", in *Expert Systems in the Micro Electronic Age*, Michie, D. (ed.), Edinburgh University Press, 1979, 122-149.

20. Clark, K. L. and F. G. McCabe, *micro-PROLOG: Programming in Logic*, Prentice-Hall, Englewood Cliffs, N.J., 1984.

21. Clark, K.L. and S.-Å. Tärnlund, "A First Order Theory of Data and Programs", *Information Processing 77*, Toronto, North-Holland, 1977, 939-944.

22. Colmerauer, A., H. Kanoui, P. Roussel and R. Pasero, *Un Systeme de Communication Homme-Machine en Francais*, Groupe de Recherche en

Intelligence Artificielle, Université d'Aix-Marseille, 1973.

23. Cutland, N. J., *Computability: An Introduction to Recursive Function Theory*, Cambridge University Press, Cambridge, 1980.

24. Dahl, V., "On Database Systems Development through Logic", *ACM Trans. on Database Systems* **7**, 1 (1982), 102-123.

25. Date, C. J., *An Introduction to Database Systems, Vol. 1*, Addison Wesley, Reading, Mass., 4th Edition, 1986.

26. Davis, M. and H. Putnam, "A Computing Procedure for Quantification Theory", *J. ACM* **7** (1960), 201-215.

27. Decker, H., "Integrity Enforcement in Deductive Databases", *Proc. 1st Int. Conf. on Expert Database Systems*, Charleston, S.C., 1986.

28. Dershowitz, N. and Z. Manna, "Proving Termination with Multiset Orderings", *Comm. ACM* **22**, 8 (1979), 465-476.

29. Dugundji, J., *Topology*, Allyn and Bacon, Boston, 1966.

30. Edman, A. and S.-Å. Tärnlund, "Mechanization of an Oracle in a Debugging System", *IJCAI-83*, Karlsruhe, 1983, 553-555.

31. Eisenstadt, M., "Retrospective Zooming: A Knowledge Based Tracing and Debugging Methodology for Logic Programming", *IJCAI-85*, Los Angeles, 1985, 717-719.

32. Eisenstadt, M. and A. Hasemer, "An Improved User Interface for PROLOG", *Interact 84*, 1984, 109-113.

33. Enderton, H. B., *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

34. Ferrand, G., "Error Diagnosis in Logic Programming: An Adaptation of E. Y. Shapiro's Method", Rapport de Recherche 375, INRIA, 1985.

35. Gallaire, H. and J. Minker (eds), *Logic and Data Bases*, Plenum Press, New York, 1978.

36. Gallaire, H., J. Minker and J.-M. Nicolas (eds), *Advances in Database Theory, Vol. 1*, Plenum Press, New York, 1981.

37. Gallaire, H., J. Minker and J.-M. Nicolas (eds), *Advances in Database Theory, Vol. 2*, Plenum Press, New York, 1984.

38. Gallaire, H., J. Minker and J.-M. Nicolas, "Logic and Databases: A Deductive Approach", *Computing Surveys* **16**, 2 (June 1984), 153-185.

39. Gilmore, P. C., "A Proof Method for Quantification Theory", *IBM J. Res. Develop.* **4** (1960), 28-35.

40. Green, C., "Applications of Theorem Proving to Problem Solving", *IJCAI-69*, Washington, D.C., 1969, 219-239.

41. Hansson, Å, S. Haridi and S.-Å. Tärnlund, "Properties of a Logic Programming Language", in *Logic Programming*, Clark, K.L. and S.-Å. Tärnlund (eds), Academic Press, New York, 1982, 267-280.

42. Haridi, S. and D. Sahlin, "Evaluation of Logic Programs based on Natural Deduction", TRITA-CS-8305 B, Royal Institute of Technology, Stockholm, 1983.

43. Hayes, P. J., "Computation and Deduction", *Proc. MFCS Conf.*, Czechoslovak Academy of Sciences, 1973, 105-118.

44. Herbrand, J., "Investigations in Proof Theory", in *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, van Heijenoort, J. (ed.), Harvard University Press, Cambridge, Mass., 1967, 525-581.

45. Hewitt, C., "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", A.I. Memo 251, MIT, 1972.

46. Hill, R., "LUSH-Resolution and its Completeness", DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

47. Jaffar, J., J.-L. Lassez and J. W. Lloyd, "Completeness of the Negation as Failure Rule", *IJCAI-83*, Karlsruhe, 1983, 500-506.

48. Kowalski, R. A., "Predicate Logic as a Programming Language", *Information Processing 74*, Stockholm, North Holland, 1974, 569-574.

49. Kowalski, R. A., *Logic for Problem Solving*, North Holland, New York, 1979.

50. Kowalski, R. A., "Algorithm = Logic + Control", *Comm. ACM* **22**, 7 (July 1979), 424-436.

51. Kowalski, R. A., "Logic as a Database Language", Research Report DOC 82/25 (Revised May 1984), Department of Computing, Imperial College, 1982.

52. Kowalski, R. A., "The Relation Between Logic Programming and Logic Specification", in *Mathematical Logic and Programming Languages*, Hoare, C. A. R. and J. C. Shepherdson (eds), Prentice-Hall, Englewood Cliffs, N.J., 1985, 11-27.

53. Kowalski, R. A. and D. Kuehner, "Linear Resolution with Selection Function", *Artificial Intelligence* **2** (1971), 227-260.

54. Lassez, J.-L. and M. J. Maher, "Closures and Fairness in the Semantics of Programming Logic", *Theoretical Computer Science* **29** (1984), 167-184.

55. Lassez, J.-L., V. Ḷ. Nguyen and E. A. Sonenberg, "Fixed Point Theorems and Semantics: A Folk Tale", *Inf. Proc. Letters* **14**, 3 (1982), 112-116.

56. Levi, G. and C. Palamidessi, "Contributions to the Semantics of Logic Perpetual Processes", Technical Report, Dipartimento di Informatica, Universita di Pisa, 1986.

57. Lifschitz, V., "Closed-World Databases and Circumscription", *Artificial Intelligence* **27** (1985), 229-235.

58. Lloyd, J. W., "An Introduction to Deductive Database Systems", *Australian Computer J.* **15**, 2 (May 1983), 52-57.

59. Lloyd, J. W., "Declarative Error Diagnosis", *New Generation Computing* **5**, 2 (1987).

60. Lloyd, J. W., E. A. Sonenberg and R. W. Topor, "Integrity Constraint Checking in Stratified Databases", Technical Report 86/5, Department of Computer Science, University of Melbourne, 1986. To appear in *J. Logic Programming.*

61. Lloyd, J. W. and R. W. Topor, "Making Prolog More Expressive", *J. Logic Programming* **1**, 3 (1984), 225-240.

62. Lloyd, J. W. and R. W. Topor, "A Basis for Deductive Database Systems", *J. Logic Programming* **2**, 2 (1985), 93-109.

63. Lloyd, J. W. and R. W. Topor, "A Basis for Deductive Database Systems II", *J. Logic Programming* **3**, 1 (1986), 55-67.

64. Loveland, D. W., *Automated Theorem Proving: A Logical Basis*, North Holland, New York, 1978.

65. Loveland, D. W., "A Simplified Format for the Model Elimination Procedure", *J. ACM* **16**, 3 (July 1969), 349-363.

66. Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

67. Martelli, A. and U. Montanari, "Unification in Linear Time and Space: A Structured Presentation", Nota Interna B76-16, Instituto di Elaborazione della Informazione, Pisa, 1976.

68. Martelli, A. and U. Montanari, "An Efficient Unification Algorithm", *ACM Trans. on Prog. Lang. and Systems* **4**, 2 (April 1982), 258-282.

69. Mendelson, E., *Introduction to Mathematical Logic*, 2nd Edition, Van Nostrand, Princeton, N.J., 1979.

70. Minker, J. (ed.), *Proc. Workshop on Foundations of Deductive Databases and Logic Programming*, Washington, D.C., 1986.

71. Mota-Oka, T. (ed.), *Fifth Generation Computer Systems: Proc. Int. Conf. on Fifth Generation Computer Systems*, JIPDEC, North-Holland, 1982.

72. Mycielski, J. and W. Taylor, "A Compactification of the Algebra of Terms", *Algebra Universalis* **6** (1976), 159-163.

73. Naish, L., ''Automating Control for Logic Programs'', *J. Logic Programming* **2**, 3 (1985), 167-183.

74. Naish, L., *Negation and Control in PROLOG*, Lecture Notes in Computer Science 238, Springer-Verlag, 1986.

75. Naish, L., ''Negation and Quantifiers in NU-PROLOG'', *Proc. Third Int. Conf. on Logic Programming*, Lecture Notes in Computer Science 225, Springer-Verlag, 1986, 624-634.

76. Nait Abdallah, M. A., ''On the Interpretation of Infinite Computations in Logic Programming'', *ICALP 84*, Lecture Notes in Computer Science 172, Springer-Verlag, 1984, 358-370.

77. Nait Abdallah, M. A. and M. H. van Emden, ''Algorithm Theory and Logic Programming'', draft manuscript, 1983.

78. Nicolas, J.-M., ''Logic for Improving Integrity Checking in Relational Data Bases'', *Acta Informatica* **18**, 3 (1982), 227-253.

79. Nicolas, J.-M. and H. Gallaire, ''Data Base: Theory vs. Interpretation'', in *Logic and Data Bases*, Gallaire, H. and J. Minker (eds), Plenum Press, New York, 1978, 33-54.

80. Paterson, M. S. and M. N. Wegman, ''Linear Unification'', *J. Computer and System Sciences* **16**, 2 (1978), 158-167.

81. Pereira, L. M., ''Rational Debugging in Logic Programming'', *Proc. Third Int. Conf. on Logic Programming*, Lecture Notes in Computer Science 225, Springer-Verlag, 1986, 203-210.

82. Plaisted, D. A., ''The Occur-Check Problem in PROLOG'', *IEEE Int. Symp. on Logic Programming*, Atlantic City, 1984, 272-280.

83. Plaisted, D. A., ''An Efficient Bug Location Algorithm'', *Proc. Second Int. Conf. on Logic Programming*, Uppsala, 1984, 151-157.

84. Prawitz, D., ''An Improved Proof Procedure'', *Theoria* **26** (1960), 102-139.

85. Reiter, R., ''Deductive Question-Answering on Relational Data Bases'', in *Logic and Data Bases*, Gallaire, H. and J. Minker (eds), New York, 1978,

149-177.

86. Reiter, R., "On Closed World Data Bases", in *Logic and Data Bases*, Gallaire, H. and J. Minker (eds), Plenum Press, New York, 1978, 55-76.

87. Reiter, R., "Towards a Logical Reconstruction of Relational Database Theory", in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, Brodie, M. L., J. Mylopoulos and J. W. Schmidt (eds), Springer-Verlag, Berlin, 1984, 191-233.

88. Robinson, J. A., "A Machine-oriented Logic Based on the Resolution Principle", *J. ACM* **12**, 1 (Jan. 1965), 23-41.

89. Roussel, P., *PROLOG: Manuel de Reference et d'Utilization*, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, 1975.

90. Sadri, F. and R. A. Kowalski, "An Application of General Purpose Theorem-Proving to Database Integrity", in *Proc. Workshop on Foundations of Deductive Databases and Logic Programming*, Minker, J. (ed.), Washington, D.C., 1986.

91. Sebelik, J. and P. Stepanek, "Horn Clause Programs for Recursive Functions", in *Logic Programming*, Clark, K.L. and S.-Å. Tärnlund (eds), Academic Press, New York, 1982, 324-340.

92. Shapiro, E. Y., *Algorithmic Program Debugging*, MIT Press, Cambridge, Mass., 1983.

93. Shapiro, E. Y., "A Subset of Concurrent PROLOG and its Interpreter", Technical Report TR-003, ICOT, Tokyo, 1983.

94. Shapiro, E. Y. and A. Takeuchi, "Object-Oriented Programming in Concurrent PROLOG", *New Generation Computing* **1**, 1 (1983), 25-48.

95. Shepherdson, J. C., "Negation as Failure: A Comparison of Clark's Completed Data Base and Reiter's Closed World Assumption", *J. Logic Programming* **1**, 1 (1984), 51-79.

96. Shepherdson, J. C., "Undecidability of Horn Clause Logic and Pure Prolog", unpublished manuscript, 1985.

97. Shepherdson, J. C., ''Negation as Failure II'', *J. Logic Programming* **2**, 3 (1985), 185-202.

98. Shepherdson, J. C., ''Negation in Logic Programming'', in *Foundations of Deductive Databases and Logic Programming*, Minker, J. (ed.), Morgan Kaufmann, Los Altos, 1987.

99. Shoenfield, J., *Mathematical Logic*, Addison-Wesley, Reading, Mass., 1967.

100. Sonenberg, E. A. and R. W. Topor, ''Computation in the Herbrand Universe'', unpublished manuscript, 1986.

101. Tamaki, H. and T. Sato, ''Unfold/Fold Transformation of Logic Programs'', *Proc. Second Int. Conf. on Logic Programming*, Uppsala, 1984, 127-138.

102. Tärnlund, S.-Å., ''Horn Clause Computability'', *BIT* **17**, 2 (1977), 215-226.

103. Tarski, A., ''A Lattice-theoretical Fixpoint Theorem and its Applications'', *Pacific J. Math.* **5** (1955), 285-309.

104. Thom, J. A. and J. A. Zobel (eds), ''NU-Prolog 1.0 Reference Manual'', Machine Intelligence Project, Technical Report 86/10, Department of Computer Science, University of Melbourne, 1986.

105. Topor, R. W., T. Keddis and D. W. Wright, ''Deductive Database Tools'', *Australian Computer J.* **17**, 4 (Nov. 1985), 163-173.

106. Ueda, K., ''Guarded Horn Clauses'', Ph.D. Thesis, University of Tokyo, 1986.

107. van Emden, M. H. and R. A. Kowalski, ''The Semantics of Predicate Logic as a Programming Language'', *J. ACM* **23**, 4 (Oct. 1976), 733-742.

108. van Emden, M. H. and M. A. Nait Abdallah, ''Top-Down Semantics of Fair Computations of Logic Programs'', *J. Logic Programming* **2**, 1 (1985), 67-75.

109. Van Gelder, A., ''Negation as Failure using Tight Derivations for General Logic Programs'', *Proc. 3rd IEEE Symp. on Logic Programming*, Salt Lake City, 1986, 127-138.

110. Warren, D. H. D., ''An Abstract PROLOG Instruction Set'', Technical Note 309, SRI International, 1983.

111. Warren, D. H. D. and F. C. N. Pereira, ''An Efficient Easily Adaptable System for Interpreting Natural Language Queries'', DAI Research Paper No. 155, Department of Artificial Intelligence, University of Edinburgh, 1981.

112. Wolfram, D. A., M. J. Maher and J.-L. Lassez, ''A Unified Treatment of Resolution Strategies for Logic Programs'', *Proc. Second Int. Conf. on Logic Programming*, Uppsala, 1984, 263-276.

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
$\emptyset$	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
!	63
2^{B_P}	37
ar	174
$A_{J,V}$	12
B_L	16
B_P	17
B_S	17

INDEX

Lloyd Foundations of Logic Programming

This is the second edition of the first book to give an account of the mathematical foundations of Logic Programming. Its purpose is to collect, in a unified and comprehensive manner, the basic theoretical results of Logic Programming, which have previously only been available in widely scattered research papers.
In addition to presenting the technical results, the book also contains many illustrative examples and problems. Some of them are part of the folklore of Logic Programming and are not easily obtainable elsewhere.
The book is intended to be self-contained, the only prerequisites being some familiarity with PROLOG and knowledge of some basic undergraduate mathematics. The material is suitable either as a reference book for researchers or as a text book for a graduate course on the theoretical aspects of Logic Programming and Deductive Database Systems.

ISBN 3-540-18199-7
ISBN 0-387-18199-7