Meta-Language in Logic Programming

Kave Eshghi

Phd. Thesis

Department of Computing

Imperial College of Science and Technology

University of London

July 1986

Acknowledgement

I would like to thank my supervisor, Professor Kowalski, for his guidance, patience and humour. Special thanks are also due to Ms. F. Sadri who meticulously read the thesis and checked the proofs. To M. Sergot, P. Vasey and all other members of the Logic Programming Group who helped me in innumerable ways I express my thanks.

Last but not least, thanks are due to my wife, Maria, who suffered 'the thesis' for far too long.

Abstract

The general outline of a system for amalgamating object-language and meta-language in logic programming is developed. It is shown that self-reference at theory level can be achieved following this approach. The utility of the amalgamated system is shown. In particular, it is shown that higher order features and non-floundering negation as failure can be easily implemented in this system.

Control of execution of object level programs from meta-level is discussed. It is shown that by the use of reflection, declarative control can be achieved without jeopardising the soundness of the interpreter.

The amalgamated system is used for developing a fault-finding system for digital circuits. The fault-finder uses the hypothesise and test paradigm.

The incorporation of sets as associative data-structures for logic programming is discussed. It is shown that by including set equality axioms, the deficiencies associated with the use of recursive data-structures can be removed. A method is developed for using set equality axioms efficiently in the context of logic programming.

Table Of Contents

Introduction..... 7
1. Objective of the Thesis..... 7
1.1 Declarative Semantics of Logic Programs
1.2 Procedural Semantics
1.3 Meta-Linguistic Extensions
1.4 Negation
1.5 Control of Interpreter's Search Strategy
1.6 Associative Data-Structures
2. Outline of the Thesis..... 11
2.1 Outline of Chapter One
2.2 Outline of Chapter Two
2.3 Outline of Chapter Three
2.4 Outline of Chapter Four
2.5 Outline of Chapter Five

Chapter One: General Framework of the Amalgamated System..... 14
1. The Development of the Amalgamated System..... 14
1.1 The Base Language
1.1.1 The Syntax of the Base Language
1.2 The Naming Relation
1.2.1 The Naming Function for Ground Terms
1.2.2 The Naming Function for Sentences
1.3 Some Points About the Naming Relation
1.3.1 The Naming of Object-Level Variables by Meta-Level Constants
1.3.2 Problems of Representing OL Variables by ML Variables
1.3.3 The Mapping of Constants and the Explicit Notification of Variables
1.4 The Naming of Sets of OL Clauses and the Meta-Predicate CLAUSE
1.5 The Representation of Relations Among OL Linguistic Objects
1.5.1 Representability and the Theory META
1.5.2 The Resolution Relation
1.5.3 Provability and Unprovability
1.6 Amalgamation of Object and Meta Languages
1.6.1 Inheritance Axioms
1.6.2 Scoping Axioms
1.7 Making META Self-Referent
1.8 Is Self-Reference Necessary?
2. Incorporation of Extended First-Order Features..... 41
2.1 Higher Order Features
2.2 Negation
3. Reflection..... 51
4. Conclusion..... 54

Chapter Two: Meta-Level Reasoning for Declarative Control..... 56
1. The Control Facilities of PROLOG..... 56
2. Control Through Re-Defined Interpreter..... 58
3. The Definition and Properties of S-DEMO..... 60
3.1 The Search Tree
3.2 Defining the Expansion Strategy of S-Demo
3.2.1 CHOOSE-NODE
3.2.2 CHOOSE-GOAL
3.2.3 CHOOSE-CLAUSE
4. Correctness of S-DEMO..... 70

5. What if User-Defined Choice-Predicates Are Non-..... 72
Deterministic
6. Examples of the Use of Choice-Predicates for .. 74
Defining Control
6.1 Clause-Preference
6.2 Goal-Preference
6.3 Lazy Co-Routining
7. Implementation Issues..... 76
8. The Efficiency of the Control Scheme..... 77

Chapter Three: An Application of Meta-level Reasoning - .. 80
Fault-Finding in Digital Circuits
1. The Set-Up of the Fault-Finding Process..... 80
2. Theories of Circuit Modules as Logic Programs..... 81
2.1 The Description of the Circuit Module (D)
2.2 The Truth-Tables (TB)
2.3 The General Laws of Logic Circuits (G)
3. Some Terminology..... 79
4. Hypothesising about the Location of Faults..... 87
5. Constructing the Hypothesis Set..... 91
6. Devising a Discriminating Input..... 96
7. Some Related Work..... 97
7.1 Comparison with Fault-Finder
8. Conclusion..... 99

Chapter Four: Comparison with Related Systems..... 101
1. Comparison with the System Developed in [Kowalski .. 101
1979] and [Bowen & Kowalski 1982]
2. Comparison with FOL..... 103
2.1 Description of the FOL system
2.1.1 The FOL Deductions
2.1.2 Syntactic Simplification
2.1.3 Semantic Attachments and Simulation Structures
2.1.4 LS Pairs or Theories
2.1.5 The LS Pair META
2.1.6 The Reflection Mechanism
2.1.7 Self-Reference
2.2 Comparison of FOL and MOL
2.2.1 Semantic Attachment vs Representability
2.2.2 Meta-Theory in FOL and MOL
2.2.3 Self-Reference in MOL and FOL
2.3 Summary of the Comparison between MOL and FOL
3. Comparison with PRESS..... 114
3.1 Meta-Level Knowledge in PRESS
3.2 Meta-Language Methodology of PRESS
3.3 Comparison with MOL
4. Comparison with the System Presented in [Konolige .. 116
1981]
4.1 The Naming Relation in KS
4.2 Important Meta-Predicates in KS
4.3 Comparison with MOL
4.3.1 The Question of Truth
4.3.2 The Question of Names
4.3.3 The Naming Relation Omega
4.3.4 The Standard Name Relation Eta
4.4 Conclusion

Chapter Five: Sets as Associative Data-Structures in Logic .. 125
Programming
1. Introduction..... 125

2. Lists and Sets..... 125
2.1 Lists
2.2 Sets
2.3 Lists and Sets in Prolog
3. The Definition of the Languages Used..... 127
4. The Axioms Defining Sets and Set Operations..... 128
4.1 The Set-Definition Axioms (SD)
4.2 The Existence Axioms (Ex)
4.3 The Membership Axiom (Fr)
4.4 The Equality Axioms (Eq)
4.5 The Clausal Form and Skolemisation
5. The Disadvantages of Representing Sets by Lists..... 131
5.1 Lists Are Accessed Serially
5.2 Hard to Share Common Subsets
5.3 Costly Set Operations
6. The One-To-One Nature of Functions in PROLOG..... 133
7. The Set Equality Axiom..... 134
8. Normal Models and Herbrand Models..... 135
9. The Alternative Treatment of the Set Axioms..... 136
9.1 Representing Sets by Numbers
9.2 The Relationship with Normal Models
10. The Definition of the Algebraic System..... 140
11. Some Implementation Points..... 144
11.1 The Data-Structures Chosen for the Constants in S
11.2 The Implementation of the Prime Code Function
11.3 The Alterations to the Prolog Interpreter
12. The Frame Problem..... 148
13. Conclusion..... 148

References..... 150

Appendix 1: The Naming Relation..... 153
Appendix 2: The Definition of NOT-DEMO..... 160
Appendix 3: The Proof of The Main Theorem in Chapter Five.... 163

1. Objectives of the Thesis

Logic programming has been developed as a methodology to address some of the problems of software engineering and artificial intelligence. The main tenet of logic programming is that formal logic, which can be used as a vehicle for the expression of problems, can also be used as a means of solving the problems so expressed. The mechanisms by which such solutions are obtained are specialised theorem proving engines. The solution is computed during the proof of the 'goal statement' from the theory, where the combination of the theory and the goal statement is a logical expression of the problem.

1.1 Declarative Semantics of Logic Programs

From a software engineering point of view, logic programming aims to provide a framework in which the programmer declaratively specifies the problem, rather than its solution through a particular algorithm. This makes the task of the programmer a more high-level, problem oriented activity. Furthermore, given the clear and unambiguous semantics of formal logic, this has the advantage that formal reasoning techinques in logic can be used to argue the correctness of the program with regard to a given specification/implementation. At the same time, formal program development techniques such as transformation can be used for deriving programs.

As for artificial intelligence, logic programming is meant to provide a vehicle for knowledge representation and reasoning. The clear formal semantics of logic and its well understood proof-theoretic properties make for a good foundation on which to build complex knowledge bases. The main attraction of the logic programming approach, as opposed to the general use of logic as a representation language, is that the inference engine can be the interpreter itself. This is in contrast with other programming

languages, e.g. LISP, were an elaborate, `one-off' reasoning module must be built from scratch by the knowledge engineer before he can build a knowledge base. Even when the limitations of the current logic programming systems forces the knowledge engineer to build his own reasoning system, his task is much easier in a logic programming environment, because he can `inherit' many of the system's facilities. For example, the expert system shell APES [Hammond 1983], which is a reasoning/communication suite of programs intended for use in expert system development, makes heavy use of the Prolog deduction facilities on which it is built. This has made APES a simpler, and more powerful, expert system shell than most other shells written in other languages.

1.2 Procedural Semantics

The field of logic programming only took off when it was discovered that a subset of first order predicate calculus, namely Horn Clause logic, has a procedural semantics as well as its normal declarative semantics [Kowalski 1974]. This meant that Horn Clause logic could be used not only as a specification language, but also as a programming language. The attempts to put this theoretical discovery into practice led to the programming language Prolog.

But Horn Clause logic has many limitations, and in an attempt to overcome these limitations the designers of Prolog added many 'non-logical' features. We call these features non-logical because their properties are dependent on the procedural behaviour of the program, and they violate the declarative semantics of logic. The inclusion of these non-logical features into Prolog, while important from a practical point of view, led to a curious situation: in many instances, Prolog programs are messier than programs written in procedural languages. Obviously, this has led to much criticism of Prolog and, by association, logic programming.

The essence of the work presented in this thesis is to extend Horn Clause

programming within the strict confines of logical semantics. The objectives of this extension have been twofold:

-to provide a richer, more expressive language and a more accommodating environment, particularly for such AI applications as expert systems, knowledge representation, induction etc.,

-to eliminate the `dirty' features of Prolog by providing alternative, clean, logical features. Generally, these features are more powerful and expressive than those they replace.

Below we mention some of the shortcomings of Horn-Clause programming, and the way we propose to deal with them in this thesis. Prolog's answers to these shortcomings, when they exist, are mentioned by way of comparison.

1.3 Meta Linguistic Extensions

Horn Clause logic does not include any meta-level facilities for access to its programs, goals, proofs etc. as data-structures, and for manipulation of these data-structures. Yet such facilities are of great importance in many types of programs. For example, in an expert system shell, most of the reasoning is at meta-level (is this rule applicable to this case? Describe how you arrived at this conclusion. What would happen if this fact were replaced with another etc.) In Prolog, a limited meta-level facility is provided through `meta-variables' and system predicates such as VAR, BOUNDTO, CLAUSE etc. But these features are both non-logical, in that they do not have a declarative semantics, and limited in scope. In this thesis, a systematic amalgamation of object language and meta-language, based on [Kowalski 1979] and [Kowalski and Bowen 1982] has been attempted. The main features of the scheme developed here are as follows:

- The language chosen for the basis of the amalgamation is a subset of Horn-Clause logic, namely list-based Horn Clause logic. The choice of this

subset is to simplify the various constructions in the system.

-The interface between object language and meta-language (the naming relation) is optimised for simplicity and expressiveness of mixed (object+meta level) sentences.

-The system is self-referential at theory level, i.e. it is possible to refer to a given theory from within itself. To achieve self-reference with finite theories and sentences, theoretical constructs called inheritance and scoping axioms are used.

-Reflection is used as a means of evaluating object level queries without efficiency overheads.

1.4 Negation

There is no negation in Horn Clause logic. In Prolog, negation-as-failure has been incorporated as a limited form of negation. While under certain conditions, a declarative reading can be ascribed to negation as failure, in practice Prolog programs frequently violate these conditions, making negation-as-failure a non-logical feature of Prolog. In our scheme, an alternative to negation-as-failure is presented which does not suffer from the same problems. This scheme is based on self-reference and the unprovability predicate.

1.5 Control of the Interpreter's Search Strategy

Horn Clause logic does not favour any particular search strategy for proving goals from programs. In most versions of Prolog, the simple `left to right, top down' strategy has been supplied as the only search strategy available. Some versions of Prolog, e.g. IC prolog, have more sophisticated control schemes such as co-routining. However, Prolog in general does not allow the programmer to specify his own control strategy declaratively. In this thesis a scheme has been developed to put this

facility at the disposal of the programmer.

1.6 Associative Data-Structures

In Horn Clause logic, the only data-structures provided are recursive data structures. Non-recursive data structures, such as arrays, sets etc. are not provided. This is a limitation which arises out of the theory on which Horn Clause programming is based. To avert this problem, Prolog allows the use of the program itself as an associative data-structure, by allowing the addition and deletion of clauses at run time. From the point of view of programming methodology, this is a disaster, because it amounts to changing the program at run time, making it extremely difficult to understand or validate such a program. In the thesis a scheme has been developed for handling sets as data-structures. Other non-recursive data-structures, such as arrays, can be built up on top of sets.

2 Outline of the Thesis

The thesis is in five chapters. It contains two related, yet distinct parts: chapters one to four are dedicated to the development and appraisal of an amalgamated (meta-language + object-language) logic programming system, while chapter five is devoted to the theoretical and practical aspects of incorporating sets into Horn-Clause logic programming. The theoretical frameworks of these two parts have not been integrated, though such an integration should not pose any serious difficulties.

Below we present a brief description of the contents of each chapter.

2.1 Chapter one is devoted to the development of the object-language/meta-language amalgamated system. The system is built up in the following stages:

a- the syntax of the base language is defined, and the reasons for adopting this syntax are explained.

b- the naming relation for mapping the constructs of the object language to the data structures of the meta-language is described.

c- self reference is discussed and the means of achieving self-reference (inheritance and scoping axioms) are elaborated.

e- the utility of the self-referential system is discussed. It is shown that the self-referential system can be used, among other things, to implement:

-`clean' negation as failure;

-generalised predicates, similar to higher order functions in functional languages;

-higher order features

f- The idea of reflection is introduced, being theoretically based on representability axioms. The use of the reflection mechanism for efficient execution of object-level programs is discussed.

2.2 Chapter two is concerned with the use of meta-linguistic knowledge to control the execution of object-level programs.

A definition of DEMO, the provability predicate is given. This definition is used in the development of a methodology for controlling the execution of object level programs. This methodology uses `upward reflection' for executing user-defined sub-modules of the provability predicate. The control scheme is compared with that developed by [Gallaire and Lasser 1981] and some examples of its use are discussed.

2.3 Chapter three is a case study of the application of the amalgamated system for the problem of fault-finding in logic circuits. The fault-finding program uses theory induction and a variant of the hypothesise-and-test paradigm for locating faults in combinatorial logic circuits. This application illustrates the utility of some of the concepts developed in the previous chapters.

2.4 Chapter four is a comparison of the amalgamated system developed in this thesis with some other prominent system which have had similar goals. The systems with which we have made comparisons are the following:

- FOL [Weyhrauch 1980]
- The system proposed in [Kowalski and Bowen 1982]
- PRESS [Bundy & Welham 1981]
- The system proposed in [Konolige 1980]

2.5 Chapter five is concerned with the incorporation of sets as data structures into Horn-Clause logic programming. After discussing the theoretical problems which have stood in the way of having sets as data structures, a technique is developed for this purpose. Initially, a number of set axioms are presented for defining sets and set operations. These axioms are not in Horn-Clause form; in particular, they include existential quantifiers. Also, these axioms include equality axioms. It is argued that the standard way of converting these axioms to clausal form will lead to great practical deficiencies. Instead, using an algebraic scheme, an infinite set of assertions is defined, and it is proved that these assertions can be used in place of the set definition axioms in any refutation proof. The practical implications of this, and some implementation issues, are discussed.

Chapter One

General Framework of the Amalgamated System

In this chapter, we will discuss the main characteristics and some of the applications of the amalgamated system proposed in this thesis. The three following sections are concerned with the development of the amalgamated (OL+ML) system, which we have called MOL, the applications of the system for representing new features such as clean negation as failure and generalised predicates (similar to higher order functions in functional languages), and the reflection rule and its use for evaluating meta-level predicates.

1. The Development of the Amalgamated System

In classical mathematical logic, the term meta-language is used to denote the language in which a certain logical theory is discussed. For example, concepts such as logical entailment, contradiction, derivation etc. belong to the meta-language of first order logic.

Ordinarily, the meta-language of logic is natural language. It is possible, however, to use logic itself as the meta-language of logic. In the context of logic programming, this is quite desirable, because we have already got the machinery for dealing with logic, and this machinery can be used for dealing with the meta-language too.

In the rest of this section we shall be discuss the following issues:

-Linguistic Objects as Data Structures

Since the meta-language is concerned with manipulating linguistic objects, it must have data-structures which represent them. For example, if we want to talk about a sentence of the object language, say S, then we must have a representation of S as a data-structure of the meta-language.

Thus, the first step in defining the interface between the meta-language and the object language is to define this mapping, i.e. the mapping from the linguistic objects of the object-language to the data-structures of the meta-language. We shall call this mapping, the naming-relation. (From now on we shall call the object language OL and the meta-language ML)

-Representation of Relations on OL Objects

Having defined the naming relation, in order to talk about the properties of the OL we need meta-level predicates which represent important relations among the linguistic objects of the OL. Such relations include provability, finite unprovability, resolution etc. The axioms of the meta theory must be such that these meta-predicates correctly represent the relations concerned.

-Amalgamation and Self-Reference

Since the ML and OL have the same syntax and semantics, it is possible to amalgamate them into one language. This is done through scoping axioms and 'inheritance axioms', which enable object theories to inherit the data-structures of the meta-theory.

1-1. The Base Language

One of our objectives is to have a meta-language (Horn Clause Logic) which is the same as the object language (Horn Clause Logic). In fact, for the purpose of amalgamation, the two languages must be syntactically and semantically identical. We shall call this language the base language. This requirement imposes the following constraints on the language:

a- The (meta) language must have rich enough data structures to represent linguistic objects of the OL,

b- The (object) language must have a simple syntax, to make the representation of its sentences etc. as simple as possible.

To satisfy these constraints, we have chosen a list-based Horn-Clause syntax for our base language. The list-based syntax is derived from micro-Prolog [Mc Cabe 1980]. Its main attraction is that it is very easy to map sentences of the language, i.e. clauses, to the main data-structures of the language, i.e. lists.

2-1-1 The Syntax of The Base Language

The syntax is based on that of micro-Prolog. Essentially,

-a sentence is a clause or a goal-list,

-a clause is a list of atoms, where the first atom is the head and the rest are the atoms of the body,

-a goal-list is a list of atoms,

-an atom is a list, the first element of which is a predicate symbol, and the rest of which are the terms which are the arguments of the atom,

-a term is a constant, a variable or a list of terms. Variable symbols are lower-case identifiers. Constant symbols are upper-case identifiers and underlined identifiers (be they lower case or upper case). Predicate symbols are upper-case identifiers.

Example: x, john and zzz are variable symbols,
X, JOHN, zzz and MORTAL are constant symbols,
P, HUMAN and MORTAL are predicate symbols.

-a list of terms is written as (head | tail), where the 'head' is a term

and 'tail' is a list. The list (a|(b|..n|())) is also written as (a b .. n)

Below are some example clauses and goal-lists.

```
((APPEND () x x))

((APPEND (x|y) z (x|z1))
         (APPEND y z z1))

((MORTAL x)
     (HUMAN x))

((HUMAN SOCRATES))
```

1-1-1. Some Points about the Base Language

The base language is a restricted Horn Clause language. The restriction is that it has only one function symbol, the list constructor function. Since it is a Horn Clause language, it has the same semantics as other Horn Clause languages.

The question arises: why did we impose this restriction on the language, and will it restrict the applicability of the language?

The reason we chose this language is to simplify the naming relation (which is defined in the next section). The absence of function symbols apart from the list constructor symbol means we do not have to worry about the naming of functions in general.

As to whether this restriction will hamper the applicability of the language, practical experience with micro-PROLOG shows that this is not so. Whenever we need to use a construct such as f(a1,a2..an) (say), we simply

use the list (f a1 a2 ..an) instead. Practice has proved this simple technique to be quite effective. Paradoxically this restriction actually increases expressive power, because now we can have variables in the first argument position, corresponding to variable function symbols.

1-2. The Naming Relation

As mentioned earlier, we need to define a naming relation to map the linguistic objects of the OL to the data-structures of the ML.

The linguistic objects in question are the following:

-Terms of the OL (terms can only be named in the context of a sentence. This point is elaborated later)

-Sentences of the OL

-Finite Sets of sentences (theories) of OL

-Bindings(i.e. substitutions), proof steps, proofs, search spaces etc.

As can be seen, our notion of 'language' is a rather loose one, and encompasses more than just sentences over an alphabet. Be that as it may, we need to define a formal mapping from the above objects to the data-structures of the meta-language.

The purpose of the naming relation is to allow us to refer, in the meta-language, to the relevant constructs of the object-language. The naming relation must have the following properties:

-It must be unambiguous. Thus if the names of two objects are the same, and they appear in the same context, those objects must be equivalent. The qualification as to the context of the names is important, for it determines the type of object that has been named.

-It must be transparent. Thus the name of an object must be as similar to the object as possible. This is to make mixed sentences more readable, and

to aid implementation.

We define the naming relation piecemeal, as a collection of naming functions over different sections of the language. We shall describe, successively, the naming relation for sentences, and for sets of sentences. For a more formal definition of the naming relation see Appendix1.

1-2-1. The Naming Function for Sentences

Let us call a term that names a sentence a 'description' of the sentence. Since the syntax of sentences is list-based, it is quite simple to map them to ground terms, which are lists. Thus the list-structure of the description of a sentence can be very similar to the structure of the sentence itself.

For reasons that will be discussed later, however, we have decided to include in the description of a sentence a list of the symbols which represent the variables in the sentence. We call this list the varlist of the sentence. Thus, the only difference between the structure of the description and that of the sentence is that the first element of the description is a list of the meta-level constant symbols standing for the object-level variables in the sentence. This technique is also used in [Bowen & Weinberg 1985] and sigma-Prolog [Mc.Cabe 1984] for describing sentences.

Before we go on to describe in more detail the rules for constructing descriptions, let us have a few examples:

Example:

The clause

((MORTAL x)

(HUMAN x))

is named by the term

((T)(MORTAL T)(HUMAN T))

Where varlist (T) at the head of the description is the list of meta-level constant symbols naming object-level variables in the clause.

Note: x is a variable symbol (because it is lower case without underlining)
T is a constant symbol (because it is upper case)
T names x

A description of the clause

((APPEND (x|y) z (x|z1))
(APPEND y z z1))

is the term

((x y z z1)(APPEND (x|y) z (x|z1))(APPEND y z z1))

Note: x is a constant symbol (because it is underlined)

We do not intend to suggest, however, by this notation that underlining is a kind of function which given a variable returns its name. Instead, we sometimes use underlining of variable names as a convenient way of choosing the constant that names that variable. Obviously, this is not always possible, as when the underlined constant already occurs in the sentence.

((HUMAN SOCRATES))

is described by the term

(()(HUMAN SOCRATES))

Now we can present the rule for constructing the description of a sentence S:

I- Replace all the variables in S with constants which do not occur in S. Call the resulting sentence S1. Call the list of constants used for this purpose V. Note that every different variable must be replaced with a different constant, but variables of the same name within the sentence must be replaced with the same constant. There is no restriction on the constants which can be used for this purpose other than the one mentioned here.

II-Replace all the predicate symbols in S1 with constants whose identifier is the underlined version of the identifier of the predicate. Call the resulting term S2. For example, the predicate symbol APPEND will be replaced with the constant symbol APPEND

III-Construct the term (V|S2) as the description of the sentence S.

Does this naming function satisfy the objectives we set out earlier? Is it unambiguous and transparent?

Let us call the name of the sentence a name(a). In order to show that it is unambiguous, we have to show that

if name(a)=name(b), then a is equivalent to b. (Proof in appendix1)

Equivalent in this case means identical up to re-naming of variables.

Note that the definition of the naming function given above is for naming sentences of the base language, i.e. (implicitly) closed formulae. Terms, too, are named by terms, in the way defined in Appendix1. It is the context which determines whether a given term is the name of a sentence, or the name of a term, or neither. The context is determined by looking at where the term appears. For example, if a term appears as an argument of a meta-predicate where we know the name of a sentence should appear, (e.g. as the second argument of DEMO) then we treat it as the name of a sentence (see point 1-3-4 below for a discussion of the reason why we did not choose to have an explicit typing scheme for determining the type of object that a term is naming)

1-3. Some Points about the Naming Relation

The naming relation described so far, although very simple, has all the properties we need. It is very similar to the mapping used by micro-PROLOG for representing clauses and goal-lists, except that it uses constants to represent variables. This issue is of fundamental importance, as explained below.

Also of practical significance is the decision to map constants to themselves and to have an explicit varlist signify the meta-level constants used for representing object-level variables.

1-3-1 The Naming of Object-Level Variables by Meta-Level Constants

Almost all Prolog implementations (with the exception of sigma-Prolog) map object level variables to meta-level variables. For example, in micro-Prolog, when the primitive CL is used to access (an object-level) clause, the data structure that is returned has (meta-level) variables standing for the variables in the clause. Thus, if we run the goal

```
?((CL z1))
```

when the program contains the clause

```
((HUMAN x)
   (MORTAL x))
```

then the goal will succeed, and z1 will be bound to the term

```
((HUMAN x)(MORTAL x))
```

where x is a variable. Similarly, if we want to use the so called meta-variable facility to run a goal, then in the data-structure that represents the goal meta-level variables must stand for object-level variables. (see example below)

The reason for adopting this method of representing object-level variables in Prolog is convenience in implementation: it is possible to simply 'inherit' the unification facilities of the meta-language.

The price that has to be paid for this convenience, however, is great. The use of meta-level variables to represent object-level variables is the source of considerable semantic and practical problems.

Example:
In micro-Prolog, we can define a predicate called RUN, which runs an object level goal-list, as follows:

```
((RUN (goal|rest-goals))
              goal
              (RUN rest-goals))

((RUN ()))
```

In the above example, the variable 'goal' is a meta-variable: at run time, it is instantiated to a list representing a goal, and is run as a goal. The list '(goal|rest-goals)' is the goal list we wish to run. As mentioned above, when we want to use RUN, the argument standing for the goal list must have variables representing the object level variables. For example, if we want to evaluate the goal (APPEND x (a b) (c a b)) using the predicate run we must execute the following goal:

```
?((RUN ((APPEND x (a b) (c a b))))
```

This is an example of the meta-level proof predicate (in this case RUN) 'inheriting' the unification mechanism of the interpreter. Had we used the naming relation defined in this thesis for naming clauses, the definition of RUN would have needed to have an explicit UNIFY subgoal.

1-3-2 -Problems of Representing OL Variables by ML Variables

a- Semantic Problems

Consider the Prolog 'evaluable' predicate (VAR x): it succeeds if x is a variable. This predicate is quite important when writing meta-level programs (such as expert system shells). It can be used, for instance, to delay the goals which have most unbound variables. But (VAR x) is not a logical predicate at all, because its behaviour is critically dependent on the sequence of execution of subgoals. In fact, strictly speaking, it has no logical meaning. This is because, logically speaking, if (VAR x) succeeds, it means that for some value of x, (VAR x) follows from the program. Let SOCRATES be the value of x for which (VAR x) follows from the program. Thus, (VAR SOCRATES) also follows from the program, i.e. it should also succeed. But SOCRATES is a constant, and (VAR SOCRATES) fails. Thus there can be no logical meaning to VAR. Its use in a Prolog program leads to the violation of the logical semantics of the program.

b- Practical Problems:

Consider the predicate (BOUNDTO x FRED): it succeeds if the (object-level) variable x is bound to the constant FRED. Its use in a meta-level program, however, leads to the following frustrating situation: if x is already bound to FRED, then it succeeds, which is OK; if x is bound to some other constant, then it fails, which is again OK; but if x is not bound to anything, then this predicate actually binds it to FRED, which is not what we want at all. (In practice, a call to BOUNDTO is explicitly or implicitly preceded by a call to VAR, to ensure that this situation does not arise)

Predicates such as BOUNDTO are essential when writing meta-level programs. The representation of object-level variables by meta-level variables means that the programmer has to be very careful not to 'accidentally' bind his object level variables through such 'test' predicates.

When meta-level constants are used for representing object-level variables, the above problems do not arise. Thus, when making meta-level analysis of an object-level goal, the object-level variables cannot get bound, because they are meta-level constants. Similarly, to see whether a symbol is an object level variable or not, we look in the list of constants representing object-level variables, an operation which yields the same result whatever the order of execution of goals.

1-3-3 The Mapping of Constants and the Explicit Notification of Variables

The naming functions for mapping the ground terms and sentences of the OL to ground terms of the ML were chosen for the following reasons:

a-to avoid undue complexity, we wanted a system in which if D is the description of a sentence, and D' is the description of D, then D' is almost the same as D. This matter is of particular significance in a self-referential system, in which there might be an indefinite number of levels.

b-We wanted to avoid the confusion between the following two

(typical)cases: whether the term (R T) describes the atom (R x), where x is a variable, or (R T), when x is a constant.

The choice made for representing constants and variables satisfies the above constraints. Thus, if D describes a clause, the description of D would be D itself. On the other hand, a description of the atom (R x) in the context of a goal-list or a clause such as (...(R x) ...) is

((T)(...(R T)...)

while a description of (R T) in the context of a clause or a goal-list is

(()(...(R T)...),

overcoming the confusion mentioned above. Furthermore, the use of an explicit varlist enhances the expressiveness of the language.

1-3-4 Why Not Use Explicit Types for Distinguishing Names of OL Objects

To avoid ambiguity, for any term that appears in a program we must be able to answer the questions:

-What type of a term it is, i.e. whether or not it is a name of an OL object,

-If it is a name, then what is the object that it is naming. (From now on we shall call the object that is named by a term the referent of the term).

We mentioned before that it is the context which decides whether a term T of the meta-language is the name of an object-level term, the name of an object-level sentence, or neither. It is also the context which determines what the referent of a term is. In other words, we use a kind of implicit typing for disambiguating the types of terms. (see Appendix1 for the details of how to determine the type of a term)

Example: In the following program, what is the type of the constant MARY, and what is its referent, if anything?

```
P:-
((HAPPY x)
   (NICE x)
   (DEMO P
      (()(GOOD x)
         (DEMO P (()(RICH x)(LIKES MARY x))

      )
   ))


((GOOD JOHN))
((RICH JOHN))
-:P
```

Answer:

the term

Q= (()(GOOD x)(DEMO P (()(RICH x) (LIKES MARY x))))

occurs as the second argument of the DEMO predicate, so it is the name of a sentence. Since MARY is a subterm of Q, it is the name of a term in the context of the varlist (). Since MARY is a constant, it is the name of itself, i.e. its referent is the constant MARY.

Now we have not said what type of a constant it is that MARY is naming; it might be an ordinary constant, or itself the name of another constant. If we want to find out which one of these cases obtains, we have to know the sentence which Q names, i.e. the referent of Q. Let us call this sentence

S. We know that the MARY in Q is naming a constant in S (whose name is also MARY). To avoid confusion, let us call the MARY occuring in S MARY1. But what is MARY1 itself?

(We know that MARY=MARY1, but in order to refer to the two unambiguously, we use the subscript)

S is a sentence such as

((GOOD TERM) (DEMO P (()(RICH TERM)(LIKES MARY1 TERM))

where TERM is some term. (Notice that since Q has an uninstantiated variable in it, it is like a name schema)

Now, in S MARY1 occurs in a term R which is the second argument of DEMO. So MARY1 is itself the name of a term. Again, since constant name themselves, the referent of MARY1 is the constant MARY1. Let us refer to the referent of MARY1 as MARY2. (again we know that MARY2=MARY1=MARY)

What is MARY2?

the referent of R, which is the term in which MARY1 occurs, is the sentence

((RICH TERM)(LIKES MARY2 TERM))

In this sentence, MARY2 is an ordinary constant, which is not a name.

Thus the MARY in Q is the name of the name of a constant.

End of Example.

The reason why we did not use an explicit typing scheme, (similar, for example, to that adopted in [Konolige 1981]) is that when there are more than two levels of meta-language, such typing greatly complicates the naming scheme. Thus, for example, one needs special sorts for naming

constants, variables, predicates, functions, names of constants, names of variables, names of predicates, names of functions, names of names of constants....

The fact is that in an amalgamated system there is an indefinite number of levels of meta-language, and this means that explicit typing schemes are counter-productive. That is why we have chosen the implicit typing scheme mentioned above.

1-4. the Naming of Sets of OL Clauses and the Meta-Predicate CLAUSE

So far, we have discussed how to describe object level clauses by meta-level terms. But we also need to describe sets of object level clauses, i.e. object level programs or theories.

Note: We sometimes use <clause> or <sentence> etc. to mean 'some clause' or 'some sentence' etc. Angled brackets are not part of the syntax of the language; we use them to save typing.

A program P is represented by a set of clauses of the following form:

((CLAUSE P <cl1>))
((CLAUSE P <cl2>))
...

where <cl1>, <cl2> .. are descriptions of the clauses in the program P. We say that the above assertions 'describe' the program P.

Furthermore, with every predicate symbol is associated a number, which specifies the number of clauses which have that predicate symbol at their head. The description of programs contains assertions of the type

((CARDINAL P R n))

where n is the number of clauses in program P which have the predicate R at their head.

Example:

The following is a description of the program P which consists of the clauses of the APPEND relation:

```
((CLAUSE P ((x)(APPEND () x x))))
((CLAUSE P ((x y z z1)(APPEND (x|y) z (x|z1))(APPEND y z z1))))
((CARDINAL P APPEND 2))
```

To make life easier, we use the following convention: The description of the program P above can also be written in the following way:

```
P:-
((APPEND () x x))
((APPEND (x|y) z (x|z1))(APPEND y z z1))))
-:P
```

Note that the above abbreviation is a purely syntactic variation. In all the theoretical discussion that follows we shall use the CLAUSE description of programs.

1-5 The Representation of Relations among OL Linguistic Objects

In order to write useful meta-level programs, we need to have at our disposal meta-level predicates which represent important proof-theoretic relationships among OL linguistic objects.

We shall first discuss representability, which is the concept used for pinning down the properties of these predicates.

1-5-1 Representability and the theory META

The concept of representability, as set out in [Kowalski 1979] and [Bowen and Kowalski 1982] formalises the notion that a predicate, in the context of a theory, can represent a relation on a domain:

Def: In the context of the theory T, the predicate P represents the relation R on the domain D iff

a- there is a naming relation mapping the objects of D to the ground terms of (the language of) T,

b- for all i1, i2, .. belonging to D, the tuple (i1 i2 .. in) belongs to the relation R iff T |- (P i1' i2' .. in')

where i1' , i2' .. are the names of i1, i2 ...

For example, in the case of integers, the domain under consideration would be the integers and the predicates of interest would be ADD, MULTIPLY etc. These predicates would represent the addition, multiplication and other relations of interest on numbers.

In our case, we want to represent relations such as provability, resolution etc. on the domain of the linguistic objects of OL. The naming relation is the one we have developed so far, and the theory in the context of which our meta-predicates represent the above relations is called META.

Thus, for example, since DEMO is supposed to represent provability, we should have

META |- (DEMO '<prog>' '<goal>') iff <prog> |- <goal>

Where '<prog>' is the description of <prog> and '<goal>' is the description of <goal>.

Now we can go on to describe some of the meta-predicates and the relations they represent.

1-5-2 The Resolution Relation

The resolution principle is a derivation rule. For our purposes, it can be thought of as a relation among three sentences.

We use the following names to identify the arguments of the resolution relation:

- the first sentence,
- the second sentence,
- the resolvent.

Logically, the resolvent is a logical consequence of the first sentence and the second sentence.

We use the three place predicate

(RESOLVE <first sentence> <second sentence> <resolvent>)

to represent the resolution relation. (The top level of the definition of RESOLVE is given in appendix1)

1-5-3 Provability and Unprovability

Provability is the central concept of logic. It is a relation between theories and sentences of the OL, where theories are sets of sentences.

The meta-predicate DEMO represents provability.

META |- (DEMO <theory> <goal>) iff the sentence <goal> is provable from <theory>. (<goal> is the name of <goal> etc.)

Unprovability is another relation among theories and sentences of the OL.

Its definition is simply the opposite of that of provability. The meta-predicate NOT-DEMO represents finite unprovability, i.e. unprovability when it is finitely demonstrable.

The top-level definition of DEMO is

```
((DEMO program goal-list)
        (CLAUSE program  clause)
        (RESOLVE goal-list clause new-goal-list)
        (DEMO program new-goal-list))


((DEMO program goal-list)
        (EMPTY goal-list))
```

Comment:to prove goal-list from program, select a clause from the program. Then resolve the goal-list with the selected clause, to get new-goal-list as the resolvent. Then prove new-goal-list.

The top-level definition of NOT-DEMO is

```
((NOT-DEMO prog goal-list)
     (CHOOSE-GOAL goal-list goal)
     (POSSIBLE-CLAUSES prog goal clause-list)
     (FAIL-ALL prog goal clause-list))
```

Comment: The goal-list is not provable from prog if it includes a goal which is a failure goal. To see if a goal is a failure goal, we get a list of all the clauses which can resolve with it. Then we show that the attempt to solve the goal with every one of these clauses fails.

```
((FAIL-ALL prog goal ()))
```

```
((FAIL-ALL prog goal (clause|rest))
     (FAILS prog goal clause)
     (FAIL-ALL prog goal rest))
```

Comment: FAIL-ALL is recursively defined in terms of FAILS. The definition of FAILS is given in Appendix 1.

Notice that NOT-DEMO above represents finite failure to prove. Thus, if it is possible to show, in finite time, that the program will fail, NOT-DEMO will show it. If the finite proof of failure is not possible, then NOT-DEMO will not terminate.

The full definition of NOT-DEMO is given in appendix2.

1-6. Amalgamation of Object and Meta Languages

So far, we have kept the meta-theory, i.e. META, and the object theory as separate entities. To summarise the previous section,

- META is a collection of clauses that define the meta-predicates DEMO, RESOLVE etc. in such a way that they correctly represent the corresponding relations on the linguistic objects of OL,

- META also includes a collection of assertions of the form

```
((CLAUSE <prog1> ((...))))
((CLAUSE <prog1> ((...))))
  .
  .

((CLAUSE <prog2> ((...))))
((CLAUSE <prog2> ((...))))
  .
```

((CARDINAL <prog1> R n))

.

which collectively describe the object theories <prog1>, <prog2> etc.

But, as we have defined it so far, we have not allowed for the case where the object theories (e.g. <prog1>,<prog2>) might need to make reference to the clauses and assertions of META. Furthermore, we do not have direct or indirect self-reference.

What we are going to develop is an amalgamated system, in which object theories can have access to the clauses and assertions of META. Since these assertions of META include the CLAUSE assertions used for defining <prog1>,<prog2> etc., this would make the object level programs self-referent.

In order to achieve amalgamation, we use 'inheritance axioms' and 'scoping axioms'. These are explained in the following sections.

1-6-1 Inheritance Axioms

An inheritance axiom is an axiom of the type

((CLAUSE P x)
    (CLAUSE Q x)).

It specifies that the the object level program P contains the object level program Q, i.e. the set of clauses of P is a superset of the set of clauses of Q. (Note: a ground term is one in which no variables occur)

1-6-2 Scoping Axioms

A scoping axiom is an axiom of the type

((CLAUSE P (()(R x)))

(R x))

it says that if the ground assertion (R T) (say) can be proved from META, then its description is part of the description of the object program P, i.e. it is assumed to be part of the program P.

Example:

Let the definition of the set S be part of META, i.e. META includes assertions of the form

((MEMBER S T1))

((MEMBER S T2))

etc.

Then the scoping axiom

((CLAUSE P (()(MEMBER S x)))

(MEMBER S x))

allows the program P to contain the definition of S. This is because, by virtue of the above scoping axiom, the description of the assertions defining S becomes part of the description of the program P.

Now, as mentioned earlier, object-level programs are represented using CLAUSE assertions. So what happens if we replace MEMBER with CLAUSE in the above scoping axiom?

((CLAUSE P (()(CLAUSE Q x)))

(CLAUSE Q x))

The above scoping axioms say that the description of the program Q is part of the program P, i.e. that the object-level program P has a

collection of CLAUSE assertions which describe Q. Notice that the program P is not a superset of the program Q; rather the program P contains a description of the program Q. In other words, the program P is at meta-level with respect to the program Q.

Letting P and Q be the same program, we can make P self-referent:

```
((CLAUSE P (()(CLAUSE P x)))
                    (CLAUSE P x))
```

i.e. the (object level) program P has a description of itself.

We can generalise the above to apply to all object level programs by letting the program name be a variable:

```
((CLAUSE p (()(CLAUSE p x)))
                    (CLAUSE p x))
```

There is also a scoping axiom connected with CARDINAL. It is:

```
((CLAUSE p (()(CARDINAL p r x))
                    (CARDINAL p r x))
```

1-7 Making META Self-Referent

The inheritance and scoping axioms refer to the CLAUSE assertions that describe the object level programs. In order to make them apply to META, therefore, we must add to META, CLAUSE assertions which describe META itself. Adding the general inheritance and scoping axioms to META, followed by adding to META a description of itself, will make our system self-referent, as explained below.

Thus far,

a- META is a collection of Horn-Clauses which define the meta-predicates DEMO, RESOLVE etc. in such a way that they correctly represent the corresponding relations on the domain of OL objects. It also includes CLAUSE assertions and Cardinal assertions which describe the object-level programs.

b- META includes the general scoping axioms

((CLAUSE p (()(CLAUSE p x)))
(CLAUSE p x))

and

((CLAUSE p (()(CARDINAL p r x))
(CARDINAL p r x))

and the inheritance axiom

((CLAUSE p x)
(CLAUSE META x))

c- META includes the CLAUSE and CARDINAL assertions which describe all its clauses, including the scoping and inheritance axioms.

d- None of the object programs should have clauses which have any of the meta-predicates at their head (for otherwise the correct representability of the meta-predicates can be impaired)

Now META is self-referent in the following sense: it includes a description of META, and its description has a description of the description of META etc. Furthermore, all object level programs are also self-referent, and they have access to the predicates defined in META. In other words, we have an amalgamated system. Notice that we have achieved an indefinite

number of levels without making the program META infinite. In fact, META can be regarded as an ordinary, finite pure PROLOG program without any frills. The qualities of self-reference, correct-representation etc. which we have sought to achieve are abstract, high level concepts which we can ascribe to META. These qualities do not change the ordinary Horn Clause semantics of the sentences of META in any way: rather, they build on these semantics to achieve a higher level of organisation.

1-8 Is Self-Reference Necessary?

We seem to have gone into a great deal of trouble to devise an amalgamated system which allows self-reference. But is this warranted?

The main reason for incorporating self-reference into the system is to make the language more expressive. On the one hand, self-reference allows us to express, within the confines of the restricted Horn Clause language, constructs such as semantically clean negation-as-failure etc. On the other, it gives us the power to express things which would otherwise need a higher-order logic.

It is worth emphasising, at this point, that the main argument here is that self-reference gives us this extra expressiveness without going outside the confines of restricted Horn-Clause logic, thereby avoiding the extra complexity involved in dealing with a more extensive first order language, or a second order language.

It is also important to stress that the scheme devised above for achieving self reference, i.e. the use of scoping and inheritance axioms etc. is central to the whole issue of having this extra expressiveness. To have self-reference without going outside the confines of a first-order, finitary language, the above scheme or something equivalent to it is needed.

1-9 The Consistency and Correctness of MOL

Self-referent systems can be prone to inconsistency and paradoxes such as the Liar's Paradox. It is therefore necessary to argue why we think MOL does not suffer from such problems.

The following points are relevant to this issue:

a- Since META is a set of Horn Clauses which does not include negative sentences, it can not be inconsistent, in the sense that the empty clause can not be derived from it. (Every inconsistent set of clauses must contain the empty clause or at least one negative clause.)

b- All object-theories in MOL, too, are sets of Horn Clauses. So they can not be inconsistent either.

c- As well as consistency, we must know that the meta-predicates of META in some sense have the properties which we intuitively ascribe to them. This notion is what we have earlier defined as representability. Thus as long as the meta-predicates in question correctly represent the relations of interest on the domain of OL objects, we should be all right. Although we have not proved that our definitions do correctly represent the relations in question, the point is that it is always possible to write a correct provability predicate etc. Thus, even if we had not given any programs for DEMO, NOT-DEMO etc., all the points made in this chapter would be valid as long as one could give such a definition. And there is no doubt that this is possible.

The fact that we have prohibited the the inclusion in our object-theories of any clause with the above meta-predicates at their head and the fact that all object-theories inherit the definition of these meta-predicates from META means that our meta-predicates are correct not only in the context of META, but in the context of any object theory.

2-Incorporation of Extended First-Order Features

In this section we show how self-reference can be used to enhance the expressiveness of the language.

2-1 Higher Order Features

In knowledge representation, as in many other applications, it is often necessary to have variables that range over predicate symbols. Traditionally, the way to achieve this has been to use higher-order languages, i.e. languages which allow quantification over predicates.

The complexities involved with higher-order languages, however, make their adoption as logic programming/knowledge representation languages problematic. By using self-reference, we can have the best of both worlds: we can have the expressive power of the higher-order languages, without the extra complexity that they incur on the implementation.

Example 1 - The Red Ball

Consider the following sentences:

- The ball is red;
- Red is a colour;
- John is happy if the colour of the ball is red

A natural way to represent the first two sentences is:

((RED BALL))

((COLOUR RED))

But representing the sentences in this way, where in one of them RED is a predicate and in the other an argument, makes the third sentence a second order sentence, unless we use self-reference.

```
((HAPPY JOHN)
    (COLOUR c)
    (DEMO B (()(c BALL)))
```

Where B is the name of the program containing these clauses. In our notation,

```
B:-

((RED BALL))

((COLOUR RED))

((HAPPY JOHN)
     (COLOUR c)
     (DEMO B (()(c BALL)))

-:B
```

Of course, we could have formulated the two sentences above in such a way as to avoid the second order - or self referential - formulation, for example by having (COLOUR BALL RED) as a two argument predicate. But in the context of a large knowledge base, the distinction as to whether RED should be a predicate or an argument, which is forced on us by this formulation, is arbitrary and harmful. Self-reference allows us to avoid this distinction.

Below we present a simplified trace of the evaluation of the query 'is John happy?'

```
<-(DEMO B (()(HAPPY JOHN)))
```

This goal is run in the context of the theory META, which includes the definition of DEMO

.

<-(DEMO B ((c)(COLOUR c)(DEMO B (()(c BALL)))))

.

.

<-(DEMO B (()(DEMO B (()(RED BALL)))))

At this point the inheritance axiom is invoked to get the definition of DEMO from META, in order to be able to solve the goal (DEMO B (()(RED BALL))) from the program B.

.

.

```
<- (DEMO B ((cl resolvent)
             (CLAUSE B  cl)
             (RESOLVE (()(RED BALL)) cl  resolvent)
             (DEMO B resolvent)



<- (CLAUSE B cl1)
   (RESOLVE
      ((cl resolvent)
         (CLAUSE B cl)
         (RESOLVE (()(RED BALL)) cl resolvent)
         (DEMO B resolvent))
      cl1
      res)
   (DEMO B res)
```

To evaluate the first goal, we use the scoping axiom:

```
    ((CLAUSE p (()(CLAUSE p x)))
                    (CLAUSE p x))
```

This returns the following instantiations:

```
  p=B
  cl1=(()(CLAUSE B (()(RED BALL))))

<- (RESOLVE
      ((cl resolvent)
         (CLAUSE B cl)
         (RESOLVE (()(RED BALL)) cl resolvent)
         (DEMO B resolvent))

      (()(CLAUSE B (()(RED BALL))))
      res)
   (DEMO res)

<-(DEMO B ((resolvent)(RESOLVE (()(RED BALL)) (()(RED BALL)) resolvent)
            (DEMO B resolvent)))


<-(DEMO B (()(DEMO B ())))

[]
```

Example 2- the general predicate

When programming, it is often helpful to pass the name of a predicate as an argument of a more general predicate. For example, consider the predicate APPLY:

(APPLY <predicate> <argument> <result>)

which applies the predicate named <predicate> to the argument <argument> to get the result <result>. (This is similar to the 'maplist' function often cited in support of higher order functions in functional languages).

How can we define APPLY?

Let the name of the program in which we need the APPLY predicate be P. Then we can define APPLY in the following way:

```
P:-
 .
 .
((APPLY predicate argument result)
              (DEMO P (()(predicate argument result)))
 .
 .
-:P
```

If we want APPLY to be a general utility, such that any object level program can have access to it, we can add the following assertion to META:

```
((CLAUSE p
   ((predicate argument result)
       (APPLY predicate argument result)
       (DEMO p (()(predicate argument result)))

   )
)
```

(Notice that in the above p is a meta-level variable)

A predicate such as APPLY enhances the expressiveness of the programming language.

2-2 Negation

Horn Clause logic does not allow negative goals. Such a facility, however, is very important for writing realistic programs. In PROLOG, this issue has been tackled through negation-as-failure: the negation of a goal is deemed to be true if that goal can not be proved from the program.

[Clark 1978] has shown that, under certain circumstances, a valid logical reading can be attributed to negation as failure. The circumstances are:

a- the program is assumed to include the 'only-if' part of the clauses,

b- the negated goal statement is assumed not to include non-instantiated variables when it is invoked.

When these conditions are satisfied, negation-as-failure is consistent with classical negation. Otherwise, it does not have a logical, declarative meaning.

The fact is, however, that negation-as-failure is often used in situations where constraint b is violated, which violates the logical semantics of PROLOG. (This can lead to a situation called 'floundering' in the literature on negation as failure [Lloyd 1984])

Using self-reference, it is possible to take a different approach to negation-as-failure, one which is free of non-logical semantics, does not flounder, and does not go outside the framework of Horn-Clause logic.

In order to define a negative goal, we use

(NOT-DEMO P <goal>)

where P is the name of the current program, i.e. the program in which this clause is appearing, and <goal> is the description of the goal which we

want to be negated. Note that since P is the name of the program in which this clause is appearing, we are using self-reference.

Example:consider the program Justice below:

```
JUSTICE:-
((INNOCENT x)
        (NOT-DEMO JUSTICE (()(GUILTY x))))      <1>

((GUILTY JACK))

((HAPPY JOHN))

-:JUSTICE
```

Note that the x in <1> is a meta-level variable (with respect to the goal-list which is the second argument of NOT-DEMO). It ranges over meta-level constants. Since x also appears in the description of the object level goal, the meta-level constants over which it ranges name object-level constants. We are sure that x does not range over meta-level constants that name object level variables because the varlist at the head of the description of the object level goal is empty, i.e. the object level goal does not have any object level variables. Since constants name themselves, this clause correctly captures the meaning of the English sentence

Every person is innocent unless he/she is proven guilty.

contrast <1> with the following clause:

```
((INNOCENT x)
       (NOT-DEMO JUSTICE ((X)(GUILTY X))))      <2>
```

Here the meta-level constant X names an object-level variable. This is because X appears in the varlist at the head of the description.

<1> and <2> have widely different meanings: <1> says that for all x, x is innocent if not proven guilty. <2> says that for all x, x is innocent if no one is proved guilty. The difference arises because in <1> x is a meta-level variable, ranging over meta-level constants that name object level constants, whereas in <2> X is a meta-level constant naming an object-level variable.

Now, Prolog's version, with negation as failure, is the following

```
((INNOCENT x)
        (NOT GUILTY x))                    <3>
```

It confuses <1> and <2>: if x is instantiated when the clause is called, <3> behaves like <1>; if x is not instantiated, it behaves like <2>.

Notice that the difference between <1> and <2> above is the difference between two ways of quantifying x: <1> is similar to (though not strictly equivalent to)

∀ x (INNOCENT(x) <- NOT GUILTY(x))

whereas <2> is similar to

∀ x (INNOCENT(x) <- NOT ∃ x (GUILTY(x))

To see how our formulation overcomes floundering, consider the query:

which x: ((INNOCENT x)(HAPPY x)) with respect to the program JUSTICE.

this should return the result x=JOHN, no matter in which order the subgoals are run. Below is a simplified evaluation of this query, running the negated goal first. (We have omitted some steps for brevity)

To evaluate this call, we evaluate the query

<- (DEMO JUSTICE ((x)(INNOCENT x)(HAPPY x)))

with respect to META.

<- (DEMO JUSTICE ((x)(INNOCENT x)(HAPPY x)))

.

.

<- (DEMO JUSTICE ((x)(NOT-DEMO JUSTICE (()(GUILTY x))) (HAPPY x)))

At this point, the inheritance axiom is invoked to get the definition of NOT-DEMO from META.

<- (DEMO JUSTICE ((x clause-list)
(POSSIBLE-CLAUSES JUSTICE (()(GUILTY x)) clause-list)
(FAIL-ALL JUSTICE (()(GUILTY x)) clause-list))
(HAPPY x)))

At this point, the scoping axiom is invoked to evaluate Possible-Clauses, i.e. to see which clauses match with (()(GUILTY x))

<- (DEMO JUSTICE ((x)(FAIL JUSTICE (()(GUILTY x)) ((()(GUILTY JACK))))
(HAPPY x))

The definition of FAIL includes the clause:

```
((FAILS program
    (goal-vlist goal-predicate|goal-argument)
    (clause-vlist (head-predicate|head-argument)|body))
 (NOT-UNIFY
    (goal-vlist|goal-argument)
    (clause-vlist|head-argument)))
```

<- (DEMO JUSTICE ((x)(NOT-UNIFY (() x)) (()(JACK))) (HAPPY x))

The definition of NOT-UNIFY includes the following clause:

```
((NOT-UNIFY (vlist1|c1) (vlist2|c2))
      (NOT-ON c1 vlist1)
      (NOT-ON c2 vlist2)
      (DIFFERENT c1 c2))
```

<- (DEMO JUSTICE ((x)(NOT-ON () x)(NOT-ON () JACK) (DIFFERENT x JACK)
(HAPPY x)

NOT-ON determines whether its second argument is a member of the varlist which is its first argument.

<- (DEMO JUSTICE ((x)(DIFFERENT x JACK)(HAPPY x)

DIFFERENT is another predicate inherited from META. It is true if its two arguments are two different constants.

<- (DEMO JUSTICE (()(HAPPY JOHN)))

<-

Thus we have used the predicate NOT-UNIFY to return values for the

meta-level variables standing for object-level constants.

The full definition of NOT-DEMO is given in Appendix1.

3- Reflection

The amalgamated system developed so far suffers from a serious problem. The use of DEMO and NOT-DEMO for running goals would seriously degrade the performance of the system, because such goals would have to be interpreted by running their definitions. To avoid this problem, we use the reflection mechanism [Weyhrauch 1980] [Bowen and Kowalski 1982] as explained below.

Recall the representability axiom stated in the previous chapter:

In the context of the theory T, P represents the relation R on the domain D iff

T|-(P t1' t2'..) iff (t1 t2..)belong to R

where t1',t2' ... are the names of t1,t2..

This rule can be used for computational purposes in two ways:

I- If we have a special purpose means of evaluating the relation R on the domain D, then we can use this mechanism to evaluate T|-P. For example, suppose that the predicate (ADD x y z) represents the relation of addition on the domain of integers, with the identity naming relation; i.e.

T|-(ADD x y z) <-> x+y=z

Then, since there is a special purpose mechanism for evaluating sum, namely the hardware integer addition mechanism of the computer, we can use this mechanism to see whether T|-(ADD n1 n2 n3). To do this, we use hardware addition to see whether n1+n2=n3.

II-If we want to ascertain that the tuple (t1 t2 ..) belongs to the relation R, we could do this by trying to show that (P t1' t2' ..) is provable from T. For example, we could evaluate x+y=z by evaluating whether T|-(ADD x y z).

We call the first computational use of the representability axiom downward reflection, and the second one upward reflection. The computational use of reflection was first discussed in [Weyhrauch 1980]. The logical justification for reflection, through representability, was worked out in [Bowen and Kowalski 1982].

When downward reflection is possible, it can have a great efficiency advantage, because it turns a theorem proving task into an evaluation task which is likely to be more efficient.

Now, as stated earlier, various meta-level predicates such as DEMO, RESOLVE etc. represent corresponding relations on the domain of OL linguistic objects. Thus, if we had special purpose mechanisms for evaluating these relations, we could use them to evaluate, by downward reflection, these predicates in the context of META.

In fact, we do have such special purpose mechanisms: they are constituent mechanisms of the interpreter itself. For example, consider the predicate RESOLVE. As stated in the previous chapter, in the context of the theory META the meta-predicate RESOLVE correctly represents the resolution relation on sentences of the OL. This means that

META|-(RESOLVE <cl1> <cl2> <resolvent>) iff

the resolvent of <cl1> and <cl2> is <resolvent>. (<cl1> is the name of

<cl1> etc.)

Now, almost all Horn-Clause interpreters have some sort of mechanism for resolving clauses. Such mechanisms can be used for evaluating the predicate RESOLVE in the context of META, without having to run the explicit definition of RESOLVE.

Another example is DEMO: instead of trying to prove

META |- (DEMO <prog> <goal>), we show that <prog>|-<goal>

Notice that in order to do downward reflection on meta-predicates we must map from the description of the clauses, i.e. <cl1>, <cl2> etc. to the actual data-structures used in the interpreter for representing clause, here symbolically denoted by <cl1>, <cl2> etc. We call this step de-referencing, and it is crucially important that it should be as simple and transparent as possible, for otherwise the overhead incurred in performing it might overshadow the efficiency advantages of using downward reflection in the first place. That is one of the reasons why in the previous section we went to some length to use a syntax and naming relation which ensured close similarity between the structure of a clause and the structure of its description. The need for this close co-relation between the data-structures used for representing clauses and the clauses themselves also arises when upward reflection is attempted, as occurs for example when the control scheme in chapter 2 is used.

Now let us see what practical issues are involved in doing downward reflection on meta-predicates:

a- We need an interpreter that is modular, with well-defined interfaces between its modules, so that the various modules could correspond to the relevant meta-predicate (e.g. the resolution module could correspond to

the RESOLVE predicate)

b- The structure of the interpreter must closely resemble the structure of the definition of DEMO. This requirement follows from the previous one, for if we have modules corresponding to various meta-predicates, we might as well link them together in the same way that their corresponding predicates are linked in the definition of DEMO.

c- The data-structures used in the interpreter for representing linguistic objects must closely resemble the data-structures used for describing the same objects in the meta-language, in such a way that moving form one to the other in either direction is straightforward. For example, to represent a clause in the interpreter, we must use a linked list that has the same structure as that of the description of the clause in the meta-language.

The use of the reflection mechanism for evaluating meta-predicates is an important practical aspect of the system developed here for amalgamating object-language and meta-language. The reflection mechanism can be put to good use for controlling the execution of object level programs through linking control information, i.e. information about the way an object-level program should be run, with the internal mechanisms of the interpreter. This issue will be taken up in the next chapter.

4-Conclusion

We have developed a methodology for enhancing Horn-Clause logic programming within the strict confines of logical semantics. This methodology relies on the amalgamation of object-language and meta-language in a programming context. We have used this amalgamated system to define 'clean' negation as failure, and some higher order features.

Chapter Two

Meta-Level Reasoning for Declarative Control

One of the main criticisms of declarative programming is that it is often inefficient; that it does not fully exploit the domain-specific knowledge that is implicitly present in procedural formulations of the same problem. It has been suggested [Kowalski 1979] that adding control information to the declarative program will to some extent alleviate this problem. In this section we are going to investigate a method for expressing such control information in a declarative way, using the same language as the language of the program.

[Davis 1980] describes a framework for using meta-level knowledge for controlling search, though this framework is not expressed in terms of logic. PRESS [Bundy and Welham 1980] [Silver 1986] uses meta-level knowledge, expressed as Horn-Clauses, to control the search-space that arises when algebraic re-write rules are applied for the solution of transcendental equations. [Gallaire and Lassere 1981] describe a methodology for incorporating declarative control into PROLOG. [Pereira 1982] also describes a technique for declarative control of logic programs.

In chapter four, the technique described in this thesis is compared to that employed in PRESS. The methodology of [Gallaire and Lassere 1981] is discussed in relation to our work later in this chapter.

1. The Control Facilities of PROLOG

The language PROLOG offers some control facilities, such as the 'cut' operator and, in some versions such as IC-Prolog, co-routining. Some of these facilities, such as the 'cut', destroy the declarative semantics of the language. All of them, however, are 'hard-wired' in the following sense: they allow the modification of the behaviour of the standard

interpreter in certain, fixed ways. For example, co-routining, with its two variants, 'lazy' and 'eager' evaluation, provides two fixed alternatives to the standard evaluation strategy. If the programmer wants a third co-routining style which is neither eager nor lazy (say), he has no recourse: he can not do it. The only choice is to re-write the program in such a way that the run-time behaviour is acceptable given the current control facilities.

A notable exception in this regard is the work proposed in [Gallaire and Lassere 1981]. They propose certain control predicates which can be used by the programmer to define, using logic, the behaviour of the interpreter. There are two main criticisms of their work, however. The first criticism is that they do not explicitly spell out the theoretical framework in which their predicates operate, but make use of an implicit 'interpreter model' to define the semantics of their control control predicates. This leads to ambiguity about the exact meaning of their proposed control predicates. The second criticism is that they do not differentiate between object level (program) and meta-level (control) variables, an omission which has important practical implications as discussed in the previous chapter.

In this section we shall present a technique for controlling the behaviour of object level programs in logic, using meta-level programming. This technique is general, i.e. it can be used for defining any top-down control strategy desired. Theoretically, it is based on the concepts of representability and reflection, which provide a firm basis for its semantics.

The work presented in this chapter is an extension of the work of [Gallaire and Lassere] in the following sense:

-It uses an explicit meta-level framework, employing the reflection mechanism, to express control knowledge,

-It treats node-selection as well as goal-selection and clause-selection, while G&L is restricted to goal and clause selection. In particular, in our scheme it is possible to control the backtracking behaviour of the system [Bruynooghe and Pereira 1981].

-It includes a scheme for ensuring that the control information does not make the interpreter unsound.

2-Control through Re-Defined Interpreter

Conceptually, the control of execution of object level programs from the meta level is very simple: just write your own special interpreter, so that it applies the particular execution strategy you are interested in.

The advantages of this approach are that:

a- the specification of control is fully declarative

b- the language used for control is the same as that used for other meta-level purposes.

c- It is possible to write the interpreter so that it bases its decision for control on the performance of the object-level program so far, i.e. it is possible to have dynamic control.

There are some serious disadvantages with this method, however, which make us adopt more elaborate schemes. These disadvantages are:

a- It is a non-trivial task to write an interpreter for every control situation,

b- The interpreter being user-defined, the reflection mechanism can not be used for its evaluation,

c- There is no guarantee that the user defined DEMO correctly represents provability.

In order to circumvent (some of) these problems and still keep the positive aspects of user-defined interpreters, one can take one of the following two approaches:

I- Define your own, special purpose interpreter in terms of subsidiary predicates such as CHOOSE, SELECT, RESOLVE etc., where these subsidiary predicates can be evaluated through downward reflection. In this way, only the top level of the interpreter needs to be defined by the user and evaluated at the meta-level. This simplifies the task of the programmer and avoids the inefficiency of full meta-level evaluation. However, since the interpreter is essentially user defined, there is no guarantee that it correctly represents provability.

II- When we do not want to define an entirely new interpreter, but only to re-define the search-strategy of the standard interpreter, we can use certain choice predicates to define our search strategy. These choice predicates are meta-predicates that are used in the definition of the DEMO and collectively define its search strategy. By defining them, the user effectively imposes his own search strategy on the interpreter. The use of reflection to evaluate DEMO can still be maintained, although there is need for upward reflection on the execution of these choice predicates, as will be explained below.

We shall not develop the first technique any more here. It is best used when there is need for a special purpose interpreter, such as a debugging interpreter, whose functions go further than what the standard interpreter is meant to provide. As far as the system is concerned, its implementation

is simply a matter of providing the necessary system-defined meta predicates, a task which, given an efficient de-referencing mechanism, is conceptually no more complex than providing predicates for doing arithmetic operations.

We shall devote the rest of this chapter to developing the second control technique mentioned above. But first we have to explain, in some detail, the definition of the provability predicate S-DEMO and its properties. Without such an explanation it would not be possible to discuss the control technique mentioned above.

3. The Definition and Properties of S-DEMO

(S-DEMO <program> <goal> <status>
is a three place meta-predicate that represents provability.
-<program> is the name of the object level program concerned;
-<goal> is the goal to be proved from <program>;
-<status> is the status of the attempted proof: SUCCESS or FAIL

S-DEMO has the following properties:

a- It is behaviourially deterministic, i.e. it does not backtrack. The only non-determinism employed in it is "don't care" non-determinism. Its handling of object level back-tracking is not through meta-level backtracking, but through the maintenance of an explicit search tree which is steadily grown. Non-determinism is handled through the CHOOSE-NODE predicate thus obviating the need for back-tracking at the meta-level. The CHOOSE-NODE predicate determines the next node in the search tree to be expanded.

b- Object level failure does not lead to the failure of S-DEMO, but to its success with the status argument bound to FAIL.

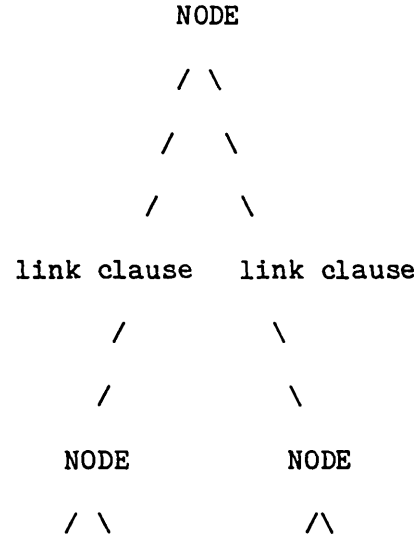
The properties a and b above are very important from the point of view of control. We shall say more about this later, in the section on control.

Before we go on to explain the definition of S-DEMO, let us briefly describe the data-structure used for representing the search-tree, as this will aid in understanding the definition.

3-1. The Search-Tree

The search-tree is a data-structure that holds all the information on the current state of the proof.

The search tree consists of a number of nodes which are linked to each other by link clauses.



Every node contains the following information:

- a goal-list,
- a focus goal,

- a list of remaining clauses for matching with the focus goal

A link clause is the clause of the program which resolves with the goal-list of the parent node to give the goal-list of the child goal. The focus goal of a goal list is the goal which matches with the head of the link clause.

Intuitively, the search-tree can be considered as an OR tree for proving the goal-list of the root node.

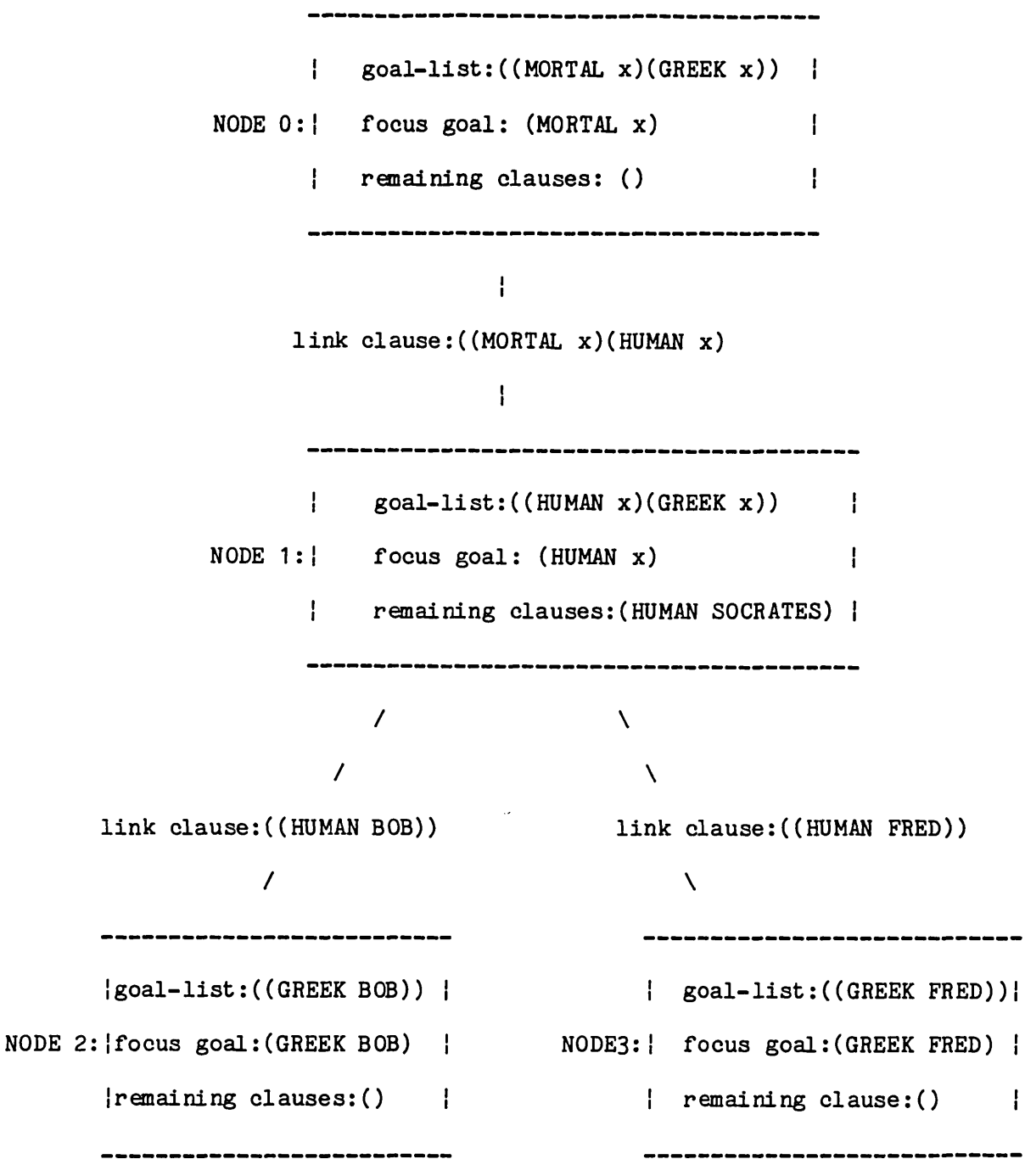
Example:
Consider the program

```
((MORTAL x)
   (HUMAN x))
((HUMAN BOB))
((HUMAN FRED))
((HUMAN SOCRATES))
((GREEK SOCRATES))
```

and the goal-list

```
?((MORTAL x)(GREEK x))
```

the following depicts the state of the search tree at a point during the proof of the goal-list. Notice that the tree has not been fully grown yet, and has room for further expansion.



In a search tree, a node is a SUCCESS node if:

- its goal-list is empty, or
- it has a child which is a SUCCESS node

A node is a FAIL node if:

- It goal-list is not empty, but its list of the remaining clauses to match

with the focus goal is empty, and

all its children are FAIL nodes , or

it does not have any children.

A node is an OPEN node if it is neither a SUCCESS node nor a FAIL node.

(In the above example, nodes 2 and 3 are FAIL nodes, while nodes 0 and 1 are OPEN nodes.)

The goal-list of any SUCCESS node is a logical consequence of the program.

S-DEMO works by growing the search tree until the root node is either SUCCESS or FAILURE.

To grow the tree, a node is chosen for expansion. If the node is an unfocussed node, i.e. its focus goal has not been chosen before, then a focus goal is chosen for the node, and the list of clauses from the program which can match with this goal is computed. This information (the focus goal and the list of clauses) is added to the node, and the expansion routine for focussed nodes is called.

The expansion routine for focussed nodes chooses a clause from the list of remaining clauses for the node. It resolves this clause with the goal-list of the node (with the focus node as the focus of resolution) and creates an unfocussed new node whose goal-list is the resolvent, whose parent is the original goal, and whose link-clause with the parent is the chosen clause. Then the link clause clause is deleted from the remaining-clauses of the parent node.

The Definition of S-DEMO:

```
1:  ((S-DEMO program goal-list status)
        (INITIAL-SEARCH-TREE goal-list in-tree)
        (EXTEND program in-tree out-tree status))
```

Comment:

Clause 1 is the top level of the definition of S-DEMO. The predicate INITIAL-SEARCH-TREE produces a search tree with one node, and that node contains the initial goal list. The predicate EXTEND recursively grows the input search tree until it is either a success tree or a failure tree.(A tree is a success or failure tree if its root node is a success or failure node)

The Definition of EXTEND:

```
2:  ((EXTEND program in-tree in-tree FAIL)
        (STATUS in-tree FAIL))

3:  ((EXTEND program in-tree in-tree SUCCESS)
        (STATUS in-tree SUCCESS))

4:   ((EXTEND program in-tree out-tree status)
        (STATUS in-tree OPEN)
        (CHOOSE-NODE program in-tree node)
        (EXPAND-NODE program node in-tree temp-tree)
        (EXTEND program temp-tree out-tree status))
```

Comment:

The predicate STATUS can return three possible values for the status of the tree: FAIL, SUCCESS, or OPEN. If the status of the tree is FAIL or SUCCESS, then the evaluation of EXTEND terminates with the relevant value returned for the status argument (clauses 2 and 3). If the status of the tree is OPEN, then EXTEND enters its recursive case: a node is chosen for

expansion, and the EXPAND-NODE predicates expand the node, appropriately modifying in-tree to temp-tree. Then EXTEND is called recursively to extend the newly formed temp-tree.

The Definition of EXPAND-NODE

```
5:  ((EXPAND-NODE program node in-tree out-tree)
        (FOCUSSED node)
        (GOAL-LIST  node goal-list)
        (FOCUS-GOAL node goal)
        (REMAINING-CLAUSES program node clause-list)
        (CHOOSE-CLAUSE  program  in-tree  goal-list goal clause-list
                                                  clause rest-clauses)
        (RESOLVE goal-list clause new-goal-list)
        (ADD-CHILD-NODE in-tree node clause new-goal-list temp-tree)
        (MODIFY-NODE node goal-list rest-clauses modified-node)
        (REPLACE-NODE temp-tree node modified-node out-tree))

6:  ((EXPAND-NODE program node in-tree out-tree)
        (UNFOCUSSED node)
        (GOAL-LIST node goal-list)
        (CHOOSE-GOAL program in-tree goal-list goal)
        (POSSIBLE-CLAUSES program goal clause-list)
        (MODIFY-NODE node goal clause-list focussed-node)
        (REPLACE-NODE in-tree node focussed-node temp-tree)
        (EXPAND-NODE program focussed-node temp-tree out-tree))
```

Comment:

Every node in the tree is either focussed or unfocussed. A node is focussed if one goal has previously been chosen from its goal-list for expansion. If such a choice has not been previously made, then the node is an unfocussed node.

Clauses 5 and 6 define the predicate EXPAND-NODE. If the node to be expanded is focussed, (clause 5) then a clause is chosen from the remaining untried clauses for that node, and the goal-list is resolved with the clause. A new node is created to accommodate the resolvent, and this node is added to the tree, with the chosen clause as the link clause . Also, the chosen clause is deleted from the list of untried clauses for the original node, and the tree is modified to reflect this.

If the node to be expanded is not focussed (clause 6), then a goal is chosen from the goal-list of the node, to be the focus for that node. The list of possible clauses to match with this goal is computed, and this information is added to the node, thus making it a focussed node. The new, focussed node replaces the original node in the tree, and EXPAND-NODE is called again to expand the new node.

3-2. Defining the Expansion Strategy of S-DEMO

As mentioned earlier, the expansion strategy of the provability predicate S-DEMO is determined by the definitions of the following three predicates:

(CHOOSE-NODE program search-tree max-steps node)

(CHOOSE-GOAL program search-tree goal-list goal)

(CHOOSE-CLAUSE program search-tree goal-list goal clause-list clause rest-clauses)

The user can specify his strategy by modifying the behaviour of these predicates. Below we shall discuss each one of these predicates in turn and illustrate their use with an example. Later we shall discuss how the user can modify their behaviour, to achieve control over the execution of the object level program, without jeopardising the correctness of S-DEMO.

3-2-1 CHOOSE-NODE

The predicate CHOOSE-NODE determines which node of the search-tree should

be chosen for expansion. It is the definition of this predicate that specifies whether the search strategy is depth-first, breadth-first or some other node expansion policy. The backtracking behaviour of the interpreter is also determined by the definition of this predicate.

Example: a depth first strategy with backtracking to the most recent expandable parent (the PROLOG strategy) can be defined as follows:

```
((CHOOSE-NODE prog search-tree node)
    (NODE-OF search-tree node)
    (OPEN node)
    (NOT-DEMO CONTROL ((OTHER-NODE)(NODE-OF search-tree OTHER-NODE)
                                   (OPEN OTHER-NODE)
                                   (DEEPER search-tree OTHER-NODE node))))
```

NOTE: CONTROL is the name of the meta-level program which contains these clauses

Note that here the constant OTHER-NODE is ranging over the constants which name object-level variables. Since this is occurring in the context of NOT-DEMO, the effect is that of an existentially quantified variable. Note that if we had used ordinary negation-as-failure, OTHER-NODE would have had to be a variable, thus violating the constraints under which negation-as-failure has a logical reading.

3-2-2. CHOOSE-GOAL

The definition of this predicate determines the goal to be chosen as the focus of the goal-list.

From a completeness point of view it is only necessary to choose a focus goal for a node once. In backtracking there is no point in changing the focus, since, if we do change the focus, the old focus will still have to

be resolved anyway. From an efficiency point of view, however, it might be useful to change the focus because we might have realised that another focus is likely to fail more quickly, thus reducing the search space. In the present formulation, however, we have not allowed the change of focus because such a feature would complicate the structure of the search tree and the definition of S-DEMO.

The definition of CHOOSE-GOAL determines the order in which the subgoals are solved. Co-routining, goal-suspension etc. can all be specified through the appropriate definition of CHOOSE-GOAL.

Example:
The standard PROLOG goal-selection strategy (left to right) can be defined as follows:

```
((CHOOSE-GOAL program tree goal-list goal)
     (GOAL-OF goal-list goal)
     (LEFT-MOST goal-list goal))
```

The two-place meta-predicate GOAL-OF is true if its first argument is a legitimate goal-list, and its second argument is a goal of the goal list, preceded by a list of the variables occurring in the goal. For example, the predicate GOAL-OF holds between the goal-list

((x y)(TALL x)(FATHER x y)(FEMALE y))

and the goal

((y)(FEMALE y))

3-2-3. CHOOSE-CLAUSE

The definition of CHOOSE-CLAUSE determines the order in which the clauses

matching with a goal are to be tried. By defining CHOOSE-CLAUSE properly, the amount of backtracking necessary can be reduced.

The PROLOG clause-selection strategy can be defined as:

```
((CHOOSE-CLAUSE program in-tree goal-list goal clause-list clause rest-clauses)
        (CLAUSE-OF clause-list clause rest-clauses)
        (LEFT-CLAUSE clause-list clause))
```

Points about the Definition of Choice Predicates.

4. Correctness of S-DEMO

When we allow the user to modify the behaviour of the choice predicates, can we still maintain that S-DEMO represents provability?

Provided that certain basic constraints are satisfied, then the correctness of S-DEMO will not be affected. These constraints are the correctness, completeness and fairness of the definition of choice predicates.

a-Correctness of the Definition of Choice Predicates

The choice predicates must return a correct result. For example, CHOOSE-NODE must return a valid, open node of the search tree, and CHOOSE-GOAL must return a goal that belongs to the goal-list

b-Completeness of Choice Predicates

The choice predicates must never fail to return a valid result if such a result exists. For example, CHOOSE-NODE must not fail to return a node if the search tree has a valid, open node.

c- Fairness of Choice Predicates

The definition of choice predicates must be such that, if the search tree is infinite, every branch eventually gets a chance for expansion. For example breadth first strategies are fair, whereas depth first strategies

can be unfair.

If the above three conditions are satisfied, then whatever the user definition of the choice predicates, S-DEMO will be correct. This is due to some fundamental results in resolution theorem proving, which stipulate that the order of expansion of goals and nodes and the choice of clauses for expanding goals is not important as far as correctness and completeness of resolution is concerned [Kowalski 1974].

So, how do we ensure that the above constraints are satisfied?

The following scheme ensures that choice predicates are correct and complete. We have not, however, devised a mechanism for ensuring the fairness of the search strategy. Thus in our scheme the search strategy can be incomplete due to the unfairness of the user-defined choice predicates.

-The choice predicates are defined in terms of the user defined predicates in the manner illustrated below for the CHOOSE-NODE predicate:

```
((CHOOSE-NODE program search-tree node)
     (USER-CHOOSE-NODE program search-tree node)
     (CORRECT-NODE search-tree node))

((CHOOSE-NODE program search-tree node)
     (NOT-DEMO CONTROL ((NODE)(USER-CHOOSE-NODE program search-tree NODE)))
     (SYSTEM-CHOOSE-NODE program search-tree node))
```

The above formulation ensures that CHOOSE-NODE is correct and complete. It ensures correctness because the result returned by the user defined predicate is tested for correctness, and the system defined predicate is

assumed to be correct. It is complete because, if the user defined predicate fails, the second clause will be activated, and will invoke the system defined choice predicate, which will return a valid result if possible.

The following are the definitions of CHOOSE-GOAL and CHOOSE-CLAUSE.

I- CHOOSE-GOAL:

```
((CHOOSE-GOAL program tree goal-list goal)
     (USER-CHOOSE-GOAL program tree goal-list goal)
     (GOAL-OF goal-list goal))

((CHOOSE-GOAL program tree goal-list goal)
  (NOT-DEMO CONTROL ((GOAL)(USER-CHOOSE-GOAL program tree goal-list GOAL)))
  (SYSTEM-CHOOSE-GOAL program tree goal-list goal))
```

II- CHOOSE-CLAUSE:

```
((CHOOSE-CLAUSE program tree goal-list goal clause-list clause rest)
     (USER-CHOOSE-CLAUSE program tree goal-list goal clause-list clause)
     (CLAUSE-OF clause-list clause rest))

((CHOOSE-CLAUSE program tree glist goal clist clause rest)
    (NOT-DEMO CONTROL ((CLAUSE)(USER-CHOOSE-CLAUSE program tree glist goal
                                                        clist CLAUSE)))
    (SYSTEM-CHOOSE-CLAUSE program tree glist goal clist clause rest))
```

5. What If User-Defined Choice Predicates Are Non-Deterministic?

As we have developed it so far, the control scheme allows the definition of non-deterministic choice predicates by the user, i.e. there can be more than one solution at a time for a given choice predicate. Will this interfere with the objectives of declarative control?

As far as the soundness of the interpreter is concerned, it does not matter whether or not the choice predicates are non-deterministic. This is because the correctness and completeness safeguards mentioned above rule out any unsound derivations.

Practically, if a choice predicate is non-deterministic, we have to answer two questions:

- Will the control scheme be of any use?
- Will non-determinism of the choice predicates lead to general non-determinism of S-DEMO itself?

The answer to the first question is that the control scheme will be useful even if the choice predicates are non-deterministic. In fact, non-deterministic control is a positive aspect of what is meant by declarative control: the control information will not have to be any more specific, or procedural, than necessary. Thus, when a range of choices is allowed by the definition of the user defined choice predicates, any choice that falls within the range is appropriate, because it follows from the user's specification. In fact, non-deterministic control relieves the user from specifying unnecessary detail.

Notice that there will never be 'conflicting' requirements at any one time: the way the control scheme is set up, any instance of the choice predicate which is a logical consequence of the definitions of the choice predicates will be valid. Since inconsistent definitions of predicates are impossible, because all definitions are Horn-Clauses, we may have more than one choice at a time, but never conflicting requirements.

The answer to the second question above is that non-determinism of choice predicates will not lead to backtracking of S-DEMO. This is because object level failure at a branch of the search tree is not handled through meta-

level backtracking. The definition of S-DEMO incorporates a kind of 'don't care' non-determinism: whatever the outcome of the choice predicates, there will never be need to backtrack at the meta level.

6. Examples of the Use of the Choice Predicates for Defining Control

In this section we shall try to show, by example, how the control scheme can be used in practice.

Some of the control situations elaborated here are those mentioned in [Gallaire and Lassere 1981]. This discussion also serves as a comparison between the system adopted here and that adopted by G&L.

6.1 -Clause Preference

Let us assume that the user decides to use the PREFER-CLAUSE predicate to express the fact that he prefers to solve <goal> by <clause1> rather than <clause2>; i.e. whenever

(PREFER-CLAUSE <program> <goal> <clause1> <clause2>)

can be proved from CONTROL, then <clause1> should be used in preference to <clause2> for solving <goal>.

He can link the PREFER-CLAUSE predicate to the interpreter via the USER-CHOOSE-CLAUSE predicate:

```
((USER-CHOOSE-CLAUSE program tree goal-list goal clist clause1)
     (PREFER-CLAUSE program goal clause1 clause2)
     (CLAUSE-OF clist clause1)
     (CLAUSE-OF clist clause2)
     (NOT-DEMO CONTROL ((CLAUSE3)
                        (CLAUSE-OF clist CLAUSE3)
                        (PREFER-CLAUSE program goal CLAUSE3 clause1))))
```

6-2. Goal Preference

If goal preference is expressed through the PREFER-GOAL clause, i.e. if the user wants <goal1> to be chosen for expansion in preference to <goal2> whenever

(PREFER-GOAL <program> <goal1> <goal2>)

is provable from CONTROL, this can be done by linking PREFER-GOAL with USER-CHOOSE-GOAL using the following clause:

```
((USER-CHOOSE-GOAL prog tree goal-list goal1)
     (PREFER-GOAL prog goal1 goal2)
     (GOAL-OF goal-list goal1)
     (GOAL-OF goal-list goal2)
     (NOT-DEMO CONTROL ((GOAL3)
                        (GOAL-OF goal-list GOAL3)
                        (PREFER-GOAL GOAL3 goal1))))
```

6-3. Lazy Coroutining

Let the LAZY predicate indicate our wish for lazy coroutining, i.e. if

(LAZY <program> <goal1> <goal2> <varname>)

is provable from CONTROL, then <goal1> should be chosen if the variable named by <varname> occurs unbound in <goal2> and <goal1>. (This would make <goal1> the producer, and <goal2> the consumer>

LAZY can be linked to the interpreter through the following clause:

```
((USER-CHOOSE-GOAL program tree goal-list (vlist1 goal1))
     (LAZY program goal1 goal2 varname)
```

(GOAL-OF goal-list (vlist1 goal1))

(GOAL-OF goal-list (vlist2 goal2))

(OCCURS varname vlist1)

(OCCURS varname vlist2))

vlist1 is a list of the object-level variables occurring in goal1, and vlist2 is a list of the object-level variables occurring in v2.

7. Implementation Issues

Earlier, we mentioned that in order to run S-DEMO efficiently, we use downward reflection to evaluate it. In practice, this means that when we want to evaluate

META⊢(S-DEMO <program> <goal> <status>),

we load the program named by <program> into the interpreter, and run the goal named by <goal>. This involves a mapping from the meta-level representation of the program and the goal (their descriptions) to the internal data-structures used by the interpreter for representing programs and goals. We called this dereferencing.

Now, when downward reflection is used to evaluate S-DEMO, how can we invoke the user defined choice predicates?

We mentioned the assumption that the structure of the interpreter closely resembles the structure of the definition of S-DEMO, with separate modules representing the various meta-predicates involved in the definition of S-DEMO. We utilise this modular structure of the interpreter to invoke the user-defined choice predicates. The manner in which this is done is explained below.

There are modules in the interpreter corresponding to the three choice

predicates. For example, if the interpreter were written in a language such as PASCAL, these would be procedures. When, during the course of evaluation of the object level goal, one of these modules is to be invoked, for example when the interpreter is trying to choose a node to expand next, the evaluation of the module is made through upward reflection; i.e. through invoking the user definition of the relevant choice predicate at the meta-level. Upon completion of the evaluation of the user defined choice predicate, control is passed back to the object-level interpreter, and the evaluation of the object level program is continued.

Earlier on, we had mentioned that the definition of S-DEMO employs "don't care" non-determinism, and that it handles object-level backtracking through operations on an explicit search-tree. This property of the definition of S-DEMO is quite useful in implementing the system in a procedural language, because it is much easier to design, in a procedural language, an interpreter which is structurally similar to a deterministic logic program than it is to design one which is similar to a non-deterministic logic program. All the recursive clauses could be simulated by loops, and the clause alternative by case statements. In the context of such an implementation, upward reflection would be a recursive call to the interpreter with the appropriate arguments (i.e. the de-referenced version of the arguments occurring in the definition of the concerned meta-predicate).

8. The Efficiency of the Control Scheme

One of our objectives in devising a control facility was to increase the efficiency of the interpreter. But will it not be the case that the invocation of the user defined choice predicates in every cycle will decrease efficiency, instead of increasing it?

Of course, in a bad implementation of the scheme described above, it might happen that efficiency is actually decreased; but the following arguments

might satisfy the reader that a judicious implementation of the scheme will aid efficiency:

a- There are many search-intensive programs whose running time increases exponentially with the size of the problem (for example unrestrained equational re-write rules). If a control scheme can make the growth of the running time polynomial with respect to the size of the problem, then the overhead of computing the choice predicates will quickly be dwarfed by the saving in the running time of the program. The PRESS symbolic equation solving system [Bundy and Welham 1982], for example, utilises meta-level control to effectively curb the combinatorial explosion arising from the unrestrianed application of algebraic re-write rules. Although the methodology of PRESS is somewhat different from the one we have expounded here, it does demonstrate that with combinatorially explosive object level programs, the overhead of meta-level control is well worth having if it tames the explosive growth of the object-level search tree.

b- A smart system can employ one or more of the following schemes to reduce the overhead of computing the choice predicates:

-At compile time, it might be able to determine what types of situations are addressed by the choice predicates, and only do upward reflection when those situations arise. For example, it might be possible to determine that the USER-CHOOSE-GOAL predicate only applies to goal-lists which include some goal with a specific predicate symbol. In such a case, it only invokes the user defined USER-CHOOSE-GOAL when such a goal is present in the goal-list.

-The greatest overhead in the use of choice predicates is when they are defined in general terms, in such a way that they potentially apply to a great number of situations, and it is necessary to run them at length each time that a choice has to be made. For example, suppose we define the PREFER-GOAL predicate in the following way:

((PREFER-GOAL g1 g2)

(PREFER-GOAL g1 g3)(PREFER-GOAL g3 g2))

Then the recursive clause above must be run every time a choice of goals is to be made, which is clearly very inefficient. It would be a good strategy to run this rule bottom up, and generate lemmas as to which goals are preferred to which, thus cutting the run time inefficiency. This lemma generation might be done at compile time or run time: if it is done at compile time, there is the problem that there might be too many lemmas, only some of which are useful. If lemma generation on choice predicates is done at run time, the system can collect statistics about the invocation patterns of the lemmas, and only keep those which are most often used.

In general, the control information is extra information about the problem. It is up to the system to make the best use of it. Since the way the control information is used will not affect the correctness of the interpreter, which is independent of control, there is great flexibility in using (or ignoring) the information as the situation demands.

Chapter Three

An Application of Meta-Level Reasoning-

Fault-Finding in Digital Circuits

Introduction

In this chapter we shall try to illustrate and develop, through an application, the meta-language programming techniques discussed in the previous two chapters. The application chosen for this purpose is the task of fault-finding in digital circuits.

FAULTFINDER is a program that diagnoses faults in combinatorial digital circuits. It relies on the input-output behaviour of the circuit to achieve this, without requiring access to the circuit elements directly.

In FAULTFINDER, fault diagnosis is viewed as a special case of induction. The goal is to derive the theory of the faulty circuit. The theory of the normal circuit is known at the outset, before the fault-finding process begins. FAULTFINDER devises a set of experiments on the faulty circuit which enable it to derive the theory of this circuit from the theory of the normal circuit. Throughout this process, it uses some assumptions about the nature of 'possible faults' to limit its search.

1- The set up of the fault-finding process

-The Objective

There exists a digital circuit whose internal organisation (circuit diagram) is known. This circuit develops a fault which causes it to misbehave.

It is required to locate the fault by devising a set of experiments and conducting them on the circuit. The program should devise the experiments,

and draw the necessary conclusions from their results to find the fault.

-The Circuit

The circuit is a combinatorial logic circuit, with a number of input and output terminals. Only these terminals are accessible to the experimenter. By combinatorial we mean that the circuit does not include any memory elements, so that the state of the output is uniquely determined by the input.

-Faults

Faults are assumed to arise from faulty gates only. It is further assumed that there is only one faulty gate in the circuit.

We assume that a faulty gate has its output 'stuck'; in other words, the output is always the same, independent of the inputs to the gate. This is an assumption made by a large number of fault-diagnosis programs, and is a fairly realistic one.

-Experiments

An experiment consists of setting up the input terminals of the circuit to a certain combination, and observing the state of the output terminals.

2- Theories of Circuit Modules as Logic Programs

In order to reason about the behaviour of circuit modules, they must be represented by a data structure that reflects their internal structure and modes of behaviour.

In FAULTFINDER circuit modules are represented by object level programs. These programs are called the theories of the corresponding circuits. The advantage of using logic programs for representing circuits is that they not only represent the static aspects of a circuit, i.e. its internal structure and interconnections, but also its dynamic aspects, i.e. the way

it would respond to inputs, and how signals are propagated in the circuit. This expressive capability is of great importance for the task of fault-finding, which is primarily the task of reasoning about circuits, be they faulty or normal, and their relationship with the signals they receive and transmit. If we had chosen a static data structure, such as a list or a set, to represent circuits, then we would have had to express the dynamic behaviour of a given circuit at the same level as the reasoning about circuits and signals, thus confusing the two different levels of reasoning.

A theory, T, of a circuit, C, consists of three sub-theories:

- The description of the circuit (D)
- The truth tables of the gates (TB)
- The general laws of logic circuits (G)

2-1 The Description of a Circuit Module (D)

D is a set of assertions that describes the internal structure of the circuit module, its inputs and its outputs.

The gates of the module are labelled by a set of names. The input and output terminals of the gates, too, are labelled by a set of names. These labelled terminals are called the nodes of the circuit.

e.g.

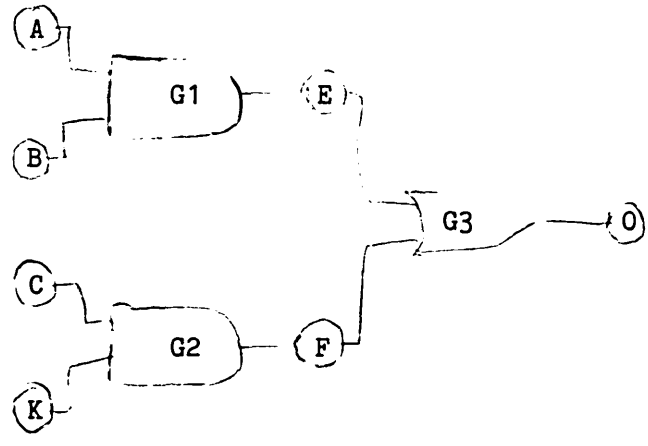


Fig. 1- the schematic diagram of C1,

showing the nodes

In C1

A,B,C,K,E,F,O are nodes,

G1 and G2 are AND gates,

G3 is an OR gate,

A,B are the input nodes of G1,

E is the output node of G1,

A,B,C,K are the input nodes of the circuit,

O is the output node of the circuit.

D is the description of a circuit iff:

-For every gate in the circuit it includes an assertion of the form
((GATE <gate-name> <gate-type> <input-nodes> <output-node>)) where:

<gate-name> is the name of the gate,

<gate-type> is the type of the gate (AND, OR etc.)

<input-nodes> is a list of the input nodes of the gate,

<output-node> is the output node of the gate.

-It includes an assertion such as

((INPUT-NODES <i>)) where <i> is a list of the input nodes of the circuit;

-It includes an assertion such as

((OUTPUT-NODES <o>), where <o> is a list of the output nodes of the circuit.

e.g this is a description of C1

((GATE G1 AND (A B) E))

((GATE G2 AND (C K) F))

((GATE G3 OR (E F) O)

((INPUTNODES (A B C K)))

((OUTPUTNODES (O)))

2-2 The Truth Tables (TB)

TB is a sub-theory that specifies the truth tables of the gates in the circuit.

The four place relation

(TABLE <gate-name> <gate-type> <input-list> <output>)

is used for representing the truth table of a gate. In the above relation,

<gate-name> is the name of the gate;

<gate-type> is the type of the gate (AND, OR etc.);

<input-list> is the list of the states of the input nodes of the gate;

<output> is the state of the output.

TB has the following property:

For every gate g in the circuit,

TB |- (TABLE <gate-name> <gate-type> <input-list> <output>) iff

when the input state specified by <input-list> is applied to the gate named by <gate-name>, the output state that results is the same as <output>. For example, for a normal AND gate named G,

TB|- (TABLE G AND (H H) H) etc.

Notice that we must be able to represent faulty gates as well as normal gates. Thus TB has a set of assertions which specify which gates are normal and which faulty. These points are discussed in more detail in section 5-1.

2-3 The General Laws of Logic Circuits (G)

Every node in the circuit can be in one of two states: High or Low. G

defines the relationship between the states of the nodes in terms of their interconnection through the gates. In particular, the state of the output nodes in terms of the states of the input nodes are derivable from G.

The main predicate of G is

(PREDICT <input-states> <output-states>)

where:

<input-states> is the list of the states of the input nodes,

<output-states> is the list of the states of the output nodes.

Here is a listing of G; it is the same for all circuits.

```
((PREDICT input-states output-states)
     (INPUT-NODES input-nodes)
     (OUTPUT-NODES output-nodes)
     (PRED1 input-nodes input-states output-nodes output-states))
```

Comment: to predict the state of the output nodes from the state of the input nodes determine what these nodes are, and invoke the four-argument relation PRED1

```
((PRED1 input-nodes input-states () ())
```

```
((PRED1 input-nodes input-states (node|restnodes) (state|reststates))
     (SINGLE-NODE-PREDICT input-nodes input-states node state)
     (PRED1 input-nodes input-states restnodes reststates))
```

Comment: to predict the state of the output nodes from the state of the input nodes, predict the state of the first output node, and then recursively predict the state of the rest of the output nodes.

```
((SINGLE-NODE-PREDICT input-nodes input-states node state)
     (MEMBER-OF input-nodes node)
     (STATE-OF input-nodes input-states node state))
```

Comment: if the node the state of which you are trying to predict is an input node, then just look up its state.

```
(SINGLE-NODE-PREDICT input-nodes input-states node state)
     (GATE gate-name gate-type gate-input-nodes node)
     (PRED1 input-nodes input-states gate-input-nodes gate-input-states)
     (TABLE gate-name  gate-type gate-input-states state))
```

Comment: if the node the state of which you are trying to predict is the output of a gate, then predict the states of the input nodes to the gate, and then look up the output of the gate, given the predicted inputs, in the truth table for the gate.

```
((MEMBER-OF (x|rest) x))

((MEMBER-OF (x|rest) y)
     (MEMBER-OF rest y))

((STATE-OF (node|restnodes) (state|reststates) node state))

((STATE-OF (x|restnodes) (y|reststates) node state)
     (STATE-OF restnodes reststates node state))
```

3- Some Terminology

Input Term

An input term is a list of the states of the input-nodes in the order

specified by the INPUT-NODES assertion.

Output Term

The output term is a list of the states of the output-nodes in the order specified by the OUTPUT-NODES assertion.

Outcome of an experiment

<O> is the outcome of the experiment E(<I>) if <I> is an input term, <O> is an output term, and when the input nodes of the circuit are set to the states specified in <I> the output <O> is observed.

IO Pairs

(<I> <O>) is an IO pair for the circuit C if <I> is an input term, <O> is an output term, and the outcome of the experiment E(<I>) on the circuit C is <O>.

Predicts

The theory T predicts the pair

(<input-states> <output-states>) iff

T|-(PREDICT <input-states> <output-states>)

Discriminating Input

<I> is a discriminating input for the theories T1 and T2 iff:

T1 predicts (<I> <O1>)

T2 predicts (<I> <O2>)

<O1> is different from <O2> in the state of at least one node.

4- Hypothesising about the Location of Faults

One way to view the fault finding process is as a process whereby a theory

of the faulty circuit, T', is derived from the known theory of the normal circuit, T. The technique used here to derive T' is a variant of the hypothesise and test paradigm.

Let C be the normal circuit, and T its associated theory. The faulty circuit, C', is identical to C, except for the fault in it. It is required to derive T', which is a theory of C', from T by experimenting on C'.

Let (<If> <Of>) be a known faulty input output pair for C'. By faulty we mean that had the experiment E(<If>) been conducted on the normal circuit C, the outcome would have been different from <Of>.

The steps involved in the derivation of T' are as follows:

a) from T and (<If><Of>) construct H, the set of hypotheses for the faulty circuit, such that :

   -all the hypotheses in H predict (<If><Of>);

   -H is guaranteed to include the theory of C'.

   We call H the hypothesis set. (A hypothesis is a potential theory of the faulty circuit)

b) Devise a discriminating input, <Id> for two elements of H

c) make the experiment E(<Id>) on the circuit C' and observe the outcome <Od>

d) remove from H all the hypotheses that do not predict (<Id> <Od>)

e) repeat from b until the hypothesis set contains just one hypothesis, or a discriminating input can not be found at step b.

Since H is guaranteed to include T', the theory of the faulty circuit, if

at the end of this algorithm only one element remains in H, then it must be T_1. It is possible, however, that no discriminating input exists for two members of H (i.e. they predict identical outputs for all input combinations). In this case, H will contain two or more hypotheses, which cannot be reduced by experimenting on the circuit.

Example-

Assume that C1 (the example circuit discussed before) develops a fault. Let us call this faulty circuit C1', to distinguish it from the normal circuit.

Assume that when the input nodes of C1' are set to the combination shown in fig. 2, we observe the output depicted. In our terminology, we say that the outcome of the experiment E((H H H L)) is (L) and that ((H H H L)(L)) is a faulty IO pair for the circuit C1'.

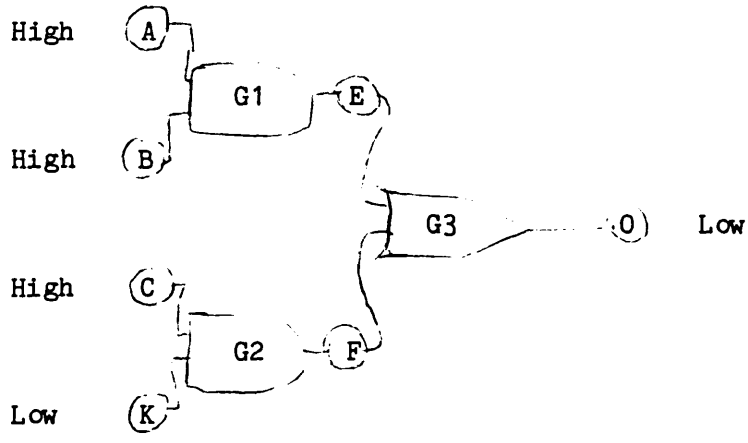


Fig. 2- the schematic diagram of C1',
showing the faulty input output

The first step in the fault finding process is the construction of the hypothesis set. To do this, we try to predict the output shown above from

the input, and analyse the unsuccessful attempt in order to derive the hypothesis set.

We have defined a special purpose provability predicate called D-DEMO for obtaining the trace of execution in the manner that we wish. D-DEMO is a four place predicate:

((D-DEMO <prog> <goal-list> <trace> <status>)

as with S-DEMO specified earlier, D-DEMO has a status parameter. If it fails to find a proof of <goal-list> from the <prog>, it succeeds with the <status> bound to FAIL, otherwise it succeeds with <status> bound to SUCCESS. In both cases, <trace> is the search tree of the (attempted) proof.

Thus we run the goal

?((D-DEMO T ()(PREDICT (A B C K) (H H H L)) trace FAIL)

and analyse the value returned for trace, which is the search-tree of the unsuccessful attempt to prove PREDICT from the theory T.

Following this analysis, we conclude that there are two hypotheses which 'explain' the faulty input output pair observed (the details of the way we analyse the trace will be discussed later). These hypotheses are:

a- The gate G3 is faulty, and its truth table is (TABLE G3 OR x L) (i.e. its output is 'stuck' at Low).

b- The gate G1 is faulty, and its truth table is (TABLE G1 AND x L)

Next, a discriminating input is devised (again, the details are discussed

later). In this case, a discriminating input is

(L L H H)

It is a discriminating input because if hypothesis a were true, then the application of this input to the circuit would result in the output being low, whereas if the hypothesis b were true, then its application would result in the output being high.

The discriminating input is applied to the circuit, and the output observed. We call this experiment a discriminating experiment. Assume that output terminal is high after the application of the discriminating input.

Following the experiment, all the hypotheses that do not predict its outcome are rejected. Here hypothesis a does not predict the observed outcome of the experiment, and it is removed from the hypothesis set. Since only one hypothesis remains in the hypothesis set at this stage, namely hypothesis b, b is the theory of the faulty circuit, C1'.

5-Constructing the Hypothesis Set

The hypothesis set contains a set of hypotheses, or prospective theories, of the faulty circuit, C'.

Every element of the hypothesis set must have the following properties:

- It must predict the faulty input-output pair (If, Of);

- It must include the D and G sub-theories intact (this is due to the fact that, by assumption, a fault can not change the structure of the circuit or the general laws of logic circuits)

- It must include a sub-theory such as TBf from which the truth tables of all the gates are derivable. Furthermore, since we assume that only one gate is faulty, the truth tables derived from TBf must be identical with those derived from TB except for the one relating to one of the gates.

The hypothesis set is built in two steps:

step 1-

Try to prove
(PREDICT input-states faulty-output)
from T and obtain the trace of the unsuccessful attempt,

step 2-

Analyse the trace and propose modifications to T that would make the proof successful. Put the modified instances of T in H.

Explanation of step 1-

The specially defined proof predicate D-DEMO is used for proving PREDICT from T. The definition of D-DEMO is such that a special goal-selection strategy is employed. This is to facilitate the analysis at step two. The sub-goal selection strategy is:

> Never choose a truth table goal for expansion as long as other goals are present in the goal set.[Truth table goals are those whose predicate is TABLE].

Thus at one stage in the proof, the goal-list will contain only truth table goals. Notice that due to the nature of the problem, for any given input this set of truth table goals is unique, whatever the search strategy for the other goals.

Explanation of step 2-

The proof tree is scanned until the point is reached where the goal

list contains nothing but truth table goals. Let us call this set of truth table goals B.

Obviously, if B were provable from TB, PREDICT would be provable from T. We know that TB can yield the truth tables of all gates in C` but one. It is the job of the Hypothesiser to find out which gate this one can be, and alter TB in a way that reflects this.

The algorithm for constructing the hypothesis set is as follows:

1- Pick one of the goals in B and call it g. Call B-g B'

2- Prove B' from TB. Call the substitution obtained **alpha**. If proof is unsuccessful skip to 6

3- Apply **alpha** to g. Call the resulting goal g'

4- Modify TB so that g' is provable from it. Call the resulting sub-theory TBf. (The way this is done is explained in the subsection 5-1)

5- Add D+G+TBf to the hypothesis set

6- Start from the beginning, picking another goal this time. Continue in this way until all the goals in B have been chosen.

Example-

We use the same example as was used in section 4.

When the proof of

?(PREDICT (H H H L) (L))

from the theory of C1 is attempted, at one stage we will have the goals

?((TABLE G1 AND (H H) x)(TABLE G2 AND (H L) y)(TABLE G3 OR (x y) L))

Obviously this set of goals cannot be proved from TB. (Try it if you do not believe this!)

This is how FAULTFINDER produces the hypotheses that were mentioned earlier:

1- select a goal in this set. e.g. (TABLE G1 AND (H H) x). Call it g. Call the remaining goals B'.
[Here B' is ((TABLE G2 AND (H L) y)(TABLE G3 OR (x y) L))]

2- Prove B' from TB. Call the substitution employed **alpha**. Here **alpha** is (y/L,x/L)

3- apply **alpha** to g'. Call the resulting goal g'. Thus we apply (y/L,x/L) to (TABLE G1 AND (H H) x) and obtain (TABLE G1 AND (H H) L)

4- Modify TB so that g' is provable from it. Add the resulting theory to the hypothesis set. This would be the hypothesis b in this example.

5- start from 1 again, this time choosing a different goal, e.g. (TABLE G2 AND (H L) y) etc.

5-1 Deriving the Hypothetical Truth Tables

TB, which is the sub-theory that contains the truth tables of the gates, is listed below:

((TABLE g AND (H H) H)(NORMAL g))
((TABLE g AND (H L) L)(NORMAL g))

((TABLE g AND (L H) L)(NORMAL g))

((TABLE g AND (L L) L)(NORMAL g))

((TABLE g OR (H H) H)(NORMAL g))

((TABLE g OR (H L) H)(NORMAL g))

((TABLE g OR (L H) H)(NORMAL g))

((TABLE g OR (L L) L)(NORMAL g))

and similarly for normal NOT, NAND, NOR gates.

TB contains the following clauses for defining the truth tables of faulty gates:

((TABLE gate type input H)(STUCK-AT-HIGH gate))

((TABLE gate type input L)(STUCK-AT-LOW gate))

TB also includes assertions about the status of the gates, i.e. whether they are normal, stuck-at-high or stuck-at-low. For the normal circuit, we would have assertions such as

((NORMAL G1))

((NORMAL G2)) etc.

for all the gates.

Recall that to generate a hypothesis, we need to modify TB such that a certain truth table goal, as determined by the process described in the previous section, is provable from it. At the same time, we want the truth tables for all the other goals to remain unaltered.

In accordance with the assumption we have made about the nature of the faults occurring in a gate, which limits us to stuck-at faults, to alter TB in the manner described above it is sufficient to delete the normality

assertion for the gate concerned, and add either a STUCK-AT-HIGH assertion or a STUCK-AT-LOW assertion for the gate. Which of the above assertions we add depends on whether we have a high or a low in the output of the truth table goal.

Example:

Consider the step 4 in the previous example [modify TB such that g' is provable from it etc.]. Here we want to modify TB such that the goal

(TABLE G1 AND (H H) L)

is provable from it. To do this, we delete the assertion

(NORMAL G1)

and add (STUCK-AT-LOW G1)

6- Devising a Discriminating Input

Take two elements of the hypothesis set, H1 and H2. We want an input term, Id, such that:

H1 predicts (Id, O1), and

H2 predicts (Id, O2),

where O1 and O2 are output terms, and the state of at least one of the output nodes is different in O1 and O2.

The discriminating input for H1 and H2 is derived by running the following (meta-level) goal:

?((DISCRIMINATE H1 H2 disc-input))

where the definition of DISCRIMINATE is:

```
((DISCRIMINATE theory1 theory2  discriminating-input)
  (DEMO theory1 (()(PREDICT discriminating-input out1)
  (DEMO theory2 (()(PREDICT discriminating-input out2)
  (DIFFERENT out1 out2))
```

where 'discriminating-input', 'out1' and 'out2' are meta-level variables ranging over ground object level terms.

On successful execution of the above goals, the instantiation to 'discriminating-input' will be a discriminating input for H1 and H2, because they predict different outputs for it.

The derivation of a discriminating input using the above method is a good example of the utility of meta-language programming. It would be difficult to formulate DISCRIMINATE without having an explicit provability predicate.

Notice that the above formulation for DISCRIMINATE, though runnable, is very inefficient. As such, it should be regarded as the specification for the DISCRIMINATE predicate. Significantly more efficient programs can be devised for evaluating DISCRIMINATE.

7- Some Related Work

The field of fault-diagnosis in digital circuits is huge, with its own specialised literature. The work presented here is not primarily intended to be a contribution to this field. Our intention has been to use this application as a testing ground for the ideas presented elsewhere in the thesis. Thus we feel justified in not making reference to the fault diagnosis literature. Even so, there has been some recent work on the application of logic to fault-diagnosis which needs to be mentioned here,

if only to contrast the techniques used.

The work in question is presented in [Reiter 1985]. It is a theoretical elaboration of earlier work reported in [DeKleer 1976]. It provides a foundation for expressing the diagnosis problem in logic. The essentials of the technique it uses are as follows:

The system in question, S, is described using a set of sentences of logic, SD . SD includes sentences which assert that the components of the system are not abnormal. SD is called the system description.

A particular behaviour of the system (e.g. an input-output pattern in a circuit) can be formulated using another set of sentences, OBS. OBS is called the system observation.

The formulation of SD and OBS is such that if the observations correspond to a faulty system then the set

OBS U SD

is inconsistent.

Once it is found that OBS U SD is inconsistent, [Reiter 1985] provides an algorithm for formulating hypotheses about which components are faulty. A hypothesis about faulty components is called a diagnosis. In [Reiter 1985] terminology, a diagnosis is a minimal set of components such that when the statements as to their normality are withdrawn from SD, consistency is restored to OBS U SD. Clearly, there might be more than one diagnosis for a given SD and OBS.

7.1- Comparison With FAULTFINDER

The main difference between the technique used in FAULTFINDER and that

reported in [Reiter 1985] is that FAULTFINDER does not derive an inconsistency when an abnormal input-output pattern is presented to it; rather, it fails to predict this I/O pattern. As a result, whereas Reiter's algorithm is based on the restoration of consistency, our algorithm is based on proof analysis, i.e. analysis of the reason for the failure of the prediction.

Reiter's work does not consider making further experiments to narrow down the number of 'diagnoses'. In the case of FAULTFINDER, however, after it has generated a hypothesis set, it goes on to an experimentation phase to eliminate as many of these as possible.

8- Conclusion

We have demonstrated how the ideas of meta-level programming, i.e. programs as data structures, proof analysis, control of the proof process etc. can be used for implementing a non-trivial diagnosis program. As such, FAULTFINDER is a successful instance of the application of the ideas discussed in the previous two chapters. Its practical utility, however, is restricted by the range of circuits it can handle, and the assumptions it makes about the nature of the faults.

To extend FAULTFINDER into a general purpose fault diagnosis program, it will have to deal with the following situations:

- It would have to work at higher levels of integration than gates. It should be able to tackle circuit modules, e.g. integrated circuits, as building blocks.

-It should be able to handle other types of faults apart from the ones we have considered, i.e. single, stuck-at faults.

- It should be able to work on sequential circuits, i.e. circuits which

have memory elements in them.

The incorporation of circuit modules as building blocks does not pose any difficult problems; they can be considered as 'generalised gates' and the same general scheme can be applied to them. Similarly, by generalising some of the predicates and algorithms, multiple faults and faults other than the 'stuck-at' type can be handled. Handling of sequential circuits, however, is intrinsically much more difficult than combinatorial circuits, and different techniques need to be employed to deal with them. This is due to the fact that with sequential circuits, it is the history of the inputs to the circuit which determine the output, not the input at the time of observing the output. This adds a whole new dimension to fault diagnosis and calls for new techniques, such as heuristic methods (as in conventional expert systems).

Chapter Four

Comparison with Related Systems

In this chapter we compare the system developed in this thesis, which we have called MOL, with other systems which share some of the objectives of MOL. The systems which we will consider are the following:

K&B - The system proposed in [Bowen and Kowalski 1982]

FOL - The FOL system [Weyhrauch 1980]

PRESS-The PRESS system [Bundy and Welham 1981]

KS -The system proposed in [Konolige 1981]

1-Comparison with the System Developed in [Kowalski 1979] and [Kowalski & Bowen 1982]

MOL is a derivative of K&B, and in this section we shall discuss how it has 'specialised' some of the general concepts set out in K&B.

K&B sets out a general framework for amalgamating object-language and meta-language in the context of a logic programming system. The following are the main devices used in K&B for achieving its aims:

a- a naming relation to map linguistic objects of the object-language to the data-structures of the meta-language;

b- the concept of representability to link relations on a domain to predicates of a language in the context of a theory,

c- the meta-predicate, DEMO, which represents the provability relation on the domain of object-language objects,

d- the amalgamation of the object-language and meta-language to provide self-reference.

The above framework also applies to MOL. The difference between MOL and K&B is that whereas K&B discusses the above concepts in general, not committing itself to any particular choice on the various possible options, MOL is more specific. Below we list some of the ways MOL has particularised the options in K&B.

a- In MOL the syntax of the language, and the naming relation adopted, have been described in some detail, and the reason for choosing this particular syntax and naming relation are discussed. K&B, on the other hand, is non-committal on the choice of naming relation, and (to some extent) the syntax of its language, though it does concentrate on Horn-Clauses.

b- In MOL, the definition of DEMO has been specialised to take account of the various demands made of it (e.g. determinancy, boundedness etc.). In K&B, the definition of DEMO is for illustration only, and is quite general.

c- In MOL, the way self-reference is included in the system is specified in some detail. The use of scoping and inheritance axioms is a novel feature of MOL. In K&B, this issue is not elaborated, largely because the details depend on the choice of the naming relation, and K&B is not committed to any particular naming relation.

d- In MOL, the use of meta-language for the purpose of controlling the execution of object-level programs is developed in some detail. In particular, the use of upward reflection for this purpose is new. K&B only

makes a passing reference to this issue by including a control parameter in the definition of DEMO.

To summarise, MOL is an elaboration of the general framework set out in K&B.

2- Comparison with The FOL system

FOL is a proof checker for a many sorted first order language with equality. It relies on human interaction for driving the derivation process.

2.1.1 The FOL Deductions

FOL provides facilities for making formal derivations using "natural deduction" rules of inference. To use this facility, one first declares the language to be used by declaring the various constant, function and predicate symbols and their sorts, if appropriate. Then one enters the axioms of the theory of interest. Having done this, FOL commands can be used for making inferences on these axioms and proving theorems.

2.1.2 Syntactic Simplification

FOL provides a syntactic simplifier. This facility allows the user to specify certain axioms as re-write rules. A re-write rule is a sentences whose main connective is an equality or equivalence sign and which has universally quantified variables whose scope is the whole sentences. The application of a re-write rule is in two steps: instantiation of the universally quantified variables, and substitution of the left hand side of the equation with the right hand side. For example, if we have the sentence

∀x ∀y (FATHER(x y)<->PARENT(x y) & MALE(x))

as a re-write rule, we can apply it to the formula

FATHER(x ,MARY) & MARRIED(x)

to get the formula

PARENT(x mary) & MALE(x) & MARRIED (x)

similarly, if we have the equation

∀ n factorial(n)=IF n=0 THEN 1 ELSE n*factorial(pred(n))

as a re-write rule, we can use it to simplify

factorial(3)

to

IF 3=0 then 1 ELSE 3*factorial(pred(3))

2.1.3 Semantic Attachments and Simulation Structures

In FOL, a mechanism called semantic attachment can be used for linking the sorts, functions and predicates of the language to LISP data-structures and programs. For example, the LISP function ADD can be attached to the function + in the language, and the LISP data structure INTEGER can be attached to the sort Nat-Number. The semantic attachment mechanism can be used for 'semantic evaluation': for example, to prove that 1+2=3,(say), we can use the semantic evaluation mechanism to calculate the value of 1+2, and then use the equality axioms to verify that the resulting integer is equal to 3.

The attachments to the sorts, functions and predicates of a language, taken as a whole, comprise what is called a simulation structure of the language.

Together with the syntactic re-write system, the semantic evaluation mechanism can be used for simplifying formulas of the language. In the words of Weyhrauch, "simulation structures are intended to be the mechanisable analogues of the notion of model" i.e. they add meaning/significance to the syntactic constructs.

2.1.4 LS Pairs or Theories

The combination of a language, a simulation structure and a set of sentences in the language is called is called an LS pair or a theory.

The FOL system always has its attention directed at an LS pair. As such, LS pairs can be considered as the objects which FOL manipulates.

The meta-level facilities of FOL use the semantic attachment mechanism. Through this mechanism, meta-linguistic sorts are attached to the corresponding underlying data-structures of FOL, and meta-level functions and predicates are attached to the programs which are an integral part of the FOL engine.

2.1.5 The LS Pair META

The LS pair META in FOL contains all the meta-level information in the system. Its language has sorts, functions and predicates which correspond to objects and relations which exist in LS pairs. For example, the language of META has predicates THEOREM and WFF, the sorts 'expression' and 'theorem' and the function 'and' and 'or' etc. (Note: throughout this section, to avoid confusion, we shall use the name METAX for MOL's META)

The axioms of META formulate the properties of LS pairs and the relationship between their various components.

The simulation structure of META consist of attachments to the sorts, functions and predicates of META. Since META is concerned with properties

of LS pairs, the routines in the FOL engine itself can be attached to its functions and predicates, and the data-structures used by FOL for representing the object LS pair can be attached to its sorts and constants. In other words, the simulation structure of META is the FOL engine itself.

2.1.6 The Reflection Mechanism

The reflection mechanism in FOL is an extension of the semantic evaluation mechanism. Downward reflection is achieved by simply evaluating the function in question using its semantic attachment. Upward reflection is more involved, and proceeds as follows:

-Assume FOL is currently working on the LS pair OBJECT. During the course of a proof, the command

REFLECT <sentence> <par1> <par2> .... <parn>

can be issued. In the above command,

<sentence> is the name of a fact in the LS pair META, (facts are sentences which are held to be true in an LS pair)

<par1>, <par2>...<parn> are names of sentences in the current LS pair, i.e. in OBJECT.

For this command to make sense, the LS pair META must have a fact whose name is <sentence>, and which has n universally quantified variables of the appropriate sort at the front.

When this command is issued, control is passed to the LS pair META. The fact whose name is <sentence> is invoked, with the universally quantified variables instantiated to the meta-level representation of the sentences <par1>, <par2> etc. Then the simplifications system is invoked to simplify the resulting sentence of META. Once this is done, the result is passed

back to the original LS pair, i.e. OBJECT, and control is also passed to this OBJECT.

Example:(from [Weyhrauch 1980])

Suppose in the LS pair META we have the axiom

ANDI: ∀ thm1 thm2 THEOREM(and(wff(thm1),wff(thm2)))

This axiom states that if we have any two theorems in an LS pair, then we get a new theorem by taking the conjunction of the two well-formed formulas associated with the theorems. (We are assuming that the variables thm1 and thm1 are declared to be of sort theorem.)

We can use this axiom of the meta-theory to assert the conjunction of two theorems as a new theorem. To do this, we proceed as follows:

During the course of a proof in the LS pair OBJECT (say), we issue the command

REFLECT ANDI T1,T2;

where T1 and T2 are the names of two theorems in OBJCET, and we want to assert that their conjunction is also a theorem.

What happens then is as follows:

FOL directs its attention at the theory META, and looks for the fact named ANDI. Once it finds this fact, it instantiates the two leading universally quantified variables to the representation of the two sentences in the theory OBJECT whose name is T1 and T2. Let us call the resulting sentence INSTAND. Then FOL runs the simplification system (encompassing both the

semantic and syntactic simplification modules) to simplify INSTAND. Once it is simplified as far as possible, it switches back to the SL pair OBJECT, and assert the result of its computation. In this case, since the simplified INSTAND at META is a sentence of the form

THEOREM(<result>)

it asserts the sentence <result> to be a theorem of OBJECT. (<sentence is the representation of <sentence>)

In practice, when META wants to work on the representation of a theorem, say T1, in OBJECT, instead of explicitly constructing a term in the language of META, it constructs a constant of type 'theorem' and attaches the FOL data structure used for representing T1 to this constant.

2.1.7 - Self-reference

Since META is an LS pair like any other, it can be the object theory to itself. Thus by issuing the REFLECT command when in the LS pair META we can have self-reference.

Suppose we issue the command

REFLECT S T1,T2;

while doing a proof in META. What happens is that FOL's attention is directed at META (i.e. its attention remains directed to META, because it was directed at META before), the sentence S is instantiated with the representations of T1 and T2, and simplified. The way this is done is that the data-structures used for representing the sentences T1 and T2 in META are attached to two constants, and the leading variables of S are instantiated to these constants. This can lead to problematic behaviour, because the process of proof at the meta-level can itself change META (for

example by adding new theorems to it), which will in turn change the object of reasoning, i.e. META itself. As documented in [Weyhrauch 1980], this can lead to incorrect results. We shall say more about this in the next section.

2.2- Comparison of FOL and MOL

FOL handles much more extensive languages than MOL. FOL's languages are many sorted first order languages, whereas MOL's language is Horn-Clause logic.

This difference in the choice of language underlies another, more important difference: FOL is essentially a proof checker, i.e. it relies on human interaction for driving the derivation process, where MOL is a logic programming system, i.e. a theorem-prover specially trimmed to run automatically and quickly.

In this sense, as in the scope of the languages involved, MOL and FOL are at the two opposing ends of the spectrum of possibilities.

Yet MOL and FOL have many things in common. The most important common feature of MOL and FOL is that they both have meta-language features extended to include self-reference. This is a fundamental similarity, because the issues involved force the adoption of many similar techniques and approaches.

The objective of the current comparison is to see how the choice of languages, i.e. a 'maximalist' versus a 'minimalist' approach to the languages, has affected the overall structure of the two systems and their properties.

2.2.1 Semantic attachment Versus Representability

In FOL, the mechanism of semantic attachment and semantic evaluation is

provided to give the system a degree of 'programmability'. As mentioned earlier, FOL is a proof checker, i.e. it relies on human interaction to drive the derivation process. The mechanisms of semantic attachment and evaluation provide some means of 'automating' this process, especially when used inside the theory META where the semantic attachment to meta-linguistic functions and predicates can effectively act as 'macro' derivation steps, thus relieving the tedium of step by step proof at the object level.

It is important to emphasise, however, that the role of semantic attachments in FOL is not a purely practical one. In fact, in a theoretical sense, they augment and specialise the axioms in a given LS pair. Thus, when the ADD function of LISP is attached to the function + in an LS pair, this function has a very different meaning and significance than in the context of an LS pair in which nothing is attached to it. We must take seriously the designation of simulation structures as 'mechanisable analogues of a model' for the language, as without these attachments the language and the axioms formulated in it will have very different theoretical properties.

The concept of representability in MOL, however, is a purely practical device. It is only introduced to warrant the use of special purpose mechanisms to evaluate predicates of the language. The crucial difference with FOL is that due to the definition of representability, we could have evaluated these predicates even in the absence of the special purpose mechanisms, and would have got the same result. The only difference between using the special purpose mechanisms and using the explicit definition of the predicates is speed.

So, in a very important sense, MOL theories are 'self-contained', whereas FOL theories (sets of axioms) are not. In fact, this is the reason for designating LS pairs, which include the semantic attachments to the

language, as the unit of knowledge in FOL.

2.2.2 Meta-Theory in FOL and MOL

As mentioned before, in FOL meta-linguistic features are provided through the LS pair META. META has a language which includes sorts, functions and predicates corresponding to various object level constructs and relationships. Through the mechanism of semantic attachment, the appropriate data-structures/routines in the FOL engine are attached to these meta-linguistic constructs.

In MOL, too, the theory METAX contains the meta-linguistic axioms which formulate the properties and relationships of the object language. As in FOL, the link between the object language and meta-language is provided through a naming relation.

Now all the comments that were made concerning the mechanism of semantic attachment in FOL apply to the LS pair META, and therefore on the formulation of meta-linguistic properties. In other words, META relies on the routines and data-structures of the FOL engine as the 'mechanised model' of the meta-language. Thus, in a theoretical sense, the axioms of META rely on the procedures and data-structures of FOL for complete articulation. This is in contrast to MOL, in which the theory META is self-contained, and can be studied independent of any particular implementation.

In particular, (downward) reflection in FOL, which is nothing more than the mechanism of semantic evaluation on meta-linguistic functions and predicates, is an integral part of the meta-linguistic unit of knowledge, (i.e. the LS pair META). In MOL, by contrast, reflection is a purely practical adjunct: it can be dispensed with altogether, at the cost of slower execution speed. In a strict, theoretical sense, MOL's METAX does not need reflection.

The reliance of FOL's META on the mechanism of semantic attachments results in a certain degree of haziness as to the theoretical properties of META. This is because, with a machine as complex and intricate as the FOL engine, it is very difficult to specify in detail the theoretical properties of all the data-structures/routines. Furthermore, since semantic attachments are only 'partial' models, it would seem to be very difficult to pin down the repercussions of this 'partialness' on the theoretical properties of META.

Notice that while these comments apply to all FOL LS pairs, their significance is much greater when applied to META. This is because, through the mechanism of upward reflection, META can effectively change (augment) the behaviour of the FOL object level proof checker. Thus it is much more important to be precise about the properties of META than any other simulation structure. To use an analogy, while it might be acceptable to have a less than perfect program, it is not acceptable to have a less than perfect compiler, because it would jeopardise the run time correctness of all the programs which are compiled by it.

By contrast, the METAX theory in MOL is self-contained: its properties are determined by the axioms contained within it. Thus, it is much easier to reason about its various aspects, e.g. the correctness of its formulations.

2.2.3 Self-Reference in MOL and FOL

As mentioned earlier, in FOL self-reference is achieved through the application of META to itself. In MOL, self-reference is achieved through the inheritance and scoping axioms, as discussed in the previous chapters.

As a result of the mechanism employed to achieve self-reference in FOL, namely the semantic attachment mechanism, the treatment of self-reference in FOL is unsatisfactory. This is because FOL literally attaches

components of the objcet of reasoning (i.e. the simulation structure META) to the subject doing the reasoning(again the theory META). Thus the process of reasoning at the meta-level changes the object of reasoning, which is a sure recipe for disaster, because there will arise cases when depending on which way you go about the derivation at the meta-level, you get different results. For example, suppose you ask the system how many theorems are known to META with the meta-predicate THEOREM appearing in them. Whatever the answer is before this question is asked, it will be one more after it is answered, because the process of answering itself adds one more such known theorem to META!

In MOL, however, the simplicity of the language and the self-contained nature of METAX enable us to have a simple and powerful mechanism for self-reference (i.e. inheritance and scoping axioms), a mechanism which is independent of any particular implementation, and open to theoretical scrutiny. In particular, MOL does not use semantic attachment (or its eqivalent in MOL, the reflection mechanism) for specifying self-reference. Therefore, there is no equivalent in MOL to the phenomenon mentioned above, i.e. the object of reasoning changing as the reasoning proceeds, which makes FOL's self-reference dubious.

2.3- Summary of the comparison between FOL and MOL

In FOL, the choice of a 'maximal' language (many sorted first order logic) has resulted in the adoption of LS pairs, which include simulation structures as the 'mechanisable analogue of the model' for the language, as the basic unit of knowledge. Due to the ambiguous theoretical properties of semantic attachments, the theoretical properties of the meta-linguistic formulation (as encapsulated in the LS pair META) are vague. This vagueness reaches unacceptable levels when self-reference is built on top of the meta-linguistic framework.

In MOL, due to the choice of a 'minimalist' language (restricted Horn-

Clause Logic) it is possible to adopt a 'logic programming' approach, i.e. to use the language itself as a programming language. As a result, the theories of MOL are self-contained, and there is no need to refer to any particular implementation (or outside programs) for elaborating their theoretical properties. When extended to the theory METAX, this property of the theories of MOL enables us to have a self-contained meta-theory of MOL. The simplicity and self-contained nature of METAx leads to the straightforward implementation of self-reference through the inheritance and scoping axioms.

3-Comparison with PRESS

As stated in [Bundy & Welham 1981], "PRESS is a computer program for solving equations and doing other kinds of algebraic manipulations on transcendental expressions, i.e. expressions involving polynomial, trigonometric, exponential and logarithmic functions". PRESS uses a set of re-write rules for solving equations. These rules, which are in fact axioms of algebraic equality, constitute the object-level knowledge of the system, i.e. its knowledge about the domain of interest.

PRESS is a large program, and has many features. For the purpose of this comparison, however, we are only interested in the meta-language features of PRESS.

3.1 Meta-Level Knowledge in PRESS

The meta-level knowledge of the system consists of knowledge about the algebraic re-write rules, and the situations in which they are to be applied. It is the duty of the meta-theory to formulate a plan for solving an equation in terms of the object-level re-write rules.

As observed in [Bundy & Welham 1981], if the object-theory, i.e. the set of re-write rules, were set loose on an equation without the guidance of the

meta-level, this would have led to an inefficient, and perhaps non-terminating, equation-solving system. This is because the re-write rules have a high branching rate, and this can lead to combinatorial explosion.

The use of the meta-level knowledge for formulating the heuristics used for guiding the object-level problem solver, however, has led to a usable equation solving system.

3.2 Meta-Language Methodology of PRESS

In PRESS, the object-level knowledge does not have a separate existence. Rather, it is encoded within the meta-level rules. The meta-level is a PROLOG program, which can be considered as a theory in predicate calculus.

The definition of the object-level re-write rules is through meta-level clauses which define the consequence of applying the rule to an equation. These clauses are complemented by other meta-level clauses which determine the way the search for a solution should proceed.

3.3 Comparison with MOL

In MOL, the object-language and the meta-language are both Horn-Clause logic, whereas in PRESS the object-language is equational algebra, and the meta-language is PROLOG. In PRESS, as in MOL, meta-level constants are used for naming object-level variables, thus avoiding the pitfalls associated with naming object-level variables by meta-level variables (which is the common PROLOG practice)

In PRESS, the object-theory is implicitly encoded within the meta-theory, whereas in MOL, the object-theory and the meta-theory are distinct. PRESS achieves control of object-level problem-solving by simulating this problem-solving through the execution of the meta-level code; MOL, on the other hand, allows the object-level deduction to go ahead independently by the use of downward reflection, and makes use of meta-level control

information by upward reflection.

The PRESS strategy for exercising meta-level control is simple, and sufficient for the task at hand. This is because it is a special-purpose system for handling algebraic re-write rules, and they can be simulated efficiently using meta-level clauses.

MOL's approach is more complex, but this complexity is necessary because it would be grossly inefficient to simulate the whole of the object-level deduction at the meta-level, i.e. to run the DEMO predicate without reflection. There are two reasons for this:

a- The kind of object-level reasoning done in MOL, i.e. Horn-Clause resolution theorem proving, is more complex than that of PRESS, leading to a more complex (and inefficient) meta-level formalisation,

b- We already have a special-purpose object-level theorem prover, i.e. the interpreter on which the meta-level program runs. Since the object-language and meta-language are the same, it would be wasteful not to use this interpreter to run object-level programs.

4 - Comparison with the System Presented in [Konolige 1981]

KS is a multi-agent planning and reasoning system [Konolige 1981]. It adopts a syntactic approach to representing agents' beliefs and actions. In KS, each agent's theory of the world is defined as a set of statements in a first order language with equality. This language is OL, or the object language.

The meta-language, ML, is another first-order language with equality. Propositional attitudes such as knowing, believing etc. are formulated in ML, and refer to agents and their theories of the world.

4.1- The Naming Relation in KS

The interface between the object-language and the meta-language is a naming relation that maps terms and sentences of the OL into variable-free terms of the ML. For example, the constant C is mapped into the constant C', the term f(a,b) is mapped to the term f'(a',b'), and the sentence P&Q is mapped to the term and(P',Q').

As for quantifiers and variables, a sentence such as

∀x P(x)

is mapped to a term such as

forall(x',P'(x'))

where x' is a constant of the meta-language etc.

4.2- Important Meta-Predicates in KS

Certain meta-predicates are used to represent important properties of object-level theories. These predicates include TRUE(f), FACT(th,f) and PR(th,f). Below we give a short description of each of these predicates.

-TRUE

This meta-predicate signifies truth in the real world. Thus TRUE(f) means that the sentence f is true in the real world.

-FACT

FACT(th,f) means that the sentence f is one of the base facts of the theory th, i.e. it is assumed to be part of th.

-PR

PR(th,f) means that the sentence f is provable from the theory th.

The meta-theory includes axioms which define these and other meta-predicates in such a way that the usual first-order semantics of object-level sentences is maintained.

4.3- Comparison with MOL

4.3.1- The Question of Truth

In KS, the ML predicate TRUE tries to capture the notion of truth. Thus the meta-theory includes axioms which define the reasoning agent's knowledge of the world through this predicate. Furthermore, the 'knowledge' so codified is used as a reference for judging the other agents' beliefs, to decide whether these are mere 'beliefs', or actual 'knowledge', which is defined to be a belief that is also true.

Now, there is no question that a reasoning system that reasons about the beliefs of other agents needs to have a notion of 'truth', if only to distinguish between those beliefs of other agents which it agrees with, and those with which it does not. But is there any grounds for demarcating between those statements that the reasoning agent itself 'believes', and those which it 'knows'? In other words, if I am the reasoning agent, can I say that I 'believe' statement A, whereas I 'know' statement B, thereby implying that B is 'true', whereas A might or might not be true?

Our view is that the reasoning agent can not distinguish between his own knowledge and belief. In other words, anything that the agent believes, he thinks is true, for otherwise he would not believe it. Thus there is no reason to have two separate codifications of the real world in ML, i.e. one that codifies the agent's beliefs, which is the meta-theory itself, and one that codifies its 'knowledge', through the TRUE axioms.

In fact, after spending a great amount of time on discussing the TRUE

axioms and their relationship with the 'real world', KS equates truth with the ML's own theory of the world, through the 'standard meaning' axioms, which are of the form

a: TRUE(P')<->P

where P' is the name of the sentence P. But even this device does not help disentangle the web which has been wound around KS by the TRUE axioms, because if we allow the reasoning agent to think about its own theory of the world, i.e. if we allow the meta-theory to be self-referential, then axioms such as a above lead to inconsistency, as proved, for example, in [Perlis 1985]. And KS does allow the agent to reason about its own theory of the world.

So what is MOL's way of talking about 'truth'?

To start with, the meta-theory codifies all the knowledge of the reasoning agent - be it knowledge about other agents and their modes of reasoning, or knowledge about the world. Thus there is no distinction between what the reasoning agent believes, and what it knows.

As in KS, we can represent other agents' theories of the world as object-theories, and represent belief as provability from these theories. Thus, if the theory T is the agent A's theory of the world, we represent A's beliefs as:

```
((Believe A x)
     (DEMO T x))
```

Now, if the reasoning agent wants to know whether a belief held by another agent is 'true', in our system this can only be done by investigating whether the reasoning agent himself believes it. Thus we can define

AGREE-WITH (a less dogmatic term for true) as:

```
((AGREE-WITH x)
      (DEMO SELF x))
```

where SELF is the name of the reasoning agent itself. Alternatively, we can define AGREE-WITH through the axiom schema

```
((AGREE-WITH A)
              A)
```

where A is any possible sentence in the logic and A is the name of A .

This approach has two virtues: simplicity and honesty. It is much simpler than the codification of truth separately from other ML axioms, with all the problems (such as inconsistency in the self-referential case) that this entails. It is honest, because it recognises the fact that there is no way the agent can have access to 'truth' as something separate from its own beliefs and prejudices.

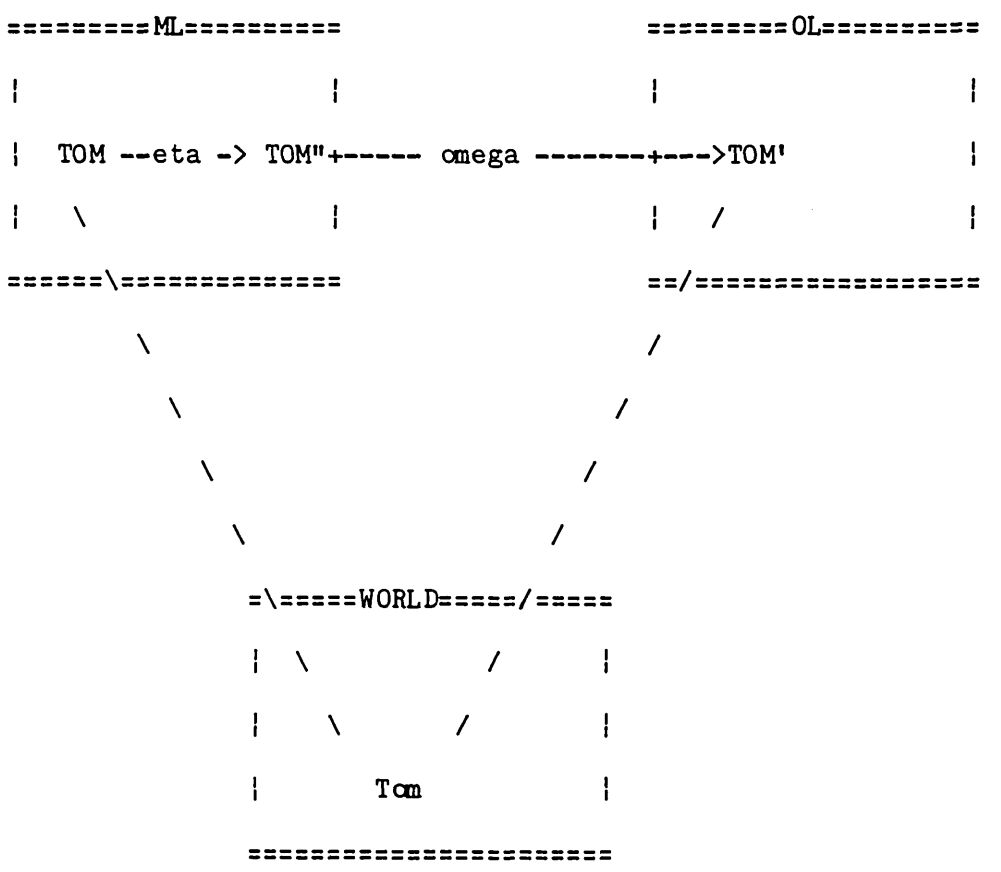
4.3.2 -The Question of Names

In KS, there are two naming relations. One is the mapping from the terms and sentences of the OL to the terms of the ML. Through this mapping ML sentences can refer to OL terms and sentences. We shall call this mapping omega. Omega is the 'sense quotes function' in KS.

The other mapping, which in KS is called the 'standard name function', is a mapping from those ground terms of ML which refer to objects of the 'real world' to the ground terms of ML which name the ground terms of OL which refer to the same objects of the 'real world'. Thus, if the ML constant TOM refers to the person named Tom, and the OL constant TOM' refers to the same person, and the ML constant TOM" names the OL constant TOM', the

standard name function maps TOM to TOM". We shall call this mapping eta.

Below is a pictorial depiction of sigma and eta and their inter-relationships with the world and the languages OL and ML.



4.3.3-The Naming Relation Omega

The naming relation omega mapping OL terms and sentences (or OL objects, for brevity) to ML terms is necessarily a function or relation outside ML. It can not be part of the ML, because if it is part of ML then its domain (i.e. the OL objects) need also be part of the ML, which means that OL needs to be part of ML, which is not the case.

Broadly speaking, there are two approaches to defining omega. In the more 'traditional' approach, which is adopted by KS, FOL and K&B, little effort is spent to make the structure of the OL object and its ML name similar.

In the case of FOL [Weyhrauch 1980] and KS [Konolige 1980], which use sorted languages, there is in fact a separate sort for names of OL objects in the ML.

When OL and ML are very similar, as in the case of KS, or identical, as in the case of MOL,FOL and KB, however, it is possible to take another approach to the naming relation, one in which there is a conscious attempt to make the structure and linguistic appearance of the OL object and its ML name as similar as possible. This approach, which we call the transparent approach, is taken in sigma-PROLOG [Mc.Cabe 1984] and in the present thesis. In MOL, the name of a ground OL term is identical with the term itself, and the name of an OL sentence is structurally very similar to the sentence. Thus omega(x)=x if x is an OL ground term. Note that this axiom is not part of the theory META, because omega is not a function of the ML.

The advantages of the transparent approach are the following:
-'mixed' sentences, i.e. ML sentences which refer to OL sentences, are much more readable. The traditional device used for making mixed sentences readable is the use of quotation marks, but there are problems with their use. One such problem occurs when reference is being made to a partially specified OL sentence. In such cases, meta-level variables need to appear within the name of the OL sentence, but this is difficult to do unambiguously using quotation marks. In MOL, due to the transparent naming relation, there is generally no need for quotation marks.

-In the traditional approach, when there is more than one level of meta-language, the naming very quickly gets out of hand. This is because every layer of naming adds considerably to the complexity of the name. In the transparent approach, by contrast, the name of the name of the sentence a is the same as the name of the sentence a, because names of sentences are ground terms, and the names of ground terms are identical to the terms themselves. Thus extra layers of naming do not add to the complexity of

the name. This issue is of particular importance in self-referential systems, where the number of levels can be large.

4.3.4- The Standard Name Relation Eta

In any system where the meta-language and the object-language need to address the same domain, there must be a mapping from the ground terms of ML which refer to the objects of the domain to those ground terms of the ML which name the terms of the OL which name the same objects. In particular, any self-referential system needs to have such a function. To see why this is the case, consider the following sentence:

Tom is innocent until proven guilty

In KS, this sentence could be represented as

INNOCENT(TOM)<-NOT DEMO(CASE, 'GUILTY(TOM)')

where CASE is the name of the theory representing the case against Tom, and the quoted sentence 'GUILTY(TOM)' is the abbreviation of the term (say)

GUILTY'(TOM")

where TOM" is the name of the constant used in the theory CASE for representing Tom.

Now, unless we link the names TOM and TOM" through the standard name function, the above sentence does not have the desired meaning. Thus we need to have an axiom such as

eta(TOM)=TOM"

Notice that unlike omega, eta is a function of ML, for it maps terms of ML

into terms of ML, and needs to appear in ML sentences.

KS leaves eta as an arbitrary one to one function of the ML. But in MOL, we have chosen to have

∀ x eta(x)=x

i.e. eta is the identity function. Thus we can represent the above sentence as:

((INNOCENT TOM)

(NOT-DEMO CASE,(()(GUILTY TOM)))

The choice of the identity function for eta simplifies the syntax of mixed sentences.

As discussed in Chapter One and APPENDIX1, the choice of identity for eta will not lead to ambiguity of reference because the context always determines whether a constant is naming an object or an OL constant.

4.4 Conclusion

There is a great deal of unnecessary complexity in KS. This complexity is due to the formulation of knowledge using the TRUE axioms, the traditional omega naming relation for referring, and the arbitrary eta relation. MOL simplifies these complexities by unifying the reasoning agent's knowledge and belief, and by using the identity relation for omega and eta.

Chapter Five

Sets as Associative Data-Structures in Logic Programming

1-Introduction

In this chapter we will consider and contrast two commonly used data structures, lists and sets, in the context of logic programming. Alternatively, we could call these two classes of data structures recursive and associative. While lists are cleanly and effortlessly incorporated into Prolog, the treatment of associative data structures has been unsatisfactory. We will argue that this is not an accidental phenomenon; rather it has its roots in the theory upon which Prolog is based. Having located the theoretical 'bug', we will propose a method of getting around it that opens the way for a clean implementation of associative data structures.

2-Lists and Sets

2-1 Lists

Recursive data structures, which we shall call lists, are data-structures that, as well as recording their membership, record a partial order among their members. In their most common implementations they are accessed serially, i.e. to access the nth element it is necessary to go through the n-1th, n-2th elements etc.

In logic programming lists are commonly represented through a list constructor function. We shall call this function cons. For convenience, we shall often use the same syntax as that used in Chapter One for respresenting lists. Thus, (h|t) is equivalent to cons(h t).

Trees are naturally implemented as a list of lists. Examples of such usage include parse trees for expressions of a language.

2-2 Sets

Sets do not record order information; they only record their membership. Sets are directly accessed data structures. To access an element it is not necessary to do a linear search. They are natural data-structures to use for implementing associative constructs such as relational data-bases and logic programs. An important application of sets is to implement arrays. An Array can be implemented as a set of n tuples, where one field in the tuple acts as the index.

2-3 Lists and Sets in Prolog

Pure Prolog, by which I mean Horn-Clauses + Negation as Failure, only offers lists as data structures. Lists are incorporated into Prolog by the use of function symbols, which act as list constructors, and unification, which acts as the engine for list manipulation.

Actual Prolog, which includes non-logical features, offers associative data-structures through the Add-Clause and Del-Clause facilities. In this approach clauses are used as entries into an (associative) data-structure, which is manipulated by the above predicates.

The use of these non-logical features is an extremely unsavoury programming practice, because it amounts to changing the program at run-time, something which even BASIC and FORTRAN do not allow. Yet, for any large application which needs associatively accessed data-structures, e.g. data-bases, the use of these operations cannot be avoided.

In this paper we will argue that Prolog's failure to provide logically 'clean' associative data-structures is rooted in the theoretical foundations on which it is based. It is further argued that the solution to this problem also lies at the theoretical level.

Our approach to the incorporation of sets into logic programming proceeds as follows:

a- we build up a collection of axioms which define sets and their properties (as far as needed for our purposes).

b- Since these axioms are not in Horn Clause form, and standard techniques for converting them to clausal form lead to implementation problems, an alternative method of dealing with them in the context of logic programming is developed. This technique is based on a theorem, proved in appendix3, which replaces the set definition axioms with an infinite set of ground assertions.

c- It is argued that the problems encountered in incorporating sets into logic programming are bypassed by the theoretical approach taken.

3- The Definition of the Languages Used

In order to simplify the exposition, we need to specify in detail the first-order Horn-Clause languages with which we are concerned.

Def:

L, the 'core' language, is a Horn Clause language whose alphabet is

C, the set of constant symbols, which includes the symbol PHI

V, the set of variable symbols,

F, the set of function symbols

P, the set of predicate symbols, which includes SET, ADD and MEMBER

- F does not contain the symbol *, the set construction function
- P does not contain the symbol = (equality)

Def:

L(*,=) is L augmented with the set construction function * and the predicate = .

L(S) is L augmented with an infinite set of constant symbols. S is the set of these symbols. S and C share only one element, namely PHI.

4- The Axioms Defining Sets and Set Operations

In order to define sets and set operations, we need the following axioms:

-Set definition axioms

-Existence axioms

-Membership axioms

-General Equality axioms

-Set Equality axioms

Henceforth we shall refer to this collection of axioms as D (for define). The axioms in D belong to the language L(*,=)

Note- For the sake of brevity, the only set operation we consider in this chapter is addition of an element. Other set operations, such as deletion, union, intersection etc. can be treated in a way similar to addition. In a practical implementation, these operations and more would certainly be included among the primitives.

4-1 The Set Definition Axioms (SD)

These axioms specify what a set is:

SD1) Set(PHI); (PHI is the empty set)

SD2) SET(s)<- SET(s') & ADD(s' e s);

4-2 The Existence Axiom (Ex)

This axiom postulates the existence of the set derived from adding an element to another set. Without this axiom, or its equivalent, we can not use the predicate ADD in a proof involving the creation of new sets by adding an element.

Ex) ∀oldset ∀e ∃newset(ADD(oldset e newset)<-SET(oldset))

Since this axiom contains an existentially quantified variable, we can skolemise it to obtain

Ex*) ADD(oldset e e*oldset)<-SET(oldset)

where * is an infix function symbol not occurring in the language L.

4-3 The Membership Axiom (Fr)

This axiom specifies the membership of a set after the addition of an element

Fr) MEMBER(e t*s)<-> (e=t OR MEMBER(e s))

Note that this axiom contains a frame axiom as discussed in [Kowalski 1979]. Later on we will discuss this matter further in connection with the frame problem.

4-4 The Equality Axioms (Eq)

The equality axioms are in fact axiom schemas. They are the following:

a) x=x

b) f(t1)=f(t2) <- t1 = t2

where f is any function symbol of the language. Only the one argument case is illustrated here for simplicity.

c) R(t1) <- R(t2) & t1=t2

where R is a predicate symbol of the language. Again, only the one argument case is illustrated here.

Proposition 1:
From the axioms defined so far it can be shown that
For all e (MEMBER(e t1*(t2*t)) <-> MEMBER(e t2*(t1*t))) (proof in appendix3)

4-5 The Clausal Form and Skolemisation

So far, in the absence of set equality axioms, the set constructor symbol * defined above is not different from any other function symbol in the program. As a result, its use for set construction would lead to the representation of sets by lists. This is because, when more and more elements are added to the empty set, the resulting data-structure will be effectively a list of all the elements of the set.

Example:
Consider the following simple reservation scheme:
-we want to represent the passengers booked on a certain flight by a set
-we want to define a predicate, BOOK, which specifies the relationship between the two sets old-booking and new-booking after a reservation has been made for a passenger

```
BOOK(passenger old-booking new-booking)<-
        Add (old-booking, passenger, new-booking);
```

```
PROCESS((p|rest) o n)<-BOOK(p o n1)&PROCESS(rest n1 n);

PROCESS(() x x);
```

In the above program, PROCESS processes a list of names by successively adding them to the empty set. When the goal

```
<-PROCESS((TOM MARY) PHI x);
```

is run, x, which is the resulting set, will be bound to TOM*(MARY*PHI) which is effectively a list, with '*' as the list constructor.

Thus, if we do not do anything about it, the axioms we have defined so far lead us to the representation of sets by lists. Later on, we shall explain how the incorporation of set equality axioms will change the function * into a true set constructor function. However, let us first discuss the disadvantages of representing sets by lists.

5-The Disadvantages of Representing Sets by Lists

5-1 Lists Are Accessed Serially

In a typical access to a set, we might want to know whether an element e is a member of a set S; i.e. we want to evaluate the call MEMBER(e s).

When sets are represented by lists, the evaluation of this call will necessitate a serial search through the list representing S. What we need is an associatively accessed data-structure in which it is possible to evaluate this call directly.

5-2 Hard to Share Common Subsets

In order to save storage, we might want to share the storage allocated to a

common subset of two sets. This issue is of considerable importance in the context of the frame problem. Yet, as we show below, when lists are used to represent sets, it is very difficult to share common subset.

The most common and straightforward method for structure sharing between two lists is for them to share a common tail.

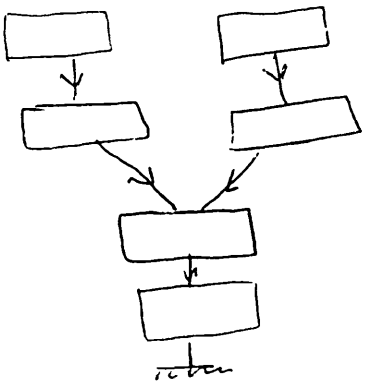


fig. 1

Any other structure sharing scheme for lists would create difficult problems of list manipulation. Yet, sharing common tails is not a good way of sharing common subsets, because quite often common subsets are not in the form of a common tail.

Example:

The two terms

1 *(2 * (3*PHI)) and

3 *(1 * (4 *PHI))

represent the two sets

{1 2 3} and

{1 3 4}

The two sets share the subset {1 3}, but the terms representing them, which in the absence of set equality axioms are effectively lists, do not have a common tail. Thus, it is not possible to use the structure sharing schemes of lists for sharing their common subset.

5-3 Costly Set Operations

When lists are used for representing sets, set operations are costly. One example of this is accessing elements, which was discussed before; another example is forming the intersection of two sets, which would cost on the order of n times m operations, where n an m are the sizes of the two sets represented by lists.

6- The One to One nature of Functions in PROLOG

The use of lists for representing sets leads to the deficiencies mentioned above. The most important cause of these deficiencies is that, as well as recording their membership, lists record a partial ordering among their elements.

Looking at this phenomenon from another angle, the peculiarities of lists arise from the one-to-one nature of the list constructor function. Consider the function cons (list constructor) applied to the two lists 1 and cons(2 3) The result is the list cons(1 cons(2 3)). But 1 and cons(2 3) are the only lists that, when the function cons is applied to them, result in the list cons(1 cons(2 3)) Thus,

cons:terms X terms -> terms,

cons is one to one

But set addition is not one to one; for example, the set {1 2 3} can be

derived from 3 added to {1 2} as well as 2 added to {1 3}.

Now, as we discussed earlier, in the absence of the set equality axioms, the function * (set constructor) is in effect a list constructor, and the same comments apply to it. Thus, for example, the two sets 1*(2*(3*PHI)) and 1*(3*(2*PHI)), although they are equal, will not be recognised as such by the system.

Why does the one to one nature of * lead to the above deficiencies?

Consider the access problem: since the elements of the set occur in the list in the order of their insertion, we need to do a serial search through the list to find an element. If we could get rid of this need to preserve order, we could use, for example, a sorted list to represent the set, cutting the access time to O(log(n)) using binary search.

There are two reasons why following the standard route to resolution theorem proving leads to a deficient set representation. These reasons are:

1- Skolemisation forces us to employ a function symbol, *, as the set constructor;

2- Unification treats all functions as one-to-one functions from termsXterms->terms

7- The Set Equality Axiom

To turn the function * into a true set constructor function, and overcome the problems mentioned earlier, we need to add a Set Equality Axiom to our theory:

SEQ:-

s1=s2 <- SET(s1) & SET(s2) & ∀x (MEMBER(x s1)<->MEMBER(x s2))

Thus, for example, the two sets 1*(2*(3*PHI)) and 2*(3*(1*PHI)) would be equal, and this would be reflected in any proof involving these two terms.

When the set equality axiom is added to our axioms for sets, * becomes a true set constructor function, and its properties will be very different from the list constructor function cons. For example, given the set equality axiom, we can use a data-structure for representing sets that ignores the order of addition of elements to the set, and represents equal sets by a unique construction (something that we could not do before). Such a data-structure, for example a sorted vector, can greatly enhance the performance of the system on set operations. For example, access to the elements of the set will take O(log(n)) instead of O(n).

But we have a problem: the equality axioms we have specified can not be used directly in a Prolog programming system. This is because their direct use would lead to explosive search and even loops. Thus we shall have to adopt them to Prolog in an indirect way.

Later we shall develop a technique for overcoming this problem. To develop this technique we shall need normal models. Thus, let us explain what a normal model is, and how it can be used for our purposes.

8- Normal Models and Herbrand Models

Traditionally, the resolution based theorem proving systems, of which Prolog is a descendant, have used Herbrand interpretations as the basis of their semantics. In a Herbrand Model, the domain of the interpretation is the set of ground terms constructible from the function and constant symbols of the language, and constants and functions are mapped to 'themselves'.

Thus, in a Herbrand interpretation of the language L(* =), the two terms A*(B*PHI) and B*(A*PHI) would have two separate interpretations, ie. A*(B*PHI) and B*(A*PHI)

By definition, a normal model is a model in which the interpretation of equality is identity. In other words, if from the theory it is provable that t1=t2, where t1 and t2 are two ground terms, then t1 and t2 must map to the same object in the domain of interpretation.

Now, a Herbrand model of the set axioms can not be a normal model, because from the axioms we can derive that T1*(T2*PHI)=T2*(T1*PHI) (say) but the Herbrand interpretation of these two terms is not the same.

It has been proved [Mendelson 1964] that whenever a theory has a model, it has a normal model. Let the theory Q have a model M. A normal model, M', of Q can be derived by taking as the domain of interpretation the set of equivalence classes generated over the domain of M by the interpretation of the equality relation in M. The interpretation of the functions and relations of the language of Q in M' is defined as a homomorphic image of the corresponding functions and relations in M.

We will not be concerned with the details of construction of normal models. All we are interested in is the fact that every consistent theory has a normal model. We will use normal models to prove the main theoretical result in the chapter.

9- The Alternative Treatment of the Set Axioms

As stated before, it is very difficult to use the set definition axioms, D, directly in a logic programming context, because the set equality axioms lead to explosive search and loops.

Now imagine that we could define an infinite set of ground assertions, T, which had the following property:

For all programs, such as prog, of the language L,

prog&D is unsatisfiable iff prog & T is unsatisfiable

Then we could use T instead of D in any refutation proof. To see this, consider the case when we want to show that the sentence G follows from the program P&D. In a refutation proof we show this by showing that

P& not G & D is unsatisfiable.

But according to the above property, this is equivalent to proving that

P & not G and T is unsatisfiable.

Thus we can use T instead of D in the proof.

The substitution of T for D in the program allows us to use orthodox logic programming proof techniques. This is because members of T are ground assertions, and although T is infinite, it is decidable and the membership of a given assertion in T can be computed efficiently. These issues will become clearer after we have defined T and discussed possible implementations.

9.1 Representing Sets by Numbers

Before we go on to describe the general definition of T, consider how sets might be represented by numbers:

-Assign to every ground term, e, a prime number, p(e). call it the term's prime code;

-represent a set through the product of its elements' prime codes

Example:

p: a->2

b->3

c->5

{a b c} represented as 2 X 3 X 5 = 30

But there is a problem here: a given set might be represented by more than one number, because such numbers as 30 and 60 (say) have the same prime factors.

In order to avoid this problem, we use the following trick:

- the empty set is represented by 1,

- sets are represented by 'monoprimal' numbers, i.e. numbers in the prime factorisation of which the power of every prime number is one;

- union of two sets is represented by @, which is defined as follow:

A@B = A X B / GCD(A,B)

where GCD(A,B) is the greatest common divisor of A and B.

Note that now the representation of sets is unique, i.e. equal sets are represented by identical names.

Through p and @ we define the infinite set of assertions, T. We say that T

is 'defined through' p and @.

Def:

-T is an infinite set of ground assertions

-Add(s e s') ∈T iff [p(e)@s]=s'

-MEMBER(e s) ∈T iff there is s' such that s'@p(e)=s

We will prove later on that T has the property mentioned earlier, i.e. that prog & T is satisfiable iff prog & D is satisfiable. (NOTE: prog includes the goal statement)

It is easy to implement the infinite assertions T in a logic programming environment using standard techniques. This would be done through evaluable predicates for ADD and MEMBER.

Notice that the relation between the two theories prog&D and prog&T is not one of equivalence; but a weaker one of being 'similarly satisfiable'. This is the same kind of relationship between a sentence and its Skolemisation.[Robinson 81]

By Herbrand's theorem, for every program there is a finite subset of T to which the above theorem applies, though this subset might be different for different programs.

9.2 The Relationship with Normal Models

We could establish a link between the assertions T and a normal model of prog & D:

- In a Least Herbrand Model M of prog & T, the relations MEMBER, ADD and SET would map to a subset of the Herbrand Base which is equal to T. Therefore, since the set names are unique in T, equal sets map into

the same object in the domain of interpretation. Now that is what happens in a normal model of prog &D, i.e. in a normal model of prog & D, too, equal sets map to the same object in the domain of interpretation. We will make use of this property in proving the 'similar satisfiability' of prog & D and prog & T.

Of course, in most circumstances it would be impractical to use prime codes and @ for representing sets. However, we can abstract those properties of prime numbers which are necessary for our purpose, and embody them in a general algebraic system. This would enable us to implement sets any way we want, as long as the implementation satisfies the requirements of the algebraic system.

10-The Definition of the Algebraic System

a- Define S, which is a countably infinite set of constant symbols such that the only common element of S and C is PHI.(C is the set of constant symbols of the language L)

Intuitively, S is the abstraction of the 'monoprimal' integers which we had used before.

b- Define a subset of S to be 'prime'. Again, this subset is the abstraction of the set of prime numbers.

P is an infinite subset of S,

PHI is an element of P

c- Define a strict total ordering >, and an operation "." on S. > is the abstraction of the "greater than" relation among integer, and "." is the abstraction of @ . Define a>=b as a>b or a=b

> is a total ordering on S such that PHI is the least element of S with respect to >,

"." is an operation on S which has the following properties:

a0 - PHI is the unit element : PHI.x=x

a1 - PHI does not have any factors: x.y=PHI -> x=PHI and y=PHI

a2 - "." is associative: x.(y.z)=(x.y).z

a3 - "." is commutative: x.y=y.x

a4 - "." is idempotent: x.x=x

a5 - "." is monotonic with respect to >= : x.y>=x

a6 - if p is prime and e=p.y and e=p.z and e<>y and e<>z then y=z
('<>' means not equal to)

a7 - prime elements have no factor other than themselves and PHI:
x $\in$ P & x=y.z -> ((x=y & z=PHI) or (y=PHI & z=x) or x=y=z)

a8 - every element of S has a unique ordered prime factorisation where an ordered prime factorisation of an element e is a sequence [p1, p2, p3 .... PHI] such that

- p1,p2 ... belong to P,

- p1>p2 , p2>p3 etc.,

- e = p1.p2....PHI

a9 - If Q=[p1,p2,..PHI] is the o.p.f of e, and p is a prime factor of

e, then p occurs in Q

Def: a prime factorisation of an element e is a permutation of the ordered prime factorisation of e

The statements a0 to a8 and all the other statements defining the algebraic system (S, . , ">") are not necessarily independent. However, they are consistent because they have a model in arithmetic, i.e. the mapping

S-> monoprimal positive integers,

P-> prime numbers,

"."-> the operation @ defined earlier,

">" -> the 'greater than' relation on integers

will satisfy all the statements made about (S, . , ">")

d- Having defined the algebraic system (S, ".", ">"), we define a mapping p from the ground terms of the language L(S) to the prime elements of S, i.e. to elements of P.

Def: p is a mapping from the Herbrand universe of L(S) to P which is one to one, and total. Moreover, p has the following property:

- If the constant s1 belonging to S occurs in the term t, then p(t)>s1

Intuitively, p maps every ground term of L(S) into a unique 'code', and a set is represented by the product of its elements' codes. Thus, if p maps

a,b and c in the following way:

p: a->p1

   b->p2

   c->p3

then the set {a,b,c} will be represented by p1.p2.p3

e- Now we can define the infinite set of assertions T which can 'replace' the set definition axioms D.

10.1 The Definition of T

Def: T is an infinite set of ground assertions belonging to the language L(S) which has the following properties:

1- SET(e) belongs to T iff e belongs to S

2- ADD(e1 t e) belongs to T iff p(t).e1=e

3- MEMBER(t e) belongs to T iff there is e1 such that e1.p(t)=e

10.2 The Main Theorem

We are now in a position to specify the main theorem.

Theorem: let prog be a program of L such that it does not have any clauses with SET, ADD and MEMBER at their head.

Then Prog & D is satisfiable iff prog & T is satisfiable. (Proof in the appendix)

To summarise,

-D is the set of axioms defining set operations, and the function symbol * is the set construction function

-D can not be directly used in a Prolog programming environment, because the direct execution of the equality axioms leads to explosive search and loops. To circumvent this problem,

-We define an algebraic system (S,".", ">")

-We define a set of ground assertions T in terms of the algebraic system,

-We use T instead of D in any refutation proof, relying on theorem 1 to ensure correctness.

11-Some Implementation Points

Thus far, we have described the theoretical basis of incorporating sets into Prolog. But how can we use this theoretical result in a practical implementation?

To implement sets using the above theoretical results, we need to devise the following mechanisms:

- We must devise a data structure to represent the constants in S, and implement the operation "." and the relation ">" on this data structure in a way that the implementation satisfies the properties of the algebra (S,".", ">")

- We need a mechanism for implementing the coding function, p. In other words, we need either some table in which the prime code for each ground term of the language L(S) is stored, or some function which can calculate the code directly. Again, the implementation of the prime code function must satisfy the definition of p above.

- We must modify the Prolog interpreter so that it can invoke the set mechanism on calls to set operations, i.e. to treat the set operations

as "system" calls.

We will now consider the above implementation issues.

11-1 The Data Structure Chosen for the Constants in S

The choice of the data structure for the elements of S and the implementation of the operation "." and the relation ">" depends on the type of application which the set mechanism is most likely to be used for. Below we will describe two of the more straightforward data structures.

I-Bit Maps

In this method, a set is represented by a binary word in which the bit in the nth position is 1 if the set includes the element with the prime code n. (We are assuming that codes are numbers).

For example, if the codes for a, b, and c are 3, 4 and 7, then the set {a b c} would be represented by the binary number 0011001. In general, assuming a 'compact' coding scheme (see section 11-2 below) the length of the word representing the set is equal to the number of the codes in use, i.e. the size of the universal set.

The operation "." is implemented by a binary OR and the relation ">" is the ordinary 'greater than' relation on the set representations considered as a binary number.

The advantage of this scheme is its simplicity, its disadvantage is that if the number of codes in use is large, large binary words must be used for representing sets (even small sets).

II-Sorted Vectors

A set is represented by a sorted vector of the codes of its elements. For example, the set {a,b,c} is represented by the vector (3 4 7). The

operation "." is implemented by the merge of two sorted vectors, deleting duplicated elements. The relation ">" is the 'alphabetical ordering' of the two vectors representing the sets.

The advantage of this scheme is that the length of the vector is proportionate to the size of the set it represents (as opposed to the bit map approach, where the size of the representation is proportionate to the size of the universal set). Its disadvantage is that set operations are more costly than the bit map scheme (though not much).

11-2 The Implementation of the Coding Function

In order to lower the storage requirements for representing sets, we need to have a coding function which keeps the size of the codes as small as possible.

A good way of achieving this is to use a hash table to represent the prime code function p. A hashing scheme such as the one described below might be used:

-Devise a hash function that takes the whole of the term as the key. This hash function should be such that the definition of p can not be violated.

-Use a straightforward clash resolution scheme.

-In the hash table itself store the value of the pointer to the term.

-The code for an element would be the address of the table location occupied by the (pointer to the) term. (see figure 5.1)

Example:

Assume the term t is a member of some set. The code of t is its address in the hash table, as illustrated below.

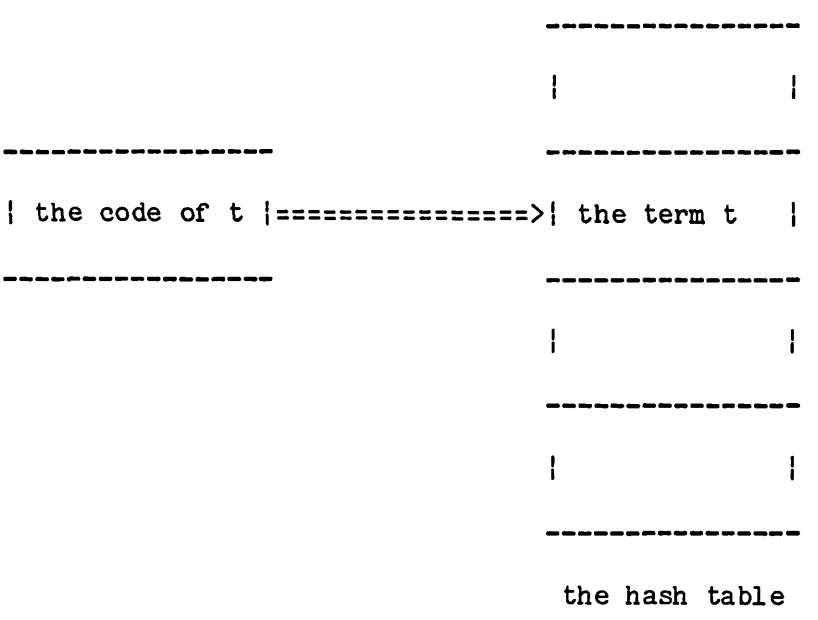


figure 5.1

Thus, to find the code for a term, we would first use the hash function to calculate the first probe for that term. Then we would look at the table location indicated by this probe. If it contains a pointer to our term, the value of the code is the probe itself. If the location does not contain a pointer to our term, we use the clash resolution scheme to compute the second probe, and so on.

11-3 The Alterations to the Prolog Interpreter

In essence, the interpreter would treat the set manipulation predicates the same way as any other system predicate, i.e. it would invoke the specialised mechanism for sets to evaluate such calls. The only extra feature that is needed is that, in the interests of efficiency, the interpreter must have a suspension mechanism for some half-instantiated set

calls. This is illustrated below by an example.

Assume that the interpreter has at one point to process the call ADD(x a y), where x and y are variables that occur in the argument places used for set names. To make such a call to the set mechanism is ludicrous, because there are an infinite number of instantiations for x and y, and the likelihood that any of them would be the correct one is remote. This situation is similar to the call SUM(x 10 y) to the arithmetic module.

Of course the option exists to treat such calls as error, and to stop the execution. But a more elegant solution is to incorporate a suspension mechanism into the interpreter, to suspend such calls until the arguments are sufficiently bound. (Same is true of MEMBER, and other set manipulation predicates)

12-The Frame Problem

The frame problem is the problem of proliferation of large data structures in declarative programming systems [Kowalski 1979]. The scheme devised here suffers from the frame problem to the extent that when dealing with large sets, the names of these sets are also large, and their proliferation costly. We feel that due to the nature of data structures chosen for representing sets (direct access structures) subset sharing, which can be used to ease the frame problem, is more straightforward in our system than in a system based on a list representation of sets. This is a matter of current investigation.

13-Conclusion:

The standard route to a resolution based treatment of set axioms, i.e. Skolemisation and the use of unification as the engine of set manipulation, will lead to the representation of sets by lists, which is inefficient in many ways. These deficiencies can be overcome by an alternative treatment of set axioms, including the set equality axioms, which enables us to use direct access data structures for representing sets.

Bowen,K. & Weinberg 1985:"A Meta-Level Extension of Prolog", Proceedings of 1984 Symposium on Logic Programming pp.126-130

Brazdil, P.B. 1986:"Transfer of Knowledge Between Systems: Some Considerations Concerning System Abilities", University of Porto, Faculdade De Economia.

Bruynooghe,M.&Pereira,L.M. 1981:"Revision of Top-Down Logical Reasoning Through Intelligent Backtracking", Centro De Informatica, University of Lisbon Report CIUNL - 8/81

Bundy,A. & Welham,B ,1980:"Using Meta-level Inference for Selective Application of Multiple Rewrite Rule Sets in Algebraic Manipulation, AI Journal 16,1980,pp 189,211

Clark,C,1978:"Negation as Failure", pp. 293-325 "Logic and Data-Bases", Plenum Press

Davis, Randal 1980: "Meta-Rules: Reasoning About Control", AI Journal 15,1980, pp 179-220

DeKleer, J, 1976: "Local Methods for Localizing Faults in Electronic Circuits", MIT Memo 394, Cambridge, MA

Gallaire,H & Lassere,C, 1981:"Meta-level Control for Logic Programs" Logic Programming, eds. Clark and Tarnlund, Academic Press, 1982

Konolige,K, 1981: "A First-Order Formalisation of Knowledge and Action for a Multi-agent Planning System", Machine Intelligence 10

Kowalski,R. 1974:"Predicate Logic as a Programming Language",pp. 572-595 Proceedings of IFIP 1974,North Holland Publishing Company,Amsterdam

Kowalski,R. 1979:"Logic for problem solving",North Holland Elsevier, New York

Kowalski,R.&Bowen,K. 1982:"Amalgamating language and meta-language in logic programming", pp.153-172 Logic Programming, eds. Clark and Tarnlund, Academic Press, 1982

Lloyd,J.W. 1984 "Foundations of Logic Programming",Symbolic Computation Series, Springer Verlag, 1984

Mendelson,E, 1964:"Introduction to Mathematical Logic" page 80, Van-Nostrad Reinhold Company

McCabe,F. 1980:"micro-PROLOG Manual", Logic Programming Associates,1980

McCabe,F. 1984:"sigma-PROLOG Manual", Logic Programming Associates, 1984

Nakashima, H: 1984:"Knowledge Representation in Prolog/KR", Proc. 1984 Symposium on Logic Programming, pp 126-130

Perlis,D:1985:"Languages with Self-Reference I: Foundations",AI Journal 25 pp 301,322

Pereira, L. M. 1982: "Logic Control With Logic", University of Lisbon Report UNL - 2/82

Reiter,R 1985:"a Theory of Diagnosis from First Principles", Department of

Robinson, A 1979:"Logic:Form and Function", North Holland Elsevier, New York

Silver, B 1986:"Meta-Level Inference: Representing and Learning Control Information in Artificial Intelligence", North Holland Elsevier, Netherlands

Takewaki, T. , Takeuchi, A., Kunifuji, S. & FuruKawa, K 1985:"Application of Partial Evaluation to the Algebraic Manipulation System and its Evaluation", ICOT Research Report

Takeuchi, A & Furukawa, K, 1985: Partial Evaluation of Prolog Programs And its Application to Meta-Programming", ICOT Research Report

Venken, R 1984: "A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source to Source Transformation and Query Optimisation", ECAI 1984, pp 91-100

Weyhrauch, R.W. 1980 "Prolegomena to a Theory of Mechanised Formal Reasoning" AI Journal 13,1980, pp 133,170

APPENDIX1:

The Naming Relation

1. Definition of the naming relation for sentences of the language

Def: let v1...vn be the variables of the sentence A in order of appearance in the sentence. Then the list ((s1 v1) (s2 v2)...(sn vn)) is a v-substitution for the sentence A iff s1..sn are distinct constant symbols that do not occur in A.

Def: let VS=((s1 v1) (s2 v2)...(sn vn)) be a v-substitution for the sentence A. Then:

-The term ((s1...sn)|A') is a name of the sentence A iff A' is a sentence-substitute for A with respect to VS.

-The term A' is the name of the atom A in the context of VS iff A' is an atom substitute for A with respect to VS

-The term A' is the name of the term A with respect to VS iff A' is a term substitute for A with respect to VS

-The constant A' is the name of the predicate A if its identifier is an underlined version of the identifier of A

- The list (atom1' atom1' ... atomn') is a sentence-substitute for the sentence (atom1 atom1... atomn) with respect to VS iff atom1',atom2'... are atom-substitutes for atom1,atom2... with respect of VS.

- The term (p' t1' t2' ..tn') is an atom-substitute for the atom (p t1

t2 .. tn) with respect to VS iff

p' is a constant whose identifier is an underlined version of the identifier of the predicate symbol p,

t1', t2' .. are term-substitutes for the terms t1,t2... with respect to VS.

- The term (HEAD'|TAIL') is a term-substitute for the term (HEAD|TAIL) with respect to VS iff HEAD' is a term substitute for HEAD and TAIL' is a term-substitute for TAIL with respect to VS.

- () is the term-substitute for ()

- the term-substitute for a constant is itself.

- the term-substitute for the variable-symbol v with respect to the v-substitution (...(s v) ..) is s.

Proposition 1: if the names of two sentences are the same, then the sentences are identical up to re-naming of variables.

Let N=(V|D) be the name of the sentence A with respect to
the v-substitution

V1=((s1 u1) (s2 u2)..(sn un))

And also let (V|D) be the name of the sentence B with respect to the v-substitution

V2=((s1 v1)(s2 v2)..(sn vn)),

Define the substitution CHI as

CHI=(v1/u1 v2/u2 .. vn/un)

Then we want to prove that CHI(B)=A. But let us first prove a subsidiary proposition.

Proposition 1.1: Let TA be a term occurring in A whose term-substitute with respect to V1 is TD. Let TB be a term occurring in B whose term-substitute with respect to V2 is also TD. Then CHI(TB)=TA

Proof of prop 1.1: by induction on structure of TB

Base Case: TB is a variable, constant or empty list

a-TB is a variable

Let TB be vk. Then CHI(TB)=uk. Thus we have to prove that TA is also uk.

The term-substitute for TB is TD, and TB=vk, therefore TD=sk.

Now, TD is also the term-substitute for TA with respect to V1. In other words, the term-substitute for TA with respect to the v-substitution

V1=((s1 u1)(s2 u2)...(sn un))

is sk.

Since the term-substitute for TA is sk, i.e. a constant, from the definition of term-substitute it can be concluded that either:

i- TA is the constant sk,

ii-TA is the variable uk

but possibility i is excluded, because the constants used in a v-substitution should not occur in the sentence. Therefore TA is the variable uk (QED)

b- TB is a constant.
Since TB is a constant, CHI(TB)=TB. Thus we have to prove that TA=TB.
Since TB is a constant, TD=TB (by definition of term-substitute).
Furthermore, we know that

TD<>s1, TD<>s2 .. TD<>sn (by definition of v-substitution) (1)

Now, the term-substitute for TA is TD. TA must be a constant, for if it were a variable, say uk, then TD=sk, which contradicts (1) above. Thus, by definition of term-substitute, TD=TA, i.e. TD=TA=TB (QED)

c-TB is the empty list
By definition of term-substitute, TD is also an empty list. Again, by definition of term-substitute, TA is an empty list. Thus TA=TB (QED)

Inductive Case:
TB=(B-head|B-body). We need to prove that CHI(B-head|B-body)=TA.

Since TD is the term-substitute for TB, TD=(t-head|t-body) where t-head is the term-substitute for B-head and t-body is the term-substitute for B-body. Thus, TA=(A-head|A-body), where t-head is the term-substitute for A-head, and t-body is the term-substitute for A-body.

Now, since t-head is the term-substitute for B-head and A-head, by inductive assumption CHI(B-head)=A-head.

Similarly, CHI(B-body)=A-body. Thus,

(CHI(B-head)¦CHI(B-body))=(A-head¦A-body)=TA. But, by nature of substitution CHI, CHI(B-head)¦CHI(B-body))=CHI(B-head¦B-body)=CHI(TB), thus CHI(TB)=TA (QED)

Now we prove the main proposition

We want to prove that CHI(B)=A

Let D be (D1 D2 ... Dn) . Then by definition of sentence-substitute,

A=(A1 A2 ... An) and
B=(B1 B2 ... Bn)

where D1...Dn are atom-substitutes for A1 ... An with respect to V1,
and D1...Dn are atom-substitutes for B1 ... Bn with respect to V2.

Let B'=CHI(B)=(B1' B2' ... Bn'). then A=B' if A1=B1', A2=B2' etc. Thus it is sufficient to prove that for all k, Ak=Bk'.

Let Dk =(pd td1 td2 .. tdn). then
Ak =(p ta1 ta2 ...tan) and
Bk =(p tb1 tb2 ...tbn)
Bk'=(p tb1' tb2'...tbn')

where pd is a constant whose identifier is an underlined version of the identifier of p,
td1...tdn are term-substitutes for ta1...tan with respect to V1,
td1...tdn are term-substitutes for tb1...tbn with respect to V2.

Now, by proposition 1.1, tb1'=ta1, tb2'=ta2...tbn'=tan. Thus Bk'=Ak (END of Proof of Proposition 1)

2- Types of Terms

In the context of a literal, a term can be of one of the following types:

A- The name of a program,

B- The name of a sentence

C- The name of an atom in the context of a varlist

D- The name of a term in the context of a varlist

E- The name of a predicate

F- Ordinary (none of the above)

Below we show how to determine the type of a term unambiguously.

- if the term does not occur as an argument of a meta-level predicate, or as a subterm of an argument of a meta-level predicate, then it is of type F;

- If a term occurs as an argument of a meta-level predicate, its type can be determined in the following way:

  With every argument place within a meta-level predicate is associated an implicit type for the term that occupies that place. This association naturally follows from the fact that these meta-level predicates are assumed to correctly represent certain relations on the domain of OL objects (see section 1.5.1).

- If term T is a subterm of the term T', we can inductively assume that we know the type of T'. Now:

- If T' is of type F, then T is also of type F,

- If T' is of type D, i.e. the name of a term in the context of a varlist V,

then T is also the name of a term in the context of V;

-If T' is of type C, i.e. the name of an atom in the context of a varlist V, then let T' be (P arg1 arg2 .. argn)

If T is P, it is the name of a predicate;

If T is one of arg1...argn, it is the name of a term in the context of V

- If T' is of type B, i.e. the name of a sentence, then let T' be (T1' T2'... Tn')

If T is T1', it is a varlist, i.e. of type F;

If T is one of T2',..Tn', then it is the name of an atom in the context of T1', i.e. of type C

_ T' cannot be of type E or A because these do not have any subterms.

Appendix2

The Definition of NOT-DEMO

```
((NOT-DEMO program goal-list)
    (CHOOSE-GOAL goal-list goal)
    (POSSIBLE-CLAUSES program goal clause-list)
    (FAIL-ALL program goal clause-list))

((FAIL-ALL program goal ()))

((FAIL-ALL program goal (clause|rest-clauses))
   (FAILS program goal clause)
   (FAIL-ALL program goal rest-clauses))

((FAILS program
        (goal-vlist goal-predicate|goal-argument)
        (clause-vlist (head-predicate|head-argument)|body))
   (NOT-UNIFY
       (goal-vlist|goal-argument)
       (clause-vlist|head-argument)))

((FAILS program goal clause)
   (RESOLVE goal clause new-goal-list)
   (NOT-DEMO program new-goal-list))

((NOT-UNIFY (vlist1 head1|body1) (vlist2 head2|body2))
   (NOT-UNIFY (vlist1|head1) (vlist2|head2)))

((NOT-UNIFY (vlist1 head1|body1) (vlist2 head2|body2))
   (NOT-UNIFY (vlist1|body1) (vlist2|body2)))
```

```
((NOT-UNIFY (vlist1 head1|body1) (vlist2 head2|body2))
   (UNIFY (vlist1|head1) (vlist2|head2) substitution)
   (APPLY substitution body1 new-body1)
   (APPLY substitution body2 new-body2)
   (NOT-UNIFY (vlist1|body1) (vlist2|body2))

((NOT-UNIFY (vlist1|c1) (vlist2|c2))
   (NOT-ON c1 vlist1)
   (NOT-ON c2 vlist2)
   (DIFFERENT c1 c2))

((NOT-ON x ()))

((NOT-ON x (y|z))
   (DIFFERENT x y)
   (NOT-ON x z))

((CHOOSE-GOAL (vlist|goal-list) (vlist|goal))
   (ON goal goal-list))

((POSSIBLE-CLAUSES program (vlist|(pred|args)) clause-list)
     (CARDINAL program pred n)
     (PS1 program pred () clause-list n))

((PS1 program pred clauses clauses 0))
((PS1 program pred clauses-sofar clause-list n)
     (PREDICATE-CLAUSE program pred clause)
     (NOT-ON clause clauses-sofar)
     (SUM m 1 n)
     (PS1 program pred (clause|clause-sofar) clause-list m))
```

```
((PREDICATE-CLAUSE program pred (vlist (pred|arguments)|body)))
```

APPENDIX 3

The Definition of the Algebraic System and Proof of Main Theorem in Chapter Five

1- the axioms of D

SD1)..... Set(PHI); (PHI is the empty set)

SD2)..... SET(s)<- SET(s') & ADD(s' e s);

Ex)..... ∀oldset ∀e ∃newset(ADD(oldset e newset)<-SET(oldset))

Ex*)..... ADD(oldset e e*oldset)<-SET(oldset)

Fr)..... MEMBER(e t*s)<-> (e=t OR MEMBER(e s))

Eq1)..... t=t

Eq2)..... f(t1)=f(t2) <- t1 = t2

where f is any function symbol of the language. (If the arity of f is more than one, then the left hand side of Eq2 is
f(..t1..)=f(..t2..)
where t1 and t2 occur at the same argument position, and all other arguments of f are identical in f(..t1..) and f(..t2..)

Eq3)..... R(t1) <- R(t2) & t1=t2
where R is any predicate of the languages. (If the arity of R is more than one, then t1 and t2 occur in the same argument position in R(..t1..) and R(..t2..), and all other arguments of R(..t1..) and R(..t2..) are identical.

SEQ)..... s1=s2 <- SET(s1) & SET(s2) & ∀x (MEMBER(x s1)<->MEMBER(x s2))

Proposition 0: SET(e*t)<-SET(t) is a theorem of D.

Proof: By resolution of SD2 and EX* we can derive SET(e*t)<-SET(t)

Proposition 1:

From D it can be deduced that

∀e MEMBER(e t1*(t2*t)) <-> MEMBER(e t2*(t1*t))

Proof:

Repeated application of the axiom Fr above to the left-hand-side results in

MEMBER(e t1*t2*t)<-> e=t1 OR e=t2 OR MEMBER(e t) (S1.1)

Repeated application of Fr to the right hand side also results in

MEMBER(e t2*t1*t)<-> e=t1 OR e=t2 OR MEMBER(e t) (S1.2)

thus, since the the right hand sides of S1.1 and S1.2 are the same, we can deduce that

MEMBER(e t1*t2*t)<->MEMBER(e t2*t1*t)

Proposition 2: ∀t1,t2,t (t1*(t2*t) = t2*(t1*t) <- SET(t)) is a theorem of D

Proof:

From the SEQ axiom it can be derived that

t1*t2*t=t2*t1*t <-

SET(t1*t2*t)&SET(t2*t1*t)&

∀x (MEMBER(x t2*t1*t<->MEMBER(x t1*t2*t) (S2.1)

Now, by proposition 1, ∀x(MEMBER x t2*t1*t <-> MEMBER(x t1*t2*t) is a theorem of D. Thus,

t1*t2*t=t2*t1*t <- SET(t1*t2*t) & SET(t2*t1*t) (S2.1)

By repeated resolution with proposition 0, we derive

t1*t2*t=t2*t1*t <- SET(t)

Proposition 2.1: t=t1*t<-SET(t) & MEMBER(t1 t)

Proof:

by SEQ, t=t1*t <- SET(t) & SET(t1*t)& ∀x(MEMBER (x t)<->MEMBER(x t1*t))

Applying Prop. 0 to the right hand side,

t=t1*t <-SET(t) & ∀x(MEMBER (x t) <->MEMBER(x t1*t))

Applying Fr to rhs,

t=t1*t <-SET(t) & ∀x(MEMBER (x t) <-> x=t1 OR MEMBER(x t))

which is equivalent to

t=t1*t <- SET(t) & ∀x(x=t1 ->MEMBER(x t)) (***)

Now, the equality axiom schema includes

∀x MEMBER(x t)<- x=t1 & MEMBER(t1 t)

This can be re-written as

∀x(x=t1->MEMBER(x t))<-MEMBER(t1 t) (+++)

thus, from (***) and (+++),

t=t1*t <- SET(t) & MEMBER(t1 t)

Definition of the Languages Used:

L, the 'core' language, is a Horn Clause language whose alphabet is

C, the set of constant symbols, which includes the symbol PHI

V, the set of variable symbols,

F, the set of function symbols

P, the set of predicate symbols, which includes SET, ADD and MEMBER

- F does not contain the symbol *, the set construction function
- P does not contain the symbol = (equality)

Def:

L(*,=) is L augmented with the set construction function * and the predicate = .

L(S) is L augmented with an infinite set of constant symbols. S is the set of these symbols. S and C share only one element, namely PHI.

Convention:

We will use the notation :Q to denote the interpretation of the constant, function or predicate symbol Q in the model M. Thus, :C will be the interpretation of the constant C, :f will be the interpretation of the function symbol f and :P will be the interpretation of the predicate symbol P. We will call :T an 'instance' of a term t in M if there is a ground term T which is an instance of t and whose interpretation in M is :T. Thus, an instance of the variable x in the model M can be any object :T in the domain of M

Notation: we use angled brackets < ... > to indicate a tuple. For example, <a,b> is the tuple whose first element is a and whose second element is b. Every relation in an interpretation is a set of tuples; thus, for example, the relation :R satisfies the sentence

∀x R(x) iff for all T, the tuple <:T> belongs to the relation :R

Convention: in the rest of the proof, t1*t2*t3 is an abreviation of t1*(t2*t3)) etc.

Proposition 3:

Let M be a normal model of D. Then for all ground terms T1, T2, and T of the language,

if <:T> belongs to :SET, then
:T1 :* :T2 :* :T = :T2 :* :T1 :* :T
(ie if SET(T) is a theorem, then the interpratation of T1*T2*T is the same as the interpretation of T2*T1*T)

Proof: proposition 2 stipulates that t1*t2*t=t2*t1*t <- SET(t). Now, since M satisfies D, it must satisfy proposition 2. Therefore, if the tuple <:T> belongs to the relation :SET in M, then for all T1 T2 the tuple <:T1:*:T2:*:T , :T2:*:T1:*T> must belong to the relation := in M. Thus, since the equality relation in M (i.e. :=) is the identity relation,

:T1:*:T2:*:T = :T1:*:T2:*:T

-The Definition of the Algebraic System

a- Let S be an infinite set of constant symbols such that the only common element of S and C is PHI.(C is the set of constant symbols of the language L)

b- Let a subset of S be 'prime'.

P is an infinite subset of S,

PHI is an element of P

We call an element of S prime if it belongs to P.

c- Let > be a strict well-founded total ordering, and "." an operation on S such that PHI is the least element of S with respect to >, and "." has the following properties:(Note: if a=b.c we call b and c factors of a)

a0 - PHI is the unit element : PHI.x=x

a1 - PHI does not have any factors: x.y=PHI -> x=PHI and y=PHI

a2 - "." is associative: x.(y.z)=(x.y).z

a3 - "." is commutative: x.y=y.x

a4 - "." is idempotent: x.x=x

a5 - for all x and y, x.y>=x

a6 - if p is prime and e=p.y and e=p.z and e<>y and e<>z then y=z

a7 - prime elements have no factor other than themselves and PHI:
x ∈ P & x=y.z -> ((x=y & z=PHI) or (y=PHI & x=z) or x=y=z)

a8 - every element of S has a unique ordered prime factorisation where an ordered prime factorisation (opf) of an element e is the sequence [p1, p2, p3 .... PHI] such that

- p1,p2 ... belong to P,

- p1>p2 , p2>p3 etc.,

- e = p1.p2....PHI

a9 - If Q=[p1,p2,..PHI] is the o.p.f of e, and p is a prime factor of e, then p occurs in Q

Def: a prime factorisation of an element e is a permutation of the ordered prime factorisation of e

Having defined the algebraic system (S, ".", ">"), we define a mapping p from the ground terms of the language L(S) to the prime elements of S, i.e. to elements of P.

Def: p is a mapping from the Herbrand universe of L(S) to P which is one to one, and total. Moreover, p has the following property:

- If the constant s1 belonging to S occurs in the term t, then p(t)>s1

Proposition 3.1

If e1=e2, then p is a prime factor of e1 iff it is a prime factor of e2.

Proof: let Q be the o.p.f of e1. By definition of o.p.f, it is unique. Since e1=e2, it is also a unique opf of e2.

Let p be a prime factor of e1. Then, by a9, it occurs in Q. Since Q is the o.p.f of e2, e2=q1.q2...p...PHI. since "." is commutative, e2=p.q1.12...PhI. Therefore by definition of factor, p is a factor of e2. Since it is prime, it is a prime factor of e2.

The same argument can be applied to show that if p is a prime factor of e2, it is also a prime factor of e1.

Proposition 3.2

Let e=p1.p2..pn. Let q1,q2,...qn be a permutation of p1,p2..pn. Let e'=q1.q2...qn . Then e'=e

Proof:by induction on n

Base case: n=1. Trivial

Inductive case:

case 1:p1=q1

by inductive assumption p2.p3...pn=q2.p3...qn. Thus p1.p2....pn=q1.q2...qn

case 2: p1<>q1

Let p1=qj. Then e'= q1.q2....qj-1.p1.qj+1...qn. By associativity of ".", e'=(q1.q2....qj-1).p1.qj+1...qn

i.e. by commutativity of ".", e'=p1.q1.q2... qj-1.qj+1....qn

Thus by case 1, e'=e.

Proposition 3.3

Let e=p1.p2.p3...PHI where p1,p2,p3... are all prime, p1,p2,...pn are all distinct, and the o.p.f. of e is Q. Then p occurs in Q only if p=p1 or p=p2 or...p=PHI

Proof: Let q1,q2...PHI be a permutation of p1.p2...PHI such that q1>q2, q2>q3...

(The existence of such a permutation is easily proved. We omit the proof).

Now, by proposition 3.1, e=q1.q2...qn. Thus, by definition of o.p.f, Q=[q1,q2,....PHI]. Now assume that p occurs in Q. Then for some j, p=qj. But since q1,q2... is a permutation of p1,p2... qj is equal to one of p1,p2...PHI.

Proposition 3.5

If x=p.y and x<>y then x>y

Proof:

Applying a5, since x=p.y, then x>=y, i.e. x>y or x=y. Since, by assumption, x<>y, then it follows that x>y

Proposition 3.8

If a prime factorisation of e is Q=[p1,p2,p3..PHI] and e1=p2.p3..PHI then e>e1.

Proof:

Let 0 be the o.p.f of e and 01 the o.p.f. of e1.

By definition, Q is a permutation of 0. Since there are no duplicated elements in 0, p1 occurs only once in Q, i.e. p1<>p2, p1<>p3 etc. Therefore, since e1=p2.p3...PHI, by proposition 3.3, p1 does not occur in the o.p.f. of e1, i.e. 01. But p1 does occur in 0 (because it occurs in Q and Q is a permutation of 0). Therefore 0<>01, thus e1<>e.

Thus, since e=p1.e1 and e<>e1, by proposition 3.5 e>e1.

Proposition 4:

let e=p1.e1 and

e=p2.e2

where p1 and p2 are prime and p1<>p2. Then there is e3 such that

e=p1.p2.e3

e1=p2.e3

e2=p1.e3

Proof: let Q=[q1,q2...PHI] be the o.p.f. of e. Then, by a9, p1 and p2 occur in Q, i.e. for some j and k, p1=qj and p2=qk.

Thus, by definition of o.p.f., e=q1.q2..qj-1.p1.qj+1....qk-1.p2.qk+1...

i.e., by proposition 3.2, e=p1.p2.q1.q2....qj-1.qj+1....qk-1.qk+1....

Let e4=q1.q2..qk-1.qk+1....qk-1.qk+1... Thus e=p1.p2.e4

Now we consider the following cases, and define e3 for each case.

1: e=e1=e2. Then e3=e trivially satisifies the above conditions.

2: e=e1 and e<>e2. Then let e3=p1.e4

We have to prove that the above conditions are satisfied.

2.1. we must prove e=p1.p2.e3

p1.p2.e3=p1.p2.p1.e4=p2.p1.p1.e4=p2.p1.e4=p1.p2.e4=e

2.2 we must prove e1=p2.e3

p2.e3=p2.p1.e4=p1.p2.e4=e=e1

2.3 We must prove e2=p1.e3

p1.e3=p1.p1.e4=p1.e4. Thus it is sufficient to prove e2=p1.e4

Now, e=p2.p1.e4 and e=p2.e2. We can easily prove e<>p1.e4, and we know that e<>e2. Thus, by a6, e2=p1.e4

3:e<>e1 and e=e2. Then let e3=p2.e4

We have to prove that the conditions are satisfied.

3.1 we must prove e=p1.p2.e3

p1.p2.e3=p1.p2.p2.e4=p1.p2.e4=e

3.2 we must prove e1=p2.e3

p2.e3=p2.p2.e4=p2.e4. Thus it is sufficient to prove e1=p2.e4

Now, e=p1.p2.e4 and e=p1.e1. We know that e<>e1 and we can easily prove that e<>p2.e4. Thus by a6, e1=p2.e4

3.3 we must prove e2=p1.e3

p1.e3=p1.p2.e4=e=e2

4:e<>e1 and e<>e2. Let e3=e4

We have to prove that the conditons are satisfied.

4.1 we must prove e=p1.p2.p3

p1.p2.e3=p1.p2.e4=e

4.2 we must prove e1=p2.e3

since e=p1.e1 and e=p1.p2.e3 and e<>e1 and we can easily prove e<>p2.e3, by a6 e1=p2.e3

4.3 we must prove e2=p1.e3

since e=p2.e2 and e=p2.p1.e3 and e<>e2 and we can easily prove e<>p1.e3, by a6 e2=p1.e3

QED

Proposition 5

If e1 and e2 are such that

for all prime elements p,

p is a factor of e1 iff p is a factor of e2

then e1=e2

Proof:

Let Q1 be the o.p.f. of e1 and Q2 the o.p.f of e2. Since all prime factors of e1 occur in Q1, and all prime factors of e2 occur in Q2, we have Q1=Q2 i.e. e1=e2.

Definition of T

T is an infinite set of ground assertions belonging to the language L(S) which has the following properties:

- SET(e) belongs to T iff e belongs to S

- ADD(e1 t e) belongs to T iff p(t).e1=e

- MEMBER(t e) belongs to T iff there is e1 such that e1.p(t)=e

- No other assertion occurs in T

Theorem1:

Let prog be a program of L such that it does not have any clauses with SET, ADD or MEMBER at their head.

Then prog &D is satisfiable iff prog & T is satisfiable

The proof is in two parts. First we prove the 'only if' part, and then we prove the 'if' part

I- the 'only if' part: if prog & D is satisfiable then prog & T is satisfiable

let prog & D be satisfiable. Then it has a normal model M

The language of prog & D is L(*,=) i.e. the language L augmented with the set construction function * and the equality relation =.

The language of prog & T is L(S) i.e. the languge L augmented with the infinite number of constants in S.

Now since M is an interpretation of the language L(*,=) it does not contain any interpretation for the constants in S.

We define another interpretation, Mo, and prove that it satisfies prog & T.

Mo is defined as follows:

the domain of Mo is the same as the domain of M,

The relations and functions in Mo are the same as those in M,

The mapping of all the constant, function and predicate symbols of L to the objects, functions and relations of Mo are the same as the corresponding mappings in M,

Since Mo is an interpretation for the language L(S), and the constants in S do not occur in the languge L (except for PHI), we define the following mapping for the constants in S.

Def: let e be an element of S,

If e=PHI, then :e=:PHI

If e<>PHI, then let [t,t1,t2..PHI] be the o.p.f of e

let e1=t1.t2...PHI

define :e = :q(t) :* :e1

Note:

:e is the interpretation of e in Mo,

q is the inverse of p, the mapping from the ground terms of L(S) to the elements of P,

:q(t) is the interpretation of q(t),

:* is the interpretation of * (the set construction function) in M,

:e1 is the interpretation of e1 in Mo

Proposition: the above interpretation of the constants in S is well defined

Proof:by induction on >

Base Case:

The interpretation of PHI is well defined because PHI belongs to the language L

Inductive Case:
Assume that for all e1 such that e>e1 the definition is well defined. Then we prove that it is well defined for e.

Proof of inductive case:
by proposition 3.8, e>e1. Thus, by inductive assumption, :e1 is well defined.

:q(t) is well defined because:
- if q(t) does not contain any constants from S, then :q(t) is well defined by definition,
- if it does contain a constant from S, such as r, then by definition of p, r<t. Thus by inductive assumption, the interpretation of r is well defined. Since the interpretation of any other functions or constants that might occur in q(t) is well defined by definition, :q(t) is well defined.

:* is well defined by definition,

therefore :q(t):*:e1 is also well defined

proposition 5.5
If e belongs to S, then the tuple <:e> belongs to the relation :SET in Mo
Proof:by induction on >

Base case: <:PHI> is a member of :SET because :SET must satisfy the axiom SET(PHI) (the axiom SD1)

Inductive case:let e=p.e1, where p is the largest prime factor of e, and e<>e1. Then
:e=:q(p):*:e1 (by definition)

Now, by proposition 3.5, e>e1. thus,

<:e1> belongs to :SET (by inductive assumption)

Now the relation :SET belongs to the model M, and M satisfies the proposition SD2* which is

SET(x*y)<-SET(y)

Thus, since <:e1> belongs to :SET, :q(p):*:e1 must also belong to :SET. Thus <:e> belongs to :SET (QED)

Proposition 6

From the definition of :e for members of S, it follows that:

a- If e<>PHI and e=p.e1 and, where p is any prime factor of e, then

:e = :q(p) :* :e1

(note that in the original definition, p was the largest prime factor of e)

b- If e=PHI, then :e=:PHI

Proof: by induction on >

Base Case: :e=:PHI by definition

Inductive case: Assume for all elements of S smaller than e, the above holds. Prove that it holds for e

Let p' be the largest prime factor of e.

If p=p' then the proposition holds by definition.

We now consider the following two cases:

case 1: e=e1

We have to prove that :e=:q(p) :* :e

Now, proposition 2.1 states that t=t1*t <- SET(:e) & MEMBER(t1 t)

thus if we can prove that <:e> belongs to :SET and <:q(p) :e> belongs to :MEMBER, we have proved this case.

<:e> belongs to :SET because e belongs to S (by proposition 5.5)

To prove <:q(p) :e> belongs to :MEMBER, we prove MEMBER(q(p) e) belongs to T. Now, by definition of T, MEMBER(q(p) e) belongs to T if there is e' such that p(q(p)).e'=e, i.e. p.e'=e

let e'=e . Then, by assumption, p.e'=e, thus MEMBER(q(p) e) belongs to T (QED)

case 2: e<>e1

If p<>p', Let e=p'.e1' where e<>e1'.

by proposition 4, since e=p.e1 and e=p'.e1' there exists e2 such that

e=p'.p.e2,

e1'=p.e2,

e1=p'.e2

By a5, e>e1' and e>e1

now, :e=:q(p'):*:e1' (by definition)

:e1'=:q(p):*:e2 (by inductive assumption, since e>e1')

Thus :e=:q(p'):*:q(p):*:e2

Since by proposition 5.5 <:e2> belongs to the relation :SET, by proposition 3

:e=:q(p):*:q(p'):*:e2

But :q(p'):*:e2=:e1 (by inductive assumption, since e>e1)

Therefore :e=:q(p):*:e1 (QED) (end of proof of proposition 6)

Now we prove that Mo satisfies prog & T .

Mo satisfies prog because:

M satisfies prog,

Mo differs from M only in that it includes the mapping for the constants in S.

prog, which is in language L, does not have any constants from S.

Thus the value of the sentences of prog in Mo would be the same as their value in M, i.e. TRUE.

Note: by the same argument, Mo also satisfies the axioms of D.

To prove that Mo satisfies T, we have to prove that it satisfies the:

a- SET assertions,

b- ADD assertions,

c- MEMBER assertions.

a- Let SET(e) be an assertion in T. Then, by definition of T, e belongs to S. Thus, by proposition 5.5, <:e> belongs to :SET, i.e. Mo satisfies SET(e)

b-ADD

Let ADD(e1 t e) belong to T. Then, by definition of T, e and e1 belong to S, and e=p(t).e1 . e<>PHI because if e=PHI, then p(t)=PHI, which contradicts the definition of p.

Now, Mo satisfies all the axioms of D, thus it satisfies

ADD(s x x*s)<-SET(s)

Let :e1 be an instance of s and :t an instance of x in the above. Then,

since e1 belongs to S, <:e1> belongs to :SET (as proved in the previous section). Thus, since Mo satisfies the above sentence, the tuple <:e1 :t :t:*:e1> belongs to :ADD. But, by proposition 6, since e=p(t).e1, :e=:q(p(t)):*:e1=:t:*:e1 . Thus <:e1 :t :e> belongs to :ADD, which is what we wanted to prove. (end of proof for ADD)

c- MEMBER

Let MEMBER(t e) belong to T. Then, by definition of T, e belongs to S, and there is e' which belongs to S such that p(t).e'=e .Again, by definition of p, e<>PHI.

Now, MEMBER(x, x*s) is a theorem of D. Thus, Mo satisfies it.
Let :t be an instance of x in Mo, and :e' an instance of s in Mo. Since Mo satisfies MEMBER(x x*s), the tuple <:t, :t :* :e') belongs to the relation :MEMBER in Mo
But, since p(t) is a prime factor of e, by propostion 6,
:e = :q(p(t)) :* :e' = :t :* :e'
Thus the tuple <:t :e> belongs to the relation :MEMBER, which means that Mo satisfies MEMBER(t e) (end of proof for MEMBER)

II-The 'if' Part of the theorem: Prog & D is satisfiable if prog & T is satisfiable

Assume that prog & T is satisfiable. Then it has a least Herbrand model, M, in the language L(S). The domain of interpretation of M is the set of ground terms of L(S).

The language L(S) does not contain the function symbol * and the relation =. Thus the model M does not contain interpretations for these symbols.

We will extend M to M* by defining interpretations for the function *

and the relation =. Note that the domain of interpretation of M* is the same as that of M, i.e. the set of ground terms of L(S). Thus, M* is not a Herbrand interpretation, because its domain of interpretation is not the set of ground terms of L(S *).

The interpretation of the function * in M* is :*

Def: :t1 :* :t2 = p(:t1).:t2 if :t2 belongs to S

:t1 :* :t2 = p(:t1) if :t2 does not belong to S

Note: the definition of :* is chosen such that ADD(t1 t2 t1*t2) holds in M* when SET(t2) holds.

Def: The interpretation of = in M* is identity

Now we prove that M* satisfies prog & D.

M* satisfies prog because it differs from M only in that the interpretations for the function * and the relation = have been added to it, and these symbols do not occur in prog. (By the same token, M* satisfies T)

So we have to prove that M* satisfies the axioms in D. Now, D consists of the following axioms:

Set Definition (SD)

Set Existence (EX)

Set Membership (FR)

General Equality (EQ)

Set Equality (SEQ)

We prove that M* satisfies each of the above axioms.

SD:The axiom SD1, which is SET(PHI), is trivially satisfied by M* because it also occurs in T and M* satisfies T.

The axiom SD2 is: SET(set) <- SET(set') & ADD(set' term set)

Let :s, :s' and :t be instances of set, set' and term in the model M*. If the tuple <:s'> does not belong to the relation :SET or the tuple <:s' :t :s>
does not belong to the relation :ADD in M*, then the value of the sentence SD2 under this instantiation is TRUE.

If <:s'> belongs to :SET and <:s' :t :s> belongs to :ADD, then since M is a least Herbrand model of prog & T, SET(:s') and ADD(:s' :t :s) belong to T. Thus, by definition of T, :s belongs to the set S, which means that <:s> belongs to the relation :SET. Thus, the value of the sentence SD2 would again be TRUE. (end of proof for SD)

EX

First we prove that M* satisfies Ex* which is

ADD(s t t*s)<-SET(s)

Let :set, :term be instances of s and t in the model M*.

If <:set> does not belong to :SET, the value of the sentence is true.

If <:set> does belong to :SET, since M is a least Herbrand model of prog & T, SET(:set) belongs to T, i.e. :set belongs to S.

Therefore, :term :* :set= p(:term).:set

By definition of T, ADD(:set :term p(:term).:set) belongs to T. Thus, ADD(:set :term :term :* :set) belongs to T, and is thus true in M*.

Ex: $\forall x,y\ \exists x'$(ADD(x y x')<-SET(x))

Let :s and :t be instances of x and y in M*. Then if <:s> does not

belong to :SET, any instantiation for x' satisfies EX.

If <:s> belongs to :SET, then :s belongs to S.

Thus, ADD(:s :t p(:t).:s) belongs to T. Since M* satisfies T, <:s :t p(:t).:s> belongs to :ADD. Thus, the instantiation p(:t).:s for x' satisfies Ex.

FR:MEMBER(x y*z)<-> x=y OR MEMBER(x z)

This sentence is equivalent to the conjunction of the following three sentences:

a-(MEMBER(x y*z) <- x=y) &

b-(MEMBER(x y*z)<-MEMBER(x z))&

c-(MEMBER(x y*z)-> x=y OR MEMBER(x z))

We have to prove that M* satisfies all three conjuncts above.

Let :e, :t and :s be instances of x,y and z in M*

a- if the tuple <:e :t> does not belong to the relation :=, which is identity, then a is satisfied. If it does, then :e=:t. Thus we have to prove that the tuple <:e :e:*:s> belongs to the relation :MEMBER in M*

Now, consider the value of :e:*:s . We have to consider two cases: (a1) :s does not belong to S, and (a2) :s belongs to S.

a1-since :s does not belong to S, :e:*:s=p(:e). Thus we have to prove that <:e p(:e)> belongs to :MEMBER.

Now, by definition of T, MEMBER(term set) belongs to T, if there is an element of S, say s', such that set = p(term).s' . Since p(:e)=p(:e).PHI, MEMBER(:e p(:e)) belongs to T. Since M* satisfies T, the tuple <:e p(:e)> belongs to :MEMBER (end of proof of a1)

a2-Since :s belongs to S, :e:*:s=p(:e).:s Thus we have to prove that

the tuple <:e p(:e).:s> belongs to the relation :MEMBER in M*

Now, by definition of T, MEMBER(:e p(:e).:s) belongs to T. Thus, Since M* satisfies T, the tuple <:e p(:e).:s> belongs to the relation :MEMBER in M (end of proof of a2)

b- If the tuple <:e :s> does not belong to :MEMBER, then b is satisfied. If it does belong to :MEMBER, then we have to prove that <:e, :t:*:s> also belongs to :MEMBER.

Now, M is a least Herbrand model of prog & T. Therefore,

<:e :s> belongs to the relation :MEMBER iff MEMBER(:e :s) belongs to T

(because, by assumption, there are no clauses with MEMBER predicate at their head in the prog, and all the sentences in T, including MEMBER sentences, are ground assertions.)

Thus, MEMBER(:e :s) belongs to T. therefore, :s belongs to S, and there is s' such that :s=p(:e).s'

Since :s belongs to S, :t:*:s=p(:t).:s . Therefore we have to prove that <:e p(:t).:s> belongs to :MEMBER

But p(:t).:s=p(:t).p(:e).s'= p(:e).p(:t).s'=p(:e).s" where s"=p(:t).s' . Thus we have to prove that <:e p(:e).s"> belongs to the relation :MEMBER in M*

By definition of T, MEMBER(:e p(:e).s") belongs to T. Thus, since M* satisfies T, <:e p(:e).s"> belongs to :MEMBER. (end of proof for b)

c- If the tuple <:e, :t:*:s> does not belong to :MEMBER, c is satisfied. If it does belong to :MEMBER, then we must prove that the right hand side of c above is also satisfied.

We must consider two cases: (c1) :s does not belong to S (c2) :s belongs to S

c1- Since :s does not belong to S, :t:*:s=p(:t). Thus, the tuple <:e p(:t)> belongs to :MEMBER, i.e MEMBER(:e p(:t)) belongs to T. Thus, by definition of T, there must be s' such that p(:e).s'=p(:t).

Thus, either

c11) p(:e)=p(:t) and s'=PHI, or

c12) p(:e)=p(:t)=s', or

(Note: p(:e)=PHI and s'=p(:t) is impossible due to definition of p) in both c11 and c12, p(:e)=p(:t), which means that :e=:t. Thus the right hand side of the implication is satisfied.

c2- Since :s belongs to S, :t:*:s=p(:t).:s. Thus the tuple <:e p(:t).:s> belongs to the relation :MEMBER. Thus, since M is a least Herbrand model of prog & T, MEMBER(:e p(:t).:s) belongs to T. By definition of T, this means that there is s' such that p(:e).s'=p(:t).:s

Now, if :e=:t, the right hand side of c is satisfied. If :e is not equal to :t, by proposition 4 there is s" such that p(:e).s"=:s, i.e. MEMBER(:e :s) belongs to T. Thus, since Mo satisfies T, <:e :s> belongs to :MEMBER, and the right hand side is satisfied.(end of proof of c2)(end of proof of c)(end of proof of FR)

EQ: the equality axioms

These axioms are trivially satisfied by M* because the interpretation of the equality predicate is identity.

SEQ: set1=set2<- SET(set1)&SET(set2)&∀e (MEMBER(e set1)<->MEMBER(e set2))

Let :s1, :s2 be instances of set1 and set2 in M* . Then if the right hand side is not satisfied, SEQ is satisfied.

If the right hand side of SEQ is satisfied, the following must be true:

a- <:s1> belongs to :SET, i.e. :s1 belongs to S

b- <:s2> belongs to :SET, i.e. :s2 belongs to S

c- for all instances of e such as :x,

<:x :s1> belongs to :MEMBER iff <:x :s2> belongs to :MEMBER

Now, <:x :s1> belongs to :MEMBER iff MEMBER(:x :s1) belongs to T, i.e. iff p(:x) is a prime factor of :s1. Thus c implies that for all n, n is a prime factor of :s1 iff it is a prime factor of :s2. By proposition 5, this means that :s1=:s2, i.e the left hand side of SEQ is satisfied (end of proof of SEQ)(end of proof of theorem)