CONSTRAINT SATISFACTION - AN ALTERNATE APPROACH TO UNIFICATION IN PROLOG

by

Renganathan Sundararajan.

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

John W. Roach, Chairman

James P. Bixler

Charles D. Feustel

December 1987

Blacksburg, Virginia

CONSTRAINT SATISFACTION - AN ALTERNATE APPROACH TO UNIFICATION IN PROLOG

by

Renganathan Sundararajan.

John W. Roach, Chairman

Computer Science and Applications

(ABSTRACT)

Prolog is a linear resolution theorem prover with restrictions on the form of the clauses to prune the search space. Although Prolog has been widely used for implementing natural language systems, database systems, knowledge based expert systems and other A.I tasks, it has many limitations. The inferencing is symbolic and all the computations are performed in the Herbrand universe. Prolog has no notion of function evaluation. The uniform parameter passing mechanism, viz unification, is syntactic and too restrictive in nature. Moreover, Prolog uses a depth first search strategy combined with chronological backtracking which renders the SLD-Resolution incomplete. In this thesis, we study the incorporation of a constraints solver into Prolog. The generalization of unification into a constraints satisfaction algorithm allows the incorporation of function evaluation into unification. Constraint satisfaction is used to replace unification thus enhancing the power of Prolog. This idea could possibly be extended to other contexts where unification is used. Constraints can also appear as goals in the body of a rule. We discuss the enhancement in expressive power leading to an efficient solution of a larger class of problems. We also discuss ways of compiling the constraint solver and the modifications that are needed to be made in the Warren abstract machine for Prolog. An interpreter for the extended

Prolog (written in Prolog) incorporating a constraint solver is presented along with examples illustrating its power.

Acknowledgements

I wish to thank Dr.John Roach for his active guidance and immense patience. He introduced me to Logic Programming and his availability for discussions at all times helped me a lot in finishing this thesis. My thanks are also due to Dr.James Bixler and Dr.Charles Feustel for serving on my committee. Dr.L.T.Watson introduced and explained many concepts from Linear Algebra. My heartfelt thanks to him. I would like to thank Dr.Deborah Mayo for my knowledge of formal logic theory and for answering many of my questions with patience.

It has been a pleasure to teach under the guidance of . I have always wondered how she finds the time for all her teaching, advising and administrative duties. My thanks are due to for the endless hours of discussions we had on Logic Programming and Computer Science and for being a good friend.

My friend deserves a special mention for his friendliness and generosity without which I would not have done my M.S. My thanks are due to my parents for supporting my decision to resume studies after a long break and my roommates for two years, and , for their understanding and accommodating nature. Finally, to my wife , for everything.

Table of Contents

List of Illustrations

1.0 Introduction

1.1 Outline of the thesis.

In this first chapter, we briefly discuss the design of programming languages and the conflict between expressive power and elegance on the one hand and efficiency on the other. Prolog is a language that is quite expressive and efficient implementations of it are becoming common. We outline its limitations from the point of view of expressive power and suggest some improvements.

In Chapter two, we describe the important features of Prolog. We then discuss in depth the limitations of unification and give some pointers to the work that has been done on unification in higher order theories and in computational and equational logic. We show the problems associated with a procedural definition of logical relationships.

In Chapter three, we introduce constraint satisfaction as a model of computation. Many problems from diverse areas can be elegantly expressed and solved using this model. We discuss various categories of constraint satisfaction and constraint propagation systems.

Chapter four contains our proposal for replacing unification with constraint satisfaction. We introduce some terminology and a constraint satisfaction algorithm to replace unification and then discuss the properties of the algorithm.

In Chapter five, we give the syntax of the language that closely resembles Prolog but differs from it in the use of a constraint-solver in place of a unification algorithm. Some example problems that can be naturally expressed and solved in the extended language are given. We also show how negation can be implemented in Prolog correctly using the constraint satisfaction approach. An outline of our interpreter is then presented. As a specific example, we discuss an efficient strategy for checking the consistency of a system of linear equations in an incremental fashion. Other efforts at extending the domain of computation are then examined.

Chapter six discusses the generality of the constraint solver followed by our conclusions.

1.2 Expressive Power vs Efficiency

One of the many recurring themes in Computer Science is the conflict between expressive power and execution efficiency faced in the design of programming languages. Ideally one would like to have a language in which the problem specifications can be naturally expressed and the properties of the specifications can be proved without reference to the machine on which they will be executed. At the same time the specifications must be efficiently executable as a program on some machine. The search for such a language is the result of the growing complexity of today's systems and the need to ensure that such complex systems conform to such specifications. It is also the result of the growing realization that software engineering

methodologies alone will not be sufficient to ensure 'correct' systems unless the tools used provide a bridge between the specifications on the one hand and executable code on the other.

In the conventional approach to the design of programming languages, expressive power has been sacrificed in favor of efficiency, and the addition of many 'useful' features is thought to make a language more 'powerful'. Both C.A.R. Hoare [Hoar 81] and John Backus [Back 78] have been highly critical of this approach in their Turing award acceptance addresses.

Logic programming languages support a declarative style of programming. A logic programming language is one whose programs consist of sentences in a well-understood logical system that has simple notions of deduction, models and completeness theorems. The operational semantics of such a language is deduction. The system, ideally speaking, must also have an initial model. An initial model is characterized by the property that only what is provable is true and every thing else is false. The 'closed world' assumption of many logic programming languages refers to the initial model.

Declarative languages are related to logic programming languages in the sense that all logic programming languages are declarative but not necessarily the other way round. In a declarative language, the statements of a program describe the desired conditions without indicating explicitly how to attain them. In fact, a declarative language need not be executable. The declarative use is associated with such applications as program specification and database query. When a problem is so formulated declaratively, no attention is paid to the actual problem solving process.

Given the set of sentences that describe the problem to be solved, the computer applies forward or backward reasoning which renders the sentences executable.

Prolog belongs to the class of logic programming languages, although it has many features that make it far from being the ideal one. Prolog is the result of efforts to automate theorem proving. By Automated Theorem Proving, we mean the use of the computer to prove non-numerical results, i.e. to establish the validity of a set of sentences of a formal language. The resolution principle invented by J.A. Robinson [Robi 65] forms the basis of Prolog.

Prolog uses a form of resolution called SLD-Resolution - Linear Resolution with a selection function for Definite Clauses. R. Kowalski and M. Van Emden [Emde 76] showed that the restricted form of clauses used in Prolog correspond to Definite or Horn clauses - a subset of First Order Predicate calculus. In fact, it has been proved that the largest subset of First Order Predicate Calculus that has an initial model is the Horn Clause Logic. By initial model, we mean that any given set of horn clauses has a model i.e, an interpretation in which all the clauses in the set are satisfied.

Prolog suffers from many limitations in terms of both expressive power and execution efficiency. Prolog performs computations in the Herbrand Universe and this restricts its applicability solely to symbolic inferencing. Quantitative reasoning cannot be readily performed. Unification used to match the current goal with the head of a clause is too restrictive. For example, we cannot unify the two terms +(2 3) and +(X 2) and arrive at a value of 3 for X where X is a variable. We will discuss in depth, the limitations of unification in Chapter two. There is also no notion of function evaluation. Function evaluation allows deterministic computations and the depth-first search strategy of Prolog with its chronological backtracking does not permit efficient

deterministic computations without resorting to features such as 'cut'. We also study the incorporation of function evaluation into unification in the next chapter.

In this thesis, we will show how to enhance the expressive power of Prolog and enlarge the class of problems that can be solved. Instead of computing in the Herbrand universe, we allow the interpretation of function symbols. We generalize unification into a constraint satisfaction problem that handles both symbolic and numerical constraints and also the unification of terms containing uninterpreted function symbols. We present an interpreter in Prolog that treats $+$, $-$, $\geq$, $\leq$, and $\neq$ as constraints. The algorithm that we give is general in scope and not restricted to the functions that we have actually implemented. The following examples show the power of the extended Prolog.

Examples.

Some simple electric circuits can be analyzed with linear equations. Suppose we have the circuit indicated in Figure 1 on page 6. The dimensions used in this example are ohms for resistors, amperes for current, and volts for potential. We use the following three facts about circuits :

1. Ohm's Law: The potential difference across a resistor is the product of the current and the value of the resistor.
2. Kirchoff's law of current: The sum of the currents flowing from a junction equals the sum of the currents flowing into it.
3. Kirchoff's law of voltages: The algebraic sum of the voltages in any closed loop is zero.

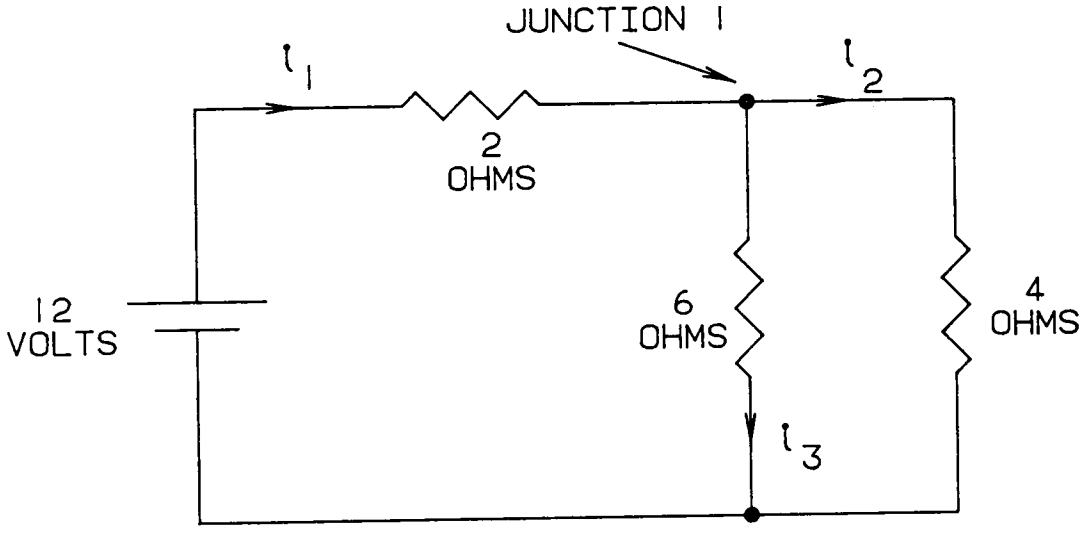

Figure 1. A simple electrical circuit.

We use the second law at junction 1 to obtain the equation

$i_1 = i_2 + i_3$

Using Ohm's law, we have that the voltage drop across the 2 ohm resistor is $2i_1$, across the 6 ohm resistor is $6i_3$ and across the 4 ohm resistor is $4i_2$. Using Kirchoff's law of voltage in the left hand loop, we have

$2i_1 + 6i_3 = 12.$

In the right hand loop we get

$4i_2 - 6i_3 = 0.$

In the larger loop we get

$2i_1 + 4i_2 = 12.$

We have therefore the following system of linear equations.

$2i_1 + 6i_3 = 12.$

$4i_2 - 6i_3 = 0.$

$2i_1 + 4i_2 = 12.$

$i_1 = i_2 + i_3$

We can find the currents by solving the above system of equations. In the extended Prolog, we express the equations as 4 constraints on 3 variables and let the system enforce the constraints and arrive at values for the variables if there is a unique solution to the system. The following is a transcript of our interaction with the extended Prolog.

Note: All the variables start with a '%' character and there is no 'if' or ':-' between the head of a clause and its body, that is, between the consequent and the antecedents of a rule. The query is to be typed in as (prove query). We will discuss in detail the syntax of the extended language in Chapter 5.

PROLOG/LISP IV.1.1 Feb 1985

; "do" accepts a file name and processes its contents as a program

```
:-(do kir)
; Echo the program as it is processed.

 ((circuit %i1 %i2 %i3)
       (= %i1 (+ %i2 %i3))
       (= 12 (+ (* 2 %i1) (* 6 %i3)))
       (= 0 (- (* 4 %i2) (* 6 %i3)))
       (= 12 (+ (* 2 %i1) (* 4 %i2))))

 (do kir)       ; done with processing
 :-             ; type in the query
 :- (prove ((circuit %a  %b %c)) )

(circuit 2.727273 1.636364 1.090909)
true
SUBJECT to the following constraints
Equations :
Inequations :
membership  :
NOT goals  :
time taken
1.840
```

Suppose we have an underdetermined system of linear equations. Normally, the system would respond that the equations are consistent, if they are, and print out a consistent set of constraints subject to which the query is satisfied. We can, however, set a flag and tell the system to print out a particular solution that together with the vectors in the nullspace span the solution space. The system would also print out the dimension of the null space, that is, the number of elements in the basis of the nullspace. The particular solution when added to any linear combination of the vectors in the nullspace constitutes a solution to the underdetermined system of equations. This is illustrated in the following example. Suppose four foods are used to make a meal. The foods have the following units per ounce of Vitamin B and Vitamin C respectively.

	Vitamin B	Vitamin C
Food 1	8	2
Food 2	4	6
Food 3	7	10
Food 4	3	6

Is it possible to combine these foods into meal of 16 oz with 96 units of vitamin B and 8 units of Vitamin C? If '%fd1', '%fd2', '%fd3', '%fd4' represent the number of ounces of foods 1, 2, 3 and 4 respectively, the equations would be expressed in the extended Prolog in the following way.

```
((balanced-meal %fd1 %fd2 %fd3 %fd4)
      (= 16 (+ %fd1 (+ %fd2 (+ %fd3 %fd4))))
      (= 96 (+ (* 8 %fd1) (+ (* 4 %fd2) (+ (* 7 %fd3) (* 3 %fd4)))))
      (= 80 (+ (* 2 %fd1) (+ (* 6 %fd2) (+ (* 10 %fd3) (* 6 %fd4))))
      )
:- ; set the trace flag on to see a trace of the execution.
:-(setq trace t)

:-(prove ((balanced-meal %a %b %c %d)) )
Q: 50-0 (balanced-meal $G5 $G6 $G7 $G8)
Q: 49-0 (= 16 (+ $G5 (+ $G6 (+ $G7 $G8))))
A: 49-0 (= 16 (+ $G5 (+ $G6 (+ $G7 $G8))))
Q: 49-0 (= 96 (+ (* 8 $G5) (+ (* 4 $G6) (+ (* 7 $G7) (* 3 $G8)))))
A: 49-0 (= 96 (+ (* 8 $G5) (+ (* 4 $G6) (+ (* 7 $G7) (* 3 $G8)))))
Q: 49-0 (= 80 (+ (* 2 $G5) (+ (* 6 $G6) (+ (* 10 $G7) (* 6 $G8)))))

; The particular solution vector is :

sol [0] = 6.727273
sol [1] = 3.454545
sol [2] = 2.727273
sol [3] = 3.090909

The null space dimension is 1::

0 :: 0.087039
1 :: -0.783349
2 :: 0.087039
3 :: 0.609272

A: 49-0 (= 80 (+ (* 2 $G5) (+ (* 6 $G6) (+ (* 10 $G7) (* 6 $G8)))))
A: 50-1 (balanced-meal $G5 $G6 $G7 $G8)

(balanced-meal $G5 $G6 $G7 $G8)
true
SUBJECT to the following constraints
Equations :
-80 + (2 * $G5) + (6 * $G6) + (10 * $G7) + (6 * $G8) = 0
-96 + (8 * $G5) + (4 * $G6) + (7 * $G7) + (3 * $G8) = 0
-16 + $G5 + $G6 + $G7 + $G8 = 0
Inequations :
membership :
NOT goals :
```

```
time taken
2.970000
```

Let us try an example where the system of equations and inequalities is inconsistent.

```
((solve %x %y %z)
      (= 7 (+ %x (+ %y %z)))
      (= 9 (+ %x (+ %y %z)))
      (# 8 (+ %x (+ %y %z))))      ; #  means ≠

:-
:-(prove ((solve (+ %a %b) %c %c)) )
Q: 50-0 (solve (+ $G13 $G14) $G15 $G15)
Q: 49-0 (= 7 (+ (+ $G13 $G14) (+ $G15 $G15)))

sol [0] =  3.000000
sol [1] =  0.500000
sol [2] =  0.500000
sol [3] =  1.000000

The null space dimension is 2::

0 :: -0.377964
1 :: 0.377964
2 :: 0.377964
3 :: 0.755929

0 :: 0.000000
1 :: -0.707107
2 :: 0.707107
3 :: 0.000000

A: 49-0 (= 7 (+ (+ $G13 $G14) (+ $G15 $G15)))
Q: 49-0 (= 9 (+ (+ $G13 $G14) (+ $G15 $G15)))

NO [ more answers ]
```

As can be seen from the above, when the trace flag is set, the interpreter displays the nullspace and its dimension for the system of equations being solved. More examples of the types of problems that can be solved are given in Chapter 5.

Finally, we look at two familiar examples, factorial and fibonacci number calculations. These two examples do not involve any equation solving if the first argument of the rules is ground.

```
(fibb 1 0)      ; base case
```

```
(fibb 2 1)      ; base case
(fibb %n %ans)                  ; naive recursive definition
        (fibb (- %n 1) %f1)
        (fibb (- %n 2) %f2)
        (= %ans (+ %f1 %f2))

:-(prove ((fibb 5 %a))  )          ; type in a query
(fibb 5 3)
true
SUBJECT to the following constraints
nil
time taken
2.65

:-(prove ((fibb 8 %a)) )
(fibb 8 13)
true
SUBJECT to the following constraints
nil
time taken
15.88

(Factorial 0 1)                 ; base case
(Factorial %n %ans)                 ; recursive case
     (Factorial (- %n 1) %int)
     (= %ans (* %int %n))

:-(prove ((Factorial  10 %A)) )
(Factorial 10 3628800)
true
SUBJECT to the following constraints
nil
time taken
5.30

:-(prove ((Factorial  5 %a)) )
(Factorial 5 120)
true
SUBJECT to the following constraints
nil
time taken
2.50
```

In the next few chapters, we will discuss the main features Prolog, the limitations of unification, the research efforts aimed at extending unification, constraint satisfaction as a model of computation and incorporating a constraint solver in Prolog. Replacing unification with a constraints solver and allowing constraints in the body of a rule

removes many of the shortcomings in the area of expressive power. We will then describe an interpreter for Prolog with a constraint solver built into it and the modifications needed in the Warren Abstract Machine for Prolog.

2.0 Prolog, the Language, its Features and Limitations.

In this chapter we will briefly discuss Prolog and its features, the limitations of unification, various efforts at extending unification and the problem with procedural interpretation of logical relationships. Prolog, a practical programming language, was created out of resolution theorem proving by adopting the following features.

- Restriction to Horn Clauses
- Unification
- Linear Resolution strategy
- Backtracking

2.1.1 Definitions.

A Prolog program is a finite set of Horn clauses or Definite clauses. Horn clause logic is a subset of First Order Predicate Calculus. We define certain terms of logic in order that the discussion about unification, resolution etc can be appreciated.

Definition. A **term** is recursively defined as :

1. A variable is a term.
2. A constant is a term.
3. If f is a n-place function symbol and t_1, to t_n are terms, then $f(t_1 , \ldots , t_n)$ is a term.
4. Nothing else is a term.

Definition. An **Atom** or a well-formed-formula (WFF) is defined as: If P is a n-place predicate symbol and t_1, ... t_n are terms, then $P(t_1 , \ldots , t_n)$ is an atomic formula or simply an atom.

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

Definition. A **clause** is a formula of the form

$$\forall x_1 , \forall x_2 , \ldots , \forall x_m (L_1 \vee L_2 \vee \ldots \vee L_n)$$

where x_1, ... x_m are the variables occurring in L_1, ... , L_n and each L_i is a literal. Given any statement of first order logic, we can arrive at the clausal form by following a systematic procedure of removing equivalences with bi-conditionals, replacing conditionals with an equivalent form, propagating negation through the complex formulae until the negation sign appears just before an atomic formula and by finally replacing existentially quantified variables with skolem functions or constants as the case may be. Thus the variables left in the clause are all universally quantified.

Definition. A **Horn** clause is a clause with precisely one positive literal and 0 or more negative literals. Since

$$(B_1 \wedge B_2 \wedge \ldots \wedge B_n) => A$$

can also be written as

$$A \vee \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_n$$

a Horn clause is conventionally written as

$$A \text{ :- } B_1, \ldots, B_n$$

where A is the consequent, $B_1, \ldots, B_n$ are the antecedents. ":-" is the reverse implication symbol, and the commas separating the antecedents denote conjunction. A horn clause with antecedents is generally called a rule and a Horn clause without the antecedents

$$A \text{ :-}$$

is called a **fact** (because it is unconditionally true). In the clause

$$A \text{ :- } B_1, \ldots, B_n$$

A is called the **head** and $B_1, \ldots, B_n$ the **body**

Definition. A **program** is a set of definite clauses. Since all variables in a clause are universally quantified

$$A \text{ :- } B_1, \ldots, B_n$$

means that "for each assignment of each variable, A is true if $B_1, \ldots, B_n$ all are true" and a clause A :- means that "for each assignment of each variable, A is true".

Definition. A **goal** clause is of the form

$$\text{:- } B_1, \ldots, B_n$$

that is, a clause with no consequent. This notation means that

$$\forall\, y_1,\ \forall\, y_2,\ \ldots\ ,\forall\, y_m\ (\ \sim B_1\ V\ \sim B_2\ V\ \ldots\ V\ \sim B_n\ )$$

where $y_1, \ldots y_m$ are the variables occurring in $B_1, \ldots, B_n$. Alternately,

$$\sim(\ \exists\, y_1\ \exists\, y_2\ \ldots\ \exists\, y_m\ )(\ B_1\ \wedge\ B_2\ \wedge\ \ldots\ \wedge\ B_n\ )$$

2.1.2 Unification

Unification is the process of generating the most common substitution instance of a set of descriptions. In actuality, it is simply an environment describing variable bindings and is applied to a description (or terms) by substituting the values for the variables occurring in that description (terms). Formally, given P_i and P_j two atoms, it is the process of finding a substitution σ

$$\sigma = \{\ V_i/T_i\ ,\ \ \ldots\ V_n/T_n\ \}$$

where $V_i, \ldots, V_n$ are distinct variables and $T_i, \ldots, T_n$ are all terms not containing $V_i, \ldots, V_n$ respectively, such that

$$P_i\ \sigma\ =\ P_j\ \sigma$$

Properties of Unification: Unification can also be thought of as finding the most general solution to the following set of simultaneous equations

$\{ s_1 = t_1, s_2 = t_2, \ldots, s_n = t_n \}$ where $s_1, s_2, \ldots, s_n$ and $t_1, t_2, \ldots, t_n$ are terms occurring in the atoms P_i and P_j respectively and in that order.

Unification of first order atoms is decidable. There exists a procedure that will succeed in unifying two atoms in finite number of steps precisely when they are unifiable and will return the most general unifier. A proof of this and the unification algorithm can be found in [Chan 73] [Lloy 84].

2.1.3 Resolution

Resolution is a generalization of 'modus ponens' and 'modus tolens' rules of inference. Resolution in its full generality can be stated as follows : If C_1 and C_2 are clauses containing no common variables, where

$$C_1 = l_1 \vee l_2 \vee \ldots \vee l_n \vee p_1 \vee p_2 \vee \ldots \vee p_m$$

and

$$C_2 = (\sim k_1) \vee (\sim k_2) \vee \ldots \vee (\sim k_n) \vee q_1 \vee q_2 \vee \ldots \vee q_j$$

and θ is the most general unifier of $(l_1, l_2, \ldots, l_n)$ and $(k_1, k_2, \ldots, k_n)$ then from C_1 and C_2 , infer

$$C_3 = (p_1\theta \vee p_2\theta \vee \ldots \vee p_m\theta \vee k_1\theta \vee k_2\theta \vee \ldots \vee k_n\theta)$$

the l's and the k's are said to be literals resolved upon and C_3 is the resolvent . Resolution is a complete rule of inference in the sense that

	From	And	Infer
modus ponens :	p	(not p) v q p => q	q
modus tolens :	(not q)	(not p) v q p => q	not p
Resolution :	(not m) m	m v n (not m) v n	n n

Figure 2. Resolution Compared with Modus Ponens and Modus Tolens.

'If a formula g follows from a set of formulas S, then there is a sequence of resolution of clauses in G (G = S $\cup$ { g }) that terminates in the empty clause $\square$'

The resolution rule is compared with modus ponens and modus tolens rules of inference in Figure 2 on page 18.

2.1.3.1 Linear resolution strategy

Most Prolog implementations use a simple linear strategy for doing resolution. This involves :

- Begin with a stated goal,
- find a clause whose head unifies with the goal,
- replace the goal with the body of the clause after applying the unifying substitution to it
- Recursively prove the goals in the body

The goal to be proven or the atom to be resolved is always resolved with one of the original clauses, never a derived one. Also no two original clauses are resolved. The clauses are chosen for resolution in the order in which they are input to the system and the goals are selected in left to right order. This depth first search strategy renders linear resolution incomplete (which is otherwise complete).

2.1.4 Backtracking

If during resolution, no clause can be found whose head unifies with the current goal, then Prolog backtracks to the previous goal resolved (undoing the substitutions in the process) and attempts to find an alternate rule/clause whose head will unify with the previous goal. If that fails, Prolog will backtrack further. Most Prolog implementations use chronological backtracking.

2.2 Limitations of Prolog

There are definitely many aspects of Prolog that need improvement. Some of these, such as a flexible search strategy (or better sub-problem selection) and problem solving strategies, are needed to make the language truly general purpose. Others like true negation, semantics for extra-logical features like cut, assert and retract are needed to bring the language closer to its theoretical model and deflect the criticisms against it. Systems like IC-Prolog, MU-Prolog, Prolog-II provide some form of control over sub-problem selection such as the use of 'wait', 'geler' declarations.

In this thesis, however, we will concentrate on the limitations of unification and the lack of inferences involving numerical quantities as against symbolic ones. Also, certain logical relationships when expressed in Prolog lead to problems that cannot be solved without using non-logical features. We give an example of this towards the end of this chapter. Then we show how the adoption of a constraints solver in lieu of a unification algorithm goes a long way toward solving these difficulties.

2.2.1 Limitations of Unification

The unification process is purely syntactic in nature. It does not take into consideration any special properties of the symbols involved, be they functions or predicates. For example, the following expressions would not unify.

{ equal (X b), equal (Y a) }

{ + (+ (a X)) b , + (a Y) }

{ + (3 4), + (5 2) }

{ + (3 4), + (5 Y) }

(Capital letters denote variables and small letters stand for constants)

The unify procedure will not return the substitutions

{ X / a, Y / b },

{ Y / X+b }

{ } (null)

{ Y / 2 }

This is because, in each of these cases, the special properties of the symbols involved must be exploited if they are to match, namely, the symmetry of equals and the associativity of plus. In addition, the unification procedure does not know the "reduction" of terms to their "canonical" forms, with the result that 3 + 4 and 5 + 2, for example, cannot be unified. Also, unification has no access to the special properties of functions and predicates. We can, however, use the associativity axiom and paramodulation to get around this difficulty. It is well known that adding the

associativity axiom to the initial set of clauses can cause the generation of many useless resolvents, make the proofs longer and detract from the naturalness of proofs. A number of examples using the axioms and paramodulation can be found in [Chan 73] and [Bundy 83].

What is the solution? There are many approaches to mitigating the difficulties caused by standard unification procedures. For example, we can build special purpose unification procedures that will have the axioms 'built-in'. In other words, we can delete the associativity axiom from the initial set of clauses and dispense with the paramodulation rule. All those formulae that were provable before will still be provable.

2.3 *Unification in higher order theories.*

In this section, we discuss unification in higher order theories where function variables and functionals are allowed. We have to build the alpha, beta and eta reduction rules of λ-calculus into the unifier. These rules are given below :

1. $\lambda x.\ X\ \ cnv_{\alpha}\ \ \lambda y.\ [y/x]\ X$

   provided there are no free occurrences of y in X

2. $(\lambda x.M)\ N\ cnv_{\beta}\ \ [N/x]\ M$

3. $\lambda x.M\ x\ \ cnv_{\eta}\ \ M$

   if there are no free occurrences of x in M.

Notes :

1. Capital letters denote arbitrary expressions in λ-calculus.
2. [y/x] X denotes an expression X in which all free occurrences of x are substituted by y .
3. A cnv_{α} B means that A is convertible to B and vice versa using the α-rule.

Implications of these rules :

α-rule : Formal parameters can be renamed freely as long as they do not introduce name conflicts.

β-rule : Substitute actual parameters for formal parameters in function abstractions.

η-rule : One can remove or introduce levels of indirection in function abstractions or applications.

Suppose we want to unify the following expressions.

F(Y) and $\sin(X).e^{x}$

We have two ways of unifying these two terms. (Please note that F and Y here are variables in a higher order theory and can be instantiated to function symbols alone or function symbols with their arguments.) But before unifying the terms, we need to form the lambda abstraction of $\sin(X).e^{x}$ so that F can be matched against that abstraction. The λ-abstracted version would be

$\lambda z.(\sin(z).e^{z})$

Now when we unify the two terms, we get the two unifying substitutions

$\{ F / \lambda z (Sin(z). e^z), Y / X \}$ and

$\{ F / \lambda z .z , Y / Sin(X). e^x \}$

The first substitution is called an **imitation** because F imitates the lambda-abstracted version of the second term. The second substitution is called a **projection (identity)** because F is defined to be an identity function whose value is its argument itself.

Note In the λ-abstracted version, we consider only closed-terms. By closed terms we mean that there are no free variables in the resulting expression. For example, an open term that contains free variable X would be

$\lambda z.(Sin(X).e^z)$

We do not consider open terms here. When a sub-expression containing free variables (say X and Y) occur within the scope of the bound variables X & Y of another expression (if any), then the free variables get bound.

2.4 *Basic Results of Decidability of Unification*

Two known results about the decidability of unification are presented.

Result 1 Unification in first order logic is decidable. There is a terminating algorithm (unification algorithm in [Robi 65]) to determine whether two given well formed

terms are unifiable. Note the difference between the decidability problem and the unification problem in first order logic.

Result 2 Unification in second order logic is undecidable.

By finding an effective method which reduces Hilbert's tenth problem to the unification problem in second order logic [Roge 67]. Since Hilbert's tenth problem is unsolvable, the unification in second order logic is undecidable [ibid].

Therefore, we know that unification for terms beyond first order (i.e. ≥ 2) is undecidable.

2.5 *Universal Unification*

Different from ordinary unification, where we only consider the types (order) of the terms being unified, universal unification is concerned with the unification problem under an equational theory, i.e., both terms and equations upon terms (such as associative, commutative, distributive properties) are considered. Since unification for terms beyond first order is undecidable, studies of E-unification (unification under an equational theory) are only meaningful for first order terms. Thus the terms in the rest of the thesis are first order terms. There are several different ways to view the E-unification problem. One approach is algebraic: terms and theorems of the equational theory are regarded as different algebras. Thus we study unification on monoids, semi-groups, groups, Abelian groups, rings, ideals and fields. Since we are interested in unification in logic, this approach will not be discussed further. The

interested reader may consult [Siek 79]. Another approach is logical. There are two main trends in studying unification in logic: One is to study unification in equational logic and the other is to study unification in computational logic. The first method has already yielded many fruitful results ([Raul 79] and [Siek 81]). The second method has been investigated thoroughly in the context of integrating logic programming and functional programming.

2.6 *Unification in Equational Logic*

Definition. T-unification. Let T be an equational theory and s and t be two given terms. Let $=_T$ stand for an equivalence relation between terms, that is, being provably equal using the equations in T. Let Σ be a set of substitutions and $\sigma \in \Sigma$. If

$$s\,\sigma =_T t\,\sigma$$

(i.e., they are equal in terms of the rules of the equational theory), then s and t are said to be T-unifiable. σ is called a unifier of s and t. The set of maximal unifiers for s and t is denoted as U_T (s,t). We say that a unifier of s and t is maximal if there does not exist two other unifiers λ , ϕ such that

$$s\,\sigma = t\,\phi$$

In ordinary unification, given two first order terms s and t, we can always decide whether s and t are unifiable, and further, when s and t are unifiable, we can always obtain a unique most general unifier. The situation is rather different in equational logic. First, unification may be undecidable. Second, even if unification is decidable,

there may not exist a unique most general unifier. That is, the cardinality of the set, $|U_T(s,t)|$ may be 0, 1, finitely many or ∞

For a complete list of known results about unification under T with various properties, see [Siek 81] and [Raul 79].

2.7 *Special Unification procedures.*

Many special purpose unification algorithms which build in equational axioms are known. In this section we examine the problems of designing an associative unification procedure. For a starter, let us look at a few examples. Consider unifying

{ X + Z , a + (Y + b) }

The regular unification procedure would return the substitution

{ X / a, Z / Y + b }

But this is not sufficient. The following substitutions also need to be generated, if the associativity axiom is to be made totally redundant.

{ X / a + Y, Z / b } ;

{ X / a + X1, Y / X1 + X2, Z / X2 + b }

For example, consider the two clauses

:- Q (a + (Y + b), b)

Q (X + Z, Z) :-.

These two are not regularly unifiable, but are associatively unifiable with the unifier being

{ X / a + Y, Z / b }

Likewise,

$$\leftarrow R(a + (Y + b), a + U, V + b)$$

$$R(X + Z, X, Z) \leftarrow$$

are associatively unifiable, where the unifier will be

$$\{ X / a + U,\ Y / U + V,\ Z / V + b \}$$

Thus associative unification entails producing not a single most general unifier but several most general unifiers. In fact, in the general case an infinite number of unifiers is required. The operation of replacing Y with U + V is variously referred to as 'variable splitting' , 'widening' etc in the literature.

The special purpose algorithms that build-in the associativity axiom have taken two different approaches. The different approaches trade off **completeness** and **termination** properties. There are no associative unification algorithms that are both complete and terminating. In the first approach, [Slag 74], [Stic 81], the unification algorithms are not complete. That is, they do not return all the most general unifiers in some cases. In their approach, there is an additional process outside of unification that alters the input expressions to cause the unification algorithm to return additional unifiers. This additional process - Slagle's widening operation for First Order Predicate Calculus and Stickel's variable-splitting operation for Qlisp - results in completeness.

The second approach by [Plot 72] is in the context of building in equational theories, of which associativity and commutativity are examples. Plotkin applies the variable-splitting operation continually inside the unification algorithm rather than outside of it. Thus his algorithm returns all the most general unifiers but may produce infinite unifiers in some cases and may never terminate.

2.7.1 Properties of associative unification

The question of whether two expressions are associatively unifiable is decidable. There is no finite and complete unification algorithm, however, for the associativity axiom.

2.8 Unification in Computational Logic

In computational logic, we consider unification in theories with a term rewriting system.

The fundamental difference between equations (in equational logic) and term rewriting rules (in computational logic) is this : equations denote equality (which is symmetric) and thus when we unify two terms, we allow the replacement of equals by equals; term rewriting systems treat equations directionally and hence when we try to unify two terms, we substitute a term which matches the left hand side of an equation by its right hand side.

Definition A term rewriting system R over a set E of terms is a set of rewriting rules, each of the form $l \rightarrow r$ where l and r are terms in E or terms containing variables ranging over E and all the variables in r occur in l.

Definition A term rewriting system is a **canonical** term rewriting system iff

- $\rightarrow$ is noetherian (terminating),

  i.e., no infinite derivation $t_1 \rightarrow t_2 \rightarrow \ldots$ exists.

- $\rightarrow$ is confluent,

  i.e., for any t_1 , t_2, and t such that $t \rightarrow t_1$, $t \rightarrow t_2$, there is a t' such that $t_1 \rightarrow t'$ and $t_2 \rightarrow t'$. (Church-Rosser property).

The problem of deciding whether a term rewriting system has the confluence property is unsolvable in the general case. However, [Rose 73] discusses the conditions that would enable one to test a given term rewriting system for the confluence property.

Under a rewriting system, a reduction is the application of a rewrite rule in order to simplify a term to which it is applied. The rewriting system must have the finite termination property, that is, after a finite number of applications of legal reductions to any term , an irreducible term - to which no further reductions can be applied - is produced. A term which is irreducible is said to be in normal form.

The normal form of a term t is denoted as R(t). Let us define reduction more formally. This definition is due to [Redd 85].

Reduction is the reflexive transitive closure of the relation $\rightarrow$, called one-step reduction, which is defined as follows.

1. If the rewriting system consists of a rewrite rule

   $s \rightarrow t$ and there is a substitution θ

   such that $s' = s\,\theta$, then

   $s' \rightarrow t\,\theta$

   where s, t, s′ are all terms.

2. If $s' \rightarrow t\,\theta$ and C is an expression in which

   s′ occurs, then

   $C \rightarrow C\,[s'/t\,\theta]$.

That is, the term s' in C is substituted by t θ

How do we obtain a term rewriting system from a given equational theory?
The **completion algorithm** of Knuth-Bendix can be used to generate a rewriting system R from an equational theory T. But such a system may not exist. The termination of term rewriting systems is in general undecidable [Ders 85].

2.8.1 Unification in first order logic with built-in addition function

The equational theory for addition (denoted by TA) is given by

$x + 0 = x$; identity

$x + -x = 0$; inverse

$(x + y) + z = x + (y + z)$; associativity

$x + y = y + x.$; commutativity

Hence it is a commutative group or Abelian group.
Unification under the equational theory TA is decidable and there are finitely many maximal unifiers. The proof of this can be found in [Siek 81].

In order to obtain a canonical term rewriting system for addition, we have to throw away the commutative equation: $x + y = y + x$, since a term rewriting system will never be noetherian if it contains a commutative rule: $x + y \rightarrow y + x$.

The equational theory for addition without the commutativity axiom has the following rewrite rules [Stic 81].

1. $x + 0 \rightarrow x$; identity
2. $x + -x \rightarrow 0$; inverse
3. $(x+y) + z \rightarrow x + (y+z)$; associativity
4. $0 + x \rightarrow x$
5. $-x + x \rightarrow 0$
6. $-x + (x+y) \rightarrow y$
7. $-0 \rightarrow 0$
8. $-(-x) \rightarrow x$
9. $x + (-x+y) \rightarrow y$
10. $-(x+y) \rightarrow (-y) + (-x)$

The above set of rules, however, will not allow us to unify the following ordered pairs of terms::

< +(X Y), -(X Y) >

< 7 , 3 >

and return the substitution { X/5, Y/2 }. Term rewriting systems are not meant to be used for solving simultaneous equations and if we want to build-in numeric inferencing, then obviously recourse to term rewriting is not going to help us.

2.8.2 Weakening the Reduction Assumption: Narrowing

Reduction is a mechanism for evaluating ground expressions. By evaluation, we mean the representation of an expression in its normal form. Reduction by itself is not very powerful because it can handle only ground expressions. That is, if there are variables in the input expression that is being rewritten, the resulting expression

will not have them instantiated. This is because, the left hand side of a rewrite rule is only matched with an expression and not unified. To find the denotations of non-ground expressions another technique is necessary. When we embed term rewriting into unification, we retain the constraints of the underlying term rewriting system, viz, directionality. We would be, for example, able to unify such terms as (2+2, x) and bind x to 4. But given the terms (x+2, 4), the directionality of the rewriting system does not allow as to derive the value of x to be 2. Narrowing is another mechanism that extends the power of reduction.

Narrowing : When an expression contains variables and hence cannot be reduced, narrowing makes an assumption about the values of the variables and proceeds to reduce the expression under that assumption. Such assumptions are represented by the substitution θ for the variables in the original expression. [Redd 85] defines narrowing as a ternary relation

$$e \rightarrow_{\theta} d .$$

That is e is narrowed to d using the narrowing substitution θ .

Narrowing is a generalization of reduction in that ,

if $e \rightarrow d$ is a reduction, then

$e \rightarrow_{id} d$ is a narrowing, where id is the identity substitution.

Narrowing is implemented by replacing matching by unification. The substitutions produced by unification for the variables in the input expression are collected and output as the narrowing substitution.

An example will help clarify the narrowing procedure. Suppose that we want to unify the two terms

{ (X + 2), 4 }.

Assume that the terms are represented in their normal form, that is, as

{ (X + (1 + 1)), (1 + (1 + (1 + 1))) }.

Normally, these two terms will not unify. Given a term rewriting system for addition, we find that (X + 2) can not be reduced since it contains a variable X. We now try to narrow the expression (X + 2) by successively unifying it with the left hand sides of the rewrite rules, preserving the substitution for X obtained in the process. The left hand sides of the rules 2, 3, 4, 5, 6 and 9 could possibly unify with term (X + 2) , but since we are interested in finding a non variable substitution for X, we are left with rules 3 and 4. When (X + 2) is unified with the LHS of the third rule, we get the substitution {X / X' + Y, Z / 2} and a new term (X' + (Y + 2)). Normalizing this term to (X' + (Y + (1 + 1))) and unifying it with 4 results in the substitution {X' / 1, Y / 1} and thus the narrowing substitution is {X / 2}. Using narrowing, we can therefore find the correct substitution instance that unifies the terms { (X+2) , 4 }. Narrowing, however, has the tendency to return infinite substitutions in some cases, though the terms after such substitutions would not unify.

Summary.

We discussed unification, its properties, its limitations, and the research efforts aimed at improving its applicability and power. We see that when the function symbols are interpreted or endowed with special properties such as associativity, we run into problems. Inferences involving numeric quantities seem particularly hard and we seem to have to be satisfied with reasoning involving symbolic quantities. Recourse to term rewriting systems allows us to rewrite terms into their canonical forms, if they exist, and unify these canonical forms. But there are problems with term

rewriting systems too. We feel that there must be a direct and intuitive way to do inferences involving numeric quantities.

2.8.3 Procedural definition of logical relationships.

Another problem with Prolog is the way in which some logical relationships are procedurally defined. For example, the set membership relationship is expressed by these two clauses :

```
member(A, (A . *rest)).
member(A, (*head . *rest))  if
        notequal (A, *head)
        member(A, *rest).
```

Although a list is only an approximation of a set, these two clauses will suffice to define membership declaratively. Let us see, however, what happens when the member relation is 'called' with the second argument uninstantiated. At the first call, the first element of the list will be instantiated to A and the call would succeed. When backtracking occurs, the second clause of the procedure would be tried and depending on the implementation one of two things will happen.

1. Since *head is a variable notequal(A *head) would succeed, thereby instantiating the second element of the list to A. In this case, there are infinite solutions to the call member(A, *list).
2. notequal(A, *head) would cause a control exception at run time and the program would abort, since notequal being a system predicate expects ground arguments.

Case 2 is clearly undesirable. What about case 1? We do not want the member rule to succeed infinite number of times since that would cause solutions to the right of the member goal in the SLD-refutation tree to be missed. Some Prolog implementations provide control facilities such as 'wait', 'geler' declarations that would delay the execution of the goal member(A, *list) until such time that *list gets bound to a non-variable term. We can also use system calls such as var/1 to determine whether the second argument to the member procedure is a variable or not and use this information appropriately. Though such declarations/system calls are useful in other contexts, it seems odd that we need a non-logical feature to enforce procedurally, what is essentially a logical relationship! The 'gap' between a declarative reading of the program and its procedural interpretation can be no more evident than in this example.

There are more powerful techniques for problem solving. Constraints Programming is one of them. In the next chapter, we define constraints and discuss the power and limitations of constraints solvers.

3.0 Constraint Satisfaction-A Model of Computation.

There are many models of computation such as Turing machines, Logic, Recursive Functions, Lambda Calculus, Post production systems and Markov algorithms. Turing machines, for example, have been extensively used in studying the complexity of algorithms and in deciding what is computable. It is well-known that whatever models of computation have been invented so far, all have been proved to be equivalent to the Turing machine in their computing power. According to Church's thesis there is only one notion of 'computability' - whether 'human' computable or 'machine' computable- and that notion is captured by the Turing machine. Still, for solving a variety of problems, we do not use a Turing machine but some model of computation that closely resembles the problem at hand and in which the problem could be expressed naturally.

Logic Programming, for example, allows the statement of relations and inference rules (albeit in a restricted form) and the deduction of various conclusions by an inference engine. Functional Programming on the other hand supports the view that all problems can be reduced to the format of functions and function applications.

Another such model is Constraint Satisfaction. In this model, a program is just a statement of relationships that must hold among symbolically named quantities. This includes a static relational model for the meaning of constraints and a computational one for enforcing the relationships. The difference between Prolog-like languages on the one hand and constraint based languages on the other is this: Prolog interprets a collection of clauses with a common head predicate as a procedure and the goals within the body of a rule as procedure calls. It imposes a backtracking control mechanism and produces 'answers' one tuple at a time. Constraints can be thought of as interacting physical devices that 'reject' 'invalid' tuples of relations and provide all the answers at one time. It can be likened to a query processing mechanism of relational data bases that produces an entire relation as the answer. [Stee 80]

In his Turing award lecture, Floyd [Floy 79] says

> When the programming language makes a paradigm convenient, I will say the language supports the paradigm. When the language makes a paradigm feasible but not convenient, I will say that the language weakly supports the paradigm..... Even the paradigm of Structured Programming is at best weakly supported by many of our programming languages.

There are numerous examples where the constraints model perfectly fits the problem at hand. They range from puzzles such as the eight queen problem, magic square, crypto-arithmetic to scheduling problems in Operations Research [Baza 77]. One may say that constraint programming languages offer the highest level of programming in which a problem can be stated without worrying about the procedural aspect of 'how to' solve it.

3.1 Networks of Constraints.

Normally, the relations among objects are represented as a constraint network in a constraint satisfaction system. The nodes of the network are quantities or terms on quantities (i.e combinations of quantities). Each node has associated with it a unary predicate or set called its label, which characterizes the range of values possible for the term. Nodes are connected by constraints that express the relationship between those terms. The constraints that are input are called explicit constraints and those that are derived are called implicit.

Given a network of constraints among quantities or terms on quantities, there are various ways of enforcing the constraints and arriving at the sets of values for the quantities that would satisfy all the constraints simultaneously. One can use inferential and non-inferential systems to find these sets of values. An important point is that all inference systems use non-inferential methods as their basic steps. For example, arithmetic is performed using machine instructions and not deduced from Peano's axioms. In fact, our search for efficient quantitative inference systems forced us to cite this as a limitation of languages like Prolog which are otherwise reasonably efficient for symbolic inferences.

Constraint satisfaction systems may involve quantitative or non-quantitative reasoning. They have been used mainly for non-quantitative reasoning within the A.I field, for example, as in Waltz's scene analysis system. [Wins 84] There are some important differences between quantitative and non-quantitative constraint satisfaction systems. The range of non-quantitative variables, for example the set of

labels for an edge or a fork in Waltz's scene analysis program, is usually a finite set of values. In the case of non-quantitative variables, when constraint propagation (to be defined later) fails to prune the sets completely, one can resort to an exhaustive search. On the other hand, quantitative variables normally have an infinite range of values. They also have an internal structure and there are a number of unary predicates we can use to limit the possible values (like positive, negative, odd, even, upper bound, lower bound, etc.).

3.2 Constraint Propagation.

Given a network of constraints, there are various methods of enforcing the constraints and arriving at the sets of values that would satisfy all the constraints simultaneously. There is, however, an algorithm which is at the core of all these methods. We will describe that algorithm now. This algorithm uses an idea known as constraint propagation. Informally, constraint propagation means that information deduced from a group of constraints is recorded as a change in the network and this change is used for further deductions. This causes the effects of constraints to spread gradually through the network. Before we give the actual constraint propagation algorithm, let us define the notion of 'refinement'. Let C be a constraint on nodes $X_1,\ldots,X_k$ and S_i be the label set for X_i. Then we define a function "REFINE" as

$$\text{REFINE}\ (C,X_j) = \{ a_j \in S_j \mid \exists\ a_i \in S_i, i = 1, \ldots, k,\ i \neq j\ C\ (a_1,\ldots,a_j,\ldots,a_k) \}$$

That is, we refine the label set of a variable participating in a a constraint to include only those values or value ranges for which we can find values or value ranges for all other variables in that constraint from their respective label sets such that the constraint is satisfied. Having defined refinement, we present two procedures [Davi 86] that together make up the constraint propagation algorithm.

```
procedure REVISE ( C (X1,...Xn))
 begin CHANGED ← φ
  for each argument Xi do
   begin  S ← REFINE (C, Xi) (* find the new set of values for  Xi *)
     if S = φ  then halt.  (* original set of constraints is inconsistent *)
     else if S ≠ Si then   (* does it equal to the old  set of values for Xi? *)
       begin Si ← S ;    (* if not, consider Xi to be changed *)
            add Xi to CHANGED
       end
     return CHANGED
   end


procedure WALTZ
  begin  Q ← queue of all constraints
    While  Q ≠ φ do        (* more constraints left? *)
      begin   remove constraint C from Q ;
        CHANGED ← REVISE ( C )   (* refine 1 constraint and find the
                                    set of variables affected *)
        for each Xi∈ CHANGED do
          for each constraint C′ ≠ C which has Xi in its domain
            add C′ to Q  (* these constraints are to be checked again *)
      end
  end.
```

In the Waltz procedure, the constraints are used to restrict the sets of values by the repeated application of a 'REFINE' operator. The control structure applies the REFINE operation repeatedly until it produces no more changes in the value sets. This state is called the network quiescence state. Since the 'REFINE' operation is sound deductively, WALTZ's algorithm which is just an iteration of refinement, is also sound.

Normally, forward inferencing in a constraint network (called assimilation) is performed using a constraint propagation algorithm. Constraint networks may also support query answering where the user may ask for the value of a term or the truth value of a proposition based on the state of the network.

There are a number of valuable properties shared by constraint propagation systems, such as :

- simple control strategy applied to a simple updating module
- graceful degradation under time limitations (interrupting the process in the middle can give partial results)
- incremental addition of constraints
- locality assumption (i.e the constraints normally affect only a small portion of the network) shared by many A.I systems / applications.

3.3 Categories of Constraint Propagation systems.

Previously we mentioned that there are various ways of enforcing the constraints in a constraint satisfaction system. We will classify these methods now so that we can compare our proposed method with these later on. The classification is due to [Davi 86] and it is based on the type of information updated in a constraint network.

Constraint Inference. New constraints are inferred from the existing ones and added to the network.

Label Inference. In this system, each node is labelled with a set of possible values. During assimilation, the constraints are used to restrict these sets.

Value Inference. In Value inference systems, nodes are labeled with constant values and the force are used to find values of unlabeled nodes from labeled ones.

Expression Inference. It is a generalization of value inference, in which nodes may be labelled with a value expressed as terms over the values of other nodes. When a node is given two different labels, these two are equated and the resultant equation is solved. CONSTRAINTS [Stee 80] uses expression inference.

Relaxation. In this, all nodes are given exact values which may not be consistent with the constraints. The assimilation processes manipulates these values so as to satisfy the constraints.

Relaxation Labelling. In relaxation labelling, nodes are assigned probabilities for various values. The assimilation process changes the probabilities by combining the previous probability on the node with the probabilities indicated by other nodes.

Label Inference systems. In label inference systems, the nodes of the constraints network are labelled with the sets of possible values. This set could be a finite set of discrete values, as in the case of Waltz's scene labelling system [Wins 84] [Char 84] or an infinite set represented in a finite way, for example an interval on the real line.

3.4 Complexity aspects.

We will now briefly discuss the complexity aspects of constraint satisfaction methods. The complexity measures differ for wholesale and incremental systems. In a wholesale system, all the constraints are available at the beginning and are fixed thereafter. In an incremental system, acceptance of constraints is alternated with answering of queries. Systems that alternate between quantitative and non-quantitative inference can be considered as incremental systems. We are concerned with the time complexity of constraint systems whether wholesale or incremental. The time complexity would, however, depend on the nature of the systems. In a wholesale system, when we want to accept a set of constraints, we may be happy with an $O(n^2)$ or $O(n^3)$ algorithm. For example, a constraint in the form of bounds on quantity differences ($x-y \in [a,b]$) can be mapped on to the well known problem of finding the shortest path in a directed graph. [Davi 86] The well known algorithm for solving this problem has complexity $O(n^3)$. Systems that allow bounds only on quantities can treat the bounds on a quantity x_i as the bounds on the term $x_i - x_0$ where x_0 is a special node with a constant value of 0. In such a case, the problem of checking whether a constraint $x_j \in [a,b]$ holds is isomorphic to constructing a single-source shortest path in a weighted, directed graph. If all upper bounds are positive and all lower bounds are negative, then Dijkstra's single source shortest path finding algorithm can be used. This algorithm has complexity $O(n^2)$. In an incremental system, we are concerned with the time to process a single constraint and we expect running times to stay relatively small as the system grows large. In such a situation, we might want an O(log n) or O(n) algorithm. Detailed estimates of the complexity measures for label inference systems whose nodes are real valued quantities can

be found in [Davi 86] The constraints on real valued quantities can range from the trivial to the uncomputable. Davis' paper [Davi 86] discusses the complexity measures associated with

- evaluating the bounds on a term of a particular kind, given the label sets of its constituent terms.
- refining a constraint of a given kind.
- solving or checking the consistency of a system of constraints.

for various types of constraints. The types of constraints are :

- Unary predicates, i.e, signs and intervals.
- Order relationships.
- Bounds on differences, of the form $x - y \geq c$, $x - y \leq d$ etc.
- Linear equations and inequalities with unit coefficients.
- Linear equations and inequalities with arbitrary coefficients.
- Boolean combinations of constraints.
- Algebraic equations.
- Transcendental equations.

For example, solving a system of Boolean constraints is NP-complete, solving a system of bounds on quantity differences is $O(n^3)$, evaluating an algebraic expression on a set of labels is NP-hard, and evaluating a transcendental function is, in the worst case, uncomputable. Since we are interested in replacing unification with constraints satisfaction in Prolog, can we hope to get an algorithm which has the same complexity as the unification algorithm or better? The complexity of the constraint solver would depend on the nature of the constraints handled and [Davis

86] discusses the various cases. For example, it is possible to get an O(n) algorithm for incrementally checking the consistency of a system of linear equations. We outline such an algorithm in Chapter 5.

In this chapter, we looked at the constraint satisfaction model of computation and various methods of constraint satisfaction. We presented Waltz' algorithm for constraint propagation and briefly discussed the dependency of the complexity measures on the nature of constraints handled. In the next chapter, we will give a constraint satisfaction algorithm that generalizes first order unification. We will suggest the changes needed in the constraint propagation algorithm and show that this would allow the handling of interpreted function symbols within Prolog without restricting the arguments of the functions to be either input or output.

4.0 A Constraints Solver to Replace Unification.

Both Logic programming languages and Constraints solvers provide a declarative, machine independent reading of their programs and are reasonably efficient in performing automatic deductions. These two are, however, complementary in many respects. Quantitative reasoning has so far not been accorded the same status as symbolic reasoning in logic programming languages whereas constraint solvers excel in quantitative inferencing. The clean semantics of logic programming languages, notwithstanding the presence of non-logical features, comes at a price. The semantics are defined with reference to a canonical domain, viz the Herbrand Universe, the set of all first order terms that can be formed from the constant and function symbols in a program. This in turn forces the unification of terms to be syntactic. In Chapter two, we looked at research efforts aimed at augmenting unification by considering the properties of function symbols involved, by incorporating term rewriting systems within the unification algorithm etc. These do not buy us adequate quantitative reasoning. It would therefore be a good idea to have a programming language that allows statements of relationships between objects / quantities, be they logical implications or numerical constraints and let the inference engine perform the deduction by using an appropriate mechanism. Although many versions of logic programming languages have been around for many years now and the constraint satisfaction technique has been applied to many problems, only in the

last two years there have been proposals to combine these two techniques to create a language that supports a declarative reading of the program and efficient deductions involving both symbolic and numeric constraints.

In this chapter we define a constraint solver and show how we can replace the standard unification algorithm with the constraint solver. We want to treat the process of making the head literal of a clause identical to a goal literal as a constraint satisfaction problem. We retain the restriction to horn clauses so that efficient symbolic deduction can be performed in a backtracking system. Prolog computes unifiers in the Herbrand universe, assigns no meaning to any of the program symbols and leaves the interpretation to the user. Here, we have associated with the symbols $+$, $-$, $\leq$, $\geq$, etc., their standard interpretation and treated them as constraints. So the main components of the system are :

- Horn clauses with constraints in the body
- Interpreted function and predicate symbols
- A constraint solver in the place of a unifier
- An inference engine for SLD-theorem proving
- Backtrackable goals and deterministic constraints

Toward that end, we first define some terms and then give a constraint satisfaction algorithm that subsumes unification.

Definition. Interpreted Functors

A functor is associated with a function on the intended domain of interpretation. An uninterpreted functor is one that is not interpreted. For example, we treat $+$, $-$, and $*$ as interpreted function symbols.

Definition. Compatible Functors

Two functors are said to be compatible if

1. both are uninterpreted functors and are syntactically identical or
2. both are interpreted functors with a common domain and the equality or inequality of terms that have these functors as their main functors is decidable.

Definition. Terms

1. A variable is a term
2. A constant is a term
3. If f is an n-place uninterpreted function symbol and X_1 ... X_n are terms, then $f(X_1 ... X_n)$ is a term.
4. If f is an n-place interpreted function symbol and X_1 ... X_n are terms that are
   - variables
   - constants from the intended domain of f
   - terms of the form g $(Y_1 ... Y_m)$ where the functors f and g are compatible and the terms Y_1 ... Y_m do not contain any function symbols at all or contain only those function symbols (along with their arguments) compatible with f and g.

Definition. Constrained terms

A term as defined by sub-clause 4 of the definition of term is called a constrained term and any variable that occurs in a constrained term is also

a constrained term. A term that is not a constrained term is an unconstrained term.

Definition. Interpreted Predicates

An interpreted predicate is an n-place predicate symbol associated with a relation on the intended domain. In our case, the relations $=, \neq, \leq, \geq$, will be the interpreted predicates.

Definition. Constraints

A constraint is a relation of the form $< f, t1, t2 >$ where f is an interpreted predicate symbol and t1 and t2 are constants from the intended domain or variables ranging over the intended domain or constrained terms whose main functors are the interpreted function symbols of the intended domain. For example, $< \neq, x, y >$, $< =, -(x,y), \times(2,q) >$, $< \geq, x, 0 >$ are all constraints, where x, y, q are variables ranging over the reals.

Definition. Consistent constraints

A constraint $C = <f, t1, t2>$ is consistent if, for the variables $v_1 ,\ldots v_n$ occurring in t1, t2, $\exists$ $a_1 , \ldots a_n \in D$ where D is the intended domain of the constraint C and the functor f, such that $<f, t1', t2'>$ holds, where t1′ and t2′ are t1 and t2 respectively with the values $a_1 \ldots a_n$ substituted for the variables $v_1 \ldots v_n$.

Definition. Consistent set of constraints

A set of constraints is consistent if all the constraints in the set are simultaneously consistent.

4.1 Constrained Unification

We view the process of making the head literal of a clause identical to a goal literal as a constraint satisfaction problem. Given a set of ordered pairs { (s_1, t_1) ,.... (s_n,t_n)} and a set of constraints S, where s_i and t_i are first order terms and the variables in s_i through s_n do not occur in any of the terms t_i through t_n , decide whether there exists a substitution / solution σ and a consistent set of constraints S' such that $\forall$ i

- either $s_i \sigma = t_i \sigma$
- or $S' = S \cup \{ < =, s_i, t_i > \}$

Operational mechanism

Let S be the set of constraints. We partition the set S into disjoint subsets $S_1,..S_m$ such that

- If the interpreted function symbols occurring in the terms of the two predicates are compatible, then the two predicates belong to the same sub-set.
- Two constraints that do not satisfy the above property belong to different sub-sets.

The above condition partitions the set of constraints into equivalence classes. The main idea is to "solve" the constraints in each of these classes and propagate the values obtained in the process through other constraints until all the constraints in the set S are consistent or we find an inconsistency and in that case there is no solution to the problem of Constrained Unification. The variables that occur in the

terms $(s_1,t_1),...(s_n, t_n)$ may be instantiated in the process of finding a solution to the Constrained Unification problem. If the variables are unconstrained terms, i.e, do not occur in any constraint, obviously there is no need to run the constraints solver whenever they are instantiated. Likewise, when a constrained variable is instantiated, we need to check the consistency of all the constraints in which it appears. This may lead to more instantiations and these are propagated through the constraints until either an inconsistency is found or no more instantiations take place.

4.1.1 Constrained Unification algorithm.

We now give an algorithm to solve the Constrained Unification problem.
Let $T = \{ (s_1,t_1), ... (s_n, t_n)\}$. Set σ (the set of substitutions) to ϕ. The set of constraints C initially may or may not be empty. The following steps will produce the substitution σ and the consistent set of constraints C', if they exist.

1. If T is empty, check the consistency of the constraints in S. If they are consistent, output σ and S and stop, else there is no solution.
2. Apply the σ to all the terms in T simultaneously.
3. Remove a pair (s_i, t_i) from T.
4. If s_i and t_i are unconstrained terms and either s_i or t_i is a variable and the variable term does not occur in the non-variable term then add either $s_i = t_i$ or $t_i = s_i$ to σ as the case may be.
5. If s_i and t_i are both unconstrained terms of the form $f(a_1,...a_m)$ and $f(b_1,..b_m)$ then add the pairs (a_1,b_1), ... (a_m,b_m) to T and go to step 1.

6. If s_i and t_i are both constants, then they must be identical or else stop (no solution).
7. If one of s_i or t_i is a variable, and the other is a constrained term, add the constraint $< =, s_i, t_i >$ to S and check the consistency of the set S. If S becomes inconsistent then there is no solution. If S remains consistent and some variables that occur in the constraint are instantiated or a substitution has been found for them, add the substitutions to σ and go to step 1.
8. If both are constrained terms of the form $f(a_1,.. a_p)$ and $g(b_1,..b_q)$, then f and g must be compatible functors. If not, there is no solution, stop. If yes, then add the constraint $< =, f(a_1,..a_p), (b_1,..b_q) >$ to S. (Note: p is not necessarily equal to q). If S becomes inconsistent, then there is no solution. If S remains consistent, add the substitutions (if any) found for the variables in S to σ and go to step 1.

The above algorithm assumes that there exists a decision procedure for checking the consistency of the set of constraints S. The constraints could be refined one at a time or a subset that forms an equivalence class of the set of constraints can be solved at a time. We modify the Waltz procedure given in Chapter 3 in the following way. The queue of constraints will be initialized to all the constraints in which the terms t_1, t_2 of the new equality constraint $< =, t_1, t_2 >$ occur. From this queue, remove and refine one constraint at a time or refine a set of constraints that form an equivalence class. This modification is proposed since efficient algorithms exist for processing a set of constraints such as linear equations over the domain of reals, set unions/ intersections, linear inequalities etc. The above algorithm does not differentiate between numerical constraints like linear equations and inequalities and symbolic

constraints like membership in a set of discrete symbolic quantities, such as labels for lines in line drawings.

4.1.2 Soundness, Completeness and Termination Properties.

Soundness. The unification algorithm for first order terms has been proved to be complete and sound. The above algorithm can be seen to have two easily identifiable parts. One part deals with solving the system of equations of first order unconstrained terms and the other with constrained terms. The former is a variant of the standard unification algorithm and the latter iterates Waltz's algorithm which is sound. Hence the above algorithm is sound.

Termination. The set T is a finite set with a finite number of variables and finite number of terms. Assuming that we have a terminating consistency checking algorithm for each of the constraints that are handled by the system, the constraint satisfaction algorithm will terminate because there are only a finite number of variables that can be assigned values. Hence the above algorithm will also terminate.

Completeness. Once again, the completeness property of the above algorithm depends on the completeness of the algorithm that would check the consistency of the set of constraints in the intended domain. If the latter would not return all the solutions or the most general solution if it exists, then the above algorithm would also be incomplete as far as solutions over the intended domain are concerned. The choice of functors interpreted over some intended domain should be such that we have a decidable algorithm for testing the equality of terms involving the interpreted functors. For example, if we allow both + and * as the interpreted functions over the

real field, there is no decidable procedure for determining the equality of arbitrary terms involving both these functors. In that case, the above algorithm would have to be incomplete.

In the next chapter, we describe an implementation of the above algorithm, discuss the implementation details and compare our work to other, similar work.

5.0 An implementation of the Constraints solver.

In this chapter, we describe a Prolog interpreter that incorporates a constraint satisfaction algorithm instead of the standard unification algorithm. We also discuss work of a similar nature aimed at incorporating a constraints solver in Prolog.

The interpreter treats the functions plus and minus adirectionally without expecting any of the arguments to be ground. It also solves problems in which linear inequalities naturally arise. We do not, however, expect the interpreter to solve maximization or minimization of a linear function subject to some constraints on the variables. Set membership is another constraint built into the interpreter, although only finite sets are handled.

5.1 Interpreted function and predicate symbols

The interpreted function symbols of the language are : + and -. Both are 2-place function symbols and are prefixed to the arguments. The interpreted predicate symbols are : $\leq$, $\geq$, =, $\neq$, and IN. The last one stands for set membership, ϵ. All these are 2-place predicate symbols. Initially these are the only interpreted functors

and predicate symbols handled by the system. We want some convenient notation to specify the coefficients of the variables in the equations / inequalities. We allow a special function symbol '*' to denote the multiplication of a variable by its coefficient. We insist, however, that at least one of the arguments of the 2-place function symbol '*' be a number. We have built in a linear equation and inequation solver and do not handle non-linear equations. The reason for not handling non-linear equations is their complexity. As mentioned in the previous chapter, evaluating an algebraic expression, given the value ranges of the variables occurring in it is NP-hard. The built-in equation solver has two parts: one that handles simultaneous equations and the other that simplifies an equation. The first part is not invoked every time an equation is encountered. There is an expression/equation simplifier that handles the algebraic manipulation of expressions and equations and reduces them to their simplest form. We show below a program segment that takes in expressions and simplifies them and a sample of the expressions simplified by the interpreter.

```
(assert0
   ((simplify-expression *a) if
       (classify     *a *b)
       (simplify-all *b *c)
       (print *c)
       )
   )

:-(simplify-expression (+  (* 5 %a)  (* 4 %b) ))
  (simplify-expression (+ (* 5 %a) (* 4 %b)))   /* irreducible */

:-(simplify-expression (* 4 5))              /* reduced to 20 */
  20

:-(simplify-expression (+ (+ (* 4 5) (* -4 5)) (* 5 %a)))
  (* 5 %a)                                   /* reduced to 5a  */

:-((simplify-expression  (+ (* 5 %a)  (* -3 %a)) )
(* 2 %a)                                     /* reduced to 2a  */

:-(= (+ (+ (*  5  %a) (* 4  %b)) (+  (- 0 (* 2  %a)) (- 0 (* -3 %b))))
( (+ (* 3  %a) (* 7  %b))                    /* reduced to 3a + 7b  */
```

5.2 *Syntax of the extended language.*

We give below the syntax of the extended language. The syntax closely follows the syntax of Virginia Tech Prolog.

- A variable always starts with a '%' sign and has one or more digits or alphabetic characters following it.
- A constant is either a number or a string of characters not starting with a digit or a '%' sign.
- A term is :
  - a variable or
  - a constant or
  - of the form (functor arg1 ... argn) where functor is an uninterpreted n-place functor and arg1 through argn are terms.
  - of the form (i-functor arg1 .... argn) where i-functor is an interpreted functor and arg1 through argn are terms that do not contain
    - ▲ constants other than those from the domain on which i-functor is defined,
    - ▲ any functor incompatible with i-functor.
  - nothing else is a term.
- A constraint has the form :

  (i-predicate arg1 arg2 ... argn) where i-predicate is an n-place interpreted predicate symbol and each argument is just a variable or a constant from the domain of interpretation of the i-predicate or a function symbol from the domain of interpretation of the i-predicate followed by its arguments.
- An atomic formula has the form :

(predicate arg1 arg2 ... argn) where arg1 through argn are terms.

- A literal is an atomic formula or the negation of it. The negation of an atomic formula is expressed as :

  (not (atomic formula)) where arg1 through argn are terms.

- A clause can have one of the following two forms :

  (atomic formula)

  (atomic formula body)

  where body is either empty or has one or more literals or constraints.

  Notice that there is no 'if' or ':-' following the head of a clause.

- A program in the extended language consists of a set of clauses.

5.3 *Examples*

We show a few examples of the problems that can be expressed and solved in the extended language. We then discuss the implementation details of the interpreter.

Circuit Analysis. Multiplication of complex numbers and some elementary properties of an electric circuit can be described by the following set of rules. This example is from [Jaffar 87].

```
Complex-mult ( c(R1, I1), c(R2, I2), c(R3, I3) ) :-
      R3 =   R1 * R2  -  I1 * I2,
      I3 =   R1 * I2  +  R2 * I1.
Ohmlaw (V, I, R) :-
      complex-mult (I, R, V).
inductor-law (I, V, L, W) :-
      complex-mult ( c(0, W * L), I, V).
capacitor-law (I, V, C, W) :-
      complex-mult ( c(0, W * C), V, I).
```

In the extended language, the rules appear as follows.

```
((complex-mult  (c %r1  %i1) (c %r2 %i2) (c %r3 %i3) )
   (= %r3  (- (* %r1  %r2)  (* %i1  %i2)))
   (= %i3  (+ (* %r1  %i2)  (* %r2  %i1)))
   )
```

```
((ohmlaw %v  %i  %r  )
   (complex-mult  %i  %r  %v)
   )
((inductor-law %i  %v %l %w)
   (complex-mult  (c 0  (*  %w %l)) %i %v)
   )
((capacitor-law %i %v %c %w)
   (complex-mult  (c 0  (* %w %c))  %v  %i)
   )
```

Given the above program, we type in a few queries and the following is the output produced by the interpreter. The following four queries involve doing adirectional arithmetic but not simultaneous equations solving capabilities.

```
:-(prove ((complex-mult %l  (c 10 20) (c 20 50) )) )
(complex-mult (c 2.40  0.20) (c 10 20) (c 20 50))
true
SUBJECT to the following constraints
nil
time taken
1.250
(prove ((complex-mult %l (c 10 20) (c 20 50))))


:-(prove ((complex-mult (c 1.5 -0.20) %c (c 20 50) )))
(complex-mult (c 1.50 -0.20) (c 8.733624 34.497817) (c 20 50))
true
SUBJECT to the following constraints
nil
time taken
1.230
(prove ((complex-mult (c 1.50  -0.20) %c (c 20 50))))

;In the following  examples, we trace the execution of the queries.

:-(prove ((inductor-law (c 20 -8) %a  5  6)) )
Q: 50-0 (inductor-law (c 20 -8) $G126 5 6)
Q: 49-0 (complex-mult (c 0 (* 6 5)) (c 20 -8) $G126)
Q: 48-0 (= $G135 (- (* 0 20) (* (* 6 5) -8)))
A: 48-0 (= 240 (- (* 0 20) (* (* 6 5) -8)))
Q: 48-0 (= $G136 (+ (* 0 -8) (* 20 (* 6 5))))
A: 48-0 (= 600 (+ (* 0 -8) (* 20 (* 6 5))))
A: 49-1 (complex-mult (c 0 (* 6 5)) (c 20 -8) (c 240 600))
A: 50-1 (inductor-law (c 20 -8) (c 240 600) 5 6)
(inductor-law (c 20 -8) (c 240 600) 5 6)
true
SUBJECT to the following constraints
nil
time taken
```

```
2.150
(prove ((inductor-law (c 20 -8) %a 5 6)))


:-(prove ((inductor-law (c 20 -60)  (c 240 %a) 5 10)))
Q: 50-0 (inductor-law (c 20 -60) (c 240 $G148) 5 10)
Q: 49-0 (complex-mult (c 0 (* 10 5)) (c 20 -60) (c 240 $G148))
Q: 48-0 (= 240 (- (* 0 20) (* (* 10 5) -60)))
A: 48-0 (= 240 (- (* 0 20) (* (* 10 5) -60)))
Q: 48-0 (= $G148 (+ (* 0 -60) (* 20 (* 10 5))))
A: 48-0 (= 1000 (+ (* 0 -60) (* 20 (* 10 5))))
A: 49-1 (complex-mult (c 0 (* 10 5)) (c 20 -60) (c 240 1000))
A: 50-1 (inductor-law (c 20 -60) (c 240 1000) 5 10)
(inductor-law (c 20 -60) (c 240 1000) 5 10)
true
SUBJECT to the following constraints
nil
time taken
2.260
(prove ((inductor-law (c 20 -60) (c 240 %a) 5 10)))
```

The following example necessitates the invocation of the simultaneous equation solver.

```
:-(prove ((ohmlaw (c 10 20)  %I  (c 5 30) )) )
Q: 50-0 (ohmlaw (c 10 20) $G27 (c 5 30))
Q: 49-0 (complex-mult $G27 (c 5 30) (c 10 20))
Q: 48-0 (= 10 (- (* $G31 5) (* $G32 30)))
A: 48-0 (= 10 (- (* $G31 5) (* $G32 30)))
Q: 48-0 (= 20 (+ (* $G31 30) (* 5 $G32)))

sol [0] var : $G31  val =  0.702703
sol [1] var : $G32  val =  -0.216216

The solution is
(($G31 . 0.702703) ($G32 . -0.216216))

A: 48-0 (= 20 (+ (* 0.702703 30) (* 5 -0.216216)))
A: 49-1 (complex-mult (c 0.702703 -0.216216) (c 5 30) (c 10 20))
A: 50-1 (ohmlaw (c 10 20) (c 0.702703 -0.216216) (c 5 30))
(ohmlaw (c 10 20) (c 0.702703 -0.216216) (c 5 30))
true
SUBJECT to the following constraints
nil
time taken
1.930
(prove ((ohmlaw (c 10 20) %I (c 5 30))))
```

In the following example, there is no need to multiply out the expression

```
(* 50 (* $G60 12) )
```

before equating $G60 to 0. The expression simplifier arrives at the value of 0 for the variable $G60 without multiplying out (* 50 (* $G60 12)) fully.

```
:-(prove ((inductor-law (c 50 0) (c 200 0) 12 %a)))
Q: 50-0 (inductor-law (c 50 0) (c 200 0) 12 $G60)
Q: 49-0 (complex-mult (c 0 (* $G60 12)) (c 50 0) (c 200 0))
Q: 48-0 (= 200 (- (* 0 50) (* (* $G60 12) 0)))
A: 48-0 (= 200 (- (* 0 50) (* (* $G60 12) 0)))
Q: 47-0 (= 0 (+ (* 0 0) (* 50 (* $G60 12))))
A: 47-0 (= 0 (+ (* 0 0) (* 50 (* 0 12))))
A: 49-1 (complex-mult (c 0 (* 0 12)) (c 50 0) (c 200 0))
A: 50-1 (inductor-law (c 50 0) (c 200 0) 12 0)
(inductor-law (c 50 0) (c 200 0) 12 0)
true
SUBJECT to the following constraints
nil
time taken
2.230
(prove ((inductor-law (c 50 0) (c 200 0) 12 %a)))
```

Let us a try case where the set of equations and inequalities is inconsistent.

```
:-(do incon)        ; incon contains inconsistent set of equations,
                    ; and inequalities
(solve %x %y %z)
(= 7 (+ %x (+ %y %z)))
(= 9 (+ %x (+ %y %z)))
(# 8 (+ %x (+ %y %z)))      ; # stands for ≠


:-(prove ((solve %a %b %c)) )
 NO [ more answers ]          ; answer from our interpreter
```

The following are examples of inconsistent systems.

```
:-(prove ((solve (+ %a %b) %c %c)) )
Q: 50-0 (solve (+ $G13 $G14) $G15 $G15)
Q: 49-0 (= 7 (+ (+ $G13 $G14) (+ $G15 $G15)))
A: 49-0 (= 7 (+ (+ $G13 $G14) (+ $G15 $G15)))
Q: 49-0 (= 9 (+ (+ $G13 $G14) (+ $G15 $G15)))
NO [ more answers ]


:-(prove ((solve (+ %a %b)  %c  2)) )
Q: 50-0 (solve (+ $G19 $G20) $G21 2)
Q: 49-0 (= 7 (+ (+ $G19 $G20) (+ $G21 2)))
NO [ more answers ]
```

```
(prove ((solve (+ %a %b) %c 2)))
:-(prove ((solve (+ %a 3) 4 %b)))
Q: 50-0 (solve (+ $G25 3) 4 $G26)
Q: 49-0 (= 7 (+ (+ 3 $G25) (+ 4 $G26)))
NO [ more answers ]

(prove ((solve (+ %a 3) 4 %b)))
:-(prove ((solve (+ %a %b) %c %d)))
Q: 50-0 (solve (+ $G30 $G31) $G32 $G33)
Q: 49-0 (= 7 (+ (+ $G30 $G31) (+ $G32 $G33)))
A: 49-0 (= 7 (+ (+ $G30 $G31) (+ $G32 $G33)))
Q: 49-0 (= 9 (+ (+ $G30 $G31) (+ $G32 $G33)))
NO [ more answers ]
(prove ((solve (+ %a %b) %c %d)))
```

Negation

The presence of a constraint solver facilitates the handling of negation in Prolog. Negation, as implemented in many compilers / interpreters, is unsound. It makes the interpreter/compiler incomplete in the sense that solutions to goals may be missed. Examples of this can be found in [Nais 85]. The problem with negation is this: a "not goal" is typically proved by showing that the "goal" fails. But the failure of the goal cannot logically imply the "not goal" unless all the arguments in the goal are ground or no variable in the goal got instantiated to a non-variable term. If any variable in the goal gets bound to a non-variable term then all we have proved is that for some x_1, .. x_n the goal is true, where x_1, .. x_n are the variables occurring in the goal. We cannot, however, conclude that forall x_1 .. x_n the "not goal" is false. Hence, the "not goal" cannot fail based on this alone. Many solutions have been proposed to remedy this problem. One efficient way of getting around this difficulty is to mark all the unbound variables in the "not goal" at run time and check if they get bound after the "goal" has succeeded. If that happens, suspend the execution of the "not goal" and proceed as if it has succeeded. If and when the variables in the "not goal" get bound, resume the "not goal". Repeat this process if necessary. If the "not goal" is still pending when no more goals are left to be proved, just output this information.

Since the above procedure is totally internal to the compiler and transparent to the user, we have implemented this treatment of "not" as another constraint. We make the checks at run-time and add the "not goal" to the list of constraints if necessary. The following example of the treatment of "not goal" will make our discussion clear.

```
((same %p %p))
((female alice))
((parents alice  victoria albert))
((parents edward victoria albert))
((sister %p1 %p2)
     (not (same %p1 %p2))
     (female %p1)
     (parents %p1 %m %f)
     (parents %p2 %m %f)
     )

 :-(prove ((sister alice %a)) )

  (sister alice edward)
  true
  SUBJECT to the following constraints
  nil
  time taken
  1.630
```

In unsound implementations of Prolog, the "not" goal in the definition of the sister predicate would fail because the system would try to prove "(same %p1 %p2)" first and since that succeeds binding %p1 to %p2, the goal " (not (same %p1 %p2)) would fail. In our implementation, we treat "not" as a constraint and this forces %p1 to be different from %p2 and produces the correct answer "(sister alice edward)". Just as we have $\neq$ as a built-in predicate which is defined for numeric arguments, there is a built-in predicate "not-equal" for non-numeric arguments. We use that in the following example with identical results. These two examples also show how constraint propagation can be used to enforce the constraints, be they numeric or symbolic.

```
((female alice))
((parents alice  victoria albert))
((parents edward victoria albert))
```

```
((sister %p1 %p2)
    (not-equal %p1 %p2)
    (female %p1)
    (parents %p1 %m %f)
    (parents %p2 %m %f)
    )

:-(prove ((sister alice %a)) )

(sister alice edward)
true
SUBJECT to the following constraints
nil
time taken
0.980
```

5.4 *Implementation details*

The interpreter for the extended language is written in Prolog. Prolog is ideally suited for writing meta-interpreters. A meta-interpreter for Prolog could be written in a few lines. Such an interpreter actually uses the unification and clause retrieval mechanisms of the Prolog interpreter/compiler on which it runs. We need, however, to write an interpreter that has constraint satisfaction in place of unification. The interpreter has six easily identifiable parts. They are

- **Classifier.** It accepts rules/goals, converts them to the internal format and stores them. Each argument of a rule/goal is classified into an atom, a number, a constrained term or an unconstrained term. As in the case of Virginia Tech Prolog, we do not distinguish between a a list and a functor with its arguments. A functor with its arguments is just a list whose first item is the functor and the rest are its arguments.

- **Clause Selector** It selects a clause whose head can be matched with the goal and calls on the constraint solver to solve the system of equations formed by the arguments of the goal and of the clause head. It also handles backtracking.

- **Constraint Propagator** This implements the constrained unification algorithm given in Chapter 4.

- **Constraint Solver** The constraint solver checks the consistency of the constraints.

- **Expression Simplifier** Simplifies expressions and equations.

- **Tracer** The tracer handles trace requests, converts the goals and rules form internal format to external format and prints them out.

Of these parts, we will discuss the constrained Unifier, the expression simplifier, the propagator and the Constraint solver in detail.

Constrained Unifier

The constrained unifier operates on a pair of terms at a time. The terms are in their simplest form. The unifier handles the unification of unconstrained terms in the usual way. When two constrained terms are to be unified, the unifier checks for the compatibility of the terms. If they are compatible (as defined in Chapter 4), it then calls on the satisfy-constraints rule. The satisfy-constraints rule checks whether the constraints hold in the ground case. If the terms are not ground, it invokes different rules for different types of constraints. For example, let the terms to be unified be:

{ (5 + %a) , (-2 + (* 3 %a) - (* 2 %b) }

it would form an equation

((5 + %a) - (-2 + (* 3 %a) - (* 2 %b))) = 0 and call the simplifier with the left hand side of the equation. The resulting simplified expression, in this case,

7 - (* 2 %a) + (* 2 %b)

will be examined for degeneracy and inconsistency. By degeneracy, we mean that the result is 0 and hence the equality constraint is already satisfied. There is no need to add this to the list of constraints. If the result is a variable, it is set to 0 and if it is a number, the constraint cannot be satisfied and hence a failure is triggered. If the result is another expression, it is added to the set of equations. The result of the constrained unification is propagated through the rest of the terms to be unified and the process continues.

Expression Simplifier

The expression simplifier simplifies expressions of whose main functor is +, -, and '*'. The simplifier is written in Prolog, and has four major sets of rules, to handle propagation of minus inside an expression, simplify minus expressions, plus expressions and multiplication expressions. The simplifier handles only linear expressions and generates warning messages on encountering non-linear equations. Each of the three functors +, -, and '*' have two arguments each. The simplifier recursively simplifies the arguments before it reduces the main expression. Internally, the 'minus' expressions are converted into a 'plus' expression. For example, if A and B are two expressions, the simplifier rewrites

A - B into A + (- B)

and the minus sign is propagated inside the expression B. The routines that handle the 'product' expressions multiply out the arguments and simplify the resulting expression. For example,

(5 * (%a - %b)) - (3 * (%a + %b))

would first be simplified to

(5 %a - 5 %b) - (3 %a + 3 %b)

and then to

(2 %a - 8 %b).

The rules that simplify the plus expression combine variables with the same names and check for degeneracies, such as coefficient zero, one or minus one and remove them or put them into the proper internal format.

Constraint Propagator

The constraint propagator maintains two lists of constraints; an active list and an inactive list. Initially, all the constraints are in the active list. The Propagator invokes a constraint solver for each type of constraint. The constraint solver may return a failure or success flag and a list of variables with their values if there is a unique solution. If there is a failure, the interpreter would backtrack. If there is a finite solution, then the values are propagated through the remaining constraints. If any constraint in the inactive list has one of its variables bound to some value, then the constraint is removed from the list and added to the active list. The constraint propagator stops when there is an inconsistency or when the list of active constraints is empty. The formal algorithm is given in Chapter 4.

Constraint Solver

Solving systems of constraints means solving systems of equations and inequations and other types of constraints. The numeric equations solved in the system yield directly to classical linear algebra techniques. We have chosen to use Gaussian elimination which is $O(n^2)$ for testing the consistency of an additional equation and $O(n^3)$ for the entire system of equations. It is satisfactory for most purposes since the number of constraint equations is typically small.

The constraint solver has separate rules for each type of constraint. In the case of equality and inequality constraints we need to order the variables in some way so that a variable occurs in the same position in each equation. This is needed because we can then read the coefficients of the variables into the coefficient matrix easily. But this ordering approach is costly. We do not expect the variables to be in the same order every time. We associate a column number with each variable. Since each variable in our system is simply an atom in the underlying Prolog compiler we use the atom pointer and simply look up its number. New variables, whenever they are created have a special number associated with them. When new variables are encountered, we allot a new column and associate the column number with them. The consistency of the existing system together with the new equation is checked and the result is reported to the Propagator along with a solution, if there is a unique solution.

The handling of inequalities is more involved. When the arguments of an inequality are not ground, we cannot check for its immediate consistency or inconsistency. We introduce a slack variable to make the inequality an equation and simplify the equation. The slack variable is constrained to be ≥ 0 for the $\leq$ and $\geq$ type constraints and $\neq 0$ for the $\neq$ type constraint. The consistency of the existing system of equations together with the new equations is checked. If there is no solution to this system, then we backtrack. If there is a unique solution, we can check whether the values of the slack variables satisfy their constraints. If the solution in not unique, then we transform each equation of the form

$$Ax = b$$

into an equation of the form

$$y_i = b - Ax$$

where y_i is a new variable. We make sure that b is non-negative, by multiplying the whole equation by -1 if necessary. This system is equivalent to the original system if and only if all the y_i are zero. We form an objective function

$$z = -y_i \ldots - y_n \quad (1)$$

which will be maximized for non-negative y_i 's only when each y_i is zero. We use the simplex algorithm [Pres 86] to maximize this function. Note that these steps are normally used to transform a set of equations and inequations into a restricted normal form where the constant in each equation in non-negative and the left hand variable y_i occurs in only one equation. The objective of this exercise is to find an initial feasible vector which is used as the starting step for finding the optimal feasible vector for the original objective function. But in our case we do not have any original objective function to maximize and hence we stop when we find an initial basic vector. We may not find one and in that case the system is inconsistent. If we find one, then some variables may be exactly zero. We check if any of the slack variables of type $\neq$ has a value zero and in that case, too, there is no solution.

Completeness

The procedure we outlined above for handling the inequalities of type $\neq$ is not complete. It is possible that the procedure may report inconsistency although the set of equations and inequalities is consistent. In the case of inequalities of type $\neq$, the slack variable introduced to transform the inequality into an equality is constrained to be strictly greater than or strictly less than 0. But, the simplex algorithm would consider only non-negative values for the slack variables. To get around this difficulty, we need to either modify the simplex algorithm or replace each slack variable constrained to be non-zero by the difference of two new slack variables that are constrained to be greater than or equal to 0, and proceed. Further, when the simplex

algorithm returns with a vector that maximizes the objective function (1), we need to check that the difference between the two new slack variables is not 0. The same problems arise in the case of strict inequalities since we can represent

$x > y$ by $(x \geq y$ and $x \neq y)$ and

$x < y$ by $(x \leq y$ and $x \neq y)$ and vice versa.

Incremental Consistency checking algorithm

We could maintain a system of equations in a solved form and check for the consistency of the system incrementally. This incremental check has complexity O(n). The idea is to maintain the coefficient matrix of the system in upper-triangular form and when an equation is added to this matrix, convert it into upper triangular form using plane rotations or Householder reflections. The plane rotations or Householder reflections necessary to zeroise the elements of the last row take O(n) number of steps, where n is the number of variables in the equation. This method however requires extra space of O(n). We need to save the changes made to the coefficient matrix in order to undo the changes when we discard the last equation on backtracking. Furthermore, it is not possible to solve the system of equations in O(n) time even though we may check its consistency incrementally in O(n) number of steps. We have not used this method in the interpreter. We incrementally maintain the coefficient matrix in upper triangular form. We use the Gaussian elimination algorithm which takes $O(n^2)$ time incrementally to maintain a matrix in upper triangular form.

Membership constraints

Treating the membership relation as a constraint is in general very difficult. Consider infinite sets expressed in some finite way. If a variable is constrained to be a member

of two infinite sets, how do we check this constraint for its consistency in general? We can express set union, intersection, difference etc. in terms of the membership relation. In our treatment of the membership constraint, we allow only finite sets that do not have other sets as their members. Further, we insist that at least one of the arguments of the membership relation be ground. Without this restriction, we can have constraints like the following

$x_1 \in x_2$, $x_2 \in x_3$, ... $x_{n-1} \in x_n$, $x_n \in x_1$.

where x_i are all variables. Checking the consistency of a system of constraints of the form given above involves the detection of cycles in a directed graph.

5.5 *Related work.*

Prolog-III.

Prolog-III is an extension of Prolog-II in which unification was replaced by an algorithm to solve a set of simultaneous equations over the domain of rational trees. Prolog-III builds on Prolog-II in that it allows complete processing of Boolean Algebra and treats equations and inequalities over the domain of rationals as constraints. In a recent article [Colm 87], Colmerauer describes the salient features of Prolog-III. The restrictions to linear equations and inequalities that we have adopted can also be found in Prolog-III. A number of solved examples that show the power of Prolog-III are also presented. Prolog-III is more powerful than our system in the sense that it handles boolean constraints and constraints over infinite, rational trees in addition to the constraints that we handle. It seems, however, that Prolog-III does not allow any constraint propagation. If more constraints are to be added, they must have

disjoint domains or a constraint propagation control structure must be introduced. Prolog-III handles boolean as well as arithmetic constraints. Since variables are allowed in arithmetic constraints, we would expect that boolean constraints are also handled in the same way. If variables participate in more than one type of constraint, then solutions to one set of constraints could lead to instantiations of variables in others which in turn may restrict the values of other variables and so on and the system must be capable of handling the chain reaction. If not, it is nothing more than a built-in equation solver and not a constraint satisfaction system.

Constraint Logic Programming

The CLP(R) interpreter [Lass 87] contains a Prolog-like engine, a constraint solver and a module that provides an interface between the two. The inference engine recognizes the constraints, evaluates complex arithmetic expressions and passes on the constraints to the solver. CLP(R) allows non-linear constraints but delays their processing until sufficient number of variables are instantiated and the constraints become linear. It is, however, the responsibility of the programmer to make sure the constraints become linear. The omission of non-linear constraint solvers from CLP(R) is for the same reason as ours. The existing algorithms for solving non-linear constraints are inefficient. Linear inequalities are solved using a modified simplex method. The linear equations are checked for consistency using an incremental algorithm. No mention, however, is made of the extra storage needed to keep track of the changes in the coefficient matrix of the equations in solved form. It is not clear whether arbitrary constraints on the existing domain could be added nor their effect on completeness, soundness and termination properties of the interpreter. As in Prolog-III, if we add more constraints to the system, then it will be necessary to incorporate the constraint propagation algorithm.

In the last chapter, we will look at the modifications needed in the Warren Abstract Machine for Prolog to incorporate the constraint solver.

6.0 Conclusions.

In this chapter, we discuss the modifications needed to be made in the Warren abstract machine (WAM in short) [Warr 83] to handle constraints followed by our conclusions. We assume that the reader is familiar with the WAM and proceed on that basis.

Warren first proposed an abstract machine for Prolog and specified the abstract instructions into which Prolog would be compiled. In other words, he considered the run time behavior of Prolog and specified the data structures and operations on these structures that would support the language. We will outline the modifications needed in the data structures and new types of instructions needed for supporting the constraint solver. We do not, however, presume that these will support any type of constraint that may be added. Internally Prolog terms are classified into variables, atoms, numbers and skeleton literals. (For terminology, please refer to Warren [Warr 83] and [Warr 77].) We suggest the addition of more types, one new type for each constraint type handled by the language. We classify each variable that participates in a constraint to have a new type introduced for that type of constraint. Likewise, we need to classify what Warren calls skeleton literals into different types. Internally, the variables are represented by some locations on a stack and each variable has associated with it a type and a value. We associate another location with the variable

that will point to the list of constraints in which it participates. Initially it would be set to nil. We need an additional stack on which the simplified forms of expressions could be represented and the constraints could be stored. These are the changes we propose to the data structures of WAM. The existing unification instructions would be modified to reflect the above changes and the constrained unification algorithm we gave in Chapter 4. Specifically, the compatibility of terms will be checked at run time and appropriate action can be taken in terms of algorithm 4.1. We need routines that would transform each equation as represented by a skeleton literal into a form needed by the equation / inequality solver and routines to simplify expressions and store the simplified form on the new stack and change the value field of a skeleton literal to point to the simplified expressions. When unconstrained terms of the head of a clause are unified, we check if any new constraints have been added. If not, we proceed in the usual way. Otherwise, the constraint propagator is invoked. The constraint propagator will carry out the two procedures outlined in Chapter 3. The above method would use the information about the type of constraints gathered at compile time to check the compatibility of terms at run time and fail quickly whenever necessary.

Conclusions and Future Research Areas

There are a number of limitations to computing in the Herbrand Universe. Constraint satisfaction, a model of computation, extends the domain of computation naturally within the framework of logical deduction. It adds expressive power to the language Prolog and elevates numeric computations to almost the same status as symbolic computation. The constraint solver that we constructed is general in scope. It is possible to add more constraints without changing the control structure. We need

only add special purpose routines to handle the simplification of the types of constraints that we may add and also to check the consistency of the system.

There are some theoretical questions and practical considerations that should be addressed here. We will look at the practical considerations first. Let us consider the factorial example of Chapter 1 again. If the second argument is instantiated and not the first, we cannot compute the first argument in our system since it involves solving non-linear equations. Moreover, the factorial rule recurses on the first argument and it is obvious that the terminating condition is not well-defined in terms of the second argument. Furthermore, what happens when we frame the factorial rule as given below ?

```
((Factorial 0 1))
((Factorial %n (* %n %int))
     (Factorial (- %n 1) %int))
```

We show below the execution of a query.

```
:-(prove ((Factorial 4 %a)) )
(Factorial 4 (* 4 (* 3 (* 2 (* 1 1)))))
true
SUBJECT to the following constraints
nil
time taken
1.180
```

It is clear that the interpreter has treated (* %n %int) as a binding and since there was no equation solving involved in this case it produced an answer which, although correct, is not what we expected. When do we treat x = y as an equation and when do we treat this as a binding? The answer to this question will not affect the soundness of the procedure but it is nevertheless an interesting question although we will not address this question in this thesis. Another practical consideration would be the development of efficient techniques for compiling constraints. We

outlined a method above for the modification of the Warren Abstract Machine for handling constraints. But compiling constraints is a research topic by itself.

On the theoretical side, how can we build a constraints solver when we have higher order constraints? For example, if we allow multiplication also as a full-fledged constraint, we get an undecidable theory. Some systems employ coroutining techniques or delay ("freeze") the processing of higher order constraints in the hope that some variables may get instantiated. Are there some special cases/conditions which would allow us to check the consistency of the constraints and allow us to maintain soundness? These are some research topics worth pursuing.

Bibliography

Backus,J. "Can a program be liberated from the Von Neumann style? A functional style and algebra of programs", *CACM* 21,1978.

Bazaraa,M.S. and Jarvis,J.J. *Linear Programming and network flows*, New York: John Wiley & Sons, 1977.

Boolos,G.S. and Jeffrey,R.G. *Computability and Logic*, London: Cambridge University Press 1985.

Bundy,A. *The Computer Modelling of Mathematical Reasoning*, New York: Academic Press 1983.

Chang,C.L. and Lee,R. *Symbolic Logic and Mathematical theorem proving*, New York: Academic Press 1973

Charniak,E. and Mcdermott,D. *Artificial Intelligence*, Reading, Mass: Addison-Wesley 1984.

Colmerauer,A. "Opening the Prolog-III Universe", *BYTE*, August, 1987.

Davis, Ernest. "Constraint Propagation with real valued quantities", *TR-189*, New York University, 1986.

De Groot,D. and Lindstrom,G. (Eds). *Logic Programming- Functions, Relations and Equations*, Englewood cliffs, NJ: Prentice Hall, 1986.

Dershowitz,N. "Computing with Rewrite Systems", *Information and Control*, 65, 1985.

van Emden,M.H. and Kowalski,R.A. "The Semantics of Predicate Logic as a Programming Language", *JACM* 32,4.

Floyd,R. "The paradigm of programming", *CACM*, August 1979.

Hoare,C.A.R. "The Emperor's Old Clothes", *CACM*, February 1981.

Jaffar,J. and Lassez,J.L. "Constraint Logic Programming", *Proceedings of the Conference on Principles of Programming Languages*, Munich 1987.

Lassez,C. "Constraint Logic Programming", *BYTE*, August, 1987.

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

Huet, G.P. "Undecidability of Unification in Third Order Logic", *Information and Control*, Vol.22, No.3, 1973.

Plotkin,G.D. "Building-in Equational Theories", *Machine Intelligence*, Vol.7, 1972.

Press, William. *Numerical Recipes,The Art of Scientific Computing*, London: Cambridge University Press, 1986.

Raulefs et al. "A Short Survey on the State of the Art in Matching and Unification problems", *SIGSAM Bulletin*, Vol.13, 1979.

Reddy,U.S. "Narrowing as the operational semantics of Functional Languages", *IEEE Symposium on Logic Programming*, 1985.

Robinson, J.A. "Computational Logic: The Unification Computation", *Machine Intelligence*, Vol.6, 1971.

Robinson, J.A. and Sibert, E.E. "LOGLISP: Motivation, Design and Implementation", *Logic Programming*, (eds. K.L. Clark and S.A.Tarnlund).

Robinson, J.A. "A machine oriented logic based on the resolution principle.", *JACM*, 12,1,1965.

Rogers,Jr. H. *Theory of Recursive Functions and Effective Computability*, New York: McGraw-Hill, 1967.

Rosen, B.K. "Tree manipulation systems and Church-Rosser theorems" *JACM*, 20, 1973.

Siekmann, J. and Szabo, P. "Universal Unification and a Classification of Equational Theories", *Sixth Conference on Automated Deduction*, LNCS 138, 1979.

Siekmann, J. and Szabo, P. "Universal Unification and Regular Equational ACFM Theories", *The Seventh International Joint Conference of Artificial Intelligence*, 1981.

Slagle,J.R. "ATP for theories with simplifiers, commutativity and associativity", *JACM*, 21, 1974

Slagle,J.R. "ATP with built-in theories including equality,partial ordering and sets", *JACM*, 19, 1972

Steele,G.L. "The definition and Implementation of a computer programming Language', *AI-TR-595*, MIT, August 1980.

Stickel,M.E. 1 "Unification algorithm for associative commutative functions", *JACM*, July 1981.

Stickel,M.E. 2 "Complete sets of reductions for some equational theories", *JACM*, April 1981.

Warren,D.H.D. "An Abstract Prolog Instruction Set", *TR-309, SRI Project 4776*, Menlo Park, CA: Stanford Research Institute 1983.

Warren, D.H.D. "Implementing Prolog: Compiling Predicate Logic Programs, Vol-1", *DAI Research Report no.39*, 1977.

Winston,P.H. *Artificial Intelligence*, Reading, Mass: Addison-Wesley 1984.

The vita has been removed from the scanned document