CONSTRAINTS, A MODEL OF COMPUTATION

by

Suryanarayana M Mantha

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:

Dr. J.W. Roach, Chairman

Dr. J.A.N. Lee

Dr. P. Bixler

July 1987

Blacksburg, Virginia

CONSTRAINTS, A MODEL OF COMPUTATION

by

Suryanarayana M. Mantha
Committee Chairman: Dr. John Roach
Department of Computer Science

(ABSTRACT)

In this thesis constraint solving/satisfaction is presented as a model of computation. Advantages of using constraints as a paradigm of programming are presented. A semantic schema for constraint based computations is given, following a brief survey of the more important systems based on constraints. These systems range from particular algorithms to problem solvers to constraint based general purpose programming languages. Finally, constraint satisfaction is applied to logic programming and theorem proving. It is shown that incorporating constraint solving in definite clause programs enhances their expressive power. Also, an alternative semantics - based on constraint satisfaction - is given for theorem proving.

Acknowledgements

I would like to thank my advisor Dr. John Roach for his help and guidance. Numerous discussions and arguments with S. Renganathan helped me understand the concepts of logic programming. I will never forget the long evenings we (Renga and I) spent debugging the PROLOG compiler being developed at Tech.

I would like to thank my housemates and other friends. They were a constant source of help and encouragement. Finally, I must thank my parents for the unconditional love they have given all these years.

Contents

List of Figures

Chapter 1

Introduction

In the last two decades, a lot of attention has been focussed on what is called the *software crisis*. It is widely agreed that the development of software packages is a difficult and thankless job. The twin problems of programmer productivity and software reliability have refused to admit of any easy solutions.

Two rather different approaches have been taken to remedy this unfortunate situation. The first of these is software engineering which sets guidelines for structured and/or disciplined programming in the hope that programs written following these guidelines will have the desired properties of

1. correctness,

2. clarity and ease of understanding,

3. other features such as modularity, portability etc.

The second approach to tackling the software crisis is somewhat more drastic. It seeks alternative models of computation and paradigms of programming. These efforts are directed towards identifying paradigms that naturally allow the expression of problems in an intuitively appealing as well as mathematically precise fashion.

The designer of a programming language has to keep two equally important issues in mind. The first of these is: it should be convenient for the programmer to express problems in a programming language. The ideal – which remains as unrealizable today as ever? – is: the user should be able to program in a very high level language such as English or Swahili! It should also be possible to efficiently execute programs written in such a language. This demands that the language designer take the features of the machine into consideration. As an unfortunate consequence, often questionable and sometimes inexcusable design decisions of the computer architect are reflected in the design of the language.

These two issues are of a rather conflicting nature. Is it possible to reduce the undesirable effects of one upon the other? Is there a model of computation that satisfactorily achieves the two goals, or at the very least, reduces the interplay

between the two. We believe that there is indeed such a model and that is the relational (constraint) model of computation.

Most problems can be naturally expressed as a collection of constraints that certain objects must satisfy. Mathematically expressed, a constraint satisfaction problem is:

1. V is a set of variables $(v_1 \ldots v_n)$,
2. Each variable v_i can take values from a domain D_i,
3. $C_1 \ldots C_m$ is a collection of constraints, constraining the values of $(v_1 \ldots v_n)$.

The problem is to find values of $v_1 \ldots v_n$ that satisfy $C_1 \ldots C_m$ simultaneously. Of course, certain classes of problems – engineering design and optimization problems among others – are best expressed in the above fashion.

In this report we study the semantic aspects of systems based on constraints and present a general semantic schema for such systems.

1.1 Why Constraints?

In recent years, constraint satisfaction techniques have received a lot of attention. Many researchers have actively worked in different areas related to constraints. Some of these areas are:

1. application of the constraint satisfaction algorithm to problems in AI. Waltz's algorithm[WALZ72], related to the labeling of edges and surfaces in three dimensional objects, is the most well known among these efforts,
2. using constraint satisfaction/manipulation methods for general problem solving,
3. Theory based heuristics for different factors that determine the efficiency of constraint satisfaction techniques. Some factors that immediately come to mind are
   (a) problem representation methods,
   (b) search order.
4. study of constraints as a paradigm of programming. In fact, the first three culminate in an effort to extend *constraints* to a general paradigm for programming.

What is it about constraints that makes them so attractive as a means of problem representation and a model of computation? Steele[STEL80] has given very persuasive arguments in favor of constraints as a paradigm of programming. He

claims that constraints form a natural and human-oriented paradigm of computation. He associates - and quite correctly at that- two images with constraints.

The first image suggests that a constraint is a *declarative statement of relationship* among symbolically denoted quantities. The constraint

$$\text{abs}(a) < \text{abs}(b)$$

is a statement of a certain *relationship* between the two objects a and b. Similarly, the equation

$$x + y = z$$

states the constraint that the value of the sum of x and y be equal to the value of z. An interesting feature of the above constraint is the following. It can be viewed in more than one way. It can be looked upon as stating the constraint that the value of x be equal to the difference between the values of z and y.

The second image is that these declarative statements also have the capability of enforcing the relationships they state. In other words, they also have a procedural interpretation. A constraint can be looked upon as an object or a device that enforces the stated relationship among the variables. In the above example, the values of x can be calculated when the values of x and y get known. Alternatively, an error message can be sent to the user if the values of x, y and z (obtained from some other constraint) do not satisfy the current constraint.

A problem can therefore be viewed as a collection of constraints among a set of variables. The problem is said to be solved when the variables are given values that are consistent with all the constraints simultaneously.

One of the most attractive features of this paradigm is its generality. It gives us an intuitively clear and appealing notion of computation. It is not restricted to any particular domain of problems, although there are certain classes of problems (engineering design problems for instance) that are undoubtedly better suited for this approach.

Constraints define *relations* among objects in a domain. The advantage of a relational semantics for a programming language is that a static meaning can be assigned to the program independent of any computational model. We do not have to appeal to the notion of a simple and completely unambiguous machine and try to interpret the program in terms of the basic operations of such a machine. The norm in the comparative study of programming languages has been to compare the performance of different programming languages on a fixed model of computation. In most cases, this model is that of a Turing machine. A relational semantics would allow different implementations to be judged uniformly.

It would also bring about a natural and desirable division in the task of programming a problem. The first subtask would devote itself to stating relationships between objects that must be satisfied in order for the problem to be considered solved. The second part would deal with efficient ways of manipulating these con-

straints or relationships, so that variables get valid instantiations. These instantiations correspond to correct answer computations. The job of deciding the correctness of the program would be assigned to the first part. Thus, a correctness proof would not have to worry about the idiosyncracies of the machine the problem is to be programmed on.

Another important feature of constraints is their locality. A constraint operates on the values of its known variables in order to deduce values for the other variables. Each constraint denotes an independent logical statement. The only way it interacts with other constraints is through shared variables. This implies that constraints can be solved in parallel on different processors. However, global consistency must be maintained.

As we observed earlier, constraints can be viewed in more than one way. They are adirectional in nature. This makes the control structure of a program more flexible than in ordinary languages. Values of variables are computed and propagated as and when variables on which they depend get instantiated. The data flow model is a special case of the constraint model in which the direction of the flow of data is predetermined. Adirectionality implies however, that the capability of viewing a constraint in more than one way must be present. Such a capability would require a sophisticated architecture or - in the worst case - explicit representation(as distinct constraints) of the various interpretations of a single constraint.

Davis[DAV87] summarizes the salient features of a constraint satisfaction system.

1. Constraint satisfaction/propagation consists of a simple control structure applied to a simple updating module. Systems based on it are therefore easy to code, analyze and extend.

2. It is easy to debug, since interrupting the process in the middle gives useful information already deduced.

3. It is amenable to parallelism, since updating can be performed all over the network simultaneously.

4. It is well suited to incremental systems. Constraints may be added incrementally, updating the values of the variables and propagating the effects.

1.2 Semantics of Languages

The semantics of a language provide a meaning to its expressions, statements, and programs. The two important considerations while defining a language are completeness and clarity. These two objectives are rather difficult to achieve together. There are three standard approaches to specifying the semantics of a programming language.

1. *The operational approach*: The meaning of each construct in the language is given by describing the effect of its execution on a *simple* and totally *unambiguous* machine.

2. *The denotational approach*: Each construct in the language is defined in terms of mathematical entities that specify their meaning.

3. *The axiomatic approach*: The meaning of each construct is specified in terms of formal statements (usually sentences in logic) that describe the effect of its execution.

The approach we take is similar to the denotational approach. It is very difficult to define - precisely and completely - the semantics of a programming language. Even in the case of constraints, it is not possible to give a single semantic interpretation that specifies all classes of constraints and computations based on them. The generality of constraints, which is an advantage in some respects, precludes such a possibility. Constraints can be of very different kinds. Such different entities as transcendental equations and sentences in first order logic can be viewed as constraints.

It is, however, possible to identify certain issues germane to most constraint based systems.

1. Is there a mathematical function that captures the constraint satisfaction process?

2. In what order are the constraints applied? Are they to be applied in a wholesale or incremental fashion? In the former, processing is started only after all the constraints have been input. In incremental systems, query answering alternates with accepting new constraints.

3. Is the constraint satisfaction algorithm complete and sound? By complete we mean that the algorithm gives all the answers to a problem. By sound we mean that the system does not compute any incorrect answers. What are the classes of constraints for which it is complete?

4. How are the constraints represented? We observed earlier that one of the chief advantages of computing with constraints is, they allow problems to be modeled visually.

In this report we restrict our attention to the first issue. The above problems are far from trivial for most classes of constraints. The *control problem*, i.e. the order in which the constraints are to be applied is probably the most difficult of them all. At this time, we do not have completely general and satisfactory solutions to the issues related to constraint based systems. It is our hypothesis, that most of these questions will have to be answered in the context of a particular application.

A general approach is however, helpful in that, it will give a better understanding of the features common to all constraint based systems and help us realize the goal of building a general purpose language based on this model.

1.3 Organization

The report is organized as follows.

Chapter 2 takes a closer look at some of the more important systems based on constraints. Chapter 3 presents a general semantic schema for constraint based computations. In Chapter 4, we look at an application in logic and show that logic programming is one form of constraint based programming.

Chapter 2

A survey of Constraint based Systems

In this chapter we shall look more closely at some of the more important work done on constraint based systems in the last two decades. This study is reported in chronological order. The reader will notice a shift in emphasis with time. In the earlier years, constraint satisfaction was looked upon as yet another algorithm that was well suited to problems in Artificial Intelligence. Later, it was used as a general problem solving technique. Only recently has the consensus grown that constraints form a natural paradigm of programming. Efforts are underway - even as this report is being written - to develop full-fledged general purpose programming languages based on constraints.

2.1 Sketchpad

The Sketchpad system developed by Sutherland[SUTH63] in 1963 was the first to be built incorporating the principle of constraints. It provided a graphic display output, a user interface and automatic satisfaction of constraints. All the constraints were equalities among linear relationships of variables. In Sketchpad, constraints were primarily used to describe geometric objects.

2.2 Waltz's Algorithm

Waltz[WALZ72] used the constraint satisfaction technique to limit the search space in the problem of assigning labels to parts of visual scenes represented as line drawings. The line drawing formed a constraint network and the goal was to assign a labeling to each junction. Waltz used local propagation and found that for many problem instances, it would converge to a unique solution or one with very few alternatives to be resolved by a global search. The efficiency of the algorithm

stimulated a lot of interest in this area. Waltz however, used a finite number of variables with finite domains. Moreover, all his constraints were binary, and his representation simultanecusly represented all valid states of the system.

2.3 Ref-Arf

Ref-Arf developed by Fikes[FIKE70] was intended to be a general problem solver. Its intellectual predecessor was GPS. Fikes used constraint satisfaction techniques to solve problems input to the system in the form of non-deterministic procedures. The system has three well-defined and distinct parts.

The first is the input language (called Ref) in which the user inputs the problem. The problem is coded as a nondeterministic procedure in Ref. Ref is a simple programming language with ordinary arithmetic and the following interesting features.

1. Identifiers stand for vectors (similar to fixed length arrays). Individual fields of the vector can be accessed and modified.

2. Each identifier has a scalar value associated with it. This is denoted by <id> where id is the identifier.

3. Ref has a special operator called "Select" that takes one operand. This operand is a range of values. Fikes used only discrete and finite ranges. "Select" nondeterministically selects a value from the given range.

4. The "set" command is similar to an assignment statement. The first operand is a variable and the second is a value. The second operand could be a select statement, in which case, the value is non-deterministic.

5. Conditional statements of the form

   if condition then code

   are allowed.

6. Goto statements are allowed.

7. Ref cannot handle infinite sets and does not have the facility of parameterized procedures.

Once a problem is entered as a procedure written in Ref, an interpreter takes over. The purpose of the interpreter is to translate the procedure into a format suitable for the application of constraint satisfaction/manipulation techniques. The third module tries to solve the problem using constraint satisfaction and heuristic search techniques. Fikes' approach is a direct descendent of Floyd's[FLOY67] work

on translating a procedure written in a nondeterministic programming language, into a procedure in the base programming language which finds acceptable values for the select function calls using a backtracking algorithm. Fikes claims that these procedures can be translated into procedures which allow the application of algorithms stronger than backtracking.

The interpreter for Ref(called Arf) is not always able to translate a problem stated in Ref into a single constraint satisfaction problem represented as a context structure (see fig. 1). This happens in the interpretation of an "if" statement, a computed goto or even an assignment statement where the target of the assignment depends on the value of some variable. The interpreter then performs what is called *case analysis*, i.e., it constructs more than one context structure corresponding to the different paths the execution could take.

Arf manipulates constraints by first simplifying them into their normal form. For example, $1 = 5$ would be replaced by FALSE and $\neg\neg(a = b)$ would be replaced by $(a = b)$. Once the constraints have been reduced to their standard form, the constraint manipulation techniques are applied.

Ref-Arf has reasonably sophisticated constraint manipulation techniques. The principal techniques are

1. elimination of variables from constraints,
2. elimination of elements from variable ranges,
3. deduction of inconsistencies among constraints.

These are carried out at each stage of the backtracking search. An interesting feature is that constraint manipulation is combined with value assignment in order to reduce the search space of the problem. Arf has a variable ordering strategy(which is applied at each value assignment step) that determines which variable ought to be instantiated next. Some intuitively justifiable criteria for determining the ordering are

1. the domain size of the variable,
2. the number of constraints in which the variable appears,
3. the constraint "tightness" of the principal operators of the constraints in which the variable occurs. For instance, an equality constraint is considered to be a very "tight" constraint.

2.4 Thinglab

Thinglab[BORN81] is a constraint satisfaction system built on top of Smalltalk 76. Although it is meant to be a graphic simulation laboratory, it is very much an object

```
Variables
    V(1) = { value1... }

    V(2) = { value1... }
      .
      .
    V(n) = { value1... }

Constraints
    C1
      .
      .
    Cn
end.
```

Figure 2.1: A Context Structure

oriented language primarily concerned with the representation and satisfaction of constraints.

In Thinglab, nonprimitive objects are constructed hierarchically from *parts* which are themselves other objects. Constraints provide a natural way to express the relation among parts and subparts. An interesting aspect of Thinglab is the integration of the use of constraints with inheritance hierarchies in the definition of new objects. Some examples of constraints in Thinglab are

1. that a triangle be twice as big as another,

2. that a node in a circuit obey Kirchoff's laws.

Constraints are represented as rules and as a set of methods that can be invoked to satisfy the contraint. The rule is a procedural test for checking whether the current state of the system satisfies the constraint or not. The methods are the different ways of satisfying the constraint.

Attached to each method is what is called a *message plan.* A message plan is an abstraction of the Smalltalk notion of sending a message. A message plan does not stand for a particular act of sending a message; rather, it is a template for any number of messages that might be sent. Each message plan specifies how to invoke the corresponding method and describes its effects. When the constraint satisfier chooses one of the methods to be invoked at runtime, the message plan that represents that method is asked to generate the code that will send the appropriate message to activate the method.

The constraints are specified by the user. It is up to the system to satisfy them. The system takes various factors into consideration.

1. The directionality of constraints.

2. Compromise between local optimization and global consistency.

3. Handling of circular and/or interfering constraints.

A notable feature of Thinglab is that it is the first system that talks about compiling constraints into plans in the base language. Further, the presence of a class-instance mechanism and inheritance allows information common to several objects to be abstracted.

2.5 Constraints

Guy Steele's[STEL80] 1980 MIT Ph.D. dissertation reports the most recent efforts in the development of a general programming language based on constraints. Steele makes a strong case for constraints as a natural paradigm for problem solving. An interesting point made by Steele is that constraints provide a graphic visual imagery

```
Class triangle
  Superclasses
     GeometricObject
Part description
  side1 : a line
  side2 : a line
  base  : a line

Constraints
   length(base) <= length(side1) + length(side2)
   length(side1)<= length(side2) + length(base)
   length(side2)<= length(side1) + length(base)
      .
      .

end.
```

Figure 2.2: Example of a constraint in Thinglab

that helps human thought processes. They are particularly well suited to expressing programs for certain applications, in particular, relationships arising in computer aided design.

Steele's system is configured as a network where each node is a constraint and arcs joining two nodes represent the variables shared by the two constraints. Each node behaves like a process running in parallel with the rest of the network. There is a library of primitive constraints or devices from which are constructed complex constraint networks. Computation is triggered by giving a value to one of the lines(or variables).

Constraints records the justifications for any computation. This helps in resolving any global inconsistencies which could arise at any intermediate stage in the computation. The language provides a form of dependency directed backtracking in which, the culprit (the subcomputation responsible for the inconsistency) is determined and its effects are undone. Dependencies are recorded by attaching a special data structure called the *repository* to each cell which contains all the information about a variable.The repository has the following fields.

1. A *supplier* which is the cell that first provided the value for the repository.

2. A *rule* which is the name of the rule(or constraint) used to compute the value. The rule component is null if the repository has no value, or if the supplier is a primitive(i.e. a constant).

3. A *mark* which is normally null but is available to serve as mark bit for the various graph algorithms that the system uses.

An important job of the dependency recording unit is to keep the dependency information from becoming circular. The second purpose of recording dependencies is to provide explanations for the computation carried out by the constraint satisfier.

It is generally agreed that a constraint system ought to have the property that it works if one gives it only a little information, and works better if one gives it more information. In other words, a constraint-based system should so far as possible be monotonic. There are certain situations however, when one would like to continue the computation in a *reasonable* direction, i.e., have the capability for *default* behavior. Constraints allows a limited form of nonmonoticity by permitting the use of assumptions. Another use of assumptions is in case analysis. If it is known that a variable must take values from a specific set, then one element of that set can be arbitrarily assumed to be the value of the variable; the value is discarded if it leads to a contradiction. *Constraints* has two special constructs *assume* and *oneof*. *Assume* assumes some fact to be true whereas *oneof* is the equivalent of Ref-Arf's *select* construct.

In the case of a contradiction, the assumptions and any computations based on them are retracted. This information is also stored for future use in what are called *nogood sets*. Nogood sets prevent the computation from repeating its mistakes.

Steele's language is built on top of LISP and uses the various facilities- garbage collection for instance- provided by it. It has several other features including the facility for the definition and use of macro-constraints. It is outside the scope of this report to give a more detailed description of the language. *Constraints* represents one of the best efforts in designing a general purpose language based on the paradigm of constraints.

2.6 Constraint Propagation with Interval Labels

Ernest Davis[DAV85][DAV87] discusses various aspects of a special kind of constraint propagation which he calls constraint propagation with interval labels. In this, a constraint satisfaction problem is configured as a network and each node is labelled with a set of possible values. In *assimilation*, the constraints are used to restrict these sets. In *query answering*, the term to be evaluated is expressed as a function of some of the nodes, and the system must calculate the range of values of the term, given the label sets of the component nodes. Davis defines an operator called *Refine* that takes a node and a constraint and produces a set of values for that node which are consistent with that constraint and all other labels. This operator is applied over and over again till refinement produces no more changes. The network is then said to have reached a state of quiescence.

Davis deals with different aspects of such systems. Some of them are

1. the representation of constraints in terms of the nodes,
2. the order in which constraints are to be run,
3. the kind of implicit constraints to be used,
4. the conditions for ensuring that refinement reaches quiescence
5. and the design of the assimilator and the query answerer and their completeness issues.

Davis also discusses the different classes of constraints and gives complexity expressions for solving a network of such constraints. He proves the completeness of the Waltz's algorithm for propagation around constraints of bounded differences and interval label inference on unit coefficient linear inequalities. The interested reader is referred to [DAV85] and [DAV87].

2.7 Summary

In this chapter, we looked at some of the more important work done on constraint based systems. It is, by no means, an exhaustive survey. Nadel[NAD86A][NAD86B]

has reported work on theory based heuristics for determining various factors such as search order, problem representation format etc. Constraint satisfaction systems have been used in other specialized domains such as planning. It is clear, that constraint satisfaction offers a rich paradigm of computation.

Chapter 3

A Semantic Schema for Computations using Constraints

3.1 Motivation

The primary thrust in this report is to find out whether constraint based computations operate on any well understood mathematical entities. There are several reasons for wanting to establish such a relationship. Constraint satisfaction methods are intuitively appealing and strikingly similar to the way we humans model problems. They are sufficiently general in nature and have a very simple and flexible control structure. A formal interpretation of these methods would afford a better understanding and also enable us to characterize them precisely.

We want to determine if the constraint satisfaction process can be captured by a mathematical operator operating on some domain. If such a operator does exist, then its properties would give us useful information about the constraint satisfaction process itself. Note, that this approach is similar to the denotational approach to specifying the semantics of a language.

We observed earlier that the term *constraints* represents a large class of very different entities. Davis[87] gives a list – by no means exhaustive – of the different kinds of constraints. Some of them – in increasing order of complexity – are

1. unary predicates,
2. order languages (consisting only of order relationships),
3. linear equations and inequalities with unit coefficients,
4. linear equations and inequalities with arbitrary coefficients,
5. algebraic equations,
6. transcendental equations.

It is difficult - perhaps impossible - to give a single function that completely specifies all classes of constraints and computations based on them. Our goal is to show that if a transformation T (not necessarily unique) can be associated with each class of constraints, then these T's satisfy certain basic properties.

It is a common practice in the study of programming languages to associate transformations with recursively defined procedures. These transformations define the *meaning* or *denotation* of such procedures[GLAS84]. It is necessary to impose some structure on the sets these transformations operate on. The minimum set of denotable values (D) in any programming language is

$$D = \text{basic constants} + [D \longrightarrow D] \tag{3.1}$$

where $[D \longrightarrow D]$ is the set of functions from D to D. If we use unrestricted sets and functions, then there is no set that satisfies the above definition. It is easy to see why this is so. For any set S, the set of functions defined on S will always contain many more elements than S itself. For example, consider the set of functions that map elements of S into $\{0,1\}$. For each element of S, we have two choices for the mapping and thus we can define 2^n distinct functions if n is the cardinality of S. For any reasonable n, 2^n is much bigger than n. The problem can be solved by imposing a structure, a partial ordering on the sets and constraining functions to preserve the structure. The transformation T associated with a procedure maps sets of input-output tuples into other sets of input-output tuples. Our interest is chiefly in transformations that preserve the structure (partial order) imposed on these sets. When the associated transformation is monotonic, the meaning of the procedure P is defined to be

$$\text{intersection}\{I : T(I) \leq I\} \tag{3.2}$$

where I is a set of input-output tuples. It can be shown[LOYD84] that (3.2) is equal to

$$\text{intersection}\{I : T(I) = I\} \tag{3.3}$$

(3.3) gives the least fixpoint of the transformation T. Intuitively, The set (3.3) completely describes the input-output behavior of P and is all we need to know about P in order to specify it completely. (3.3) consists of all the valid input-output tuples for the procedure P. At this point we need some definitions and simple concepts of partially ordered sets.

3.2 Definitions

3.2.1 Definition 1

Let S be a set of elements $a, b, c \ldots$ where a relation of equality $x = y$ is already defined; then a relation O of partial order over S is any dyadic (binary) relation over S which is

1. *reflexive*: for every a in S $a \mathcal{O} a$;

2. *antisymmetric*: if $a \mathcal{O} b$ and $b \mathcal{O} a$, then $a = b$;

3. *transitive*: if $a \mathcal{O} b$ and $b \mathcal{O} c$, then $a \mathcal{O} c$.

The symbol $\mathcal{O}$ in $a \mathcal{O} b$ will generally be replaced by "$\leq$"; if $a \leq b$ we say that a is less than or equal to b.

3.2.2 Definition 2

A set S over which a relation $\mathcal{O}$ of partial order is defined is called a partially ordered set.

3.2.3 Theorem 1

If a set S is partially ordered by a relation $\mathcal{O}$, then S is partially ordered by the converse relation $\mathcal{O}'$ where $x \mathcal{O}' y$ iff $y \mathcal{O} x$.

3.2.4 Definition 3

g is a lowerbound for a set T iff $g \leq t$ for all t in T. If g is such that $b \leq g$ for all lowerbounds b of T then g is called the greatest lower bound of the set T.

An analogous definition exists for the least upper bound for a set T.

3.2.5 Definition 4

A lattice is a partially ordered set A with the ordering relation denoted by $\leq$ such that for any two elements a and b in A , there is a least upper bound (join) $a \cup b$ and a greatest lower bound (meet) $a \vee b$. The lattice $(A, \leq)$ is said to be *complete* if every subset B of A has a least upper bound and a greatest lower bound in A. Such a lattice has two special elements.

1. $0 = \vee A$.

2. $1 = \cup A$.

The last definition is taken from [TARS55] and the rest are from [DONN68]. We shall consider functions on A to A and relate them with the deductive techniques used in constraint satisfaction and theorem proving. A function f on a subset B of A to another subset C of A is said to be increasing if for any 2 elements s, t in B $s \leq t$ implies $f(s) \leq f(t)$. By a fixpoint of a function f we understand an element x of the domain of f such that $f(x) = x$.

3.3 Constraint Satisfaction

We observed earlier that the basic constraint satisfaction algorithm tries to find instantiations for variables that satisfy all the constraints simultaneously. The *refine* operator discussed in [DAV87] suggests that we can associate a transformation with constraint satisfaction that operates on a partially ordered set. *Refine* is defined as follows.

> Let C be a constraint on variables $x_1, \ldots, x_n$. Let S_i be the label (value) set for x_i. Then
>
> $$\text{REFINE}(C, x_j) = \{a_j \text{ elementof } S_j | \text{ there exists } (a_i \text{ elementof } S_i,\ i = 1, \ldots, n, i \# j)\ C(a_1, \ldots, a_j, \ldots, a_n)\}.$$

That is, refine(C, x_j) is the set of values for x_j which is consistent with all the labels S_i. A value a_j is in refine(C, x_j) if a_j is in S_j and it is part of some k-tuple $a_1, \ldots, a_n$ which satisfies C and all the S_i. This operator is sound deductively; if a tuple satisfies the constraint and the starting value sets, then it satisfies the refined value sets.

In the following section, we present two ways of constructing the partially ordered set and the associated transformation T. As we shall see, the least fixpoints of the transformations in the two systems, have direct correspondence with the assimilation and query answering systems of [DAV85].

Assimilation refers to forward inference on constraint networks, in which all the constraints are assimilated into the value sets of the variables. Value sets of variables are refined using local groups of constraints. These changes are propagated through the network of constraints. Computation reaches an end whenever no more refinement can be performed. In *query answering* the value of a term has to be calculated using the current value sets.

3.3.1 A lattice theoretic approach

> Let $V = \{v_1, \ldots, v_n\}$ be a set of variables.
> Let $D = \{D_1, \ldots, D_n\}$ be the domains from which each v_i can take values.
> Let $C = \{c_1, \ldots, c_m\}$ be a set of constraints in which one or more of the variables occur.

- Define the set U as follows.

$$U = \{(u_1, \ldots, u_n) | u_i \text{ subsetof } D_i \text{for} 1 \leq i \leq n\}.$$

Let the partial order "$\leq$" be the superset relationship. For u, v in U, $u \leq v$ iff $v_i \subseteq u_i$ for $1 \leq i \leq n$. The least upper bound of two elements u, v is the element $(u_1 \vee v_1, \ldots, u_n \vee v_n)$, where the operator $\vee$ represents set intersection. The greatest lower bound of two elements u, v in U is the element $(u_1 \cup v_1, \ldots, u_n \cup v_n)$, where $\cup$ represents set union.

Theorem 1: The system $\{U, \leq\}$ forms a complete lattice. Further, it forms a Boolean algebra with the top element equal to (nil,...,nil) and the bottom element equal to $(D_1, \ldots D_n)$

We can associate an operator T with the constraint satisfaction/propagation algorithm. T maps elements of U to elements of U. It closely mimics the operator *refine* defined earlier. Every application of T takes us upward in the lattice. T is clearly monotonic. After some number of applications of T, no more *refinement* takes place and a state of *quiescence* is reached. This corresponds to a fixpoint of the operator T.

Theorem: T has a least fixpoint. Suppose not. Let F_1 and F_2 be two minimal fixpoints such that $F_1\neg \leq F_2$ and $F_2\neg \leq F_1$. Consider $F = \text{glb}(F_1, F_2)$. $F = (F_{11} \cup F_{21}, \ldots F_{1n} \cup F_{2n})$. F itself is a fixpoint of T. T cannot refine F since it cannot refine its constituents F_1 and F_2. But $F \leq F_1$ and $F \leq F_2$ which contradicts the minimality of F_1 and F_2.

Example

Let $V = \{x, y\}$. Let $D_x = \{2, 3, 4\}$. Let $D_y = \{1, 2, 3\}$.
$C_1 = \{x = y\}$.

The refined label sets are $Dx' = \{2, 3\}$ and $Dy' = \{2, 3\}$. Assimilation has reached quiescence and $(Dx'\ Dy')$ is the fixpoint of the corresponding T. The answer however, is a proper subset of $(Dx' \times Dy')$. Further processing needs to be done at the time of query answering. This could involve the addition of new (explicit or implicit) constraints to the system. In the present example, the constraint $x - y = 0$ may have to be added to get the correct answer.

- Define U in a slightly different fashion.

$$U = \{x | x \text{ subsetof } \times D_i 1 \leq i \leq n\}$$

where $\times D_i$ denotes the cross-product of $D_1, \ldots, D_n$. Each element of U is a set of n-tuples. Each n-tuple can be looked upon as an interpretation of the variables in the constraint satisfaction problem. Each element of U is, therefore, a set of such interpretations. We are interested in finding the maximal set of such interpretations that are consistent with all the constraints simultaneously.

The system $\{U \leq\}$ forms a complete lattice. The partial order is the superset relationship as before. $\{U \leq\}$ is also referred to as the powerset lattice. The operator T throws out an inconsistent interpretation at each step. It is monotonic and has a least fixpoint. The least fixpoint corresponds to the set of all consistent interpretations.

Note that the T here is more powerful than the one previously defined. The stage at which quiescence is reached corresponds to the completion of both *assimilation* and *query answering.*

Example

Let us consider a particular problem from the viewpoint of constraint satisfaction. The problem we shall look at is unification. Unification is a very important problem and a lot of work has been done on it. It is widely used in resolution based theorem proving systems, where equality between complementary literals is established using unification. The reader is referred to [LOYD84] for details.

We state the problem of unification in its general form. Let $s_1 = t_1, \ldots, s_n = t_n$ be a set of equations where $s_1, \ldots, s_n, t_1, \ldots, t_n$ are arbitrary terms. A term is defined by the following grammar.

term:== variable | functor(term*)

where variable and functor are from disjoint sets. Let $x_1, \ldots, x_n$ be the set of all distinct variables occurring in the equations. Let $D_1, \ldots, D_n$ be the domains of each of these variables. Note that D_i, D_j, $i \# j$ may not be disjoint and/or finite. The problem is to find instantiations of $x_1, \ldots, x_n$ that satisfy the set of equations simultaneously.

The basic unification algorithm is syntactic in nature and does not take into account, the properties of the functors. For example, +(4 2) would fail to unify with the term +(3 3). The only constraints that the algorithm handles are syntactic equality constraints.

It is possible to extend the unification algorithm to a general constraint satisfaction algorithm with the capability for handling other constraints such as set membership, order relationships etc. Capability for handling functors that compute relations and those that are used to create complex data structures can be incorporated into the algorithm. Sundararajan[SUND87] has developed an interpreter for Prolog which allows *constraints in the body of a clause* and also views unification as constraint satisfaction. The procedures for computing the relations that the constrained functors represent should, however, be decidable, in order to ensure the termination and correctness of

the unification algorithm. The reader is referred to [SUND87] for details. The least fixpoint of the transformation associated with such a system would then be the set of all consistent instantiations of the variables. This corresponds to the maximal set of unifiers for the system.

3.4 Summary

In this chapter, we saw that a general characterization could be given for constraint satisfaction problems. A partial order can be defined on the vector of values of the variables. The ordering relation - set inclusion - is very well behaved and makes it easy to establish the existence of various properties. The schema presented above can be fruitfully applied to specific classes of constraints; the properties of the operator T can help determine the behavior of the system for a particular class of constraints. In the next chapter, we shall study logic programming from the viewpoint of constraint satisfaction. We shall show that constraint solving can be incorporated into logic programs in more than one way. The unification algorithm can be replaced by a general constraint satisfaction technique as described in [SUND87]. Also, we can view theorem proving as constraints among atomic predicates and use constraint satisfaction techniques to derive contradictions.

Chapter 4

An Application in Logic

Constraint satisfaction techniques have traditionally been used in the context of numerical computations. [DAV87] describes most of the interesting classes of problems that can be solved using CS/CP algorithms. In this chapter, we shall apply the constraint schema from Chapter 3 to logic programming.

Today, one of the most interesting and active areas of research in computer science is computation using logic. The use of logic in computer science is not a recent phenomenon. It has always been used as a specification or declarative language. In the early 1960's, it was the basis of work in automated theorem proving and Artificial Intelligence. The most important result in this phase of automated deduction was the discovery of the resolution principle by Robinson. It was not until 1971-72, however, that the idea that sentences in logic could be used as computer programs took shape. Kowalski [KOWA74] and Colmerauer showed independently, that statements of a subset of first order logic had a procedural interpretation as well. A program clause $P \leftarrow P_1, \ldots, P_n$ can be considered to be a procedure definition, and a goal clause $\leftarrow G_1, \ldots, G_m$ can be looked upon as a series of procedure calls. The most important practical outcome of this research is the the development of the language Prolog and other similar languages. In the rest of this chapter, we shall assume the reader's familiarity with elementary logic.

4.1 Prolog and its Semantics

Prolog is the most popular logic programming language. A Prolog program consists of a collection of definite clauses. A definite clause(or Horn clause) is a statement of the form

$$P \leftarrow Q_1 \text{ and } Q_2 \text{ and} \ldots \text{and } Q_n (n >= 0).$$

in which $P, Q_1, \ldots, Q_n$ are atomic. An atom is defined by the following simple grammar.

atom :== pred(term*)
term :== variable | functor(term*).

Pred, functor, and variable here are disjoint sets of symbols representing predicates, functors and variables. The above clause represents the universally quantified formula

Q_1 and Q_2 and ...and Q_n => P.

$n = 0$ results in an unqualified assertion(or fact or axiom); $n > 0$ results in an inference rule. A Prolog program can be viewed as a set of such axioms and inference rules. Another view is to consider all the clauses to be axioms with resolution as the only inference rule.

Given a program of the form described above, one would like to determine the truth of a conjunction of atoms relative to it. There are two ways of doing this. The first is to construct a model for the program and determine whether the atom belongs to the model. This corresponds to the model-theoretic approach in which we try to establish *semantic entailment.*The second is to derive the atom as a theorem of a first order theory using the clauses in the program as a set of axioms and using some set of inference rules. This corresponds to the proof-theoretic approach where the notion of formal derivability is important.

The abstract interpreter for Prolog takes the latter approach. The only inference rule used is *resolution.* The proof system takes the negative approach,i.e., it is set up as a refutation system. The negation of the theorem to be proved is added as an axiom and the system tries to derive the empty clause, which implies that the negated theorem is inconsistent with the axioms.

As a consequence of Godel's completeness theorem for first order logic, the two approaches are equivalent. In fact, there exists a canonical model for definite clause programs, which is the least Herbrand model and is equal to the set of all logical consequences of the program. An equivalent fixpoint semantics can also be given for Prolog programs. The set of theorems is exactly equal to the least fixpoint of a monotonic and continuous operator over the complete lattice formed by $\{2^{Bp}$, set inclusion$\}$ where Bp is the Herbrand base associated with the program p. This operator captures the intuitive notion of implication. A thorough treatment of the above can be found in [VAND76], [APT82] and [LOYD84].

It is remarkable that the three approaches to describing the semantics of Prolog programs coincide. Such a neat semantics however, does not come without a price. The expressive power of Prolog programs leaves much to be desired. The use of definite clauses makes it very difficult to express negative information, i.e., the notion of "falsehood". We have to turn to some very non-intuitive interpretations. Some of them are the closed world hypothesis and a somewhat weaker notion of negation as finite failure.

All Prolog interpreters achieve negation as finite failure. Negation as failure states that if A cannot be proved from the progrm P then not(A) is a theorem.

This does not correspond with our intuitive notion of falsehood. We would like to believe that something is false if any belief to the contrary would lead to a contradiction(i.e. something being inferred as true and false at the same time). Failure to establish A implies that the truth or falsity of A is independent of P. Also, the interpreter with a depth first left to right control strategy might fail to terminate for very obvious and simple programs. Consider the following program.

$P \leftarrow Q$
$P \leftarrow \text{not}(Q)$
$Q \leftarrow P.$

Given the goal P the above program would fail to terminate. It is obvious that P follows from the program(at least in classical logic). In fact, the interpreter may go into a loop while trying to prove theorems as well as while trying to show that something is not a theorem.

There is another feature that limits the expressive power of logic programs in a crippling way. At the core of all logic programming languages is the unification algorithm. The unification algorithm was first given by Robinson in [ROB65] and most logic programming systems still use it with few or no changes. Unification determines the syntactic equality of a set of terms that belong to the Herbrand universe. By restricting itself to uninterpreted terms over the Herbrand universe, the algorithm fails to establish the equality between terms such as $+(3\ 3)$ and $+(4\ x)$. Some of the extensions proposed to the unification algorithm incorporate a set of axioms for establishing equality ([PLOT72]). They, however, deal with the simplest of arithmetic terms outside the system in an ad hoc fashion. We discussed this problem earlier, in the previous chapter.

Sundararajan ([SUND87]) has developed an interpreter for PROLOG that replaces unification by the solution (most general) to a network of constraints. The interpreted symbols he handles are $(+, <, >,$ disequality, and set membership). Terms containing uninterpreted symbols are related by syntactic equality. The Herbrand universe is replaced by a many sorted domain similar to the one described in the last chapter. The elements of the new Herbrand base (Bc) are obtained by taking the cross-product of the predicate symbols and elements of the many sorted domain. Consider $\{2^Bc, \text{set inclusion}\}$. It forms a complete lattice. Describe Tc as follows.

if I is an element of 2^Bc then
$Tc(I) = \{$ s element of Bc |
there is a clause
$P \leftarrow P_1$ and P_2 and $\ldots P_n$
and the constraint solver outputs
the set of solutions delta such that
$(P)\delta = s$ and

$(P_i)\delta$ belongs to I, $1 <= i <= n$.
}

An important issue is whether we can give a least model semantics that corresponds to the least fixpoint of Tc defined above. We do not have theorems to show that the above property holds. Jaffar and Lassez ([JAFF87]) have proposed a more general approach by allowing constraints in the body of clauses as well. [SUND87] also allows constraints in the body. In fact, he does not require the constraints to appear before the goals. According to Jaffar et al, prolog with constraints fits in well with their logic programming scheme.

A second approach is to consider arbitrary well formed formulas of first order logic to be constraints among atomic predicates. The constraint symbols are all the logical operators (and, or, not $\leq$). Instead of treating negation as finite failure, we have true logical negation.

It is well known that the largest subset of first order logic that has an initial model is the definite clause subset. We shall therefore require that the set of formulas in our program be consistent. Also, there are no least models for the system that we describe here. The least fixpoint of the associated transformation, however, gives the set of logical consequences of the set of formulas (or program).

4.2 A Constraint Satisfaction Approach

We shall treat formulas of first order logic as constraints and show that the least fixpoint of the transformation associated with the truth-functional calculus computes the set of logical consequences of a program. It will be generally agreed that the clauses in a Prolog program express constraints. In fact, all well formed formulas of any first order language express constraints between objects in an intended domain. We shall not restrict our constraints to be Horn clauses but shall require that the following preprocessing be done on the constraints.

1. $A <=> B$ is replaced by $A => B$ and $B => A$.
2. Negation signs are moved inward.
3. Variables are renamed if necessary.
4. Existential quantifiers are removed by replacing existentially quantified variables with unique constants or functions of universally quantified variables.
5. All remaining variables are understood to be universally quantified.

We would like to restate the problem as a constraint satisfaction problem. There is more than one way of doing this. Consider a set of first order constraints(with at least one predicate symbol and one ground term).

1. The variables of the CS problem are all the predicate symbols appearing in the constraints. For a finite set of constraints, there are a finite number of such variables. The domain of values of an n-ary predicate symbol is

$$\text{Dom}(P_n) = \{(x_1, x_2, \ldots, x_n) | (x_1 \ldots x_n) \text{ element of } \times (H)\}$$

where H is the Herbrand universe for the set of constraints and $\times(H)$ represents the cross-product of H taken n times. $\text{Dom}(P_n)$ will be infinite for constraints containing at least one function symbol. The denotation of a predicate P_n is the set of tuples $(x_1 \ldots x_n)$ such that $P_n(x_1 \ldots x_n)$ is logically implied by or is consistent with the constraints. The problem is to determine the denotation of each predicate. $\text{Den}(P_n)$ is a subset of $\text{Dom}(P_n)$. Therefore, the constraint satisfaction operates on the following complete partial order.

$$U = \{(Y_1 \ldots Y_m) | Y_i \text{ subsetof } \text{Dom}(P_i)\}$$

The program has m predicate symbols and $\text{Dom}(P_i)$ is the domain of P_i. The partial ordering is the superset relationship. Note that U is very similar to 2^{Bp} where Bp is the Herbrand base for the set of constraints P. In fact, it is the inverse of the lattice formed by $\{2^{Bp}, \text{set inclusion}\}$.

2. In classical logic there are two truth values. Given a set of premises, we would like to know if an atom is semantically entailed by the premises or not, i.e., given a set of true sentences what is the truth value of an atom A. Is it true or is it false? Or are the constraints not tight enough to impose a truth value upon the atom. This is the most intuitive and appealing approach to all logical arguments. The approach adopted in the semantic tableaux method is very similar.

   The variables in our problem then are all the ground atoms that can be constructed from the predicate symbols and the elements in the Herbrand Universe. All atoms take values from a single domain $D = \{T, F\}$. Define

$$U = \{(u_1, \ldots, u_n) | u_i \text{ subsetof } \{T, F\}\}.$$

   where n is generally infinite.

   U forms a complete lattice with the superset relation as the partial order. The *lub* of two elements is given by set intersection and the *glb* is given by set union.

   Define an operator $\mathcal{O}$ from U to U as follows

   (a) If (P_1 and P_2 and ... and P_n => P) is a ground instance of a constraint ($P_1 \ldots P$ are atoms) and F is not present in the domains of $P_1 \ldots P_n$ in I, then remove F from the domain of P in $\mathcal{O}(I)$.

(b) If *(not(P))* is a ground instance of a constraint then remove T from the domain of P in $\mathcal{O}(I)$.

(c) If (P or Q) is a ground instance of a constraint and T is not present in the domain of $P(Q)$, then remove F from the domain of $Q(P)$ in $O(I)$.

(d) If $((P => Q) => R)$ is a ground instance of a constraint, and T is absent from the domain of P in I, then remove F from the domain of R in $\mathcal{O}(I)$. Similarly if F is absent from the domains of both P and Q then remove F from the domain of R in $\mathcal{O}(I)$.

Proofs for following theorems are simple and along the lines of [LOYD84].

Theorem1: The operator $\mathcal{O}$ is monotonic.

Theorem2: All I's in U for which $\mathcal{O}(I) \leq I$ are *models* for the set of constraints.

Theorem3: The least fixpoint of $\mathcal{O}$ is equal to

$$\text{glb}\{I|\mathcal{O}(I) \leq I\}.$$

Theorem4: $\mathcal{O}$ is continuous. By continuous we mean if $I_1 \leq I_2 \leq \ldots \leq I_n$ then

$$\text{lub}(\mathcal{O}(I_i)) = \mathcal{O}(\text{lub}(I_i)) \quad 1 \leq i \leq n.$$

We sketch the proof for the case when (3.1) above applies to $\mathcal{O}$.

PROOF: Consider the ground constraint $(P_1 \text{and} \ldots \text{and} P_n) => P$.
F is absent from the domain of P in $\mathcal{O}(\text{lub}(I_i))$
iff F is absent from the domain of $P_1, \ldots, P_n$ in $\text{lub}(I_i) 1 \leq i \leq n$
iff F is absent from the domain of $P_1, \ldots, P_n$ in some $I_j 1 \leq j \leq n$
iff F is absent from the domain of P in $\mathcal{O}(I_j)$ for some j $1 \leq j \leq n$
iff F is absent from the domain of P in $\text{lub}(\mathcal{O}(I_i))$ $1 \leq i \leq n$.

Evans[EVAN87] has developed a theorem prover that is based on the above characterization. The important features of her system are

1. The axioms of the system are consistent sets of well-formed formulas first order logic.

2. The axioms are pre-processed as described earlier.

3. The theorem to be proved is negated, skolemized and added to the system.

4. Let C_1 and C_2 be two formulas. Let P_1 and P_2 be literals in C_1 and C_2 respectively. If P_1 and P_2 can be unified with θ as the most general unifier, then form two clauses $(C_1)\theta$ and $C_2(\theta)$ with the following changes. If T was absent from the value set of $P_1(P_2)$, then remove T from the value set of $(P_2)\theta((P_1)\theta)$ and a similar operation is carried out for F. If the literals that were unified are complementary, then the refinement is made in a slightly different way. If T were absent from the value set of $P_1(P_2)$, then remove F from the value set of $(P_2)\theta((P_1)\theta)$. A similar operation is carried out for F.

5. The above is interleaved with truth-functional simplification of the clauses. This corresponds to propagation of the constraints.

6. A contradiction is deduced when both T and F are knocked out of the value set of some literal.

For details about control strategies, case analysis and splitting rules and completeness, the reader is referred to [EVAN87]. Note that Evans views unification too as constraint satisfaction. Thus, there are networks of constraints at two different levels. The literals of the clauses form a network of constraints. Each unification is also set up as a network of constraints among the terms appearing in the set of equations. This gives us a uniform basis of computation.

4.3 Summary

In this chapter we have seen that constraint solving techniques can be effectively utilized to extend the expressive power of logic programming languages. Also, an algebraic semantics can be given for theorem proving treated as constraint satisfaction.

Chapter 5

Conclusions

The constraint model offers a rich paradigm of computation and programming. In this report, we saw that the constraint satisfaction process defines a transformation operating on a structured domain. The properties of this transformation can give useful information about the process itself. Constraint satisfaction as a deductive engine is sound and quite powerful. There are many issues, of course, which require a lot more work. Some of them are listed below.

1. Are there any general control strategies or is the problem of control application dependent?
2. Is it possible to order the different control strategies in some space?
3. For what classes of constraints can we ensure completeness?
4. Development of a powerful syntax that allows the expression of constraints on complex objects.
5. Exploration of ways to compile constraints, and form plans of execution to achieve efficiency.

References

[APT82]Apt K.R,vanEmden M.H,"Contributions to the theory of logic programming" *JACM 29, 841-862* 1982

[BORN81]Borning A,"The programming language aspects of Thinglab, A constraint oriented simulation laboratory" *ACM Transactions on Programming Languages and Systems Vol3. No4. Oct 1981 Pages 353-387*

[CHANG73]Chang C, Lee R.C,"Symbolic Logic and Mechanical Theorem Proving" *Academic Press,1973*

[COH87]Cohen J,"A view to the Origins and Development of Prolog" *Computer Sc. Dept. Brandeis Univ. 1987*

[DARL82]Darlington et al,"Functional Programming and its Applications" *Cambridge University Press 1982*

[DEGR86]Degroot D, Lindstrom G,"Logic Programming: Relations Functions and Equations" *Prentice Hall 1986*

[DAV85]Davis E,"Constraint Propagation on Real-Valued Quantities" *Technical Report # 189, Courant Institute of math. 1985*

[DAV87]Davis E,"Constraint Propagation with Interval Labels" *Artificial Intelligence vol 32 1987 Pages 281-331*

[DONN68]Donnellan T,"Lattice Theory" *Pergamon Press 1968*

[EVAN87]Evans E,"Forthcoming Master's Thesis" *Virginia Tech 1987*

[FIKE70]Fikes R,E"REF-ARF A System for Solving Problems stated as Procedures" *Artificial Intelligence 1 1970 27-120*

[FITT85]Fitting M,"A Kripke-Kleene Semantics for Logic Programs" *The Journal of Logic Programming 1985 4 295-312*

[FLOY67]Floyd R.W,"Non-Deterministic Algorithms" *Journal of the ACM Vol 14 4 1967, 636-644*

[GASCH74]Gaschnig J,"A Constraint Satisfaction Method for Inference Making" *12th Annual Allerton Conference on Circuit and System Theory Oct1974*

[GLAS84]Glaser et al,"Principles of Functional Programming" *Prentice Hall International 1984*

[JAFF87]Jaffar J, Lassez J L,"Constraint Logic Programming" *Proceedings of the Conference on Principles of Programming Languages Munich 1987*

[JAFF83]Jaffar J, Lassez J,L, Maher M,J,"Completeness of the Negation as Failure Rule" *Proc. 8th IJCAI Karlsruhe 1983 500-506*

[KOWA74]Kowalski R,"Predicate Logic as a Programming Language" *Proc. IFIP Cong. 1974, North Holland Pub Co. Amsterdam, 569-574*

[KOWA79]Kowalski R,"Logic for Problem Solving" *North Holland 1979*

[KRIP75]Kripke S,"Outline of a Theory of Truth" *the Journal of Philisophy 72, 690-717*

[LOYD84]Lloyd J,W,"Foundations of Logic Programming" *Springer Verlag 1984*

[MALC86]Malaachi Y,"Nonclausal Logic Programming" *PhD Dissertation Stanford University 1986*

[MANN80]Manna Z,Waldinger R,"A Deductive Approach to Program Synthesis" *ACM Transactions on Programming Languages and Systems vol2 no1 Jan1980, 90-121*

[MANN76]Manna Z,Shamir A,"The Theoretical Aspects of the Optimal Fixpoint" *SIAM Journal of Computing vol5 No3 September 1976*

[MURR82]Murray N,V,"Compeletely Nonclausal Theorem Proving" *Artificial Intelligence 18 1982, 67-85*

[NAD86B]Nadel B,"The General Constraint Satisfaction Problem" *DCS-TR-170 Rutgers University Jan 1986*

[NAD86A]Nadel B,"Theory Based Search Order Selection for Constraint Satisfaction Problems" *University of Michigan CRC-TR-6-86*

[PLOT72]Plotkin,"Building in Equational Theories" *Machine Intelligence Vol 7 1972*

[ROB65]Robinson J,"A Machine-printed Logic Based on the Resolution Principle" *JACM 12 1 Jan.1965 23-41*

[STEL80]Steele G,"Constraints: The Design and Implementation of a general purpose language based on Constraints" *PhD Dissertation MIT 1980*

[SUND87]Sundararajan R,"Forthcoming Master's Thesis" *Virginia Tech, September 1987*

[SUTH63]Sutherland I,E,"Sketchpad: A man-machine graphical communication system" *MIT Lincoln Labs Tech Rep.296 1963*

[TARS55]Tarski A,"A Lattice Theoretic Fixpoint Theorem and its Applications" *Pacific Journal of Math 5 1955, 285-309*

[VAND76]vanEmden,Kowalski,"The Semantics of Predicate Logic as a Programming Language" *JACM vol23 no4. Oct76, 733-742*

[WALZ72]Waltz D,"Understanding Line Drawings of scenes with shadows" *in P. Winston, The Psychology of Computer Vision*, McGraw Hill 1975, 19-91.

The vita has been removed from the scanned document