UNF UNIVERSITY of NORTH FLORIDA

UNF Digital Commons

UNF Graduate Theses and Dissertations | Student Scholarship

1987

Compiling Unit Clauses for the Warren Abstract Machine

George D. Herbert
University of North Florida

Suggested Citation

Herbert, George D., "Compiling Unit Clauses for the Warren Abstract Machine" (1987). *UNF Graduate Theses and Dissertations*. 571.
https://digitalcommons.unf.edu/etd/571

This Master's Thesis is brought to you for free and open access by the Student Scholarship at UNF Digital Commons. It has been accepted for inclusion in UNF Graduate Theses and Dissertations by an authorized administrator of UNF Digital Commons. For more information, please contact Digital Projects.
© 1987 All Rights Reserved

UNF UNIVERSITY of NORTH FLORIDA

COMPILING UNIT CLAUSES FOR THE WARREN ABSTRACT MACHINE

by

George D. Herbert

A thesis submitted to the
Division of Computer and Information Sciences
in partial fulfillment of the
requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA

December, 1987

The thesis "Compiling Unit Clauses for the Warren Abstract Machine" submitted by George D. Herbert in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

3/14/88

Thesis Advisor and Committee Chairman

Signature Deleted

3/14/88

Signature Deleted

3/18/88

Accepted for the Division of Computer and Information Sciences:

Signature Deleted

4/7/88

Division Director

Accepted for the University:

Signature Deleted

4/8/88

Vice-President for Academic Affairs

ACKNOWLEDGEMENT

The author extends his sincere thanks to Dr. Ralph Butler for his guidance and support throughout the writing of this thesis. I wish to thank Argonne National Laboratories for the use of their computer facilities through Dr. Butler's account.

The efforts of Dr. Charles Winton and Dr. Paul Mullenix, as members of the thesis committee, are also greatly appreciated.

I especially wish to thank my wife, Sallie, for her support throughout my graduate education.

TABLE OF CONTENTS

ABSTRACT

This thesis describes the design, development, and installation of a computer program which compiles unit clauses generated in a Prolog-based environment at Argonne National Laboratories into Warren Abstract Machine (WAM) code. The program enhances the capabilities of the environment by providing rapid unification and subsumption tests for the very significant class of unit clauses. This should improve performance substantially for large programs that generate and use many unit clauses.

Chapter 1

INTRODUCTION

1.1 THESIS ORGANIZATION

This thesis deals with the design, development, and use of a computer program which was written to enhance the capabilities of a PROLOG-based environment at Argonne National Laboratories (ANL).

Part 1.2 of chapter 1 of this thesis presents an overview of the problem. Part 1.3 of chapter 1 presents a review of literature pertinent to the topic.

Chapter 2 presents a discussion of the design of the programming procedures used. It also provides a discussion of the methods used in testing the completed program.

Chapter 3 describes the installation of the program, and discusses changes that were necessary for its successful operation.

Finally, chapter 4 presents an evaluation of the obtained results and suggests changes that may enhance the program's operation.

1.2 PROBLEM REVIEW

1.2.1 Computational Logic

Computational logic has proved to be valuable in dealing with a wide range of applications. The obvious application would be automated reasoning, in particular, automated theorem proving. It is not a great leap to see the application of logic to the inference capabilities of expert systems. Logic is an obvious tool when dealing with formal languages. It is therefore not surprising to find logic used to handle natural language processing as well. What may be surprising is that any computational task can be reduced to proving a theorem in first order logic [Levesque84]. This makes logic one among the many formalisms that support general computation.

Why should logic be chosen rather than one of the competing formalisms for tasks that are not obviously "logical"? One of the great services logic has provided to computation has been to bring to it a declarative semantics [Cohen82]. The clauses of a logic program can be given a declarative

reading as descriptive statements about entities and relations. This means that programs about real world problems can be written by making assertions about real world entities and their relationships to one another. This is a very natural and powerful way to view a problem and it has been argued that for some problems, it is the only way [Moore82]. Logic programs take problems out of the procedural thickets common to other computational formalisms. This is not to say that logic programs do not have a procedural interpretation. When executed by an interpreter, these programs behave as if they were performing a deduction in a very formal manner, using the clauses of the program as axioms. So logic separates the declarative and procedural components of a problem, which is to say, problem representation and control become distinct issues.

Given these excellent properties then, why would anyone consider any approach other than logic? Perhaps the most abiding criticism of logic is its perceived inefficiency. Theorem-proving can be seen as a search of a "theorem-space". As ordinarily implemented, backtracking is employed to exhaustively search this space. For even relatively small problems, the search space can be enormous. Under some circumstances, the search may never terminate. Furthermore, the pattern-matching capability known as

unification used by logic interpreters, can require exponential time [Lloyd84].

There are several paths to improved performance in computational logic [Butler86]. They are clause compilation, multiprocessing, database indexing, and clause-set "compaction". Using Sam's Lemma, a classic problem in lattice theory for the theorem proving literature, as a benchmark problem, it was shown that a better than order-of-magnitude improvement in processing time could be achieved using these techniques. The clause compilation approach is taken in this paper, and, to understand it better, it will be useful to look more carefully at the techniques employed by typical computational logic environments.

1.2.2 Resolution and Unification

Before moving deeper into our topic, let us define some of the terms with which we shall be dealing. A Horn clause is a clause of the form

$$Ux_1 \ldots x_n [P_1 \ \& \ \ldots \ \& \ P_m \ \text{->} \ P_{m+1}]$$

where $m > 0$ and each P_i is atomic. Horn clauses are the basic representation used in PROLOG. A unit clause is a Horn Clause with $m = 0$. That is, there are no antecedents

to the above implication. Unit clauses are therefore unconditional clauses.

Proof in a computational logic environment usually means establishing a contradiction. Starting with the negation of the assertion to be proved, consequences are derived until one of these consequences yields a contradiction. The contradiction always evidences itself when two unifiable unit clauses, one of which is positive and the other negative, are derived. Testing for unit clause conflict is therefore a basic automated inference procedure.

The inference mechanism employed by most computational logic environments to generate consequences is the resolution principle [Robinson65A]. This principle generalizes the classical inference rules of modus ponens and modus tolens [Winston84]. To get a clear idea of how this principle works, let us first restrict our attention to the propositional calculus rather than the full first-order predicate calculus. Consider modus ponens in this arena:

$$((P \rightarrow Q) \ \& \ P) \vdash Q.$$

Implication can be translated into a disjunctive form by the rule:

$$(P \rightarrow Q) \leftrightarrow (\neg P \vee Q).$$

Now let us rephrase modus ponens by putting its implication in disjunctive form or synonymously, clausal form, that is:

$$((\neg P \vee Q) \wedge P) \vdash Q.$$

Let us do the same for the modus tolens rule:

$$((P \rightarrow Q) \wedge \neg Q) \vdash \neg P,$$

which becomes:

$$((\neg P \vee Q) \wedge \neg Q) \vdash \neg P.$$

Finally, take the case of the transitivity of implication :

$$((P \rightarrow Q) \wedge (Q \rightarrow R)) \vdash (P \rightarrow R),$$

which becomes:

$$((\neg P \vee Q) \wedge (\neg Q \vee R) \vdash (\neg P \vee R).$$

What has happened in each of these instances is that in each case where both the positive and negative of some proposition occurs in different conjuncts on the left of the

"¦-", they "cancel out" yielding the disjunction of the remaining propositions to the right of the "¦-". This is the essence of the resolution principle [Cohen82].

To generalize this to the first-order predicate calculus, more apparatus is needed. In the first-order predicate calculus, there are variables, predicates, and quantifiers with which to contend. The expression P(x) means that the predicate P is true for x. If P means "is prime" then P(3) is true but P(6) is false. The universal and existential quantifiers will be represented here as Ux P(x) and Ex P(x), and will be read as "for all x P(x)" and "for some x P(x)", respectively. In order to apply the resolution principle in first-order predicate calculus, it will be necessary to convert to clausal form and eliminate quantifiers. The conversion to clausal form is very similar to that used in propositional calculus earlier. To eliminate quantifiers, skolemization is used.

Skolemization is essentially the elimination of existential quantifiers in favor of appropriately chosen functions. A function is appropriate if it is unique to the particular quantifier being eliminated and does not occur elsewhere in any clause. Also the function must have an arity equal to the number of universal quantifiers that precede the replaced existential quantifier in the clause (functions of

arity zero are constants). Once this is done, the universal quantifiers can be eliminated as well by subscribing to the convention that all remaining variables are universally quantified. For example, consider the clause:

$$Ex\ Uy\ Uz\ Ew\ (-P(x,y,z)\ V\ Q(y,z,w)).$$

When skolemized, this becomes:

$$(-P(a,y,z)\ V\ Q(y,z,f(y,z))),$$

where "a" is a constant and "f" is a function of arity two.

Once our predicates have been skolemized and put in clausal form, there is one further complication that must be handled. When using the propositional calculus, the notion that two propositions are the same but of opposite polarity, that is to say, complementary, is quite clear, but variables, constants, and functions muddy the situation somewhat. Instead of requiring that two predicates (often called literals in this context) be identical in every respect, we now require only that they be unifiable. By this we mean that there is a substitution for the variables in both predicates that makes the predicates identical. A valid substitution must not contain within it the variable for which it is being substituted. The test in a

unification algorithm for this condition is known as an occurs check. The occurs check is necessary in order to preserve the soundness of resolution. Nonetheless, it is often omitted in PROLOG implementations. This is a pragmatic matter, for the most part. The occurs check is rarely needed in non-theorem proving applications, and its omission makes for a faster unification algorithm. Some practitioners [Colmeraurer82] [Eggert83] have even used the absence of the occurs check to work with infinite terms! The following are examples of the unification process: P(a,x,h(g(z))) and P(z,h(y),h(y)) are unifiable via the substitution {z/a, x/h(g(a)), y/g(a)} ("/" means "is substituted by"), but P(f(a),g(x)) and P(y,y) are not unifiable [Lloyd84].

Now all the elements are present to complete the picture of resolution in the first-order predicate calculus. First two complementary literals are unified. The substitution generated by the unification is applied to the remaining literals and the disjunction of these becomes the inference. For example:

(-P(x) V Q(x)) and
(P(a) V R(z))
yield (Q(a) V R(z)).

1.2.3 Subsumption

Another technique commonly used in theorem proving environments is subsumption. Theorem provers can generate a huge number of clauses. Many of the clauses generated are actually instances of more general clauses that are derived earlier in the deduction. Subsumption eliminates these less general clauses. A clause subsumes another clause if the variables in the first clause can be instantiated in such a way that the resulting literals all occur in the second clause [Wos84]. The procedure by which the variables are instantiated is called half-matching and is very similar to unification except that substitutions can only be generated for variables in one of the clauses.

1.2.4 Applications and Unit Clauses

Now that some of the techniques of computational logic have been reviewed, we can take a look at how some important applications use unit clauses. These applications are theorem proving, expert systems, and databases.

1.2.4.1 Theorem Proving

We have already made mention of the notion of unit clause conflict in theorem proving. Since unit clause conflict is

so basic, the generation of unit clauses must also be a priority of a theorem prover. Inference rules that generate unit clauses rapidly are hence of great value. Further, unit clauses feed inference rules effectively. If one or more of the clauses involved in a particular inference are unit clauses, an inference made by a resolution-type inference rule will be shorter than the longest clause involved. Indeed, the Unit-Resulting Resolution (UR-Resolution) inference rule generates only unit clauses and feeds on at most one non-unit clause at a time.

Strategically, unit clauses are once again in the forefront. One of the simplest strategies for improving performance in theorem proving is the unit preference strategy. Here the theorem prover attempts to resolve unit clauses before non-units. Even weighting strategies, while they do not specifically mandate the selection of unit clauses, will tend to prefer them by virtue of the simple fact that fewer symbols generally mean a smaller weight.

Non-resolution inference rules also utilize unit clauses heavily. Paramodulation and Demodulation are driven by unit equality clauses.

Generation of unit clauses is not sufficient for effective theorem proving. As mentioned earlier, many clauses are

generated which are redundant versions of clauses already present. Also new clauses may be more general than clauses currently in the database. If these clauses are added to (or respectively, left in) the database, there will be many more clauses to resolve against, but which will provide no more information than is already available. Forward Subsumption is the technique which prevents redundant new clauses from being added to the database and Backward Subsumption deletes from the database old clauses which are less general than a newly generated clause. Without Subsumption, the number of clauses in the database for even a reasonably sized problem will increase explosively and the progress toward a proof will be slowed dramatically.

Unit clauses are ideal for subsumption. If a unit clause half-matches any literal in a multi-literal clause, that clause is subsumed. Also more general unit clauses will subsume less general unit clauses. Thus a database with many unit clauses will tend asymptotically to contain only quite general unit clauses. This unit clause-rich database then provides a fertile situation for unit clause conflict, which is another way of describing an environment ideal for proving theorems.

1.2.4.2 Expert Systems

Expert systems bear certain superficial similarities to theorem provers. Both are based on inference and work with a kind of database. The theorem prover's initial database generally consists of several lists of clauses, known as axioms, set of support, have-been-given, and demodulators. This database is then augmented as inferences are made. The initial database of an expert system is referred to as a knowledge base [Jackson86]. It generally does not change in the course of the inferences made by the system, but rather affects the state of the expert system often by modifying a special type of memory called the working memory. One knowledge representation commonly found in expert systems is the production rule. A production rule has the form of a set of conditions called antecedents or if-parts, together with a set of actions which are called the then-parts. Production rules therefore look very much like logical implications. Other forms of knowledge representation have a far less logical look to them. Among these are frames, objects, and semantic nets. The form of knowledge representation which will most interest us in this discussion is called a fact. Facts are essentially unit clauses. In production rule systems, it is common to use a resolution-like mode of inference which matches antecedents to conditions in the working memory and facts. So

unification with unit clauses is important in production systems. Systems based on frames, objects, and semantic nets can be thought of as having two separate components: a control component and an information component. The control component dictates the conditions under which access to data in the information component is allowed. But once access to the information component has been achieved, the data will generally be factual data; that is, unit clauses.

Subsumption plays little role in the operation of a typical expert system; although, given what we know about theorem provers, perhaps it should. The literature on knowledge acquisition, the process by which the knowledge base for a particular expert system is acquired, does make mention of subsumption. Knowledge bases tend to be built up on a rather ad hoc basis, and often redundant knowledge and less general knowledge than is already contained in the knowledge base is added. This sort of knowledge only makes the operation of an expert system less efficient. Systems have been proposed that will, among other things, check knowledge to be added to a knowledge base for subsumption [Nguyen87].

1.2.4.3 Database Systems

Database systems present a somewhat different picture than theorem provers or expert systems. The suitability of logic for database applications has long been recognized [Gallaire78] [Gallaire84]. The relational database model fits particularly neatly into the logic programming paradigm [Codd70]. The definition of the logic programming form of a database given in [Levesque84] is as a collection of Horn clauses (as defined earlier) where m = 0 and the arguments to the predicates are all constants. Thus the database form is nothing other than a collection of unit clauses with constant arguments! A database query is typically a conjunction of clauses which may contain variables. If these clauses can be satisfied by unification with clauses in the database, the query succeeds. Technically, it should be noted that full unification is not necessary, since only the query can contain variables. Half-matching will suffice for database queries.

These by no means exhaust the applications of logic. Nor have the uses of unit clauses, unification, and half-matching within logic been used up. However, these applications are very important, and they all make heavy use of unit clauses and unification. Half-matching has found significant use in these applications as well. If there is

some means of improving the performance of unification and half-matching for unit clauses, there will be great benefits for computational logic.

1.3 LITERATURE REVIEW

1.3.1 The Prehistory of Computational Logic

The idea of a mechanical procedure for deciding the truth or falsity of a given proposition dates from the seventeenth century with the Analytic Geometry of Descartes [Davis83]. Descartes' method introduced a coordinate system with which geometrical figures could be represented using equations and those equations could be manipulated algebraically. Descartes contrasted his method with the axiomatic method of Euclid:

> ... it is possible to construct all the problems of ordinary geometry by doing no more than the little covered in the four figures that I have explained. This is one thing which I believe the ancients did not notice, for otherwise they would not have put so much labor into writing so many books in which the very sequence of the propositions showed that they did not have a sure method of finding all ...

Descartes had in an important way mechanized geometry. It was Leibniz, however, who envisioned the mechanization of reasoning. To this end, he proposed a calculus of reason (calculus ratiocinator) imbedded in a universal language

(lingua characteristica). A problematic proposition would be formulated in the lingua characteristica and subsequently decided by manipulations using the calculus ratiocinator (in the words of Leibniz, "Let us Calculate"). Though Leibniz made little progress toward the achievement of his grand conception, he had planted the seed that was to germinate into mathematical logic and ultimately into automated reasoning.

The first substantive progress toward the realization of Leibniz' program was the work of George Boole two centuries later. That Boole had indeed mechanized logic was recognized by Stanley Jevons, an economist and logician who constructed a cash register-like machine capable of verifying Boolean identities.

The next landmark on the way toward Leibniz' dream was the Begriffsschrift of Gottlieb Frege. Frege develops the predicate calculus by explicating the use of quantifiers about which there had been no clear conception. Frege's work is the first example where the syntax of an artificial language is laid out in detail and thus is the ancestor of all formal languages, especially computer programming languages. He also pointed out the importance of modus ponens as a rule of inference. Unfortunately, Frege's work was regarded as too obscure and it is the notation developed

by Peano that we use today. However, Peano's work, done a decade after Begriffsschrift, lacks the syntactic clarity and appreciation for quantifiers and rules of inference of the earlier work.

The reputed obscurity of Frege's work was the least of the problems mathematical logic faced in its formative years. Mathematics itself was in a great ferment over its foundations. The work of Cantor in set theory and Weierstrass and Dedekind in analysis was regarded as expanding the boundaries of mathematics by many of their contemporaries, while Kronecker, Poincare and Brouwer heaped contempt on this same work. A key issue in this dispute revolved around the role of existence proofs in mathematics. The classical camp felt free to accept a proof in which mathematical existence was proved without the construction of an actual example. Brouwer, the founder of Intuitionism, on the contrary demanded that every mathematical proof purporting to demonstrate existence do so by constructing an example in a finite number of steps. In opposition to Brouwer was David Hilbert who felt the Intuitionists were rejecting too much that was valuable in mathematics. Hilbert therefore proposed a dramatic program, called metamathematics, to provide a basis for classical mathematics that even an Intuitionist would be forced to accept. What was needed first was a formal calculus in

which classical mathematics could be expressed. This step was accomplished by Whitehead and Russell in their monumental Principia Mathematica. Then a constructive consistency proof would be provided for this calculus. Although this program never achieved its aim, it was nonetheless highly influential.

Hilbert and Ackermann posed two key problems for the metamathematical program. The first is the problem of completeness: that every valid sentence is derivable from the axioms. The second is the Entscheidungsproblem: that there is an algorithm for determining whether or not a given sentence is valid.

Skolem showed that a quantified predicate has no interpretation that makes it true if and only if a finite conjunction of sentences which contain Skolem functions in place of existential quantifiers is unsatisfiable. This is essentially the proof procedure used in automated theorem-provers. This method relies on the axiom of choice however and is not therefore constructive because there is no algorithm providing the value of the Skolem functions given some constant arguments.

Kurt Goedel settled the completeness problem by establishing the equivalent of Skolem's result without recourse to the

axiom of choice. Herbrand also provided a proof of the completeness theorem which is valid for a wider class of sentences than are the results of Skolem and Goedel. He also produced the basic idea for a unification algorithm which is fundamental to the operation of automated theorem-provers.

Doubts about the solvability of the Entscheidungsproblem were raised by Goedel's undecidability theorem. His famous proof establishes that all consistent formulations of number theory include valid sentences for which there can be no demonstration in a finite number of steps. The actual unsolvability of the Entscheidungsproblem was established independently by Alan Turing and Alonzo Church. Turing used his well-known "machines" to show the unsolvability of the halting problem. If the Entscheidungsproblem were solvable, its algorithm could be used to solve the halting problem. Since the halting problem is unsolvable, so is the Entscheidungsproblem. Church derives a similar undecidability result using his lambda-calculus.

1.3.2 Early Theorem Provers

At about this time general purpose digital computers were invented and it was not long before attempts to test the potential of these devices by programming theorem provers

was made [Davis83] [Loveland78] [Chang73]. The two earliest attempts at theorem proving on a digital computer were the "logic machine" of Newell, Shaw, and Simon and a system implementing a decision procedure for Presburger arithmetic by Davis. The former took the approach of a human problem solver to prove theorems in the propositional calculus using the axiomatization in Russell and Whitehead's Principia Mathematica. Something approaching the notion of unification came out of this program. The latter program took a more rigorous approach but proved to be very slow. This is not surprising since it is now known the Presburger decision procedure is worse than exponential in complexity.

These programs set the tone for work that was to follow by emphasizing heuristic sophistication in the first case, and mathematical sophistication in the latter.

The next important attempt was the "geometry machine" of Gelernter. This program was more in the spirit of the "logic machine" and managed to rediscover a proof unknown to Gelernter of a theorem on isosceles triangles. In order to make any real headway though, the program had to rely on guidance from the techniques of analytical geometry.

The idea of using methods based on Herbrand's theorem can be attributed to Abraham Robinson where he made suggestive

remarks about the constructions used to prove geometry theorems as being elements of the Herbrand universe for the problem. Davis and Putnam proposed a theorem prover based on these ideas and introduced Skolem functions and the clausal form for the initial clauses in the database. The work proved disappointing, however, since it unleashed the combinatorial explosion inherent in these procedures, leading them to comment:

> "... the most fruitful future results will come from ... excluding ... 'irrelevant' quantifier-free lines from the Herbrand expansion."

The logician Hao Wang used methods he had developed through proof theory and solvable cases of the Entscheidungsproblem. He wrote a program that proved all the theorems in Principia Mathematica that belonged to the pure predicate calculus with equality. What Wang showed was not that the techniques used were particularly powerful, but rather that the problems being attacked were fairly easy, requiring very little of the resources of the domain.

Prawitz in 1960 came up with the idea to produce terms of the Herbrand expansion only when they were actually needed. This is a very powerful idea on which all later work depends. His idea leads to the notion of a unification algorithm but he was not quite able to see this. The

observation was made independently by Dunham and North in 1962 and Davis in 1963.

The road was now paved for J. A. Robinson to introduce the resolution principle [Robinson65A] [Robinson83]. He was further able to show that this rule of inference is complete. With resolution, a single combinatorial principle was shown to be adequate for all inference. Related principles were also developed including hyperresolution [Robinson65B].

The resolution principle does not entirely eliminate the problem of combinatorial explosion, however. To limit this problem, heuristics are still necessary. The team of L. Wos, D. Carson, and G. A. Robinson at Argonne National Laboratories (ANL) developed approaches that give certain clauses special treatment and thereby dramatically limit search [Wos64] [Wos65].

An important theorem proving system was developed by Boyer and Moore [Boyer79]. The central formulas operated on by the theorem prover are treated as functions rather than predicates. The system operates by rewriting the current formula and never backtracks or changes any decision once made. The rewriting process is guided by heuristics, which though sound, render the system incomplete. The system is

capable of performing induction, a capability not common in predicate calculus-based theorem provers.

The Interactive Theorem Prover (ITP) was developed by Argonne National Laboratories as a general purpose theorem proving environment [Wos84] [Lusk84]. It is a descendant of the Automated Reasoning Assistant (AURA). AURA was a very fast and powerful theorem prover. Because it was written in IBM 360/370 Assembly Language and PL/I, it was not portable. To address this lack of portability, Logic Machine Architecture (LMA) was written in Pascal. LMA is not itself a theorem prover, but provides procedures that can be tailored into automated reasoning programs. ITP was the first major system implemented within the LMA framework [Lusk82]. The crowning achievement of ITP was to settle some open questions in mathematics, logic, and circuit design [Winker81] [Winker82] [Kabat82] [Wojciechowski83]. Here was a theorem prover that was surpassing human capacities, not just demonstrating a few human-like problem solving capabilities. This is not to say that ITP is somehow an ultimate theorem proving system. There are still many problems for which it is not suitable or that are intractable.

1.3.3 Logic Programming

Theorem proving was an obvious application for computational logic. However, John McCarthy [McCarthy63] realized that the execution of an applicative program can be thought of as proving an identity ($f(x_1,...,x_n)$ = result) by applying various axioms of identity according to a fixed control regime [Cohen82]. The idea of logic programming per se is attributable to C. Green in his thesis [Green69], where he used a method now referred to as Green's trick to derive operator sequences. The idea of logic programming was popularized by R. Kowalski [Kowalski79a]. He advanced the concept of an algorithm as being made up of a logic component and control component [Kowalski79b]. The logic component describes the problem and the control component specifies the manner in which the definitions will be used. Kowalski argued that once these components are isolated, programs can more readily be improved and modified. Nils Nilsson has proposed that artificial intelligence is most properly thought of as applied logic [Nilsson80].

1.3.4 PROLOG

Today, the primary vehicle for logic programming is the PROLOG language, which was developed by Alain Colmerauer and his associates [Colmerauer73A] [Colmeraurer73B] who were

primarily interested in a vehicle for natural language processing. PROLOG resembles the Microplanner language [Sussman71] which in turn derives from the Planner language, proposed by Carl Hewitt [Hewitt69]. Eventually, the syntax and techniques of PROLOG became relatively standardized [Clocksin84]. Terry Winograd's blocks world program SHRDLU demonstrated the power of logic programming [Winograd72]. However, there were many who scoffed at the slowness of early PROLOG interpreters arguing that LISP was the only language for serious artificial intelligence programming.

The rebuttal to this position came in a paper by David Warren, Luis Pereira, and Fernando Pereira [Warren77a]. Several linguistic advantages of PROLOG over LISP are given:

1. General record structures take the place of LISP S-expression.
2. Pattern matching takes the place of selector and constructor functions in LISP.
3. PROLOG procedures can have multiple outputs as well as multiple inputs.
4. Inputs and outputs do not have to be distinguished in advance, so PROLOG procedures are multi-functional.
5. Through backtracking PROLOG can present many alternative results. This is a high-level form of iteration.

6. Unification in conjunction with the logical variable is much more powerful than simpler forms of pattern matching.
7. There is no inherent distinction between program and data.
8. There is a natural declarative semantics in addition to a procedural semantics.
9. The procedural semantics of a syntactically correct program is totally defined.

The really crucial point, however, is that all these advantages can be had without a significant sacrifice of performance. In particular, through the compilation of logic, PROLOG compares favorably with LISP. This performance is achieved through various implementation approaches, such as the compilation of "special purpose" unification procedures, clause indexing, structure sharing and a distinction between local and global stacks. The innovations described in Warren's paper were further honed in subsequent approaches to PROLOG implementation [Warren77b, Warren80], culminating with the Warren Abstract Machine [Warren83].

1.3.5 Warren Abstract Machine

The Warren Abstract Machine (WAM) provides a framework into which any PROLOG program can be mapped. This machine could

be implemented as an interpreter for a bytecode into which the WAM instructions have been translated, as a set of macroinstructions which could be compiled, or in hardware or firmware. The bytecode interpreter is the approach most often adopted. WAM makes use of several distinguishable types of data. These are variables, constants, lists, and structures. The data areas used are the code area, the heap (or global stack), the local stack, and the trail. There is also a small push-down list used for unification. The heap contains all the complex data structures (lists and structures). The local stack contains information used only by the current procedure. The trail contains information about variables that have been bound but will have to be unbound when backtracking occurs. WAM uses a number of registers to keep track of the various data areas, to pass arguments to procedures, and to hold the values of temporary variables used by a clause. The WAM instruction set is made up of get instructions, put instructions, unify instructions, procedural instructions, and indexing instructions. The first three types of instructions handle unification, and the last two types of instructions deal with control. The get instructions are used for matching against the head of a clause. Conversely, the put instructions load the arguments that will be passed in a procedure call. The unify instructions handle unifications with the arguments of a structure or list whether the

structure or list already exists or is being created. The procedural instructions handle control transfer and environment allocation. The indexing instructions filter out those clauses in a procedure definition that cannot possibly match a given procedure call.

This is exactly the toolbox needed to improve the performance of logic programs. One of the hallmarks of the WAM architecture is its set of specialized unification primitives. These primitives avoid the overhead of a general unification algorithm by focusing only on exactly what is needed in a particular situation. For example, a variable will unify with anything. It makes no difference that the symbol with which the variable is to be unified is a constant, a variable, a structure, or a list, unification will succeed. So a primitive for unifying with a variable can take this knowledge into account. Similarly, a constant will only unify with a symbol that is exactly equal to that constant or a variable. Once again this knowledge can be used to make constant unification as simple and straightforward as possible.

The primitives are further specialized as to the context of the unification. Head unification, which is used only for arguments of a called procedure, is distinguished from a kind of unification designed to build arguments for calling

routines and a third type of unification for any other purpose. Each of these typically will have certain implementation consequences when WAM is coded for a particular machine.

Chapter 2

METHODS AND PROCEDURES

2.1 PLAN OF ATTACK

2.1.1 Background.

We have seen that unit conflict checking and inference rules rely on a pattern matching capability called unification. Subsumption relies on a related form of pattern matching called half-matching. In a typical theorem prover, for example, these pattern matching activities are done in a manner similar to an interpreter. Every time a clause is unified (or half-matched) with a second clause, a general algorithm for unification (or half-matching) is invoked. Unit clause compilation builds a special purpose unification (or half-matching) algorithm tailored specifically to a particular clause. This reduces the computational cost of performing pattern matching.

Unit clause compilation may therefore be viewed as an investment. The cost of compiling a clause is amortized against pattern matching efficiency. To achieve the best

"return on investment", one must get the best gains possible while holding down the cost of compilation.

This is an important reason for having a special compiler for handling only unit clauses. The compilation process for unit clauses is simpler than that for non-unit clauses. Therefore, a unit clause compiler will require less resources than a full WAM compiler. So the "return on investment" is amplified by reducing initial compilation costs. Also unit clauses tend to be more enduring than non-unit clauses. Clearly, it is less likely that a unit clause will be subsumed (only a more general unit clause can subsume a unit clause, while non-unit clauses can be subsumed by unit clauses or even other non-unit clauses). Therefore, the gains from unit clause compilation will tend to accrue for a larger proportion of the time during which the theorem prover is active. Also gains from unit conflict checking are simply not available to non-unit clauses. The sheer prevalence of unit clauses guarantees that little will be lost by excluding non-units from compilation. Of course, it is still possible to compile non-unit clauses using a full WAM compiler. The expected return will simply be much lower than the that for unit clauses.

WAM appears to be a vehicle for generating custom unification code. But to be applicable to a theorem proving

context, there must be some modifications. Theorem proving and PROLOG, while related, are not the same thing. The most important consideration is the absence of the occurs-check in PROLOG. Also subsumption requires half-matching and not unification. Otherwise, unit clauses in PROLOG and a theorem prover are very similar. Further, the nature of the problem of compiling unit clauses into WAM requires that only a small subset of the WAM instruction set be used.

Fortunately, ANL has implemented a version of the Warren Abstract Machine referred to as ANLWAM. The ANLWAM environment has facilities for switching an occurs-check on and off or switching from unification to half-matching and vice-versa. ANLWAM also has excellent facilities for handling the interface between special programs like the unit clause compiler and the external environment which includes the theorem prover.

It should be noted that this is not the only reasonable approach to handling the occurs-check and half-match problem. An alternative would be to extend WAM to include additional special instructions that have the occurs-check and still more instructions to perform half-matching. There is a tradeoff in these approaches. The latter approach will be more efficient at execution time since no switch checking needs to be done in the generated code. However, the code

generated by this approach can be used for only one purpose, so if both unification with occurs-check and half-matching are needed, two separate compilations must be done. In the first approach the same code serves all purposes. This tradeoff needs careful consideration when choosing a particular implementation. For example, a database application would probably need only unification without an occurs check. So one would be best advised to take the latter route. Theorem proving clearly needs both unification with occurs-check and half-matching. So the former path would be preferable. In an expert system application the decision would depend on details of the particular system.

The ANLWAM interface is designed for use by programs written in C. This is consistent with the earlier mentioned goal of portability. C has the advantages of high-level language constructs with low-level access to machine functions. This provides the ability to write comprehensible programs that sacrifice little to assembler language programs with regard to function and performance. This is ideal for the current application where both portability and speed are important.

In summary, this paper describes a program that compiles unit clauses generated by an application into WAM to speed unification and half-matching. This will find application

in computational logic problems of sufficient size to warrant the investment of time necessary for unit clause compilation. For very large problems, the investment should pay-off handsomely. Perhaps problems that were once too large will become accessible to computational logic.

2.1.2 Interaction with Environment

The ANLWAM environment that was used for this project runs on a 16 processor Balance system running the UNIX operating system. The unit clause compiler is a built in predicate of the ANLWAM environment, named "ucc". It can therefore be called from PROLOG code run in the ANLWAM environment or from other predicates in the ANLWAM environment that use the foreign subroutine facility.

The "ucc" predicate has arity two. The first argument passes the unit clause to be compiled. This will look like a structure to the unit clause compiler. Anything other than a structure in the first argument will cause the "ucc" predicate to fail. The second argument is a variable which will be instantiated by "ucc" to a structure one of whose arguments is a list containing the WAM code for the unit clause passed as the first argument. If the second argument fails to unify with the list that "ucc" builds, the predicate will fail.

The format of the structure containing the WAM code is the same as that produced by the full WAM compiler in the ANLWAM environment. This means that the same tools that would be used in conjunction with the full WAM compiler, can be used with "ucc". In particular, the same assembler is used to generate bytecode. This bytecode can then be executed on the bytecode interpreter.

2.1.3 Basic Design

One of the primary concerns of a compiler writer is how to handle parsing. Given the nature of this problem, with relatively few productions with which to deal, the recursive descent approach was taken [Aho79] [Calingaert79]. The ability to write recursive functions in C, made this a very feasible approach.

The lexical analysis of the incoming text is another problem that compiler writers need to face. In this case, the problem was greatly simplified by the C macros provided in the foreign subroutine interface to ANLWAM. It is impossible to say enough about these macros. One would expect that the input would come in the form of a string that would have to be broken down into tokens. With the ANLWAM foreign subroutine interface, the situation is

somewhat different. The arguments passed by ANLWAM are of necessity PROLOG data types. The available list of types is an extension of the list of data types presented earlier in the discussion of the Warren Abstract Machine. They are variables (known as value cells or simply vcells), lists, structures, and constants; but constants are of three subtypes: strings, integers, and floating point. In addition, there is a special type for the nil list. To access an argument, one first determines the type of data by using the TYPE_FORMULA macro. This returns an integer which represents the data type of the argument. This integer can be used to vector to a routine for handling the data type which the integer represents. If the data type is a string, integer, floating point constant, one uses the ACC_STRING, ACC_INTEGER, or ACC_FLOAT, respectively to gain access to the actual value of the argument. If the data type is a nil list, there is no further need to access data, since the exact nature of the data is known. For a non-nil list, one must use the ACC_HEAD and ACC_TAIL to gain pointers to the head and tail of the list respectively. The head and tail of the list can be accessed by going through the TYPE_FORMULA macro again and proceeding as above. Structures also have two macros ACC_ARITY and ACC_STRUCT. The ACC_ARITY macro gives the number of arguments in the structure. The function/predicate symbol can be accessed by using the argument number zero with the ACC_STRUCT macro.

The arguments are accessed by giving to the ACC_STRUCT macro the argument number of the argument to which access is desired. One of the great beauties of the ANLWAM foreign subroutine interface is that the arguments of a structure can be processed using a "for" loop, something which can hardly be imagined in a typical lexical analysis situation! The recursive nature of the list and structure data types is clearly highlighted through these macros. It is strongly recommended that anyone attempting to work with PROLOG data structures in a C or for that matter any other language environment make use of any macro or subroutine facilities for operating on these data structures that may be provided. If such facilities do not exist, the effort to create them will be time well spent.

The mechanism for code generation is accomplished through the difference list technique [Bratko86] [Sterling86] and implemented through more facilities of the ANLWAM foreign subroutine interface. For each data type, there is a macro to build an element of that data type. The macros of interest for the current application are BLD_VCELL, BLD_SYMBOL, BLD_NIL, and BLD_LIST. Initially, the pointer to the vcell which will contain the instruction list that will be embedded in the second argument of "ucc" is stored. Each time a new instruction is to be output, BLD_SYMBOL establishes a pointer to the instruction and BLD_VCELL is

invoked to create a new variable. A BLD_LIST is then done giving the symbol which contains the instruction as the head of the list, and the new variable as the tail. The stored vcell pointer referred to earlier is then retrieved and the newly created list is bound to that vcell. The pointer to the variable that became the tail of the new list then replaces the pointer to the vcell to which the list was bound. When all of the code has been generated, a BLD_NIL is used to create a nil list and this is bound to the vcell at the tail of the code list. The list is then a complete list of the generated WAM code.

2.1.4 Design Considerations

An important consideration in designing a Warren Abstract Machine code compiler is register usage. There are several excellent references on the Warren Abstract Machine [Warren83] [Gabriel85] [Turk85]. There is even a good reference on efficient register usage [Debray84]. The difficulty is that these references do not focus on the problem that becomes most acute when working with unit clauses. Under many circumstances, unit clauses will be well-behaved with respect to register usage. Since the Warren Abstract Machine has only a finite number of registers, unit clauses which are lush with complex arguments, that is, structures and lists, pose a potential

problem of register exhaustion. These kind of unit clauses commonly occur in theorem proving applications. Because some situations will require registers beyond those used to pass arguments, it is important to reclaim registers that are no longer needed as soon as possible.

Consider a nested structure in a unit clause:

```
p(f(g(a,b),h(c,d))).
```

Here, the structure f has two arguments, g and h, which are themselves structures with two arguments. We can use the WAM instruction get_structure to process f. The get_structure instruction cannot, however, be applied to g, the first argument of the structure f, since get_structure can only handle arguments which are in a WAM register. The solution is to move the structure g to its own register, using the WAM instruction unify_x_variable, and then apply the get_structure instruction to g in its new location. The case of the h structure is somewhat more hospitable. The WAM instruction unify_structure can be employed directly to h, without moving h to a new register. Because it is the last argument of the structure, the unify_structure instruction does not need a new register. It can treat the current register as a scratch register for the new structure

since there are no subsequent arguments of the old structure with which there might be interference.

Lists, in turn, being a kind of structure, are handled in a similar way. A list can be thought of as a structure of arity two with the name ".". The head of the list is the first argument of the structure, and the tail of the list is another list which is the second argument of the structure. The tail may be the special list "[]", the empty list. WAM does not employ the structure "." to represent lists, but if one keeps the structure representation in mind, it makes the approach very comprehensible. Consider the list:

[[a],b].

Here we have a list whose head is in turn the list [a], and whose tail is the list [b]. Analogously to structures, a list is processed with the WAM instruction get_list. Also the head of a list, if it is itself a list, must be moved to a new register to be processed. So in the example, the list [a] must be moved with a unify_x_variable instruction, and then processed with the get_list instruction in the new register. The list [b], can be processed with the unify_list command, since it is the tail of our original list, or, to put it another way, the last argument of the "." structure.

It should be noted that lists can be arguments of structures, and vice versa, so we must be prepared to handle all these instances within the guidelines put forth. All complex arguments occurring as the last argument, may be handled with the appropriate unify instruction; otherwise, they must be moved to a new register and processed with the appropriate get instruction. In the latter case, the move must be done immediately, but the get processing must be delayed until sometime after the processing of all the other arguments is completed. This delay is implemented by enqueueing the information needed to process the moved argument. When the processing of a structure or list is completed, the register which it occupied is freed.

Variables pose an additional problem. When a variable appears as the argument of the unit clause, one of two things must be done. If it has not appeared earlier, it is only necessary to note the name of the variable and the register in which it is located. No code needs to be generated. However, if it has appeared earlier, the WAM instruction get_value is generated and the register it occupies may be marked as available for use. But when variables appear as arguments in structures or lists, the situation changes somewhat. If the variable has appeared earlier, the instruction unify_x_value is generated, and

processing continues; but, if the variable has not appeared earlier, there are two cases. If the variable does not appear anywhere else in the unit clause, the unify_void instruction is generated; otherwise, the variable must be moved to a new register in order to preserve its value by generating the unify_x_variable instruction. The problem here is that one does not yet know whether the variable at hand will appear later. One could make an initial scan of the entire unit clause to determine exactly which variables occurred more than once. The implementation described here does not take this approach. Instead, a unify_x_variable instruction is generated in all situations. The justification for this is that situations where the unify_void instruction are useful are relatively infrequent. Generating a unify_x_variable instruction causes no harm beyond the use a register that would otherwise be free and the minor run time consequences due to the differences in the two instructions. On the positive side, there is the saving of a complete scan of the unit clause. This is important since this unit clause compiler is built for speed. An alternative approach will be described later in the recommendations for future enhancement. To mitigate the effects of this design decision somewhat, variables used as arguments to the unit clause are located first. Since these variables already occupy a register, they require no new register. Subsequent occurrences of these variables in

lists and structures can be handled by generating the get_x_value instruction for argument variables and the unify_x_value instruction elsewhere.

Constants and the empty list, when they occur as arguments of the unit clause, provide the opportunity to free registers immediately after generating the appropriate WAM get instruction. When they occur in lists and structures, the appropriate WAM unify instruction needs to be generated, and processing can simply continue.

In summary, structures, lists, and variables that appear in structures and lists as other than the last argument consume additional registers. Completion of processing of an argument, other than a variable that has not occurred before, frees the register occupied by that argument.

2.2 CODING AND IMPLEMENTATION

2.2.1 Detailed Design

The particulars of the design of this unit clause compiler are dictated by the discussions of the previous sections. There are some noteworthy data structures employed.

An array, indexed by register number, is used to keep track of register usage and variables. To accomplish both of these functions simultaneously, it was necessary to make an assumption which is valid in the ANLWAM environment and probably in most other conceivable environments, but which should be checked by implementors following this approach. The data in the array may be the name of a variable, or an indicator that the register is either in use or free. The assumption is that zero can indicate a free register, and that the number one can indicate a used register and that no valid variable name is either zero or one. A variable name is taken to be a pointer to the dereferenced value of the variable. To understand what the dereferenced value is, it is important to realize that variables can be bound to other variables. Thus the pointer of a variable may not point to the value of the variable, but to another variable. The process by which a pointer to the actual value of a variable is obtained is called dereferencing. In the ANLWAM environment, the pointers provided to the unit clause compiler are always dereferenced, and importantly to this discussion, never have the values zero or one. A pass is made through the arguments of the unit clause to determine if any are variables. Those that are not variables are marked as being in use. For those that are variables, the name of the variable is placed in the array, unless that name is already in the array, in which case the register is

marked as free after generating a get_value instruction. To check whether a variable has already been encountered, the array is searched from zero to max_var. Max_var is the highest register known to contain a variable and is initially set to negative one. Once all the arguments have been examined, the remaining registers are marked as being available. When any routine completes processing on a register, it simply sets the value in the array for that register to zero. The get scratch function is used to acquire a free register. The routine scans the array for the first available register, marks it as used, and returns the register as the value of the parameter passed to the function. If the register can be acquired, the function returns TRUE; otherwise, it prints an error message and returns ERROR.

The unit clause which is input to the compiler determines to a large extent the flow of control. The input is examined to determine if in fact it is a unit clause. If it is not, the compilation is terminated with an error message. It has already been described how a pass is made of the arguments of the unit clause for the purpose of initializing the register array. After this, the business of compilation really begins. Each non-variable argument is examined and code appropriate to it is generated. The non-trivial case is that of structures and lists. These are recursively

defined data structures, and fittingly the functions which process them are recursive. The recursion occurs only in the case that a structure or list is encountered as the last argument in a structure or non-nil list (the last argument of a list is always a list). This is not to say that lists or structures cannot occur in other than the last argument of a list or structure. That case is handled later. There is a difference in the first call to process a list or structure and later recursive calls. The first call always generates a get instruction. The later calls generate a unify instruction. Also the first call must free its register when its processing is complete. Otherwise they are identical. The difference is handled by passing a switch as a parameter to the processing function. The initial call passes zero, and all recursive calls pass the value one.

Another significant data structure is the queue. As indicated in the previous section, when a structure or list is encountered as other than the last argument to a structure or list, it must be moved to a new register and processed later. The pointer to the argument, its type, and the register to which it has been moved are stored in the queue for later processing. The queue is implemented as a simple linked list. To place an entry on the queue, the function enqueue is used. After all of the arguments of the

unit clause have been processed, the items on the queue, if any, are processed one at a time until the queue is exhausted. It should be noted that items on the queue may cause new items to be placed on the queue.

The final data structure worth note is the one associated with the output variable of the unit clause compiler. It has already been described how the code is placed into what amounts to a difference list, which when complete, is instantiated to an ordinary list by binding the final variable to the empty list. This list, in turn, is bound into a structure which represents the output of the unit clause compiler. Finally, the structure is bound to the output variable. This data structure has a variety of routines associated with it. There is essentially one function for each type of WAM instruction. There are also some supporting routines for handling general problems, like converting strings and integers to symbols and structures which can be used by WAM.

2.2.2 Testing

Testing of the unit clause compiler was accomplished via a suite of test unit clauses. The clauses ranged from a variety of relatively trivial instances, to some very complex, deeply nested cases. These cases were compiled

using the full WAM compiler, and the results compared with the output from the unit clause compiler. There was no requirement that the code match exactly, since there are many ways to generate correct code for a typical unit clause. The key consideration is that the code be functionally equivalent. For example, the arguments of a unit clause may be handled in any order. The assignment of variables, lists, and structures, can be made to an arbitrary work register so long as that register is not currently in use. Some WAM instructions are interchangeable in certain circumstances. Finally, it has been noted earlier that the unit clause compiler in some instances will generate sub-optimal code. Given that this is taken into account by the design of the compiler, the sub-optimality is tolerated for the sake of speed.

Beside accuracy of results, the other significant factor in the unit clause compiler is speed. The standard of comparison is once again the full WAM compiler. Timings of the results were made in order to establish whether a special purpose unit clause compilation process can indeed achieve superior performance. The unit clause compiler averaged fifty times faster than the full compiler.

Chapter 3

RESULTS

3.1 INSTALLATION

The unit clause compiler was installed in the ANLWAM environment in 1987. The unfortunate situation is that the ANLWAM environment is being superceded by a new WAM environment.

ANLWAM was developed as a research tool and has served that purpose well. Nonetheless, attention is now shifting to the new environment, and the prospects of there being any significant use made of the unit clause compiler in the ANLWAM environment are quite dim. It is hoped that a unit clause compiler will be written for the new environment using the experience gained through the ANLWAM implementation described here.

3.2 NECESSARY CHANGES

The changes in the unit clause compiler resulted mainly from the exigencies of working in a research setting. Documentation is sometimes incomplete or in flux. The

working documents for the ANLWAM foreign subroutine interface, for example, were drafts, not final documents. Given this, it is not too surprising that there were occasional problems.

The only serious problem occurred when it was discovered that the unit clause compiler would generate correct code for certain deeply nested unit clauses in one instance and incorrect code in other instances. Sometimes an infinite loop would occur and sometimes a hard failure due to a pointer error would occur. How one situation differed from the others was not at all clear. The solution to the problem came with the realization that ANLWAM was not reloading the unit clause compiler each time it was invoked. Initializations not made by run time assignments were not done after the first invocation of the program. When all initializations were made by run time assignments, the condition disappeared.

Chapter 4

CONCLUSIONS AND RECOMMENDATIONS

4.1 EVALUATION

Because of the move to replace the ANLWAM environment, there is little to report about the experiences of users of the unit clause compiler. In spite of this, testing reveals that the idea of unit clause compilation is sound. By providing a special purpose compiler for unit clauses, it is possible to reduce the cost associated with compilation. The research presented by [Warren77] and particularly [Butler86] provides the justification for the compilation of logic.

Since the compilation of logic is an important component of Argonne National Laboratory's efforts to achieve a high performance logic environment, the concepts advanced here should receive considerable attention in the future.

4.2 RECOMMENDATIONS FOR FUTURE ENHANCEMENT

The implementation described here is satisfactory in most respects, except that it accepts the generation of sub-optimal code. The justification presented earlier was that this was a conscious trade off of compile time speed for execution time speed. The idea was that it would take a second pass of the entire input unit clause to gain the information necessary to generate optimal code. The cost of this second pass was not felt to be justified since there would be no improvement in the great majority of input instances even after the extra pass.

An insight into this problem came after the current implementation was completed. The value of the ANLWAM foreign subroutine facilities has been pointed out earlier. They essentially embed PROLOG capabilities in a C program. The key insight is precisely that thinking of the above problem in PROLOG terms provides an elegant solution.

To solve the problem in one pass, it is necessary to modify the code generation process a little. When the situation arises where the decision must be made as to whether to generate a unify_x_variable instruction or a unify_void instruction, the information needed to make this decision

will not be available until the unit clause has been completely processed. So one delays the decision by generating an uninstantiated variable as the "code" and storing in a list the information that procedures invoked later in the compilation will use to generate the correct code and then instantiate the variable to that code. The information stored in the list would be the pointer to the "code" variable and a scratch register which will be used in the event the unify_x_variable instruction is ultimately generated. If later in the compilation, it is seen that one of the variables in the list appears again, a unify_x_value instruction is generated for that variable and the "code" variable in the list is instantiated to a unify_x_variable instruction which uses the register stored in the list. The entry is then deleted from the list. At the end of the compilation, there will be only unreferenced variables remaining in the list. The "code" variables for these are all instantiated to unify_void instructions.

The hope is that the above discussion provides additional impetus for future implementors to develop and utilize the sort of macros and routines in the ANLWAM foreign subroutine interface. It is obviously not impossible to accomplish the above without such facilities, but it is certainly not desirable. The delayed binding exemplified above provides support for the power of logical variables and the

desirability of providing them even in conventional procedural language environments.

Another improvement to the unit clause compiler would be to generate object code directly. Once a compiler can generate WAM code successfully, it is not a huge step to generate "byte code", which could be interpreted directly without the intermediate step of assembly. Indeed, given some of the complications involved with generating WAM code, it may well provide even faster compilation. With the success of WAM implementations, it will not be surprising to see firmware or even hardware implementations of WAM in the near future. These provide further motivation toward the attainment of very high performance logic environments.

4.3 CONCLUSIONS

Logic is becoming an extremely important computational paradigm. Logic provides a clear declarative and procedural semantics that lends itself to a wide variety of applications.

The most frequent criticism of logic, is that implementations of logic are too slow. This criticism has been addressed by the work of David Warren in the compilation of logic. A unit clause compiler such as the

one described in this paper, further refines the advantages of logic compilation by providing a low overhead method for compiling the very significant class of unit clauses.

An implementation of a unit clause compiler, particularly one with the recommended enhancements, will help computational logic environments to achieve high levels of performance.

BIBLIOGRAPHY

[Aho79]
Aho, Alfred V., and Jeffrey D. Ullman, Principles of Compiler Design, Addison-Wesley Publishing Co., Reading Massachusetts, 1979.

[Boyer79]
Boyer, Robert S., and J. Strother Moore, A Computational Logic, Academic Press, New York, 1979.

[Bratko86]
Bratko, Ivan, PROLOG Programming for Artificial Intelligence, Addison-Wesley Publishing Company, Wokingham England, 1986.

[Butler86]
Butler, Ralph, et al,"Paths to High-Performance Automated Theorem Proving", Technical Report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne Illinois, 1986.

[Butler87]
Butler, et al, "ANLWAM: A parallel Implementation of the Warren Abstract Machine", Technical Report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne Illinois, 1987.

[Calingaert79]
Calingaert, Peter, Assemblers, Compilers, and Program Translation, Computer Science Press, Rockville Maryland, 1979.

[Chang73]
Chang, Chin-Liang, and Richard Char-Tung Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

[Charniak85]
Charniak, Eugene, and Drew McDermott, Artificial Intelligence, Addison Wesley, Reading Massachusetts, 1985.

[Clocksin84]
Clocksin, W. F., and C. S. Mellish, Programming in PROLOG, Springer Verlag, Berlin, 1984.

[Codd70]
Codd, E. F., "A relational model for large shared databases", Communication of the ACM 13 (6), pp. 377-387, June 1970.

[Cohen82]
Cohen, Paul R., and Edward A. Feigenbaum, The Handbook of Artificial Intelligence, William Kaufmann, Inc., 1982.

[Colmeraurer73A]
Colmeraurer, Alain, "Les systemes-Q ou un Formalisme pour Analyser et Synthesizer des Phrases sur Ordinateur", Publication Interne No. 43, Dept. d'Informatique, Universite de Montreal, Canada, 1973.

[Colmeraurer73B]
Colmeraurer, A., et al, "Un Systeme de Comunication Homme-machine en Francaise", Research Report, Groupe Intelligence Artificielle, Universite d'Aix-Marseille II, 1973.

[Colmeraurer82]
Colmeraurer, A., "PROLOG and Infinite Trees", in <C. L. Clark and S.-A. Tarnlund>, eds., Logic Programming, Academic Press, London, 1982.

[Davis83]
Davis, M., "The Prehistory and Early History of Automated Deduction," in <Jorg Siekmann and Graham Wrightson>, eds., Automation of Reasoning, Springer Verlag, Berlin, 1983.

[Debray84]
Debray, Saumya K. "Efficient Register Allocation for Temporary Variables in the Warren PROLOG Engine", Technical Report, Dept. of Computer Science, SUNY at Stony Brook, August 1984.

[Eggert83]
Eggert, P. R., and K. P. Chow, "Logic Programming Graphics with Infinite Terms", Technical Report, University of California, Santa Barbara 83-02, 1983.

[Gabriel84]
Gabriel, J., et al, "A Short Note on Achievable LIP rates Using the Warren Abstract PROLOG Machine", Technical Report, Mathematics and Computer Science Division Technical Memo #36, Argonne National Laboratory, Argonne, Illinois, September 1984.

[Gabriel85]
Gabriel, John, et al, "A Tutorial on the Warren Abstract Machine for Computational Logic", Technical Report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne Illinois, June 1985.

[Gallaire78]
Gallaire, H., and J. Minker, Logic and Databases, Plenum Publishing Co., New York, 1978.

[Gallaire84]
Gallaire, H. J. Minker, and J. M. Nicolas, "Logic and Databases: A Deductive Approach", Computing Surveys (16), pp. 153-185, 1984.

[Jackson86]
Jackson, Peter, Introduction to Expert Systems, Addison Wesley, Wokingham England, 1986.

[Kabat82]
Kabat, W. C., and A. S. Wojcik, "Automated Synthesis of combinational logic using theorem proving techniques," Proceedings of the Twelfth International Symposium on Multiple-values Logic, pp. 178-199, May 1982.

[Kelley84]
Kelley, Al, and Ira Pohl, A Book on C, The Benjamin/Cummings Publishing Co., Menlo Park, 1984.

[Kowalski74]
Kowalski, Robert, "Predicate Logic as a Programming Language," in <J. L. Rosenfeld>, ed, Information Processing 74, North-Holland Publishing Co., Amsterdam, 1974, pp. 569-574.

[Kowalski77]
Kowalski, Robert, "Algorithm = Logic + Control", Communications of the ACM, pp 424-436, July 1977.

[Kowalski79]
Kowalski, Robert, Logic for Problem Solving, Elsevier North Holland, New York, 1979.

[Kowalski85]
Kowalski, Robert, "The relation between logic programming and logic specification", in <C. A. R. Hoare>, ed., Mathematical Logic and Programming Languages, Prentice-Hall International, Englewood Cliffs New Jersey, 1985.

[Levesque84]
Levesque, Hector J., "A Fundamental Tradeoff in Knowledge Representation and Reasoning," Proceedings CSCCSI-84, London Ontario, pp. 141-152, 1984.

[Loveland78]
Loveland, Donald W., Automated Theorem Proving: A Logical Basis, North-Holland Publishing Co., Amsterdam, 1978.

[Lloyd84]
Lloyd, J. W., Foundations of Logic Programming, Springer Verlag, Berlin, 1984.

[Lusk82]
Lusk, Ewing, and Ross A. Overbeek, "An LMA-based theorem prover," Technical Report, ANL-82-75, Argonne National Laboratories, Argonne Illinois, December 1982.

[Lusk84]
Lusk, Ewing, and Ross A. Overbeek, "The Automated Reasoning System ITP," Technical Report, ANL-84-27, Argonne National Laboratories, Argonne Illinois, April 1984.

[McCarthy67]
McCarthy, John, "A basis for a mathematical theory of computation," in <Bratfort, P. and D. Hirschberg>, eds., Computer Programming and Formal Systems, North-Holland Publishing Co., Amsterdam, 1967.

[Moore82]
Moore, Robert C., "The Role of Logic in Knowledge Representation and Commonsense Reasoning," Proceedings AAAI-82, Pittsburgh Pennsylvania, 1982, pp. 428-433.

[Nguyen87]
Nguyen, Tin A., Walton A. Perkins, Thomas J Laffey, and Deanne Pecora, "Knowledge Base Verification", AI Magazine, Summer 1987, pp. 69-75.

[Pugh85]
Pugh, Kenneth, C Language for Programmers, Scott Foresman and Company, Glenview Illinois, 1985.

[Robinson65A]
Robinson, J., "A machine-oriented logic based on the resolution principle," Journal of the ACM (12), 1965, pp. 23-41.

[Robinson65B]
Robinson, J., "Automatic deduction with hyper-resolution," International Journal of Computer Mathematics (1), 1965, pp. 227-234.

[Robinson83]
Robinson, J., "The Generalized Resolution Principle," in <Jorg Siekmann and Graham Wrightson>, eds., Automation of Reasoning, Springer Verlag, Berlin, 1983.

[Sobell85]
Sobell, Mark G., A Practical Guide to UNIX System V, Benjamin/Cummings Publishing Co., Menlo Park, 1985.

[Sterling86]
Sterling, Leon, and Ehud Shapiro, The Art of PROLOG, The MIT Press, Cambridge Massachusetts, 1986.

[Turk85]
Turk, Andrew K., "Compiler Optimizations for the WAM," Technical Report, School of Computer and Information Science, Syracuse University, November 1985.

[Walker87]
Walker, Adrian, ed., MIchael McCord, John F. Sowa, and Walter G. Wilson, Knowledge Systems and PROLOG, Addison-Wesley, Reading Massachusetts, 1987.

[Warren77A]
Warren, David H. D., "Applied Logic - its use and implementation as programming tool.," Ph.D. Thesis, University of Edinburgh, Scotland, 1977.

[Warren77B]
Warren, David H. D., Luis Pereira, and Fernando Pereira, "PROLOG -The language and its implementation compared with LISP", Proceeding of the Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices ACM 12 (8); SIGART Newsletters ACM (64), August 1977, pp. 109-115.

[Warren80]
Warren, David H. D., "An improved PROLOG implementation which optimizes tail recursion," Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, 1980.

[Warren86]
Warren, David H. D., "Optimizing Tail Recursion in PROLOG," in <van Caneghem and Warren>, eds., Logic Programming and its Applications, 1986.

[Warren83]
Warren, David H. D., "An Abstract PROLOG Instruction Set," Technical Note 309, SRI International, Menlo Park CA, October 1983.

[Winston84]
Winston, Patrick Henry, Artificial Intelligence, Addison-Wesley, Reading Massachusetts, 1984.

[Winker81]
Winker, S., L. Wos, and E. Lusk, "Semigroups, antiautomorphisms, and involutions: a computer solution to an open problem, I," Mathematics of Computation 37 (156), pp.533-545, October 1981.

[Winker82]
Winker, S., "Generation and verification of finite models and counterexamples using an automated theorem prover answering two open questions." Journal of the ACM 29 (2), April 1982, pp. 273-284.

[Wojciechowski83]
Wojciechowski, W. S, and A. S. Wojcik, "Automated design of multiple-valued logic circuits by automated theorem proving techniques." IEEE Transactions on Computers, September 1983.

[Wos64]
Wos, L., D. Carson, and G. Robinson, "The unit preference strategy in theorem proving," Proceedings of the Fall Joint Computer Conference, Thompson Book Company, New York, 1964, pp. 615-621.

[Wos65]
Wos, L., D. Carson, and G. Robinson, "Efficiency and completeness of the set-of-support strategy in theorem proving," Journal of the ACM (12), 1965, pp. 536-541.

[Wos84]
Wos, Larry, et al, Automated Reasoning: Introduction and Applications, Prentice-hall, Inc., Englewood Cliffs New Jersey, 1984.

VITA

George Donald Herbert was born on September 13, 1948 in Chicago, Illinois. He received his primary education in Chicago and Elk Grove Village, Illinois. He received his secondary education at St. Viator High School in Arlington Heights, Illinois, where he received the Auxilium Latinum award for Latin scholarship and the school Mathematics award. He received his Bachelor of Science degree in Mathematics from the University of Illinois in Champaign-Urbana. He has worked for 17 years with J. M. Family Enterprises, mostly with their data processing subsidiary Carnett. He currently holds the position of Manager of Special Projects.

He married his wife, Sallie, in 1973, and they have three daughters: Roberta, age 9, Catherine, age 7, and Lyndon, age 4.

He has been enrolled in the Graduate School of the University of North Florida since August 1985. He is a member of the Institute of Electrical and Electronics Engineers and the Computer Society of the I.E.E.E. He is also a student member of the Association for Computing

Machinery, American Association for Artificial Intelligence and the American Mathematical Association. Recently, his paper, "Relating Two Forms of Ackermann's Function", was accepted for publication subject to revision by the American Mathematical Monthly.