TOWARD A THEOREM PROVING ARCHITECTURE

Sheue-Ling C. Lien

Computer Science
California Institute of Technology
4653:TR:81
CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science Department

Technical Report #4653

Toward a Theorem Proving Architecture

by

Sheue-Ling C. Lien

In Partial Fulfillment of the Requirements for the Degree of Master of Science

July, 1981

The research described in this report was sponsored by the Defense Advanced Research Project Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

Copyright, California Institute of Technology 1981.
TABLE OF CONTENTS

1. INTRODUCTION

2. LOGIC PROGRAMMING and THEOREM PROVING
   2.1 Conventional Programming Languages
   2.2 Predicate Logic and Horn Clauses
   2.3 Logic Programming
   2.4 Prolog Programming Language
   2.5 Algorithm = Logic + Control

3. UNIFICATION COMPUTATION
   3.1 Unification Computation
   3.2 Unification Algorithm
   3.3 Modification and Simulation

4. HARDWARE IMPLEMENTATION
   4.1 System Overview
   4.2 Equation-table Part
   4.3 Stack-memories Part
   4.4 Data-registers Part
   4.5 The Controller Part
   4.6 System Design

5. NONDETERMINISM
   5.1 Non-determinism
   5.2 Sequence of Procedure Invocations
   5.3 Scheduling of Procedure Calls

6. CONCLUSIONS
ACKNOWLEDGEMENT

APPENDIX
A. A Special Purpose Unification Algorithm
B. The Transformed Assembly Program

REFERENCES
Section 1 INTRODUCTION

The unification computation is a pattern-matching process operating on a set of expressions to search for the variable binding relations which turn this set of expressions into a singleton. The unification computation occurs at the very heart of most modern deduction algorithms. An enormous amount of unification computation has to be done during a deduction calculation or theorem proving computation. Unfortunately, this computation tends to be the most time consuming step during the calculation.

The subject of the research described in this thesis is the design of a chip, called UNIF chip, with VLSI tools, which executes unification computation. The major purpose for designing such an unification chip is to implement the unification computation on hardware such that, whenever the computation is needed, it can be executed in a high speed way. This unification chip can improve the execution efficiency of those systems where the unification computation takes the major part during the execution time.

Section 2 of this paper includes a discussion of logic programming, since the unification computation plays the central role in the logic programming. For those who are not familiar with logic programming, this section can supply them a better idea of this field. At the beginning of the section, we compare conventional programming with logic programming to distinguish differences and advantages of logic programming. Then, predicate logic, Horn clauses and the Prolog language are discussed separately. Prolog is a logic programming language whose main computation mechanism is based on the unification computation, which makes it an ideal language for discussion. Finally some examples written in Prolog are presented to demonstrate the unification computations involved during the solving of these problems.

Since the major function of the UNIF chip is to execute unification computation, the UNIF chip follows a unification algorithm during its execution. In section 3, we discuss an algorithm which unifies a set of expressions [Robinson 1965]. This algorithm, called the unification algorithm, is the basis of the microprogram stored in the controller of the UNIF chip. This algorithm constitutes the skeleton of the controller. The discussion in this section gives more idea about the unification computation and the unification algorithm, and clarifies how the UNIF chip executes its computation.
The system description of the UNIF is included in section 4 of this paper. In this section we look at the overall system, including its floor plan, block diagrams and function blocks. The UNIF chip is divided functionally into four parts: the controller, the data-registers, the stack-memories, and the equation-table. Each part is responsible for one major function. The controller is the heart of the chip, which controls the behavior of the whole chip. The controller is implemented as a PLA. The data-registers are responsible for storing data items temporarily during the computation. This part contains seven data-registers. The stack-memories hold multiple arguments of the expressions, replacing the software recursion with the hardware stack. The equation-table is a random access memory where expressions to be unified are stored. All the information is stored in this table and the final output result is also recorded in this table. The equation table is implemented as a standard part (RAM) external to the chip. The procedure for designing the UNIF, the VLSI tools which have been used are also described in this section.

In section 5 of this paper, we discuss the nondeterministic problems associated with the overall deduction procedure. There are two cases of nondeterminism. One is the nondeterminism on the sequence of procedure invocations. The other is the nondeterminism on the scheduling of procedure calls. Different decisions on the nondeterminism usually lead to different results.

In the final section, section 6, we give some conclusions on this research. A concurrent system is also presented to implement multiple UNIF chips on an array model. This model shows the idea of a concurrent deductive system.
Section 2 LOGIC PROGRAMMING and THEOREM PROVING

This section presents a brief introduction to logic programming, suggesting another view of the programming field. We start with the definition of predicate logic and clausal form, and discuss their syntax and semantics separately, since logic programming is based on predicate logic, where sentences are represented in clausal form. After that we talk about a practical logic programming tool, Prolog, to give more concrete idea about logic programming. A short introduction to Prolog programming language, including its syntax and semantics, is presented. Finally, some simple examples written in Prolog are carried through to illustrate the unification computation and the procedure invocation associated with each example during the execution time.

2.1 Conventional Programming Languages

Computer programming languages today are often referred to as being in the midst of a software crisis, as they are growing ever more complex but not stronger. Can programming languages be liberated from the conventional style which is based on the von Neumann model computer? This has been recently a lively topic for discussion [Winnograd 1979], [Backus 1978].

We say that conventional programming languages, while solving a problem, usually combine the knowledge about the problem with the way this problem is solved in the algorithm. To make this clear, we first think of an algorithm as composed by a logic component with a control component [Kowalski 1979]. The logic component specifies the knowledge to be used in solving the problem, and the control component determines the problem-solving strategies by means of which that knowledge is used. For instance, in sorting a list, there are several sorting algorithms like quick sort, bubble sort, binary sort, sequential sort, etc. We can think of those algorithms as composed of the same logic component but different control components, since they use different ways to do the same job [Van Ronden 1977].

However, it is more important to discuss separating the logic components with the control components. Because, if those two components can be separated, the efficiency of an algorithm can often be improved by improving solely its control component without
changing its logic component. Conversely, we can also improve its logic component without affecting its control component at all. In conventional languages, since both logic and control components are not separate, one might change the algorithm's meaning while trying to improve its execution efficiency. It means that unknown errors can be created in an unexpected situation. In contrast to this, logic and control components are separate in logic programming. It gets rid of the mutual interference of the two components as in conventional languages. It, therefore, reduces the fear of creating unexpected errors while one is trying to improve an algorithm's efficiency.

Besides this, conventional languages use variables to imitate the computer's storage cells; assignment statements to imitate its fetching and storing; and use control statements to control the execution sequence of the algorithm. So that, what is left to the execution mechanism is only the most rudimentary problem-solving capabilities. However, in logic programming, the solving strategies are decided by the executor instead of specified by the programmer. The execution mechanisms can provide more powerful problem-solving facilities of the kind provided by the intelligent theorem-proving systems. The programmer is relieved from specifying detail source codes and controlling execution sequence. It becomes much easier to write a program without having to control the execution sequence step-by-step through the whole algorithm. It also saves much time without having to specify the machine-level codes. And it lessens the occurrence of errors in programs. In this way, computer programs will be more often correct, more easily improved, and more readily adapted to new problems.

2.2 Predicate Logic, Horn Clauses

Before talking about logic programming, we discuss predicate logic and Horn clauses first, since predicate logic supports the semantics of logic programming and Horn clauses frame its syntax. This section is presented as an introductory part to logic programming.

Logic studies the relationship of implication between assumptions and conclusions. It is concerned not with the truth or falsity of the individual sentences, but with the relations between them. For example, the assumption that
"Helen is Mary's mother"

implies the conclusion that

"Mary is the child of Helen".

What is concerned about, in logic, is the implication relation that if the statement "Helen is Mary's mother" is true, then it implies that the statement "Mary is the child of Helen" is also true, but not the truth whether Helen is really Mary's mother.

The above statement can also be expressed in clausal form as

\[
\text{CHILD (Mary, Helen)} \leftarrow \text{MOTHER (Helen, Mary)}
\]

with the function name written in front of the atomic formula, followed by the sequence of names of individual to which the relation applies. The meaning of the clause is the same with before. \(\text{MOTHER (Helen, Mary)}\) means that Helen is the mother of Mary. The arrow mark in the center means the implication between the assumption part \(\text{MOTHER (Helen, Mary)}\) and the conclusion part \(\text{CHILD (Mary, Helen)}\).

In predicate logic, sentences are always represented in clausal form. A sentence in clausal form is a sentence represented in a set of clauses. A clause is an expression of the form

\[
\text{B1, B2, ..., Bm } \leftarrow \text{A1, A2, ..., An}
\]

where \(\text{B1, ..., Bm}\) and \(\text{A1, ..., An}\) are two sets of atomic formula with \(m \geq 0\) and \(n \geq 0\). The atomic formula \(\text{A1, ..., An}\) are the joint conditions (the assumption part) of the clause and \(\text{B1, ..., Bn}\) are the alternative conclusions (the conclusion part).

An atomic formula is an expression of the form \(p(t1, t2, ..., tk)\), where \(p\) is a \(k\)-ary predicate symbol and \(t1, ..., tk\) are terms. A term is either a variable, a symbol or an expression \(f(t1, t2, ..., tk)\) where \(f\) is an \(k\)-ary function symbol and the \(t1, ..., tk\) are terms.

The clause

\[
\text{B1, B2, ..., Bm } \leftarrow \text{A1, A2, ..., An}
\]
with variables \( x_1, x_2, \ldots, x_k \), is interpreted as "for all variables \( x_1, \ldots, x_k, B_1 \) or \( B_2 \) or \( \ldots \) or \( B_m \), if \( A_1 \) and \( \ldots \) and \( A_n \)", or "for all variables \( x_1, \ldots, x_k, B_1 \) or \( \ldots \) or \( B_m \) is implied by \( A_1 \) and \( \ldots \) and \( A_n \)".

In the special case where \( m = 0 \),

\[
\langle \langle - A_1, A_2, \ldots, A_n, \rangle
\]

we interpret it as "for no variables \( x_1, x_2, \ldots, x_k, A_1 \) and \( A_2 \) and \( \ldots \) and \( A_n \) are true".

and the special case where \( n = 0 \),

\[
B_1, B_2, \ldots, B_m \langle \langle - \rangle
\]

we interpret it as "for all variables \( x_1, x_2, \ldots, x_k, B_1 \) or \( B_2 \) or \( \ldots \) or \( B_m \) are true".

If both \( n = 0 \) and \( m = 0 \), it is a null clause. We treat it as a halt statement. Following is an example expressed in clausal form.

**Example: Appending List Program**

\[
(F1) \text{APPEND } [\text{NIL}, Y] \quad \langle \langle - \rangle
\]

\[
(F2) \text{APPEND } [\text{cons}(x, Y), Z, \text{cons}(x, W)] \quad \langle \text{APPEND } [Y, Z, W]\]

According to the explanation above, the first clause (F1) asserts that appending the null list NIL to any list Y always gets list Y. The second clause (F2) asserts that if appending a list Y to a list Z gets list W, then appending the list cons(x,Y) to Z will get cons(x,W).

**Horn Clauses:**

Horn clause is a subset of predicate logic, where each clause contains at most one conclusion atomic formula. The example mentioned above is also expressed in Horn clause form. There are totally four kinds of Horn clause, each one associated with an explanation. It is very useful to interpret a Horn clause as a procedure declaration.

1. \( R \quad \langle \langle - \rangle \quad A_1, A_2, \ldots, A_n \quad (\text{where } m = 1, n \not= 0) \)
We can interpret this clause as a procedure declaration. The conclusion B is interpreted as the procedure name and the conditions A1,...,An are interpreted as the procedure body. The procedure body consists of a set of procedure calls A1's, and the procedure name B is implied by this set of procedure calls.

\[(2) \quad B \leftarrow (\text{where } m=1, n=0)\]

This clause is interpreted as an assertion of fact. It can also be thought of as a procedure with empty body.

\[(3) \quad \leftarrow A1,A2, ..., An (\text{where } m=0, n<>0)\]

The third clause is interpreted as a goal statement which asserts the goal of successfully executing all the procedure calls A1,..,An.

\[(4) \quad \leftarrow (\text{where } m=0, n=0)\]

Clause (4) is a null clause. It can be regarded as a satisfying goal statement.

2.3 Logic Programming

Predicate logic codifies rational thought. Recently its potential as a language was studied and developed into a practical logic programming language, based upon the interpretation of sentences in predicate logic as programs; the interpretation of derivations as computations; and the interpretation of proof procedures as feasible executors of predicate logic programs.

According to the procedure interpretation, resolution can be treated as procedure invocation.

\[(C1) \quad \leftarrow A1,A2, ..., A_{i-1}, A_i, A_{i+1}, ..., A_n,\]
\[(E1) \quad B \leftarrow B1,B2, ..., B_m\]

For a given goal statement (C1) and a procedure (E1), the procedure name B matches the selected procedure call Ai under some general substitution set G. We say that the procedure B is invoked and the resolution derives a new goal statement (C2), by
substituting the procedure body $B_1,\ldots,B_m$ with the procedure call $A_i$ under the corresponding substitution set $G$.

$$(C2) \quad \leftarrow (A_1,A_2,\ldots,A_{i-1},B_1,B_2,\ldots,B_m,A_{i+1},\ldots,A_n)G.$$  

We use the appending-list example mentioned before to illustrate the substitution set, the activation of matching clauses, the procedure invocation, and the derivation of new goal statements.

Example 1: Appending List Program

$$(F1) \text{APPEND}[\text{NIL},Y,Y] \leftarrow$$

$$(F2) \text{APPEND}[\text{cons}(x,Y),Z,\text{cons}(x,W)] \leftarrow \text{APPEND}[Y,Z,W]$$

$$(C1) \leftarrow \text{APPEND}[\text{cons}(a,\text{cons}(b,\text{NIL})),\text{cons}(d,\text{NIL}),V].$$

$(C1)$ is the goal statement for computing appending a list $\text{cons}(a,\text{cons}(b,\text{NIL}))$ to the list $\text{cons}(d,\text{NIL})$. Goal statement $(C1)$ cannot match with the first statement $(E1)$, since its first argument is not NIL. Comparing $(C1)$ with $(E2)$, we can see that under the following variable binding relations

$$
x \longrightarrow a,
Y \longrightarrow \text{cons}(b,\text{NIL}),
Z \longrightarrow \text{cons}(d,\text{NIL}),
V \longrightarrow \text{cons}(a,U),$$

the two expressions $\text{APPEND}[\text{cons}(a,\text{cons}(b,\text{NIL})),\text{cons}(d,\text{NIL}),V]$ and $\text{APPEND}[\text{cons}(x,Y),Z,\text{cons}(x,W)]$ are turned into the same expression. We call these variable binding relations the substitution set $G$, where $G = \{ a/x, \text{cons}(b,\text{NIL})/Y, \text{cons}(d,\text{NIL})/Z, \text{cons}(a,U)/V \}$.

In this case, we say that $(C1)$ matches the second statement $(F2)$, under the substitution set $G$, and the procedure $(F2)$ is invoked. The resolution derives a new goal statement $(C2)$ by substituting the procedure body into the goal statement under the above substitution set.
(C2) \( \leftarrow \text{APPEND}[\text{cons}(b,\text{nil}), \text{cons}(d,\text{nil}), U] \).

(C2) still matches the second statement (F2) under the substitution set \( \{ b/x, \text{nil}/Y, \text{cons}(d,\text{nil})/Z, \text{cons}(b,U')/U \} \), and derives the new goal statement

(C3) \( \leftarrow \text{APPEND}[\text{nil}, \text{cons}(d,\text{nil}), U'] \).

The goal statement (C3) matches the first statement (F1) under the substitution set \( \{ \text{cons}(d,\text{nil})/Z, Z/U' \} \).

Since (F1) is a procedure with empty body, the new goal statement derived is a satisfying halt statement. The computation is thus finished.

Formally speaking, given a set \( S \) of Horn clauses and an initial goal statement \( C_1 \) in \( S \), the computation is a sequence of goal statements \( C_1, C_2, \ldots, C_n \) where \( C_{i+1} \) is derived from \( C_i \) through the procedure invocation. Each time, a new goal statement \( C_{i+1} \) is derived from the old one \( C_i \) by first looking for a procedure in \( S \) whose name matches with some selected procedure call in \( C_i \) and then substituting the body of the procedure into \( C_i \). If a final halt statement can be derived the computation is "successful", since it ends with a satisfying halt statement. Otherwise, it terminates without success if the selected procedure call in the end goal statement \( C_n \) matches no procedure in \( S \).

The generation and application of new goal statement, during procedure invocation, has to do with the transmit of input and output. The part of substitution which affects variables in the original procedure calls \( A_1, \ldots, A_n \) transmits output. The part of substitution which affects variables in the new procedure calls \( B_1, \ldots, B_m \) transmits input.

Instantiation of variables occurring in the procedure \( B \) by terms occurring in the procedure call \( A_i \) corresponds to passing input from \( A_i \) to the procedure body \( B_1, \ldots, B_m \) through the procedure name \( B \). The instantiated procedure body \( (B_1, B_2, \ldots, B_m)G \) is the result of input transfer. Instantiation of variables occurring in the procedure call \( A_i \) by terms occurring in the procedure name \( B \) corresponds to passing output back to the procedure call \( A_i \) which distributes to the remaining procedure calls \( A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \). The instantiated procedures \( (A_1, A_2, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n)G \) is the result of output transfer.
Next we go through the calculation of the factorial program to show the input/output transmission. We will see that outputs are transmitted throughout calculation. Partial outputs accumulate and determine successively the approximation to the final result. The approximations are generated whether or not the computation eventually succeeds.

Example 2: Factorial Program

(E1) FACTORIAL [ 0, s(0) ] <-
(E2) FACTORIAL [ s(x), u ] <- FACTORIAL [ x, v ],
    TIMES [ s(x), v, u ]

We start with the goal statement (C1), which asserts the goal of computing the factorial of 2:

(C1) <- FACTORIAL [ s(s(0)), y ]

The goal statement (C1) matches (E2) under the substitution set

G = { s(0)/x, y/u },

and derives a new goal statement (C2) by substituting variable x in (E2) with s(0) and u with y

(C2) <- FACTORIAL [ s(0), v ], TIMES [ s(s(0)), v, y ]

The goal statement (C2) still matches the second statement (E2). With one more procedure invocation with (E2), under the substitution set { 0/x, v/v' }, resolution derives a new goal statement

(C3) <- FACTORIAL [ 0,v' ], TIMES [ s(s(0)), v, v' ],
    TIMES [ s(s(0)), v, u ]

Finally the procedure call FACTORIAL [ 0,v' ] matches with the first statement (E1) under the substitution set { 0/x, v'/s(0) } and derives the halt statement.

We get the final result that "the factorial of 2 is s(s(0))", by accumulating the outputs transmitted out during the above computation process:
\[ v := s(0) \times v' \\
:= s(0) \times s(0) \\
:= s(0); \\
u := s(s(0)) \times v \\
:= s(s(0)) \times s(0) \\
:= s(s(0)); \\
y := u \\
:= s(s(0)); \]

In the logic interpretation, computations are resolution derivations. The end goal statement of the computation is a logical consequence of the original set of specifications, which either proves the original assumption or disproves it with a demonstration of refutation. Let's look at the interpretation for the goal statement first.

\[(C1) \quad \leftarrow A1, A2, \ldots, An\]

The interpretation for this statement, as explained in the clause (3) in the predicate logic section, is

"for no variables x1,..,xk, A1 and .. and An are true"

As we know, the goal statement is the one which asserts the problem to be solved. According to the interpretation above, it is assumed that there exist no variables which make the goal statement true. It is quite a strong assumption that entirely refutes the possibility of the existence of A1,..,An. However, the computation is a sequence of resolution according to the logic relations you submit, trying to derive an end goal statement which either proves or disproves this assumption.

If the final goal statement arrives at a set of variables x1,..,xk which make A1,..,An true, the computation is successful. We say that the result is a refutation and demonstration of the unsatisfiability of the original assumption. Since originally, it is assumed that there exists no variables x1,x2, .. ,xk which make A1 and .. and An exist. At this moment the answer to the problem is found, the set of variables x1,..,xk is the result which satisfy the goal statement.
On the other hand, if the derivation is not successful, it means that the original assumption is correct; there exist no variable set $x_1..x_k$ which can make $A_1..A_n$ exist. Therefore, in this case no answer is obtained.

In the factorial example, goal statement (C1) can be interpreted as "there exists no any integer $y$ which is the factorial of 2". However, the final result refutes this assumption by showing the existence of the factorial of 2, i.e. $s(s(0))$.

It is also worthy to notice that there is no definite distinction between the input/output parameters in the predicate programs. The difference lies on the context in which procedure is invoked. Any subset of the parameter list can be treated as input, and the remaining parameters as outputs. In the APPEND example:

\[
\begin{align*}
(F1) & \text{ APPEND [ nil,Y,Y] } \leftarrow \\
(F2) & \text{ APPEND [ cons(x,Y),Z,cons(x,W)] } \leftarrow \text{ APPEND [Y,Z,W]} \\
(C1) & \leftarrow \text{ APPEND [ cons(b,nil),cons(d,nil),u ]},
\end{align*}
\]

With the goal statement (C1), the third parameter $u$ is treated as an output parameter for storing the result, and the second input list cons(d,nil), and the first and second parameters are treated as input parameters.

However, if with the goal statement (C2)

\[
(C2) \leftarrow \text{ APPEND [ x, y, cons(a,cons(b,cons(c,nil)))]},
\]

the third parameter $u$ is used as an input parameter instead of output. The first and second parameters $x$ and $y$ are treated as output parameters for storing any two lists which constitute the third flat cons(a,cons(b,cons(c,nil))).

### 2.4 Prolog Programming Language

Prolog is the first logic programming system, which was developed at the University of Marseille as a practical tool of logic programming [Roussel 1975]. It is a simple but powerful language. A Prolog compiler written in Prolog was implemented at the University of Edinburgh by Warren, Pereira and Pereira [1977]. It showed that Prolog compiler
executed LISP-like programs as efficiently as compiled LISP.

As a programming language, Prolog is entirely user-oriented. It differs from existing high-level languages in that it possesses no features which are meaningful only in machine-level terms. It differs from functional language like LISP in that it derives from the normative study of human logic, rather than from investigation into the mathematical logic of functions. Prolog has simple syntax, clear and declarative semantics. From a user's point of view, its major attraction is ease of programming. Prolog is also an ideal language for programming in the field of artificial intelligence. There have already been several very complex programs written in Prolog in a very concise way [Shapiro 1980].

Syntax and semantics of Prolog is based on the interpretation of predicate logic as a programming language and Horn clauses as a procedure declaration. (Discussion of predicate logic and Horn clauses is included in section 2.3 and section 2.4). A Prolog program is composed of a set of Horn clauses which express the logic relations of the algorithm, and activated by an initial goal statement which specifies the problem to solve. The basic computation mechanism in Prolog is the unification computation.

Next, two factorial examples written in Algol and Prolog are presented for illustrating the distinction between logic programming and conventional programming.

**ALGOL EXAMPLE:**

```plaintext
integer PROCEDURE factorial(n); integer n;
BEGIN integer j, fact; fact := 1;
   for j:=2 step 1 until n do fact:=fact*j;
   factorial:=fact;
END of factorial;
```

**PROLOG EXAMPLE:**

```plaintext
(E1) FACTORIAL [0,1] <-
(E2) FACTORIAL [s(x),u] <- FACTORIAL [x,v],
     TIME[v,s(x),u]
```

The meaning of the Algol example is quite obvious. No explanation is needed. The interpretation for the Prolog example is as follows:

(E1): Factorial of zero is one.
(E2): If the factorial of x is v and v times successor of x is u,
then the factorial of the successor of $x$, $s(x)$, is $n$.

The differences between them are striking. In order to compute the factorial, the Algol example uses the assignment statements to imitate data storing and fetching and a FOR-loop statement to control the computer cycling through loops and calculating the value of the factorial of $n$. However, the Prolog example specifies only the definition of the factorial without describing how this algorithm should be executed. From these two examples, we can see that, unlike the conventional program, a logic program specifies only the meaning of the algorithm and leaves the execution sequence to the run-time system.

Both Lisp and Prolog are interactive languages founded on formal mathematical basis - Lisp on Church's lambda calculus, Prolog on Horn clauses. Comparing with Lisp, Prolog is similar to this list-processing language in the way that data in Prolog can be represented in terms or relations. However, Prolog uses the pattern-matching mechanism to operate on data structures instead of the selectors and constructors in Lisp.

Another feature of Prolog is that, like Lisp, programs and data are identical in form. Clauses can usefully be employed for expressing data. Database systems thus become another appreciable application of Prolog. It's able to implement a database directly in Prolog and use unification for data-items searching, instead of simple pattern matching. The following example illustrates the identity of program and data in Prolog, showing its potential as a natural medium for database system.

```
student(John,m,20).  student(Jack,m,10).
student(Mary,f,20).  student(Linda,f,22).
student(Bruce,m,20). student(Helen,f,16).
student(Jen,f,21).    student(Amy,f,19).
student(Dick,m,23).
```

```
MALE-ADULT(name,sex,age) <- student(name,sex,age),
   greater(age,20),
   sex=type(sex,m)
```
This database of unit clauses stores the information about students, including their names, sex, and ages. The procedure MALE-ADULT looks for the student whose age is greater than 20 and whose sex is male. This procedure returns data items of all those students who are qualified.

2.5 Algorithm = Logic + Control

Conventional programs combine the logic of the algorithms with the control of the execution of the algorithms. Logic programs are more abstract. They control neither the order in which different procedures are invoked, nor the order in which procedure calls are executed. We can represent the analysis of an algorithm into a "logic component", which defines the logic of the algorithm, and a "control component", which specifies the manner in which definitions of algorithm are used [Kowalski 1979].

ALGORITHM = LOGIC + CONTROL

Logic programs express only the logic component of the algorithm. The control component is determined by the program executor, either following its own control decision or following the control instructions provided by the programmer. Therefore, logic programs can be thought about as an entirely machine independent and user-oriented language.

The separation of logic from control has several advantages:

(1) Algorithms can be constructed and generated by defining its logic component before control component, such that, the algorithm can be verified by applying deductive rules to their logic component alone.

(2) Efficiency of the algorithms can be improved by simply modifying their control components without changing their logic components, or by transforming their logic components into more efficient ones without changing their control components.

(3) The control component can be expressed separately, or it can be determined by the program executor. It allows the programmer ability of specifying about program execution. On the other hand, the determination of control by the program executor relieves the programmer of the need to specify control together with logic.
Section 3  UNIFICATION COMPUTATION

Computational logic is a branch of artificial intelligence which deals with how to make machines do deduction efficiently. There are more and more applications of computational logic: in robot control, question-answering, program-writing, program verification, etc. These applications are very useful to today's world. However, the current deductive power of those algorithms is still quite weak. In order to develop the computational logic into a genuinely strong and useful science, it is important to focus on the unification computation. Since the unification computation plays the central role of most modern deduction algorithms.

In this section, we are going to discuss a unification algorithm [Robinson 1965]. This is a general purpose unification algorithm which takes a finite collections of finite sets of expressions as input and calculates the substitution set which unifies these sets of expressions. This algorithm is based upon as the skeleton of the controller in the Unif-chip.

Following this algorithm another special purpose and chip-simulation algorithm is derived, through modifying the original one. The newly derived algorithm takes only two expressions as input and computes the substitution set which unifies this pair of expressions. This algorithm is transformed into a microprogram, which is stored in the UNIF-chip.

3.1 Unification Computation

Following is a simple example showing how two expressions are unified.

A system $P$, where $P = \{ P_1, P_2 \}$,

$P_1 = \{ f(x, g(y)), z \}$

$P_2 = \{ h(n, b), h(p(q), r) \}$.

If in the system $P$, we substitute variable $z$ with $f(x, g(y))$

variable $a$ with $p(q)$.
and variable \( r \) with \( b \),

P1 and P2 are changed into singleton sets after substitution.

\[
\begin{align*}
P1 &= \{ f(x, g(y)) \}, \\
P2 &= \{ h(p(q), b) \}.
\end{align*}
\]

In this case, we say that the system P is unifiable under the variable binding set G, where \( G = \{ f(x, g(y))/z, p(q)/a, b/r \} \). Since G turns each expression in P1 (or P2) into the same expression.

However, for the system \( P' = \{ P3 \} \), where

\[
P3 = \{ r(x, t(x)), r(t(x), x) \},
\]

there exists no variable binding set which change \( P' \) into a singleton. The system \( P' \) is said to be non-unifiable.

We call the process of changing a system into a singleton the unification process or the unification computation; and the variable bindings set the unification set.

Formally speaking, a substitution \( \{ t_1/x_1, t_2/x_2, \ldots, t_n/x_n \} \) is an operation which can be performed on expressions by replacing each occurrence of variable \( x_i \) in the expression with the corresponding term \( t_i \). For a finite collection \( P = \{ P1, P2, \ldots, Pn \} \) of finite sets of expressions, each finite set \( Pi \) is made up of either terms or atoms. Each expression either consists of a symbol and a list of terms as arguments, or is a variable. We say that a substitution \( G \) unifies \( P \) if for each \( i, i = 1, \ldots, n \), \( G \) turns each expression in \( Pi \) into the same expression. We express \( EG \) as the result of performing the substitution \( G \) on the expression \( E \). If \( S \) is a set of expressions, then \( SG \) is the resulting set of \( \{ EG / E \in S \} \).

There are three cases of variable binding during the unification:

(1) variable to variable, \( v1 \rightarrow v2 \)

This is the most simple case where the substitution set can be either \( \{ v1/v2 \} \) or \( \{ v2/v1 \} \).
(2) variable to function, \( v_1 \rightarrow f \)

If the variable \( v_1 \) is not contained in the function \( f \), then \( v_1 \) can be binded to function \( f \), with the substitution set \( \{ f/v_1 \} \). However, if \( v_1 \) is contained inside \( f \), then \( x \) can not be unified with \( f \).

(3) function to function, \( f_1 \rightarrow f_2 \)

If \( f_1 \) and \( f_2 \) are different functions, then they can not be unified, since functions can not be substituted. However, if \( f_1 \) and \( f_2 \) are of same function name with different arguments,

\[
\begin{align*}
    f_1 &= f(t_1, t_2, \ldots, t_k) \\
    f_2 &= f(r_1, r_2, \ldots, r_k).
\end{align*}
\]

Whether they can be unified or not depending on the success of the unification of the argument pairs \( ti \) with \( ri \), (i=1...k), correspondingly. If any one pair can not be unified, then the unification of \( f_1 \) with \( f_2 \) is failed.

3.2 Unification Algorithm

The central idea of the unification is to 'shrink' the given system \( P \) into a set of singletons, through the successive substitutions. Each substitution equates two corresponding subexpressions which need to be made identical if \( P \) is to be unified. Therefore, a unification algorithm will allocate these two subexpressions and perform the necessary substitution through each cycle. Following steps describe systematically the sequence of the execution of the unification algorithm.

Step1: Given system \( P = \{ P_1, P_2, \ldots, P_n \} \) as input, set \( j = 0 \), Go = identity set, goto step 2;

Step2: If \( P_{ij} \) is a singleton for each \( i, i = 1, \ldots, n \), then STOP with output \( G = G_j \), else goto step 3;

Step3: Let \( k \) = the earliest number such that \( P_k G_j \) is not a singleton. Let \( E, F \) be any two expressions in \( P_k G_j \). Scan \( E \) and \( F \) in parallel, from left to right, to locate the
leftmost position in which \( E \) and \( F \) do not have identical symbols. Let \( e \) and \( f \) be the subexpressions of \( E \) and \( F \), respectively, which begin in that position. If neither \( e \) nor \( f \) is a variable then STOP, else goto step 4; Step 4: If one of \( e \) and \( f \) is a variable which is properly contained in the other then STOP, else goto step 5;

Step 5: Choose \( x \) and \( t \) so that \( \{ x, t \} = \{ e, f \} \). Set \( G_{j+1} = G_j \{ t/x \} \), \( j = j+1 \), goto step 2;

End;

Through each cycle, \( 1 \leq k \leq n \), \( P_{kG_j+1} = (P_{kG_j}) \{ t/x \} \), We can update the sets \( P_{1G_j}, \ldots, P_{nG_j} \) by performing the substitution \( \{ t/x \} \) on them, i.e. locating each occurrence of \( x \) in each expression, and replacing it by an occurrence of \( t \), until the final step is arrived at. However, if the algorithm stops in either step 3 or step 4, the unification fails and the given system \( P \) can not be unified.

Besides the efficiency of the unification algorithm, how to store expressions in the computer memory is also very important. Since we want to minimize the amount of scanning, copying and rearranging which have to be done during the computation. The process must be capable of detecting differences between expressions as quickly as possible.

Table 3-1 with eight fields "SYMBOL, VBLE, ARGS, ARITY, SUBST, TERM, EQUALS, and PARTS ", shows how the expressions are represented. The first column stores the symbols of the expressions. VBLE is a boolean array. VBLE indicates whether the corresponding symbol is a variable. ARGS stores its arguments list, and ARITY counts the number of elements contained in the arguments list. SUBST is also a boolean array, indicating whether the symbol is substituted or not. If it is, the substituting term is stored in the field TERM. Column PARTS have the meaning that expression\( (i) \) is to be made identical with the expression\( (\text{PARTS}(i)) \). During the computation, the column EQUALS is modified such that it always has the meaning:

\[
\text{if EQUALS}(i) = \text{EQUALS}(j) \text{ then EXPRESSION}(i) = \text{EXPRESSION}(j).
\]

Each field in this table has one column, with the exception of ARGS. ARGS has as many columns as the arity of that symbol. The way in which these expressions are
represented can be explained by the function `EXPRESSION` as follows.

```plaintext
EXPRESSION (i) = if SUBST(i) then EXPRESSION( TERM(i) )
    else SYMBOL(i) / ARGLIST( i, ARITY(i) );

ARGLIST (i, j) = if j=0 then NIL
    else ARGLIST(i,j-1)/EXPRESSION(ARGS(i,j));
```

For example, a system \( P = \{ P1, P2 \} \), where

\( P1 = \{ f(x), g(x,y), z \} \),

and \( P2 = \{ h(z,y), h(f(a,b), f(d,c)) \} \)

can be represented in TABLE 3-1 as defined by the function `EXPRESSION` as follows:

```
<table>
<thead>
<tr>
<th>symbol</th>
<th>args</th>
<th>arity</th>
<th>vble</th>
<th>subst</th>
<th>term</th>
<th>equals</th>
<th>parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f</td>
<td>2</td>
<td>3</td>
<td>F</td>
<td>F</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>0</td>
<td>T</td>
<td>F</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>g</td>
<td>2</td>
<td>4</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>y</td>
<td>0</td>
<td>T</td>
<td>F</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>z</td>
<td>0</td>
<td>T</td>
<td>F</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>h</td>
<td>5</td>
<td>a4</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>h</td>
<td>8</td>
<td>11</td>
<td>F</td>
<td>F</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>9</td>
<td>10</td>
<td>F</td>
<td>F</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>a</td>
<td>0</td>
<td>T</td>
<td>F</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>0</td>
<td>T</td>
<td>F</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>f</td>
<td>12</td>
<td>13</td>
<td>F</td>
<td>F</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>d</td>
<td>0</td>
<td>T</td>
<td>F</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>c</td>
<td>0</td>
<td>T</td>
<td>F</td>
<td>13</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>
```

---

*TABLE 3-1:

\( P = \{ P1, P2 \} \) where \( P1 = \{ f(x), g(x,y), z \} \),

\( P2 = \{ h(z,y), h(f(a,b), f(d,c)) \} \)
TABLE 3-1 is called the "equation table" of the system P. Each row of the table represents one expression, row 1 represents $\text{EXPRESSION}(i)$. The following tree structure representation of the system $P$ makes the $\text{EXPRESSION}$ clearer.

$$P_1 = \{ f(x), g(x,y), z \}, \quad P_2 = \{ h(z,y), h(f(a,b), f(d,c)) \}$$

```
  f   g   z   h
  / \ / \ / \
 x   x   y   z   y   f  f
  / \ / \ / \ / \  
 a   b   d   c   
```

Following is the unification algorithm. The whole algorithm is carried out by executing the function $\text{UNIFY}$. This function yields TRUE or FALSE as a result, according to whether the system represented in the table is unifiable or not. At the same time the table is modified as a side-effect of the computation. If the item $\text{SUBST}(i)=\text{TRUE}$, it means that $\text{EXPRESSION}(\text{term}(i))$ is to be substituted for this variable.

```plaintext
function UNIFY:boolean;
  label 1;
  var j:integer;
begin
  j:=1;
  1: if j>n then UNIFY:=true
     else if part(j)=j then begin
           j:=j+1; goto 1;
     end
     else if EQUATE(j,part(j)) then begin
           j:=j+1; goto 1;
     end
     else UNIFY:=false;
end  (function UNIFY)

function EQUATE(i,j:integer):boolean;
  label 1;
begin
  1: if equals(i)=equals(j) then EQUATE:=true
     else if subst(i) then begin
           i:=term(i); goto 1; end
     else if subst(j) then begin
           j:=term(j); goto 1; end
     else if symbol(i)=symbol(j)
           then EQUATE:=EQUATEARGS(i,j)
end  (function EQUATE)
```
else if vble(i) then begin
  if OCCUR(i,j) then EQUATE:=false
  else begin subst(i):=true;
     term(i):=j;
     IDENTIFY(i,j);
     EQUATE:=true;
   end;
end;

else if vble(j) then begin
  if OCCUR(j,i) then EQUATE:=false
  else begin subst(j):=true;
     term(j):=i;
     IDENTIFY(j,i);
     EQUATE:=true;
   end;
end;
else EQUATE:=false;
end;  \{function EQUATE\}

function EQUATEARGS(i,j:integer):boolean;
label 1;
var k:integer;
begin  k:=arity(i);
  1: if k=0 then begin IDENTIFY(i,j);
     EQUATEARGS:=true;
     end
  else if EQUATE(args(i,k),args(j,k))
     then begin k:=k-1;
     goto 1;
     end
  else EQUATEARGS:=false;
end;  \{function EQUATEARGS\}

function OCCUR(i,j:integer):boolean;
label 1;
begin
  1: if subst(i) then begin j:=term(j);
     goto 1;
     end
  if arity(j)=0 then
     if i=j then OCCUR:=true
     else OCCUR:=false
  else OCCUR:=OCCURARGS(i,j);
end;  \{function OCCUR\}
function OCCURARGS(i,j:integer):boolean;
label 1;
var k:integer;
begin
  \[ k := \text{arity}(i); \]
  1: if \( k = 0 \) then OCCURARGS := true;
  else if OCCUR(i,\text{args}(j,k))
    then OCCURARGS := true
  else begin
    \[ k := k - 1; \]
    goto 1;
  end;
end; \{function OCCURARGS\}

procedure IDENTIFY(i,j:integer);
label 1,99;
var k,e,f:integer;
begin
  \[ k := 1; e := \text{equals}(j); f := \text{equals}(i); \]
  if \( e = f \) then goto 99;
  1: if \( k > n \) then goto 99;
  if \( \text{equals}(k) = e \) then begin
    \text{equals}(k) := f;
    \[ k := k + 1; \]
    goto 1;
  end;
end; \{function IDENTIFY\}

All the functions and procedures above are written in PASCAL programming language.

Function UNIFY is quite simple. It consists of computing EQUATE(j,part(j)) for each row.

Function EQUATE is the heart of the program. It possesses the side-effect of modifying the equation table. If the value of EQUATE(i,j) is true, then during its evaluation, the equation-table will have been modified such that, after evaluation is completed, EXPRESSION(i) = EXPRESSION(j). Besides, EQUALS will have also been modified. Function EQUARARGS is a recursive execution of function EQUATE on all the arguments of the symbol currently being executed. Function OCCUR is for checking whether a variable is properly contained in the expression which is to be substituted for this variable.

After the execution of unification algorithm, the equation-table in TABLE 3-1 is modified as shown in TABLE 3-2. The SUBST field of rows 4,5,9,10 in the TABLE 3-2 are filled. From the TERM field of the corresponding rows, we can find the substitution relations.
TERM(4)=4, TERM(5)=1, TERM(9)=2, TERM(10)=3.
a <- x;  b <- g(x,f(d,c));
y <- f(d,c);  z <- f(x,g(x,f(d,c)));

Therefore, the substitution set which unifies system P is
G = \{ x/a.f(d,c)/y/g(x,f(d,c))/b.f(x,g(x,f(d,c)))/z \}

The EQUALS of rows 1,2,3,5,6 and 11 are also changed. We have
EQUALS(1) = 8,  EQUALS(2) = 9,
EQUALS(3) = 10,  EQUALS(5) = 8,
EQUALS(6) = 7,  EQUALS(11) = 4,

After the substitution of set G, we can see that
E1 G = E8 G = E5 G,  E9 G = E2,
E6 G = E7 G,  E4 G = E11.

**************  TABLE 3-2  **************

P = \{ P1, P2 \} where
P1 = \{ f(x,g(x,y), z) \},
P2 = \{ h(z,y), h( f(a,b), f(d,c)) \}

<table>
<thead>
<tr>
<th>symbol</th>
<th>args</th>
<th>arity</th>
<th>vble</th>
<th>subst</th>
<th>term</th>
<th>equals</th>
<th>parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f</td>
<td>2</td>
<td>F</td>
<td>F</td>
<td>8</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>g</td>
<td>2</td>
<td>F</td>
<td>F</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>0</td>
<td>T</td>
<td>(T)</td>
<td>10</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>y</td>
<td>0</td>
<td>T</td>
<td>(T)</td>
<td>11</td>
<td>(4)</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>0</td>
<td>T</td>
<td>(T)</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>h</td>
<td>5</td>
<td>F</td>
<td>F</td>
<td>7</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>h</td>
<td>8</td>
<td>F</td>
<td>F</td>
<td>(7)</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>9</td>
<td>F</td>
<td>F</td>
<td>(8)</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>a</td>
<td>0</td>
<td>T</td>
<td>(T)</td>
<td>2</td>
<td>(9)</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>0</td>
<td>T</td>
<td>(T)</td>
<td>3</td>
<td>(10)</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>f</td>
<td>12</td>
<td>F</td>
<td>F</td>
<td>4</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>d</td>
<td>0</td>
<td>T</td>
<td>F</td>
<td>(12)</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>c</td>
<td>0</td>
<td>T</td>
<td>F</td>
<td>(13)</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>
We summarize all the substituted expressions in TABLE 3-3 as follows:

| P = { P1, P2 } where P1 = { f(x, g(x, y), z), P2 = {h(z, y), h( f(a, b), f(d, c))} |
|-----------------|-----------------|
| ( E1 )          | ( E1 0 )        |
| 1 f (x, g(x, y))| f (x, g(x, f(d, c))) |
| 2 x             | x               |
| 3 g (x, y)      | g (x, f(d, c))  |
| 4 f(d, c)       | f(d, c)         |
| 5 z             | f (x, g(x, f(d, c))) |
| 6 h (z, y)      | h(f (x, g(x, f(d, c))), f(d, c)) |
| 7 h (f(a, b), f(d, c)) | h(f (x, g(x, f(d, c))), f(d, c)) |
| 8 f (a, b)      | f (x, g(x, f(d, c))) |
| 9 a             | x               |
| 10 b            | g(x, f(d, c))   |
| 11 f (d, c)     | f (d, c)        |
| 12 d            | d               |
| 13 c            | c               |

### 3.3 Modification and Simulating

After having explained the unification algorithm next we discuss a special purpose algorithm. The previous algorithm is a general-purpose one, which unifies sets of expressions. We will simplify this algorithm to a special purpose one which unifies only a pair of expressions. The new algorithm and the old one are very similar except that the EQUALS and PARTS fields in the original algorithm are omitted. Since there are only one set of expressions to be unified, these two fields are not necessary any more.
This simplified unification algorithm is also written in PASCAL. It is included in Appendix A. It unifies two expressions following the same steps as before. However, instead of comparing by field EQUALS, it compares two expressions according to their addresses and TERM field.

Following this simplified algorithm, we derive another algorithm written in SIMULA, which is closer to the hardware implementation. The distinction is that it replaces the recursive features with CLASS STACK. Whenever we need to equate arguments by recursively calling the function EQUIATE, we can first push all the arguments into a stack-memory. We then equate these arguments one by one, popping the arguments out of the stack memory until there is no more arguments exist.

CLASS STACK;
begin REF(member) first;
    integer procedure length;
    begin REF(member)x; integer j;
        x:=first;
        while x /= none do
            begin j:=j+1; x:=x.next; end;
        length:=j;
    end;
    REF(member) procedure pop;
    begin REF(member)x;
        if first /= none then begin
            x:=first; first:=x.next;
            x.next:=none; end;
        pop:=x.data;
    end; end;

CLASS MLMBLH;
begin REF(member) next; integer data;
    procedure push(stk); REF(stack)stk;
    begin if stk /= none then begin
        next:=stk.first;
        stk.first:=this member;
    end;
    end;

    procedure loadstack(i,j); integer i,j;
    begin REF(member)x; integer k;
        ...
for k:=1 step 1 until arity(l) do
    begin
      x:= new member; x.data:=arg(k,l);
      x.push(rstack);
      x:= new member; x.data:=arg(k,j);
      x.push(istack);
    end; end;

procedure loadoccstack(m); integer m;
    begin
      REF(member)x; integer k;
      for k:=1 step 1 until arity(m) do
        begin
          x:= new member; x.data:=arg(k,m);
          x.push(occstack);
        end; end;

boolean procedure OCCUR(i,j); integer i,j;
    begin
      boolean stopped;
      while stopped do
        begin
          while subst(j) do j:=term(j);
          if arity(j)=0 then
            begin if i=j then begin OCCUR:=true;
              stopped:=true;
              end
            else if occsk=0 then begin
              OCCUR:=false; stopped:=true;
              end
            else begin occsk:=occsk-1;
              j:=occstack.pop;
            end;
          end
        else begin
          LOADOCCSTACK(j);
          occsk:=occsk+arity(j);
          j:=occstack.pop;
          occsk:=occsk-1;
        end;
      end; end;
    end of while loop;
  end of procedure OCCUR;

boolean procedure UNIFY(i,j); integer i,j;
    begin
      boolean stopped;
      while stopped do begin
        while subst(i) do i:=term(i);
        while subst(j) do j:=term(j);
      end;
    end;
if i = j then begin if sk=0 then begin
    unify:=true; stopped:=true;
    end
    else begin sk:=sk-1;
    i:=rstack.pop; j:=lstack.pop;
    end;
end
else if symbol(i)=symbol(j) then
begin if arity(i)=0 then begin
    if sk=0 then begin
        unify:=true; stopped:=true;
        end
    else begin sk:=sk-1;
    i:=rstack.pop; j:=lstack.pop;
    end;
    end
else begin LOADSTACK(i,i):
    sk:=sk-1;
    i:=rstack.pop;
    j:=lstack.pop;
    end
end
else if visible(i) then begin
    if OCCUR(i,j) then begin
        UNIFY:=false; stopped:=true;
        end
    else begin subst(i):=true;
        term(i):=j;
        if sk=0 then begin
            unify:=true; stopped:=true;
            end
        else begin sk:=sk-1;
        i:=rstack.pop;
        j:=lstack.pop;
        end;
        end;
    end
else if visible(j) then begin
    if OCCUR(j,i) then begin UNIFY:=false;
        stopped:=true;
        end
    else begin subst(j):=true;
        term(j):=i;
        if sk=0 then begin
        unify:=true; stopped:=true;
        end
    end;
end
else begin sk:=sk-1;
i:=rstack.pop;
j:=lstack.pop;
end;
end

else begin UNIFY:=false; stopped:=true; end;
end of procedure UNIFY;

Since the arity of any expression is variable. The ARG field of the equation-table needs to be either dynamic or large enough to take the maximum number of arguments. In order to save the memory space for holding this table, we rearrange the equation-table into a new one, called the eq-table, as shown in TABLE 3-4.

TABLE 3-4 is the eq-table of system p. This new table stores the arguments with a linked list using field LINK and separates two lists with a zero valued element. The ARG field, instead, stores the pointer pointing to the first element of the corresponding argument list. For searching the arguments of a symbol, we sequentially take arguments starting from the position pointed by the ARG until a zero pointer is found. This eq-table is the representation used in the UNIF-CHIP, storing the two expressions to be unified. There is a random access memory located outside the UNIF-CHIP with four fields corresponding to the fields in TABLE 3-4, respectively:

field1 -> SYMBOL, VBLE
field2 -> ARGS
field3 -> LINK
field4 -> TERM, SUBST

Boolean field VBLE is combined into field SYMBOL and field SURST is combined into TERM.

The chip-simulating algorithm written in SIMULA is translated into an assembly program (Appendix B), then to a microprogram (Appendix C) which is used to be stored in the controller of the UNIF-CHIP. Such that UNIF-machine can execute unification algorithm following this microprogram.


<table>
<thead>
<tr>
<th>symbol(vble)</th>
<th>args</th>
<th>link</th>
<th>term (subst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>h</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>h</td>
<td>F</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>F</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>F</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>y</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Section 4 HARDWARE IMPLEMENTATION

The "UNIF-machine" or "UNIF-chip" we are going to describe is a special purpose microprogrammed machine. As mentioned before, the main purpose for designing this machine is to implement the unification computation in hardware, such that this machine can work as a coprocessor in a system to execute the unification computation more efficiently. This machine is designed to take a pair of expressions as input and compute the substitution set G for this pair of expressions. During the computation, the variable bindings of the substitution set are produced and the equation table is modified as a side-effect. After the execution of UNIF-machine, the binding information is completely recorded in the off-chip RAM. However, if there exists no general substitution set for these input expressions, the final output of the machine will indicate the failure of the computation.

4.1 System Overview

Fig 4.1 shows the application of an UNIF-chip in a Prolog system, working as a coprocessor. The user submits a program to the system, and the system starts running the program. During the execution of the Prolog program, whenever the system needs to do unification computation it stores pairs of expressions to be unified in the external RAM (called equation table). The system stores the data in the way as described in section 3 and directs the UNIF-coprocessor to unify them. Once the UNIF-chip receives the request, it starts computing the substitution set by reading in data from the equation table and, during computation, recording the variable bindings back to the equation table. After the computation is done, the UNIF-chip will notify the system the result of the unification. If the expressions can be unified, then the system can search through the equation table for the variable-binding informations and do the substitution to derive a new goal statement according to the given information.

Fig 4.2 shows the pinout of the Unif-chip. There are ten output pins connecting to the address pins of the external RAM, which holds the address of the input/output data. Another nine tristate pins are for bidirectional dataflow between the UNIF and the RAM such that data can be either read or written. The other nine pins consist of the power (vdd), ground (gnd), clock (phase1, phase2), control signal (read, write, reset),
Prolog run time system in host machine

RAM

DATA-REGISTERS

ROM

DATA-BUS

STACK-MEMORY

Fig 4.1 UNIF coprocessor
Fig 4.2 Pinout of UNIF-chip
terminating signal (stop) and the result signal (unified).

During the execution period, the stop-pin output is always low, until the computation is finished it becomes high. After the computation is done. If the unified-pin output is high, the two expressions can be unified, otherwise they are un-unifiable. The reset-line is used to initialize all those data registers in the chip which need to be reset to proper initial values. The read and write lines are signals indicating reading or writing the external RAM.

Once the machine starts execution, it reads in the first two symbols of the expressions from the external eq-table, as directed by its internal microprogram. It computes the substitution set step by step and records the result back into RAM until the final answer is reached. After the computation is finished with the machine indicating successful unification result, system searches for the variable binding relations from the "subst" and "term" fields of the eq-table, and performs the necessary substitution and derive a new goal statement.

From the pinout of the chip, we can see that the communication between Unif-Chip and the external RAM is bounded by the nine data-lines and 10 address-lines, under the control of the read/write signals.

Next we are going to look at the floor plan of the Unif-chip and discuss each function block in the chip and the data flow between them. Fig 4.3 is the floor plan of the "Unif-Chip". The chip is composed of four major parts as follows. The equation table is implemented as an external standard RAM. The stack-memories are also implemented as a standard part. Except for the controller which is made by PLA generator, all the leaf cells in the UNIF chip are made with REST, which is a leaf cell design system for NMOS stick diagrams. The higher level composition cells are made with RIOT or LAP. The last step of wiring are done with LAP.

1. EQUATION-TABLE is a standard external random access memory. It stores the pairs of expressions to be unified and also stores the final result of the unification computation. It contains four fields at all. Each field is 9-bits by 256-bits.

2. STACK-MLMURILS are used to store multiple arguments of the two expressions. It replaces the software recursion with the hardware stack. This block contains three
Fig 4.3 Floor plan of UNIF-chip
8-bit wide, 256-bit long stack memories with two 8-bit wide counters counting the number of data elements stored in the stack memories. There are a total of $24 \times 206 = 6144$ stack cells in the stack-memories.

3. DATA-REGISTERS hold several information about the two expressions, including symbols, addresses, arguments, etc., during the unification computation. It contains seven 9-bit wide data registers, and two 8-bit wide comparators for comparison between symbols or addresses.

4. CONTROLLER stores the unification algorithm and controls the behavior and the execution sequence of the chip. Once the UNIF chip starts working, the execution is determined by the controller following its internal microprogram. Controller is composed of a microcode memory with 7 inputs, 31 outputs, 113 minterms and zero feedbacks, and is implemented as a PLA. There is a 7-bit-wide program counter storing the address of the microprogram, which can count-up or directly load.

Fig 4.4 shows the layout geometry of the chip. Fig 4.5 is the block-diagram of the Unif-Chip. All the data registers and the three stack memories share one data-bus. Externally, the chip can communicate with the external RAM only through the two registers MENADDR and DATAREG. Two comparators indicate the results of comparison between registers ADDR1 with ADDR2, and SYMB1 with SYMB2. Eight flags indexed with A, B, C, .....H indicate the current state of each register. Micorocode memory ROM, addressed by the program counter PC, distributes the control signals to control each action of the chip and the data-flow on the data-bus.

Following this we are going to discuss the details of each block, the operation and function of each component in the chip. We divide the discussion into nine steps as follows:

1. Equation-table.
2. Stack-memories.
3. Data-registers.
4. Controller.
5. Data-bus. Eight flags and check-cell.
7. Input/output signals.
Fig 4.4 Layout Geometry of UNIF-CHIP
Fig 4.5 Block diagram of UNIF-CHIP

4.2 Equation-table

We have described the format of the equation table in section 3. TABLE 3-4 is the eq-table of the system P, where \( P = \{ h(f(a,b), z), \ h(y, f(d,c)) \} \). This table stores the arguments of the expressions with linked list, using the field LINK. Each list is terminated with a zero valued element. The ARGS field stores the pointer pointing to the first element of the corresponding list. For searching the arguments of a symbol, we sequentially take arguments starting from the position pointed by the ARGS until a zero pointer is found.

<table>
<thead>
<tr>
<th>symbol (vble)</th>
<th>args</th>
<th>link</th>
<th>term (subst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>h</td>
<td>z</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>h</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>T</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>F</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>l</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>z</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>y</td>
<td>T</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
TABLE 3-4 is the representation we use in the UNIF-chip to store two expressions. We implement the equation-table as a standard RAM. A random access memory is located outside the UNIF-chip, with each data word divided into four fields corresponding to the four fields in TABLE 3-4, respectively:

- field1: SYMBOL, VBLE
- field2: ARGS
- field3: LINK
- field4: TERM, SUBST

Each part of the RAM is 9-bits wide and 256-bits long. The SYMBOL field takes 8 bits and the boolean VBLE field takes one bit. The TERM field takes 8 bits and the boolean SUBST field takes one bit. The ninth bit of fields 2 and field 3 are not used.

4.3 Stack-memories

The stack-memories block is composed of three last-in-first-out stack memories, called OSK, LSK and RSK, and two stack-memory-counters named OCC and SKC. Each stack memory is 8-bit wide and 256-bit long. The counter is 8-bit counter, counting either up or down.

As mentioned in section 3, there are three cases of variable bindings during the unification:

1. variable to variable, \( v_1 \rightarrow v_2 \)
2. variable to function, \( v_1 \rightarrow f \)
3. function to function, \( f_1 \rightarrow f_2 \).

OSK is used for storing the arguments of the two expressions during the execution of the OCCUR procedure. In the second case above, a variable \( v_1 \) can be bound to a function \( f \), where \( f = f(t_1, t_2, \ldots, t_k) \), only if the variable \( v_1 \) is not contained in this function. Therefore, in order to bind \( v_1 \) to \( f \), we need to check first whether \( v_1 \) is contained in the function \( f \), by pushing all the arguments of \( f \) onto the stack memory OSK and then pop them one by one to check whether any argument \( t_i \) equals to \( v_1 \) or
contains v1.

RSK and LSK are used for storing the arguments during the execution of the UNIFY procedure. In the third case of binding a function f1 to another function f2, if f1 and f2 are different functions, then they can not be bound together. However, if f1 and f2 are of same function name with different arguments, f1 = f(t1,t2...,tk), f2 = f(r1,r2,...,rk). The unification computation recursively unifies each the arguments pairs t1 with r1, ..., and tk with rk. If any one pair cannot be unified, the unification of f1 with f2 fails. Therefore, we perform this pairwise computation by pushing all the arguments pairs sequentially onto the two stack memories RSK and LSK. Now, pairs are popped out and unified until no more argument pairs are left.

\[
t1, t2..., tk \rightarrow \text{RSK} \\
r1, r2..., rk \rightarrow \text{LSK}
\]

Each stack memory has four control lines: PUSH(SHL), TRL, POP(SHR), TRR. SHL is the shift left control line corresponding to pushing down and SHR is the shift right control line corresponding to popping up. TRR is the transfer right control line and TRL is the transfer left control line. During clock phase2, SHR can be driven high followed by driving the transfer right (TRR) high during phase1, which causes a pop-up operation to be performed. The first data element stored in the stack memory is shifted onto the data-bus and the remaining data in the stack memory are shifted one position closer to the stack top. If instead SHL is driven high during phase2, followed by the TRL being driven high during phase1, a push-down operation is performed. The data on the data-bus are pushed down into the stack memory and the data originally in the memory are shifted one position deeper into the stack.

Fig 4.6 shows the timing sequence of these two operations. The pop-up and push-down operations are executed only during phase2. During clock phase1, data in the stack memories will be either transferred right or transferred left or refreshed, depending on the preceding operation. If there is no pop or push operation being executed during preceding phase2, the data will only be refreshed by driving TRL high during phase1. However, if in the preceding phase2 a pop-up operation is executed, instead of TRL, TRR control line is driven high to transfer data right. If instead a push-down operation is executed, TRL control line is driven high to transfer data left.
Fig 4.6 Stack control timing sequence

Fig 4.7
With two operation, data registers and stack memories can communicate with each other through popping up the data in stack memory onto data-bus and writing them into the data register, or through reading the data in the data registers onto data-bus and pushing them into the stack-memories.

Fig 4.8 is the stick diagram of a stack-memory cell. It is designed such that these cells can be packed vertically and horizontally. There are 256 cells packed in the vertical direction and 8 cells in the horizontal direction, making an 8-bit-wide and 256-bit-long stack memory. The four control lines of the stack-memory-cell are separated on both edges such that two neighboring stack-memory-cells can share the two control lines between them. Since stack memories take much space, sharing control lines is one of the ways to reduce down the area it takes.

There are two stack-memory counters, OCC and SKC. Both are reset to zero at the beginning of the UNIF process. Whenever a push-down or a pop-up operation is executed the corresponding stack-memory counter value is incremented or decremented, controlled by the control lines SHL and SHR. There are two flags, FLAG-E and FLAG-H, which indicate whether the corresponding values in the counters are zero. Whenever counters count to zero, the flag turns high to indicate that no more data elements are stored in the stack memory.

4.4 Data-registers

The data-registers block contains seven 8-bit wide registers, named SYMB1, SYMB2, ADDR1, ADDR2, TEMP, MENADDR, and DATABEG, and two comparators named COMP1 and COMP2. Each register has two control lines, 'control-read' and 'control-write', to control the input/output of data. With its control-read high, the register puts its data onto the data-bus; with control-write high it writes data on the data-bus into itself.

These operations can be executed only during phase2. During phase1, the data contained in each register will be refreshed automatically. Therefore, no operation can take place during this cycle. Since all the data-registers share the same data-bus, only two registers can communicate with each other during a single clock cycle.
Fig 4.8 Stack-memory cell
SYMB1 and SYMB2 are used for storing the symbols of the two expressions fetched from the eq-table to be unified. The 9th bit of SYMB1 and SYMB2 indicates whether these symbols are variables or not. ADDR1 and ADDR2 store the addresses of the symbols in SYMB1 and SYMB2. (The 9th bit of ADDR1 and ADDR2 are not used). Comparator COMP1 outputs high if the addresses in ADDR1 and ADDR2 are equal, otherwise it outputs low. COMP2 outputs high if symbols in SYMB1 and SYMB2 are the same, otherwise low. These two comparators are latently capable of performing comparison without being clocked. The TEMP register is used to store the address of either SYMB1 or SYMB2 temporarily during the execution of the OCCUR function.

DATAREG and MENADDR registers are the interface between Unif-Chip and the external RAM. Data to be output to the RAM or input from the RAM are stored in DATAREG and the corresponding address is stored in MENADDR. When the "read" signal is enabled, data in the eq-table addressed by MENADDR are read out and stored in DATAREG. Whichever register requests data from the external RAM can transfer the data from DATAREG. If the "write" signal is enabled, the data stored in DATAREG will be written into the external RAM at the address indicated by MENADDR. The external RAM contains 4 fields, each field is 256-bits long. The address-register MENADDR is, therefore, divided into two parts. The first two bits point to the field, and the other eight bits indicate the position in the field.

4.5 Controller

The controller is the heart of the UNIF chip. It is composed of a microcode memory, a program counter and a super buffer register. A 31-bit wide microcode memory stores the microinstructions of the unification program. The program counter, PC, stores the microcode memory address. The super buffer register is 30-bit wide, through which control signals are amplified.

The functions performed by the microcodes are listed in Table 4-1. Since there are only 31 kinds of microinstructions needed in the microprogram, five microcode bits are enough to encode these operations. However, with a 5-bit microinstruction, we would need to decode the microinstructions before sending them to the super-buffer. All of these steps have to be done within one clock cycle. To prevent the timing delay incurred by the decoding, we store the 31 control signals directly as microcodes in the ROM.
Among the 31 microcodes, the first one is a special signal, called "jump-signal", which defines the interpretation of the other 30 microcodes. If signal jump=0, the microcontrol word is in the 'normal' state, and the 30 microcodes are treated as control signals. But if jump=1, only the first 16 microcontrol are meaningful. The first seven codes become the new address for the jump-instruction, and the next eight bits are the conditional codes which are sent to the CHECK-comparator to be matched with the eight flags, representing the current state.

Microinstructions are always fetched during phase 1. If jump=0, the 31 control signals are sent directly to the super-buffer, and during phase 2, these control signals are distributed across the chip. The PC counter simply increments. However, if jump=1, the eight conditional codes are sent to the CHECK-comparator and the seven address bits are stored in an address-buffer in the PC counter. The CHECK-comparator takes codes both from the eight flags and the eight conditional codes in the microcode, comparing whether the current state matches the conditional codes. If a match occurs, the jump-instruction is approved and the PC counter is loaded with the new address stored in the address-buffer. The next fetching of microinstruction will then be read from the new address. However, if no match occurs, the jump-instruction is not approved. PC counter simply increments. Therefore, in this case, no control signals are sent out during phase 2, since the 31 microcodes are interpreted as new address and conditional codes.
TABLE 4-1 MICROCODES-TO-MICROINSTRUCTIONS

<table>
<thead>
<tr>
<th>microcodes</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 jump=0</td>
<td></td>
</tr>
<tr>
<td>2 stop</td>
<td>b0 (new address bits)</td>
</tr>
<tr>
<td>3 unified/failed</td>
<td>b1</td>
</tr>
<tr>
<td>4 read</td>
<td>b2</td>
</tr>
<tr>
<td>5 write</td>
<td>b3</td>
</tr>
<tr>
<td>6 chang-field</td>
<td>b4</td>
</tr>
<tr>
<td>7 field1</td>
<td>b5</td>
</tr>
<tr>
<td>8 field2</td>
<td>b6</td>
</tr>
<tr>
<td>9 write/lenaddr</td>
<td>A (conditional codes)</td>
</tr>
<tr>
<td>10 read/lenaddr</td>
<td>B</td>
</tr>
<tr>
<td>11 find/lenaddr</td>
<td>C</td>
</tr>
<tr>
<td>12 write/datareg</td>
<td>D</td>
</tr>
<tr>
<td>13 read/datareg</td>
<td>E</td>
</tr>
<tr>
<td>14 write/symbol 2</td>
<td>F</td>
</tr>
<tr>
<td>15 read/symbol 2</td>
<td>G</td>
</tr>
<tr>
<td>16 write/symbol 1</td>
<td>H</td>
</tr>
<tr>
<td>17 read/symbol 1</td>
<td>I</td>
</tr>
<tr>
<td>18 write/addr 2</td>
<td>J</td>
</tr>
<tr>
<td>19 read/addr 2</td>
<td>K</td>
</tr>
<tr>
<td>20 write/addr 1</td>
<td>L</td>
</tr>
<tr>
<td>21 read/addr 1</td>
<td>M</td>
</tr>
<tr>
<td>22 write/temp</td>
<td>N</td>
</tr>
<tr>
<td>23 read/temp</td>
<td>O</td>
</tr>
<tr>
<td>24 push-occ-stack</td>
<td>P</td>
</tr>
<tr>
<td>25 pop-occ-stack</td>
<td>Q</td>
</tr>
<tr>
<td>26 push-left-stack</td>
<td>R</td>
</tr>
<tr>
<td>27 pop-left-stack</td>
<td>S</td>
</tr>
<tr>
<td>28 push-right-stack</td>
<td>T</td>
</tr>
<tr>
<td>29 pop-right-stack</td>
<td>U</td>
</tr>
<tr>
<td>30 transfer-push-stack</td>
<td>V</td>
</tr>
<tr>
<td>31 transfer-pop-stack</td>
<td>W</td>
</tr>
</tbody>
</table>
4.6 System Design

Data-bus:

There is a nine-bit wide data path flowing around the chip through the data-registers and the stack-memories. Every register can only get data from this data-bus or write data onto it by driving one of its two control lines "control-read" or "control-write" high. The three stack memories are also restricted to communicate only with the data-bus by popping out the data onto data-bus or pushing down the data on data-bus into the stack memory. All these sequences of operations are controlled by the microinstructions from the controller, i.e. their control lines are driven by the control signals from the controller.

Eight-flags:

There are eight flags in the chip to indicate the current state of each register, the number of data elements stored in the stack memory, or the current state of computation.

Flag A: If the data stored in DATAREG[1-8] are zero, then flag A = 1 else flag A = 0.

Flag B: If DATAREG[9] = 0 then flag B = 1 else flag B = 0.

Flag C: If data in SYMB1 and SYMB2 are equal, then flag C = 1 else flag C = 0.

Flag D: If data in ADDR1 and ADDR2 are equal, then flag D = 1 else flag D = 0.

Flag E: If data in stack-counter OCC equals zero then flag E = 1 else flag E = 0.

Flag F: If data in stack-counter SKC equal zero then flag F = 1 else flag F = 0.

Flag G: If SYMB1[9] = 1, i.e. symbol 1 is variable, then flag G = 1 else flag G = 0.

Flag H: If SYMB2[9] = 1 then flag H = 1 else flag H = 0.
Check-comparator:

Whenever a jump instruction occurs, the CHECK-comparator compares the eight flags with the eight conditional codes in microcode to decide whether this jump-instruction should be executed. For the eight conditional codes, the state of the flag is neglected if the corresponding conditional code is zero; however, if the conditional code is 1, the state of the corresponding flag must be equal to 1. Therefore, only those current state of flags whose conditional codes equal to one are checked. For an "unconditional jump" instruction, the eight conditional codes are all zero, i.e. it is always executed, neglecting the current state of all the eight flags. After the checking is done, the CHECK comparator will either approve the jump instruction by loading the new address into PC counter or disapprove the jump instruction by simply increasing the value of PC counter by one.

Reset and I/O:

When we look at the pinouts of the Unif-chip, there are three input signals: 'PHASE1, PHASE2, RESET', and four output signals: 'READ, WRITE UNIFIED, STOP'.

The reset pin need to be driven high to reset the registers to their proper initial values before Unif-machine can start working. Among all the registers, ADDR1 and ADDR2 need to be reset with the initial values one and two, respectively, since the equation-table always stores the two expressions to be unified starting with the the first two positions. Counters PC, OCC, SKC need to be set to zero at the beginning. The reset line is driven back to low once the machine starts working, and kept low for the entire period of execution.

Within the four output signals, the first two, 'read' and "write", control reading and writing the external eq-table; the other "stop" and "unified", the 'stop' pin indicates the finish of the computation and the 'unified' pin indicates the result of the computation.

Output signal 'stop' is always low during the computation of Unif-Machine. Once the computation is finished, 'stop' becomes high. If the 'unified' signal is high at this moment, it means that the unification computation has been successful, otherwise the computation fails.
Timing:

The synchronization of the UNIF chip is controlled by a two-phase clock. During clock phase 1, data in the data registers and stack-memories are refreshed, and the microinstruction in ROM is fetched, with no operations executed during this phase. During phase 2, program counter, PC, either increments or loads a new address, depending on the current microinstruction. In this phase, microinstruction is executed. It can either read or write a data register, push or pop a stack memory, or read or write the external eq-table. The microinstruction is executed only during phase 2, and must be finished before the ending of clock phase 2.

Fig 4.9 Timing sequence of UNIF-CHIP
Section 5  NONDETERMINISM

In this section we discuss two kinds of nondeterministic problem associated with the resolution procedure. One is the nondeterminism on the sequence of procedure invocations, and the other is the nondeterminism on the scheduling of procedure calls.

5.1 Non-deterministic problems

While solving a Horn clauses program, non-deterministic problems usually appear, because the computation process is based upon repeated procedure invocations. Followings are two cases where non-determinism occurs:

(1) During the unification process, when several procedures match a given procedure call, which one of the alternative procedures is executed is non-deterministic.

(2) When a goal statement contains several procedure calls needed to be executed. The order of the execution of these procedure calls is non-deterministic.

5.2 Sequence of Procedure Invocation

In this section we discuss the first case where several procedures match a selected procedure call. In this kind of situation, execution of the different procedure calls may result in alternative outputs, and in what order the outputs will be generated is also undefined. It depends on the order in which the procedure calls are executed. We use the APPEND program, which ever appeared in section 2 before, to illustrate the resulting of different outputs by executing different procedure calls.

APPEND example:

(F1) APPEND [nil, Y, Y]
(F2) APPEND [cons(x, Y),Z,cons(x,U)] <- APPEND [Y,Z,U]
With the goal statement (C1) above, we search for two lists which combine into a list cons(a,cons(b,cons(c,nil))). The two parameters W and R are output parameters for storing the final result. Statement (E1) means that "Searching for two lists W and R which constitute the third list cons(a,cons(b,cons(c,nil))), by appending the list R to the list W". We use the graphic tree structure below, where each branch represents one kind of final result, to explain the solution.
In this example the goal statement (E1) matches both (F1) and (F2). If the first statement (F1) is chosen for procedure invocation, we can get a set of result (Ans1):

\[ W := \text{nil}; \]
\[ R := \text{cons}(a, \text{cons}(b, \text{cons}(c, \text{nil}))); \]

However, if instead the second statement (F2) is chosen, we derive a new goal statement

\[ (C2) \leftarrow \text{APPEND} [Y, Z, \text{cons}(b, \text{cons}(c, \text{nil}))]. \]

The statement (C2) also matches both (F1) and (F2). Based on the same reason, we can either derive another set of result (Ans2):

\[ W := \text{cons}(a, \text{nil}); \]
\[ R := \text{cons}(b, \text{cons}(c, \text{nil})), \]

or derive another goal statement

\[ (C3) \leftarrow \text{APPEND} [Y', Z', \text{cons}(b, \text{cons}(c, \text{nil}))]. \]

Keep going on like this, we can get four sets of different results. We can see that different choice of the procedure invocations produces different results.

Back to the original topic, if several procedure calls appear in a goal statement body, we are concerned with the scheduling of the execution of the procedure calls. In the next part, we are going to discuss this case of non-determinism.

### 5.3 Scheduling of procedure calls

In conventional programming languages, the program controls the order of procedure calls. However, in logic programming the body of the procedure specifies only the collections of procedure calls it contains. The order of the procedure calls being specified has nothing to do with the meaning of the algorithm. Different strategies for scheduling procedure calls affect only the efficiency of the program but not the meaning of the program.
Following is a sorting program which reveals the effect of scheduling of the procedure calls on the efficiency of the execution.

\[
\begin{align*}
(E1) \text{SORT} \ [x, \ y] &\ <\ \text{PERMUT} \ [x, \ y], \ \text{ORDER} \ [y] \\
(E2) \ \text{PERMUT} \ [\text{nil}, \ \text{nil}] &\ <\ \\
(E3) \ \text{PERMUT} \ [\text{cons}(x, \text{nil}), \ \text{cons}(a, \text{nil})] &\ <\ \\
(E4) \ \text{PERMUT} \ [\text{cons}(a, \text{cons}(b, \text{nil})), \ \text{cons}(b, \text{cons}(a, \text{nil}))] &\ <\ \\
(E5) \ \text{PERMUT} \ [w, \ w'] &\ <\ \text{PERMUT} \ [u, \ u'], \\
&\ \text{PERMUT} \ [v, \ v'], \\
&\ \text{PERMUT} \ [w, \ w'], \\
&\ \text{APPEND} \ [u, \ v, \ w], \\
&\ \text{APPEND} \ [u', \ v', \ w'] \\
(E6) \ \text{ORDER} \ [\text{nil}] &\ <\ \\
(E7) \ \text{ORDER} \ [\text{cons}(a, \ \text{nil})] &\ <\ \\
(E8) \ \text{ORDER} \ [\text{cons}(a, \text{cons}(b, Z))] &\ <\ \text{LESS}[a, \ b], \\
&\ \text{ORDER}[\text{cons}(b, Z)]
\end{align*}
\]

Statement (E1) asserts that y is a sorted version of x if y is a permutation of x and y is ordered. Statement (E2) through (E5) specify the definition of permutation and (E6) through (E8) specify the definition of order.

For a top-down interpretation of statement (E1), we can say that

1. In order to sort a list x, first generate a permutation y of x, then test that whether y is ordered. If it is, then y is a sorted version of x.

On the other hand, if the procedure ORDER[y] is executed before the procedure PERMUT[x,y], we can say that

2. In order to sort a list x, first generate an ordered list y, then test that whether y is a permutation of x. If it is, then y is a sorted version of x.

The third case, we can have the two procedure calls executed semi-dependently. The partial output from the procedure call PERMUT is translated to the procedure call ORDER and this partially specified input can initial the computation of procedure ORDER.
(3) In order to sort a list $x$, first break $x$ into two sublists $u_1$ and $u_2$. If both $u_1$ and $u_2$ are sorted, and the maximum component of $u_1$ is less than the minimum component of $u_2$. Then appending sorted $u_1$ to sorted $u_2$ results in a sorted version of $x$.

In principle, the procedure calls in the procedure body can be executed in any sequence. However, it is quite obvious to see that the execution of procedure call PERMUT before ORDER would be much more efficient than the execution of procedure call ORDER before PERMUT. It would be even more efficient to execute procedure calls as soon as sufficient input is available. The test for order can be initiated once the first two elements of the permutation have been generated, and the failure can be detected as soon as possible.
Section 6 Conclusion

In the previous sections we have given a basic introduction to logic programming, including predicate logic, Horn clauses and Prolog language. We also described the unification chip which is a hardware implementation of the unification computation. In this final section, some applications of the unification chip would be presented.

There are already a few Prolog systems on the market which are intended for use on microcomputer systems. However they are very slow because, during the execution of a Prolog program, the deductions are done by the pattern matching processes (unification computations) and these most computational intensive steps must be performed in software. There is every indication that a unification coprocessor would bring moderate performance even on very modest systems.

Aside from the more obvious applications of the unification chip, architectures which directly execute Prolog like languages or which are special purpose theorem provers, there are a number of other applications too. The first concerns a rather modest use of a unification chip which acts much like a numerical coprocessor in a microcomputer system. Here, we envision a chip which connects to a microprocessor bus and would perform the unification computation either synchronously or asynchronously upon the receipt of an activating code in the instruction stream. This would allow efficient execution on such microcomputer systems.

The second non-obvious application of a unification processor is with parallel streams of assertions, as in a logic-per-track disk drive. It would be a very powerful successor to today's back-end relational database architectures. The important difference comes through the use of unification rather than simple pattern matching. It is very exciting to contemplate the construction of very intelligent database machines which are capable of making the logical deductions necessary in inferential databases.

In these applications, the unification chip improves the efficiency of the system by working as a coprocessor, helping execute the unification computation. However, if we can implement many unification chips on a concurrent environment, a highly efficient concurrent deductive system would improve the efficiency even more. In the following we present such a concurrent model.
Given a logic program specified by a set of Horn clauses $S$

$$S = B_1 \leftarrow B_{11}, B_{12}, \ldots, B_{1r} \quad B_2 \leftarrow B_{21}, B_{22}, \ldots, B_{2s} \quad \ldots \quad B_m \leftarrow B_{m1}, B_{m2}, \ldots, B_{mt}$$

and activated by an initial goal statement (C1)

$$(C1) \quad \leftarrow A_1, A_2, \ldots, A_n,$$

to execute the goal statement (C1), the system searches for a clause in $S$ which matches the goal statement. There are $m$ statements in $S$ and $n$ procedure calls in (C1). Since any procedure call in (C1) has the chance to unify with the statements in $S$, there are totally $M \times N$ kinds of procedure invocation can occur. Instead of executing $M \times N$ times of unification computation sequentially, they are executed concurrently on a two dimensional array.

The mesh-type model below imbeds one UNIF-chip between each cross-section of the rows and the columns. The UNIF-chip locating at row $i$ cross column $j$ executes the unification computation between the expressions $Ai$ and $Bj$ and indicates the final result whether $Ai$ can be unified with $Bj$ or not.
Surrounding this concurrent system should be a communication system which handles the transmission of input and output, manipulates the data structure, such that the results of the first computation can be transferred to the next one through this surrounding system. The important job for the communication system is to define a feasible strategy for solving the communication problems and the nondeterministic problems involved during the computation. This communication system is not discussed in this paper. It would be an interesting and worthy topic for future work.
ACKNOWLEDGEMENTS

Sincere gratitude is honored to my advisor, Jim Kajiy, for his enthusiastic and patient guidance.

Thanks to my husband, Chuen-Der, for his encouragement and support.

Thanks is also extended to Johenson D. and Mosteller R.C. for their helpful discussion during the designing of the unification chip.
Appendix:

A. The Special Purpose Unification Algorithm

```plaintext
function UNIFY(i,j:integer):boolean;
begin    while subst(i) do i:=term(i);
         while subst(j) do j:=term(j);
         if i = j then UNIFY:=true
         else if symbol(i)=symbol(j)
         then UNIFY:=EQUATEARGS(i,j)
         else if vble(i) then begin
                   if OCCUR(i,j) then UNIFY:=false
                   else begin subst(i):=true;
                           term(i):=j;
                           UNIFY:=true;
                   end;
         end
         else if vble(j) then begin
                   if OCCUR(j,i) then UNIFY:=false
                   else begin subst(j):=true;
                           term(j):=i;
                           UNIFY:=true;
                   end;
         end
         else UNIFY:=false;
end;  \{function UNIFY\}

function EQUATEARGS(i,j:integer):boolean;
var k:integer;
    equal:boolean;
begin    k:=arity(i);    equal:=true;
    1:    if k=0 then EQUATEARGS:=true
         else while ( k>0 and equal ) do
               begin equal:=UNIFY(args(i,k),args(j,k))
                  k:=k-1;
               end;
    end;
    EQUATEARGS:=equal;
end;  \{function EQUATEARGS\}

function OCCUR(i,j:integer):boolean;
begin    1:    while subst(j) do j:=term(j);
```
if arity(j)=0 then
  if i=j then OCCUR:=true else OCCUR:=false
else OCCUR:=OCCURARGS(i,j);
end;  {function OCCUR}

function OCCURARGS(i,j:integer):boolean;
  var k:integer;
  occ:boolean;
  begin  k:=arity(i);  occ:=false;
  1: if k=0 then OCCURARGS:=false
    else while (k>0 and not occ) do
      begin  occ:=OCCUR(i,args(j,k))
      k:=k-1;
      end;
  OCCURARGS:=occ;
end;  {function OCCURARGS}
Assembly Program transferred from the unification algorithm

0 addr1 -> menaddr
1 read
2 if subst = F jump 6
4 datareg -> menaddr
5 jump 2
6 set field 00
7 read
8 datareg -> symb1
9 addr1 <- menaddr
10 addr2 -> menaddr
11 set field 11
12 read
13 if subst = F jump 16
14 datareg -> menaddr
15 jump 12
16 set field 00
17 read
18 datareg -> symb2
19 addr2 <- menaddr
20 if addr1=addr2, sk=0 jump 111
21 if addr1=addr2 jump 45
22 if symb1=/=symb2 jump 48
23 addr1 -> menaddr
24 set field 01
25 read
26 if arity(1)=0, sk=0 jump 111
27 if arity(1)=0 jump 45
28 datareg -> menaddr
29 set field 10
30 read
31 if link=0 jump 35
32 datareg -> rstack
33 menaddr+1
34 jump 30
35 addr2 -> menaddr
36 set field 01
37 read
38 datareg -> menaddr
39 set field 10
40 read
41 if link=0 jump 46
42 datareg -> lstack
43 menaddr+1
44 jump 40
45 rstack -> addr1
46 lstack -> addr2
47 jump 0
48 if vble(i)=/=T jump 80
49 addr2 -> temp
50 addr2 -> menadd
51 set field 11
52 read
53 if subst=/=T jump 56
54 datareg -> menaddr
55 jump 52
56 set field 00
57 read
58 datareg -> symb2
59 menaddr -> addr2
60 set field 01
61 read
62 if arity(j)=0 jump 70
63 datareg -> menaddr
64 set field 10
65 read
66 if link=0 jump 72
67 menaddr+1
68 jump 65
70 if i=j jump 109
71 if occsk=0 jump 74
72 occstack -> addr2
73 jump 50
74 temp -> datareg
75 addr1 -> menaddr
76 set field 11
77 write
78 if sk=0 jump 111
79 jump 45
80 if vble(j)=/=T jump 109
81 addr1 -> temp
82 addr1 -> menadd
83 set field 11
84 read
85 jump 84
86 datareg -> menaddr
87 jump 84
88 set field 00
89 read
90 datareg -> symb1
91 addr1 <- menaddr
92 set field 01
93 read
94 if arity(j)=0 jump 102
95 datareg -> menaddr
96  set field 10
98  if link=0 jump 104
100  menaddr+1
102  if i=j jump 109
104  occstack -> addr1
106  temp -> datareg
108  jump 76
110  jump 112
112  jump 112 *

97  read
99  datareg -> occstack
101  jump 97
103  if occsk=0 jump 106
105  jump 82
107  addr2 -> menaddr
109  set failed=1, stopped=1
111  set unified=1, stopped=1
REFERENCE


Mosteller R F (1966) Data user's guide SSP file #4030, Caltech.

Robinson, J.A. (1965) Computational logic: The Unification

SSP file # 3710, Caltech.

Research Report 192, Yale University, C.S. Department.

Prolog - The Language and Its Implementation Compared with LISP.


CACM v.22, 7 pp. 391-401.

Machine Intelligence 8, pp. 266-299.