[redacted]

PRICES SUBJECT TO CHANGE

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND

(NASA-CR-139212) GLOBAL PARALLEL UNIFICATION FOR LARGE QUESTION-ANSWERING SYSTEMS (Maryland Univ.) 43 p HC ~~$5.25~~ CSCL 09B

N74-29545

Unclas
G3/08 54894

43

Technical Report TR-307 June, 1974
NGR 21-002-270

GLOBAL PARALLEL UNIFICATION FOR LARGE QUESTION-ANSWERING SYSTEMS

by

J. Gary Auguston and Jack Minker

Department of Computer Science
University of Maryland
College Park, Maryland 20742

The support of the National Aeronautics and Space Administration, under NASA grant NGR 21-002-270 is gratefully acknowledged.

TABLE OF CONTENTS

ABSTRACT

An efficient means of storing data in a first-order predicate calculus theorem-proving system is described. The data structure is oriented for large scale Question-Answering Systems. An algorithm is outlined which uses the data structure to unify a given literal in parallel against all literals in all clauses in the data base. The data structure permits a compact representation of data within a QA system. Some suggestions are made for heuristics which can be used to speed-up the unification algorithm in such systems.

1. Introduction

In recent years much research has been conducted in applying formal theorem proving techniques to produce deductive Question Answering (QA) systems [5, 6, 9, 12, 13, 14]. Historically, the capabilities of such systems have suffered deficiencies in producing deep deductions in an efficient and timely manner. The systems exhibiting deductive capabilities that have been developed to date have been shown to operate effectively onlv with small data bases. Two major obstacles to handling large data bases in a QA system have been the amount of space required to store large numbers of facts, and the amount of processing time required to search the data base to find clauses which are capable of resolving with one another.

In this paper we introduce concepts aimed at providing a more efficient means of storing data while at the same time representing data in a manner which will allow easier processing. A data structure oriented for large scale QA systems is introduced along with a compact scheme for the internal representation of data base clauses. A unification algorithm, using this data structure, which enables unification to be performed in parallel across all clauses which have literals complementary to a given literal is presented. Finally, some suggestions are presented for the type heuristics which could be used to guide the unification process in such a system to a more efficient solution.

2. Related Work

Several papers relate directly to the work described here. Burstall [3] presented a unification algorithm with a somewhat similar approach to the one presented in this paper. He defined an abstraction of a literal for each literal appearing in the data base, representing the structure of each literal and all variants of the literal. Associated with each literal was a list of all clauses in which the literal appeared. He then constructed a tree of abstractions such that all abstractions occurring below a given node on a tree could be obtained from the abstraction at that node by a substitution. Unification was attempted with the top node of each tree. When successful, a subset of all possible resolvents over the entire data base had been found. However, in order to find all possible resolvents in the data base, unification had to be attempted at each node of each tree until a point was found for all paths of every tree where no unification could be performed successfully.

The system presented in this paper takes the approach of identifying all clauses within the data base based upon the arguments of the literals of each clause. Unification is performed globally across the entire data base one argument position at a time. When a unifiable situation is found for a particular item at a particular argument position, the clauses which are unifiable are continued for consideration in the unification process as long as no non-unifiable situation has been found for those clauses at a previously processed argument position. Each argument position need be processed only once by the unification algorithm (as opposed to many times by the Burstall approach) and after all arguments have been processed all those clauses, and only those clauses, which do unify with the input clause have been identified.

The approach presented in this paper is an outgrowth of work by Fishman and Minker [8] who developed the concept of Π-notation in order to permit the introduction of parallel processing to theorem proving. We have extende

Π-notation slightly (see Section 4) to permit the unification of clauses across the entire data base for a set of input literals. Additionally, due to the partitioning of the data base which is imposed by the data structure, many of the unsuccessful attempts at unification which are apparent in Burstall's approach can be avoided.

The importance of perfoming searches in a parallel fashion for OA Systems has also been noted by Sussman and McDermott [17] in their work on CONNIVER. The approach to parallelism taken by Fishman and Minker [8], differs from that of CONNIVER. Within CONNIVER parallelism is achieved by statements in the language of the FOR ALL variety, while Fishman and Minker embed parallelism into a new representation and a modification to the Robinson unification algorithm. We describe here how to exploit the representation devised by Fishman and Minker to achieve a parallel unification over all clauses in the entire data base, given a specific literal.

3. Π-Representation

In a conventional QA system based upon the first-order predicate calculus, each separate fact input to, or generated by the system is represented in clause form. Thus, in a general data base, the fact that an individual has several children is represented by a series of distinct clauses. For example, the statement:

[FACT:] "Sam is the parent of Sue, Sally, Bill and Jim" is represented by the four facts:

[3.1] P(Sam, Sue) ,
P(Sam, Sally) ,
P(Sam, Bill) ,
P(Sam, Jim) ,

where the predicate letter P represents parent.

A similar structure would be required to indicate the fact that Mary is the mother of each offspring. No advantage is taken of the fact that the general structure (template) of these repetitive clauses is the same with the only difference being the specific entities substituted for the two arguments x and y .

Fishman [7] and Fishman and Minker [8] introduced an extension to the clause form of the first-order predicate calculus to take advantage of such structural similarities. The representation, called "Π-representation", allows a set of similarly structured clauses to be represented in a single clause termed a "Π-clause."

[3.2] DEFINITION:

A Π-clause P is a pair (C,ϕ) where C is a first order predicate calculus clause which contains no constants and ϕ is a finite non-empty set of Π-substitutions.

A Π-substitution set consists of expressions of the form $[a_1,\ldots,a_n]/v$ where the a_i are specific constants and v represents a variable in C . Each Π-substitution set may contain one expression for each variable in C . The constants associated with a variable in such an expression represent the set of constants which can be substituted for the particular variable in the represented clause. Any variable occurring in C which has no associated entry in a particular Π-substitution set is universally quantified over that particular Π-substitution set. A Π-clause (C,ϕ), where ϕ consists of the single Π-substitution set,

$$\phi = \left\{[a_{11},\ldots,a_{1n_1}]/v_1,\ldots,[a_{m1},\ldots,a_{mn_m}]/v_m\right\}$$

actually represents the set of $\prod_{i=1}^{m} n_i$ clauses

$$C\{{}^{a_{11}}/v_1,\ldots,{}^{a_{m1}}/v_m\}$$

$$C\{{}^{a_{12}}/v_1,\ldots,{}^{a_{m1}}/v_m\}$$

$$\vdots$$

$$C\{{}^{a_{1n_1}}/v_1,\ldots,{}^{a_{mn_m}}/v_m\}$$

For example, consider the Π-clause

[3.3] $$(P(x,y),\ \{{}^{[a,b,c]}/x,{}^{[a,e]}/y\})$$

This Π-clause represents the following six first-order predicate calculus clauses:

i. $P(a,a)$
ii. $P(a,e)$
iii. $P(b,a)$
iv $P(b,e)$
v. $P(c,a)$
vi. $P(c,e)$

Note that the first-order predicate calculus clauses represented by a Π-clause may not be unique. For example, if the Π-clause in [3.3] had another Π-substitution set consisting of $\{{}^{[b]}/x,{}^{[e,f,g]}/y\}$ the clause $P(b,e)$ would have two representations in the same Π-clause.

The power of Π-representation can be demonstrated by constructing a Π-clause to represent the clauses in [3.1]. The resultant Π-clause would be as follows:

[3.4] $(P(x,y), \{[Sam]/x,[Sue, Sally, Billy, Jim]/y\})$

The additional facts that Mary is also a parent of these children can be represented by simply expanding the substitution component for x to include Mary.

[3.5] $(P(x,y), \{[Sam,Mary]/x,[Sue, Sally, Bill, Jim]/y\})$

Fishman and Minker also introduce extensions to unification, resolution, and factoring which apply to Π-clauses. They point out that each inference step between a pair of Π-clauses in a deductive proof is equivalent to performing the same step in parallel across all the individual clauses represented by the Π-clauses. Thus, the meaningful characteristics of a particular inference system (e.g., completeness, soundness) are preserved when operating on Π-clauses.

The value of Π-representation to deductive QA systems is twofold; parallelism in operation and reduction in storage requirements. The availability of parallelism is inherent in the structure of the Π-clause. By unifying Π-clauses one can achieve the work of many times the number of similar unifications that would have had to be performed on the corresponding first-order predicate clauses. While a Π-unification may be a somewhat lengthier process than a single standard unification, the overall time advantage of Π-unification will become quite apparent in any system that has even a modest degree of clauses which can be correlated to a single template.

As can be seen by the above examples, use of Π-clause representation in a QA system will help reduce the proliferation of clauses within the data base. A great deal of savings in space can be realized in storing the basic data clauses of the system. More significant, however, will be the savings introduced at each level of deduction, as the number of Π-clauses in the system will tend to grow more slowly.

4. Extended Π-Representation

The Π-representation presented in Section 3 did not attempt to take advantage of the structural similarity between literals which have differing predicates. We provide in this paper an extension to Π-representation to take advantage of this similarity.

[4.1] DEFINITION:

An extended Π-clause is a pair (C,ϕ) where C is a first order predicate calculus clause which contains no constants where each literal of degree n is represented as an $n+1$ tuple *and ϕ is a finite non-empty set of Π-substitutions.*

A literal of degree n is represented by including the predicate letter in an $n+1$ tuple. Such a literal will be said to have degree n, but the associated tuple will have $n+1$ argument positions. The predicate letter will be considered to be in argument position one (of the tuple) and the $i^{\underline{th}}$ argument of the predicate letter will be in argument position $i+1$.

For example, the Π-clause in [3.1] would be represented in extended Π-notation as

[4.2] $((\beta,x,y),\{{}^{[P]}/_{\beta},\ {}^{[Sam]}/x,\ {}^{[Sue,\ Sally,\ Bill,\ Jim]}/y\})$

By definition, each of the n positions in the template portion of a literal of an extended Π-clause can represent only one type of data item; it must represent either a variable for which there is a replacement set in the associated substitution set (a placeholder variable), or a variable for which there is no replacement set in the substitution set (a free variable) or it must be a function symbol along with its associated arguments.

The first argument position of any template is only permitted to be a free variable since this represents the predicate of the literal. Note that although we allow a free variable to appear in the predicate position, we are not attempting to generalize the concepts presented here to second-order logic.

Extended Π-representation provides the capability of representing axiom schemas in the data base. For example, consider the transitivity relationship which can be represented in the first-order predicate calculus as:

[4.3] $P(x,y) \wedge P(y,z) \Rightarrow P(x,z)$

Changing this to clause structure we arrive at:

[4.4] $\sim P(x,y) \vee \sim P(y,z) \vee P(x,z)$

This clause would be represented in extended Π-notation as:

[4.5] $(\sim(\beta,x,y) \vee \sim(\beta,y,z) \vee (\beta,x,z),\{[P]/\beta\})$

The transitivity expression for the predicate P represented by [4.5] could be expanded to additional predicates by adding elements to the replacement set for β. In general, the property of transitivity could be represented for all predicates in the data base which are transitive by simply including the Π-clause template of [4.5] in the data base with a replacement set for β containing all transitive predicates. Similarly, other characteristics of other sets of predicates (such as associativity, communitiativity, etc.) could be represented just as easily. This provides a capability to include a great deal of useful information in the data base concerning the predicates of the data base in a very compact manner.

5. Substitution Set Number of Base Clauses

It is necessary to establish a numbering convention for substitution sets in order to provide an exact identification of a Π-clause within the data structure. This numbering system assigns a unique number to the application of a particular substitution set to a particular literal of a particular clause. These numbers (referred to as substitution set numbers) are used throughout the data structure developed in this paper to identify the clauses and literals to which data items belong. The substitution set number has the following forms:

x-y-z ,

where x is a unique number assigned by the system to a particular clause

y is a relative position of the literal within clause x (counting from left to right)

and z is a relative number of the particular substitution set within the set of substitution sets applied to clause x

For example, consider the following extended Π-clause:

[5.1] $((\rho,x) \vee (\beta,y),\{S_1,S_2\})$

where $S_1 = \{{}^{[P]}/_{\rho}, {}^{[Q]}/_{\beta}, {}^{[a]}/_{x}, {}^{[b]}/_{y}\}$

and $S_2 = \{{}^{[P]}/_{\rho}, {}^{[R]}/_{\beta}, {}^{[d,e]}/_{x}, {}^{[a]}/_{y}\}$

Assume that the clause in [5.1] has been assigned the unique clause number 10 by the system. Then the substitution set number (SSN) associated with the application of substitution set S_1 to the literal (ρ,x) is 10-1-1 . Correspondingly, the SSN associated with the application of S_2 to the liter (β,y) is 10-2-2 .

By establishing such a numbering convention, the application of a substitution set to a particular literal has a unique internal representation. For example, the replacement of ρ by P in the first argument position and x by a in the second argument position of [5.1] would be known to have occurred in the same literal since they both would be represented by the same unique substitution set number within the data base. No other substitution represented in the data base will be assigned the same number thereby making the substitution identifiable. The fact that substitution sets with different numbers apply to elements of the same clause is indicated by the first part of the substitution set number. The second part of the SSN similarly provides a unique identification of the literal within a clause to which a particular substitution set has been applied.

6. Data Structure for Global Representation

In a deductive QA system the most critical operation performed is unification. Unification is essentially a pattern-matching operation used to identify clauses of the data base which can be resolved with one another in an effort to produce a problem solution. Given that the search strategy has by some means selected the literal upon which it wishes to unify, several unifications with clauses in the data base may be attempted (some successfully, other unsuccessfully) before the selected literal can be resolved satisfactorily. Each of these unification efforts requires a separate invocation of the unification process which is a large and time consuming program.

In this paper we present an indexing scheme to eliminate the need for repetitive calls of the unification process for the same selected literal. This indexing scheme is structured such that all unifiers for a particular selected (input) literal will be produced in one call to the unification process. Information is organized in this data structure so that unification action can be taken without having information presented concerning the entire clause involved. If unification is successful with a literal of a particular clause, information is available as to where the entire clause may be found.

The general data base is partitioned into disjoint subsets with a separate segment for literals of differing degrees. The data base description that follows represents a standard data base format which is applicable to all the independent subsets of the data base. (See Appendix A for a sample representation of clauses stored in a data base partition).

6.1 Index Level

The top level of the data structure is the Index Level. The Index Level contains an entry for each of the n argument positions in the associated n-tuple. Each of these n entries contains a pointer for each of the three types of data items which may appear at a particular argument position of a clause (placeholder variable, free variable and function letter). These pointers indicate the location of the list of all the entries of a particular item type which have occurred at a specific argument position.

6.2 Placeholder Variables

The placeholder variable portion of a specific argument position of the data structure consists of the set of unique constants (or predicate letters in the first argument position) which have occurred as a member of a substitution set for any literal of degree n at that argument position. Each entry contains the constant value itself and for each separate occurrence of the constant in a unique substitution set, the substitution set number in which the constant appeared. Consider the following extended Π-clause:

[6.2.1] $((\beta,x),\ \{[P]/\beta,\ \ [a,b]/x\})$

The placeholder variable portion of the data base for the second argument position of this n-tuple might be

a	1-1-1
b	1-1-1

where 1-1-1 is the unique substitution set number assigned the application of the substitution set in [6.2.1] to the template in [6.2.1].

6.3 Free Variables

The free variable portion of the data structure is similar to the placeholder portion. For each argument position within an n-tuple of a specific degree, a separate entry is made for each unique free variable that occurs. Accompanying each entry is a list of the unique substitution set numbers in which this variable appears as a free variable. For example, consider the following literal:

[6.3.1] $((\beta,x,y),\{\{[P]/\beta,\ [a]/y\},\{[Q]/\beta,\ [b]/y\}\})$.

The corresponding free variable entries for the second argument of literals of degree two might be as follows:

x	2-1-1
	2-1-2

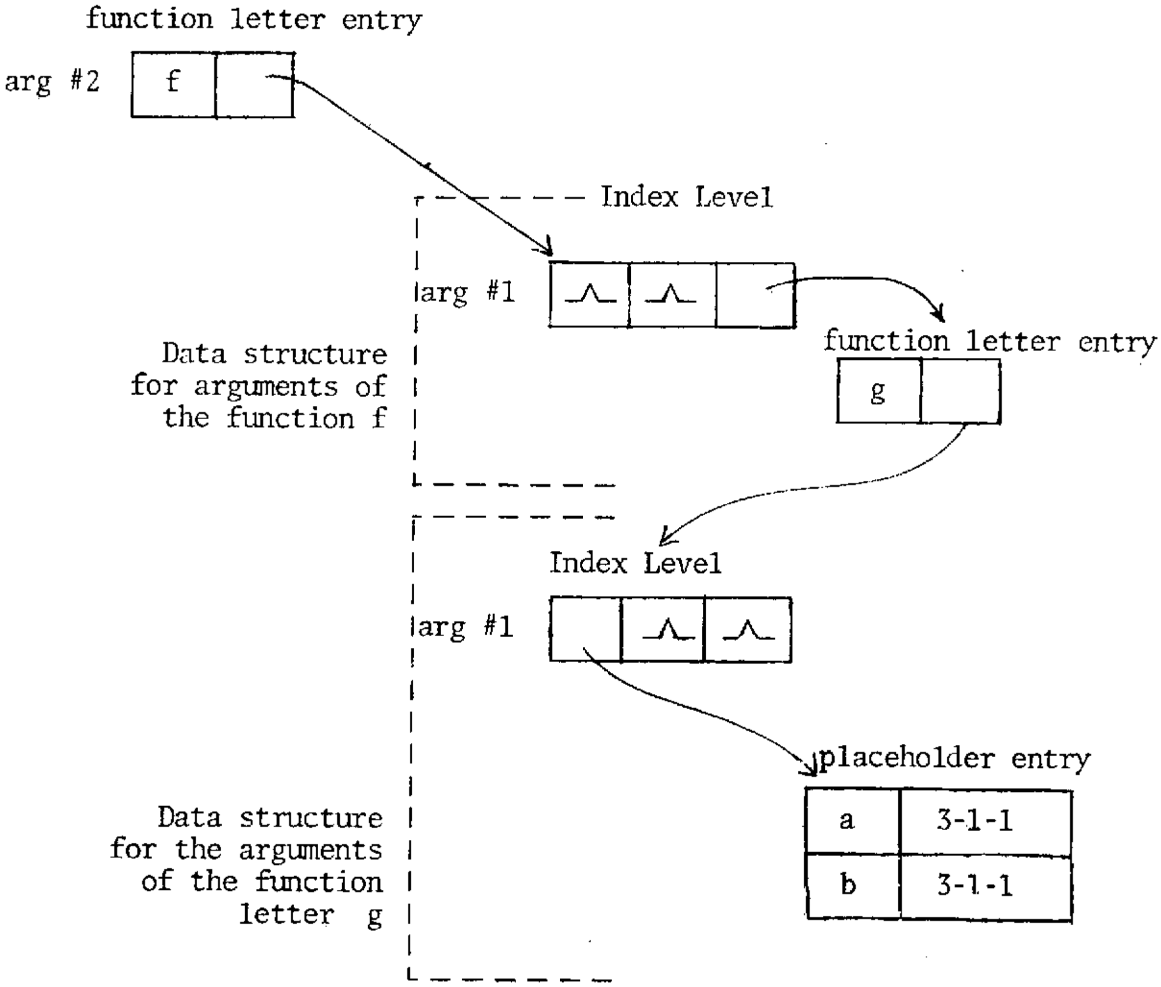
where 2-1-1 and 2-1-2 are the unique substitution set numbers assigned to the application of the substitution sets in [6.3.1] to the template in [6.3.1].

6.4 Function Letter

The third type of basic data base entry is the <u>function letter</u> entry. Each function letter entry consists of two items; the function letter itself and a pointer to another data structure identical in format to the basic data structure describing the arguments of the function. For example, consider the following extended Π-clause:

[6.4.1] $((\beta, f(g(x)), y), \{[P]/\beta, [a,b]/x\})$

The corresponding <u>function letter</u> entry for the second argument position would appear as follows:



NOTE: ⋏ = null pointer

The number 3-1-1 in the placeholder entry is the unique substitution set number assigned by the system to the application of the substitution set in [6.4.1] to the template in [6.4.1]. Note that the occurrence of a new Index Level entry defines a new level in the recursive definition of the data structure. The null pointers ($\underline{\Lambda}$) present in the placeholder and free variable positions of the first level entry in [6.4.2] are caused by the absence of any free or placeholder variables in the substitution set for the first argument position of the function f . Similar reasons cause the null pointers in the Index Level entry for the arguments of the function g .

7. Global-Parallel Unification

The majority of the work which must be performed in a QA system using a theorem proving approach, is based upon the principle of resolution (basically a pattern matching operation). Robinson [15] developed an algorithm which successfully adapted the principle of resolution to automatic theorem proving (see Chang and Lee [4] for a more detailed description). The unification algorithm described below is an adaptation of the Fishman-Minker algorithm [8](which is based on Robinson's algorithm), adapted in order to allow it to handle parallelism in unification and to work efficiently on the particular data structure described in this paper. In a QA system, the single most frequently used function of the system is the unification algorithm. As a result, efforts have been made at optimizing the efficiency of the unification process [1, 2, 10, 16]. What we have tried to do in this paper is to describe an efficient unification algorithm which can work with a large data base and by working with the special data structure can perform unification in parallel across all clauses of the data base.

The usefulness of the data structure (described in Section 6) to our unification algorithm results from the partitioning that it performs on the data items. The data items have been divided into three clases: placeholder variables, free variables, and function letters. Furthermore, this division has been performed at each argument position within each n-tuple partition of the data base.

7.1 Basic Properties of Unification

The global-parallel unification algorithm makes use of this partitioning of the data base items and the following basic properties of unification to enable it to identify all the literals (and their associated clauses) in the data base which are unifiable with a specific input clause with only one application of the unification process. (For clarity, the following examples are represented in standard first-order predicate calculus form):

1. Two literals in which non-matching constants appear at the same argument position cannot be candidates for unification.

   For example, the two literals

   $P(a,x)$, and

   $P(b,y)$,

   cannot unify regardless of what x and y are because the constants a and b in the first argument position can never be made the same.

2. Given a literal which has an arbitrary constant in its $n^{\underline{th}}$ argument position and a second literal which has an arbitrary function letter in its $n^{\underline{th}}$ argument position, the two literals can never unify regardless of the remainder of each literal.

   For example, the two literals,

   $P(a,x,f(x),g(y))$,

   $P(a,x,b,g(y))$,

can never unify since the third argument entries b and f(x) can never be made the same.

3.a Two literals in which non-matching function letters occur in the same argument position can never unify.

.b Additionally, literals with matching function letters in the same argument position can unify only if none of conditions 1 through 3.a holds for any of the arguments of the two functions. For example, the two literals,

P(x,f(y),k) , and

P(x,g(y),k) ,

cannot unify because the function letters f and g in the second argument position do not agree. In addition, the two literals

P(x,f(g(h(a))),z) , and

P(b,f(g(h(b))),z) ,

cannot unify because the second argument positions are not the same since the constants a and b do not agree and can never be made to agree.

4. Unification can only occur between literals of the same degree.

7.2 The Unification Process

Consider now the data structure we have developed in view of these basic rules. We have divided the data base into disjoint subsets forming a separate subset for each set of literals of degree n . By considering only elements of a particular subset during unification, we have assured ourselves that we will consider for unification only literals of the same degree and that we will consider all literals of the same degree. Recall that for all literals of the same degree we have partitioned, by argument position, the elements of the entire data base into placeholder variables, free variables and function letters.

Suppose that a particular literal upon which we want to unify (termed the input literal) is also represented in such a form. Then, by virtue of the data structure, while searching for a unifier for the input literal, all literals within the data base which will unify with the input literal and the most general unifier associated with each unifying clause will be found. This is achieved by keeping track at each argument position of the substitution set number-pairs (one substitution set number from both the input literal and the data base item) which have not violated any of the conditions set forth in rules 1 through 3.

For example, suppose we are examining the second argument position during a particular unification process. If the input literal representation indicates that the constant b is a value which may be substituted in the second argument position in the input literal (with an associated substitution set number of 1-1-1) and the data base indicated that the constant b can be substituted in argument position two for substitution set numbers 5-1-1, 10-1-2 and 15-1-3, then a valid substitution has been found for the substitution set number-pairs (1-1-1, 5-1-1), (1-1-1, 10-1-2) and (1-1-1, 15-1-3). The fact that a valid substitution for these pairs has been found will be recorded in a data area internal to the unification process (we term that area the mgu set). The fact that a valid substitution entry has been found at this argument position will be indicated to subsequent argument positions by the presence of the mgu set for the particular substitution set number-pair (SSN-pair). If in the example just cited one of the SSN-pairs with a valid entry of b just found had not produced a valid entry at a previously processed argument position, this would have been indicated by the absence of a mgu set for the particular SSN-pair.

Unification is performed in parallel across the entire data base one argument position at a time. When processing at one argument position is complete, a check is made of the SSN-pairs which are still candidates for unification and those, and only those, SSN-pairs are considered at all future argument positions for possible unification. When all argument positions have been processed, the SSN-pairs not eliminated from consideration represent the unifiable clauses and the associated mgu sets represent the most general unifiers.

7.3 Unification Algorithm

Unification is achieved at a particular argument position by unifying only between the particular subsets (placeholder variables, free variables, function letters) which might possibly produce valid unifiers. The general unification algorithm, modified to work with clauses represented in extended π-notation using the data structure just described consists of a set of subroutine calls. One of the particular subroutines is called depending upon the type of sets between which unification is being attempted. It is assumed that the variables in the template of the input literal have been standardized apart from all variables occurring in the data base. The subroutine of the unification algorithm can be outlined as follows:

a) <u>Unify data base placeholder variables and input literal placeholder variables</u>

Compare the two sets element by element. When the sets are both ordered alphabetically, the comparison, which is in fact a set intersection of the two sets, becomes a simple process.

Where there is an element match, consider the cross-product of the two sets of substitution set numbers listed under both the data base and input entries. If the SS-number pair satisfied unification criteria in all previous argument positions processed (indicated by a current mgu entry for the SSN-pair) but not for this argument position, make an entry in the mgu for this argument position indicating the placeholder variable match found. If a previous unifier for this argument position has been found, indicating in this case that the intersection of two placeholder variable sets has produced more than one common element, this additional entry should be added to the already current mgu. If no mgu exists for an SSN-pair this indicates that at some previous argument position unification between the two Π-clauses represented by the SSN-pair was found to be impossible. Therefore, the placeholder variable match just found should be ignored.

b) Unify data base placeholder variables and input literal free variables

Each free variable being considered for unification will automatically allow the substitution of any placeholder variable for it (unless the free variable has already previously been restricted - see Section 7.4). Therefore, this subpart of the unification process is quite straight-forward. The cross product of the SS-numbers of each placeholder variable and each free variable are formed and the corresponding entries made into all current mgu's as was described in (a).

c) *Unify data base free variables and input literal placeholder variables*

This operation is identical to that described in (b).

d) *Unify data base free variables and input literal free variables*

Again, each element of these two sets automatically provides for valid unification, regardless of which variable is substituted for which (again assuming that neither free variable has previously been restricted). Arbitrarily one variable is chosen to be substituted for the other and the entries for the global mgu are generated identically to that described in Section (a).

e) *Unify data base free variables and input literal function letters*

Each free variable in the data base can have as a legal substitution an arbitrary function provided that the variable does not occur in the function (again assuming the free variable has not been restricted). Therefore, inclusion of all possibilities of function letters being substituted for all free variables is made in the mgu sets which are still current for this argument position.

f) *Unify data base function letters and input literal free variables*

This operation is identical to that described in (e).

g) *Unify data base function letters and input literal function letters*

No unification for a particular argument position can be achieved by inspecting two function letters; only entries which *might* unify can be identified. This is achieved by comparing the two sets of function letters element by element. When these sets are in alphabetical order this comparison, which is in fact a set intersection, becomes a simple process. When a match is found, this indicates that the two argument positions agree at least to the function letter, but no more can be told. The entire unfication algorithm must now be recursively called to process the argument sets of the two functions to determine if they can contribute unifiable sets to any mgu. Upon return from the recursive call of the unification process, the processing of the two function letter sets continues.

Note that no searching for unifiers is performed between elements of placeholder sets and elements of function letter sets. Such unifiers do not exist and comparing items from these two sets in search of possible unifiers would be a waste of time. We are able to avoid such useless processing due to the manner in which we have structured the data base. By isolating the various types of entities at each argument position into disjoint subsets, such non-productive searches can be avoided by never processing the entities of the two sets against one another.

It should be noted that the input literal to global parallel unification is an element of an extended π-clause and therefore may have more than one substitution set attached to it. Thus global parallel unification is actually performing unification globally across the entire data base for a set of similarly structured input literals.

7.4 Indirect Restriction of Free Variables

Most unifications which are performed will include the unification of a free variable with either a placeholder variable or a function letter. Such a unification has the effect of placing an induced restriction on the free variable concerned for unification at all future argument positions and for all argument positions previously processed. In effect, the free variable in question can no longer be considered a free variable for the duration of this specific unification.

For example, consider the following two literals:

[7.4.1] $((\beta,x,y),\{[P]/\beta,\ [b]/y\})$

[7.4.2] $((\beta,u,z),\{[P]/\beta,\ [a]/u,\ [b]/z\})$

The free variable x in argument position 2 of [7.4.1] must be restricted to the constant a in order to allow the literals to unify at argument position 2.

Problems may occur in such cases when the newly restricted free variable occurs at another argument position in the same clause. Care must be taken to assure that previous restrictions made will be taken into account in unification attempts later in the same SSN-pair, and unifications made at previous argument positions which were accepted as valid must be rechecked for validity and changed if necessary.

For example:

[7.4.3] $((\beta,x,y,x),\{[P]/\beta,[b,c,d]/y\})$

[7.4.4] $((\beta,u,u,v),\{[P]/\beta,\ [a,b,c]/u,\ [c,d,e]/v\})$

a. in the second argument position, x in [7.4.3] is restricted to the set of constants [a,b,c] by the placeholder variable u in [7.4.4].

b. in the third argument position, u and y are restricted to the set [b,c] (the set containing the common elements of the substitution sets for the two placeholder variables). This in turn affects the previous restriction on x which must be changed to be the same as the new restriction for u .

c. the fourth argument position restricts x in [7.4.3] to the set [c,d,e]. However, in view of previous restrictions made upon x , both x and v must be restricted to the element c (the common element of the previous restriction of x by u and the new restriction imposed by v). However, in order for the two literals to unify, x and u must unify in argument position 2. Therefore, the previous restriction on u ([b,c]) must also be changed to the element c (the same as the restriction on x). This also causes y in argument position 3 of [7.4.3] to be restricted to the element c since u and y must agree in the third argument position.

Note that if the last element of the substitution set for [7.4.4] had been [d,e]/v instead of [c,d,e]/v , unification would have been impossible between the two literals even though unification at each argument position, independent of all other argument positions, is possible. This reduced substitution set for v would have caused a restriction on x in step (c) of the previous description that would have made x inconsistent with restrictions placed on it in step b .

Whenever a free variable is restricted by the unification algorithm presented in 7.3, an entry is made in the mgu set associated with the particular unification indicating which variable has been restricted and how it has been restricted. A check must be made to see if this previously unrestricted variable has been used at a previous argument position. If so, the effects of the newly placed restrictions must be reflected in all previously processed argument positions using this variable.

Prior to using any free variable in the unification algorithm a check is made to see if the variable has previously been restricted. If it has, the information stored in the mgu set at the time of its restriction is used by the unification algorithm. This will have the effect of changing the variable type of the restricted free variable to the type variable dictated by the restriction made, and thus affect the subroutine called by the unification algorithm. For example, if a unification between two free variables was about to take place and one of the free variables was found to have been restricted to a placeholder variable, the unification operation that would take place would be that of unifying a placeholder variable and a free variable (subroutine b) even though the two variables originally presented to the algorithm were free variables.

Note that the operations described here dealing with how the unification algorithm handles restricted free variables are the parallel of the operations that are performed in the composition of unifiers in a conventional approach to unification. What we refer to as the mgu set contains substantially the same information as a most general unifier in a standard system.

8. Advantages and Disadvantages of Global Parallel Unification

There are several advantages for the approach to unification presented in this paper. The most obvious one is the globalness of the unification itself. Once the problem-solving portion of the system decides which particular problem is to be solved next (which particular literal is to be unified upon) it is possible to find all operators (unifiers) in the entire data base in one operation. It is never necessary to return to the data structure to produce another unifier for the same literal. If a previously produced unifier for a particular literal does not lead to a satisfactory problem solution, one can simply select another unifier from the set previously produced.

Along the same line, by having all the unifiers for a particular literal available at one time it is possible to select the best unifier (most likely to produce a problem solution) and use that unifier first. This gives the added capability of choosing the best clause from the data base to resolve with the selected literal. This will of course require the development of heuristics to determine what are the characteristics of a good unifier (see Section 9.2). By having all unifiers available at the same time, it will be possible to vary the criteria for determining the best unifier based upon the state of the partially solved problem at the particular instant required and the characteristics of the input literal itself.

The unification process is position independent: that is, the unification at any particular argument position does not depend upon unification being performed on any preceding argument position. Arguments may be processed in any order, and this order may be changed arbitrarily from one unification to the next. Advantage could be taken of this capability by determining the order in which arguments are to be processed based upon the characteristics of the data contained in the various argument positions of the input literal and the data base items. Since each succesive argument position can continue for consideration in the global mgu set only those SSN-pairs which have produced valid mgu set entries at the previously processed argument positions, a judicious ordering of the argument positions could have a great effect upon limiting the number of candidates considered at each state of unification. Heuristics could be developed to aid in selecting the order in which argument positions should be processed (see Section 9.1).

It is important to note that this approach to providing a data structure and a formal mechanism for parallel unification is independent of any other aspect of a question-answering system. Therefore, any refinements of resolution, such as those presented in Chang and Lee [4], is still applicable and the various possible search strategies [11] developed can still be used.

The global parallel approach to unification has several potential disadvantages. Given a system in which there is not a great deal of similarity between the clauses of the data base, the complex data structure required by this system could significantly increase the amount of overhead required to store the clauses. This could impact heavily on the amount of storage area required to store the items of the data base. Similarly, the processing capabilities of global parallel unification would be wasted if each step of the unification were working in effect on individual unique literals rather than on a batch of syntactically similar ones. The processing requirements for such a system could quickly escalate to many times that which would be required to process the data in a conventional QA system. The point at which the characteristics of the data base indicate that a global parallel approach would be wasteful is unknown. Research in this area must be conducted to determine where such a breaking point is and how likely it is that data bases exhibiting characteristics useful to a global parallel approach exist.

Given that there are data bases which could be more efficiently handled in a global parallel manner, there still is a question as to whether or not such a scheme provides an overkill capability. That is, must such an exhaustive approach be used in order to achieve the increased efficiency. For example, if a specific problem can be solved by unifying the input literal with only those data base clauses which are unit clauses, is it necessary to take the time to find all other unifiers of lengths greater than one? Possibly an approach should be taken where unification is performed globally for all unifiers of a specific clause length with the shortest being done first. This would then, of course, require a separate pass through the applicable portions of the data base for each unification performed. This is another area in which some experimental testing should be done to determine the tradeoffs in the various approaches.

9. Heuristics

As was mentioned in the previous section, global parallel unification requires the introduction of some new heuristics in order to guide the unification process to a reasonably efficient solution. In this section some suggestions for heuristics are presented and a tentative evaluation of their usefulness is made.

9.1 Argument Selection

Global parallel unification is done globally across the entire data base one argument position at a time. Any SSN-pairs which do not successfully unify at any one particular argument position cannot possibly produce a valid unifier at a later point. All SSN-pairs which produce a valid unifier at the first argument position processed must be included for consideration at all subsequent argument positions until they fail to produce a valid unifier. In order to reduce the number of candidate SSN-pairs at each argument position, it would be beneficial to develop a heuristic which would be capable of ordering the argument positions for processing such that those producing the least number of unifiable SSN-pairs would be processed first.

There are several approaches which could be taken to achieve some level of this capability by using the characteristics of the data stored in the data base. These approaches differ as to the computational complexity required to compute the heuristic value, the amount of additional storage media required to store the data used by the heuristic, and the confidence that can be placed in the derived heuristic values.

For example, argument positions could be selected for evaluation based on an upper bound on the total number of possible substitution set number pairs which could be generated by unifying a particular argument position of the data base with the associated argument position of the input literal. This upper bound could be computed in several ways. The total number of unique substitution set numbers occurring under the argument positions of the data base and input literal could simply be multiplied. This would give a very rough upper bound which could be computed cheaply. The information required to compute this heuristic value requires minimal storage space. The confidence level of such a heuristic may be questionable due to the very gross characteristics of the measure supplied.

A more sophisticated approach for generating an upper bound on the number of possible SSN-pairs which takes advantage of the characterists of function letter and placeholder variable items could be developed. First of all, this limit need not include in the number of generated SSN-pairs any consideration for possible unifications between function letter and placeholder variables since such unifications are impossible. Additionally, the manner in which items from these two sets interact with themselves have characteristics which can reduce the total possible number of generated SSN-pairs. For example, given two placeholder variable or function letter sets one consisting of n unique entries and the other consisting of m unique entries, it is possible to find no more than minimum (m,n) occurrences where the two sets have identical entries. Additionally if the input clause has no specific entry with more than h associated substitution sets and the data base has no specific entry with more than k associated substitution sets, then the matching of any two particular entries from the two sets can produce no more than $k \cdot h$ SSN-pairs. Since we have already found that no more than minimum (m,n) matching entries can be found, there can be no more than minimum $(m,n) \cdot k \cdot h$ valid SSN-pairs produced by intersecting the two sets.

A heuristic measure taking advantage of the properties of placeholder variable and function letter sets noted above could provide a much more restrictive upper bound on the number of unifiable SSN-pairs which can be generated at any argument position. This would especially be true in cases where the size of one function letter (or placeholder variable) set is expected to be much larger than the other, as one would expect to find in comparing a global data base with the elements of an input literal. This heuristic requires the storing of a significant amount of information in the data structure in order to compute the heuristic value. The calculation of the heuristic value will also require a significant amount of additional computation at the time of unification.

9.2 Unifier Selection

The purpose of finding a unifier in the course of a deduction is to provide an operator which is capable of reducing the problem (eliminating a literal) being solved. Depending upon the structure of the unifier and the data base clause to which it applies, the resulting reduced problem may have widely varying characteristics. As presented in this paper, data base information used in the unification process is based solely on the literals of a clause and contains no specific information concerning the clause with which the literal is associated. Such information could be included in the data base and characteristics of the clause (such as clause length, complexity, etc.) could be used in determining the most desirable unifier to select. For example, given two unifiers to select from, if one involves using a unit clause from the data base, it is the most desirable to use since the problem solution step involving a unit clause will reduce the problem to be solved (eliminate a literal) without introducing additional subproblems. This characteristic can be generalized for any unifier regardless of the length of the associated clause; the shorter the unifying clause, the fewer literals the resultant clause will have.

Other characteristics of the clause may also be meaningful. For example, the particular literal upon which one has successfully unified may have such a simple structure that it appears to be the natural candidate to be selected as the operator for the problem to be solved. However, the remainder of the clause may have characteristics (such as being extremely complex) that make the clause as an entity a poor selection as a problem reducer. Since any use of a unifier developed will cause the inclusion of all other literals in the same clause into subsequent problems to be solved, it would be worthwhile to be able to investigate the clause-oriented ramifications of using a unifier before actually using it.

The unifier itself, independent of the clause it is associated with, also contains information which is useful in selecting the most desirable unifier. The more variables of the literal template which are specifically known (variables for which there appears a specific substitution set of constants), the fewer free variables there will be at the next step of the problem solution. The fewer free variables at the next step, the less difficult the subsequent unification steps will be and the smaller the number of subsequent unifiers that will be produced. This provides the rationale for investigating a heuristic which enables unifiers which leave the fewest variables of the template literal unspecified to be expanded first.

Similarly, consideration should be given to the number of substitutions in the unifiers which will cause the introduction of additional function letters into the solved clause. Solving problems which have function letters as arguments requires recursive calls of the unification process for each level of function complexity (function nesting). The more complex the function, the more complex the unification process will be for solving the resultant clause. An attempt should be made to defer processing unifiers which will introduce new functions at argument positions; the more complex the functions introduced, the longer the use of the unifier should be deferred. This helps to keep from introducing unnecessarily complex problems in an attempt to solve fairly simple problems especially if a simpler solution is readily available.

10. Conclusion

In this paper we have presented a new approach to handling large data bases in a QA system environment. We have presented a modification to the unification procedure which can produce all the unifiers for the entire data base for a specific input literal in parallel. We have defined a data structure to handle our parallel search requirements in a powerful and efficient manner.

We are now at a point where significant experimentation should be undertaken in an effort to aid in formalizing and verifying our concepts. Various trade-offs in terms of time and space must be explored. The question of creating hybrid systems, combining the concepts of a totally parallel system and that of a standard QA system, should be explored. New areas of capability using the data base structure and the concept of parallelism should be investigated. Such a task is presently being conducted at the University of Maryland[1] in attempting to use semantic information within a system structured such as this.

The problem of determining which approach to take is unresolved. To date, we have developed what we believe to be some promising techniques for structuring and manipulating large data bases in a QA environment. The fact that these techniques are independent of the search strategy employed in the host QA system leaves a wide latitude of areas of possible application.

[1] Work being conducted by J. McSkimin, G. Wilson, G. Augustson and J. Gishen under the direction of Dr. Jack Minker.

Acknowledgment:

The authors would like to extend their appreciation to Gerald Wilson whose advice and comments provided a significant contribution to the preparation of this paper. They also thank Jim McSkimin for his suggestion that Π-representation be extended to include the predicate name.

REFERENCES

1. Baxter, L.D., "An Efficient Unification Algorithm," Technical Report CS-73-23, University of Waterloo, Ontario, Canada.

2. Boyer, R.S. and Moore, J.S., "Sharing of Structure in Theorem-Proving Programs," in *Machine Intelligence 7*, Edinburgh University Press, 1972, p. 101-116.

3. Burstall, R.M., "A scheme for indexing and retrieving clauses for a resolution theorem-prover," Memorandum: MIP-R-45, Department of Machine Intelligence and Perception, University of Edinburgh, December, 1968.

4. Chang, C.L. and Lee, R.C.T., *Symbolic Logic and Mechanical Theorem-Proving*, Academic Press, New York, 1973.

5. Coles, L.S. et al., "Design of a Remote-Access Medical Applications of Remote Electronic Browsing," Final Report, Contract NLM-69-13, SRI Project 7963, Stanford Research Institute, Menlo Park, California (November 1969).

6. Darlington, J.L., ""Theorem-Proving and Information Retrieval," In *Machine Intelligence 4*, Edited by Meltzer, B. and Michie, D., American-Elsevier Publishing Co., New York, 1969, p. 173-182.

7. Fishman, D.H., Experiments with Resolution Based Question-Answering Systems and a Proposed Clause Representation for Parallel Search. Ph.D. Thesis, Computer Science Department, University of Maryland, College Park, Maryland, 1973.

8. Fishman, D.H. and Minker, J., " Π-Representation: A Clause Representation for Parallel Search," Internal Report, University of Maryland, November 1973.

9. Green, C.C. and Raphael, B., "The Use of Theorem Proving Techniques in Question-Answering Systems," In: *Proc.-1968 ACM National Conference*, Brandon/Systems Press, Princeton, N.J., 1968, p. 169-181.

10. Hoffman, G.R. and Veenker, G. "The Unit-Clause Proof Procedure with Equality," *Computing* Vol. 7, 1971, p. 91-105.

11. Kowalski, R., "Search Strategies for Theorem-Proving." In: Meltzer, B. and Michie, D. (Eds.) *Machine Intelligence 5*, American-Elsevier, New York, 1970, p. 181-200.

12. Minker, J., Fishman, D.H., and McSkimin, J.R., "The Maryland Refutation Proof Procedure System (MRPPS)" TR-208, Computer Science Center, University of Maryland, College Park, Maryland, December 1972.

13. Minker, J., Fishman, D.H., and McSkimin, J.R., "The Q* Algorithm - A Search Strategy for a Deductive Question-Answering System," *Proc. 3rd Int'l. Joint Conference on Artificial Intelligence*, Standford, California, August 1973.

14. Minker, J., McSkimin, J.R. and Fishman, D.H., "MRPPS - An Interactive Refutation Proof Procedure System for Question-Answering," TR-228, Computer Science Center, University of Maryland, College Park, Maryland, February 1973

15. Robinson, J.A., "A Machine-Oriented Logic Based on the Resolution Principle," *J. ACM* 12, 1 (Jan., 1965), p. 23-41.

16. Robinson, J.A., "Computational Logic: The Unification Algorithm," In: Meltzer, B. and Michie, B. (Eds.), *Machine Intelligence 6*, Edinburgh University Press, 1971. p. 63-72.

17. Sussman, G.J. and McDermott, D.V., "From PLANNER to CONNIVER - A Genetic Approach," *Proc. FJCC*, AFIPS Press, Montvale, N.J., 1972, 1171-1179.

Appendix Sample Data Structure

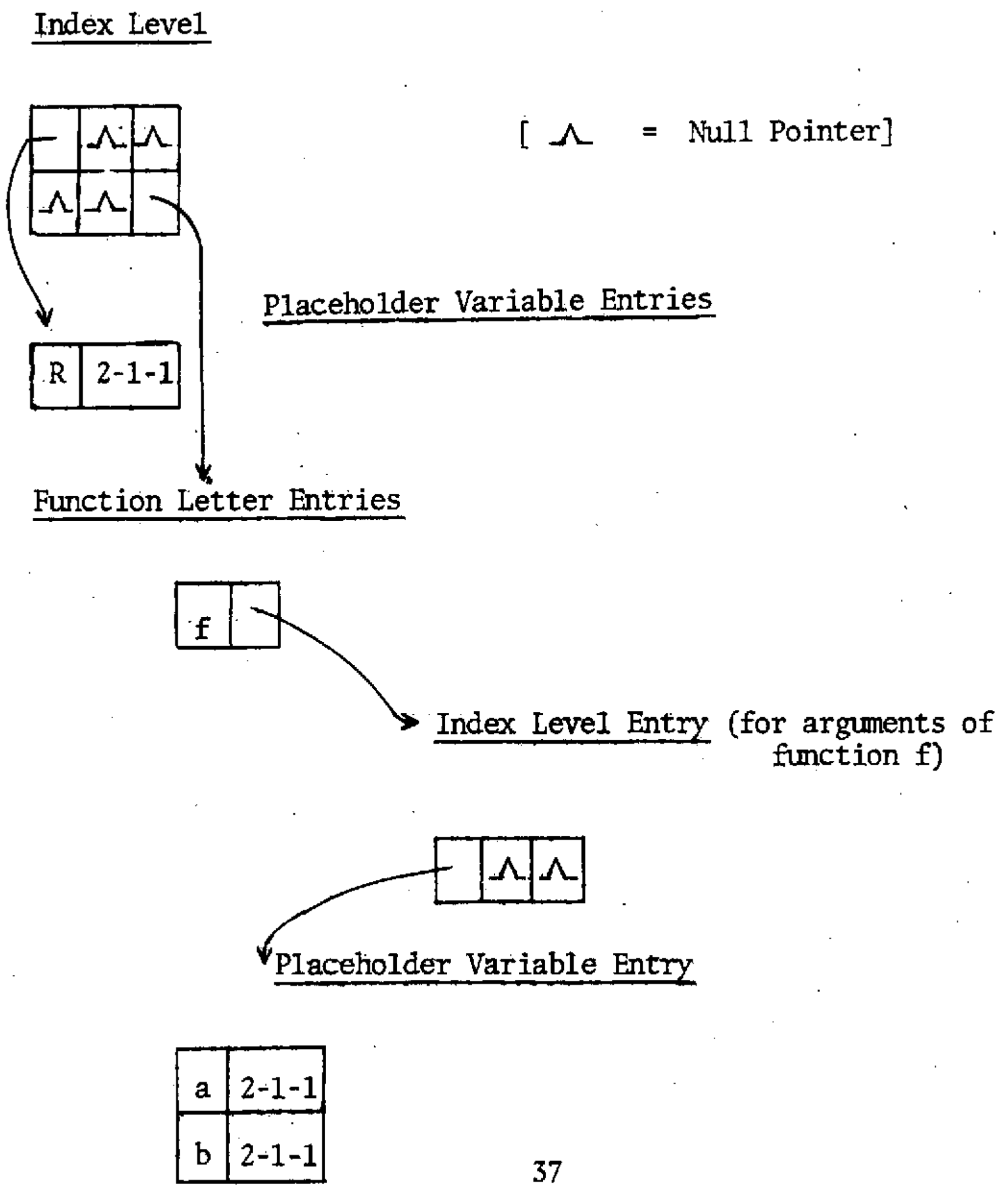
In order that the reader might better comprehend how the various elements of the data structure fit together, we present in this appendix the data structure that would be required to store two fairly simple clauses. These clauses contain only first and second degree literals. Therefore, the corresponding data structure has only two partitions; one for literals of degree one, and one for literals of degree two. In order that we can reflect the occurrence of substitution set numbers in the data structure, we have assumed that the system-assigned unique clause number for clause [A.1] is 1 and for clause [A.2] is 2 .

Sample Clauses in Extended Π-Notation

[A.1] $$\left((\beta,x,y) \vee (\rho,y,z), \left\{\left\{[P]/_{\beta}, [Q]/_{\rho}, [a,b,c]/_{x}, [c,d,e]/_{y}\right\}, \left\{[P,Q]/_{\beta}, [M]/_{\rho}, [a,b]/_{y}, [c]/_{z}\right\}\right\}\right)$$

[A.2] $$\left(\left(\psi,f(u)\right) \vee \left(\alpha,t,f(h(w))\right), \left\{[R]/_{\psi}, [P]/_{\alpha}, [a,b]/_{u}, [c,e]/_{t}\right\}\right)$$

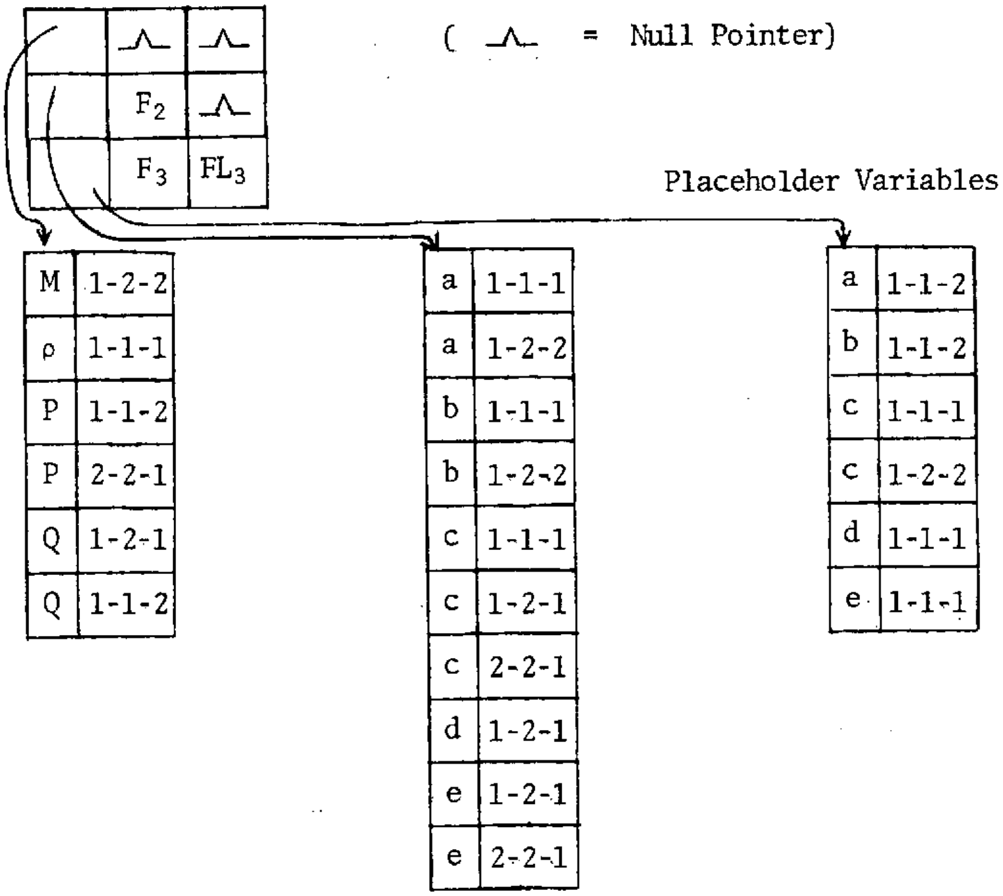
Data Structure For Literals of Degree 1



Index Level
[⋀ = Null Pointer]
⋀ ⋀
⋀ ⋀
Placeholder Variable Entries
R 2-1-1
Function Letter Entries
f
Index Level Entry (for arguments of function f)
⋀ ⋀
Placeholder Variable Entry
a 2-1-1
b 2-1-1

Data Structure for Literals of Degree 2

Index Level



(Λ = Null Pointer)

	Λ	Λ
	F_2	Λ
	F_3	FL_3

M	1-2-2
ρ	1-1-1
P	1-1-2
P	2-2-1
Q	1-2-1
Q	1-1-2

a	1-1-1
a	1-2-2
b	1-1-1
b	1-2-2
c	1-1-1
c	1-2-1
c	2-2-1
d	1-2-1
e	1-2-1
e	2-2-1

Placeholder Variables

a	1-1-2
b	1-1-2
c	1-1-1
c	1-2-2
d	1-1-1
e	1-1-1

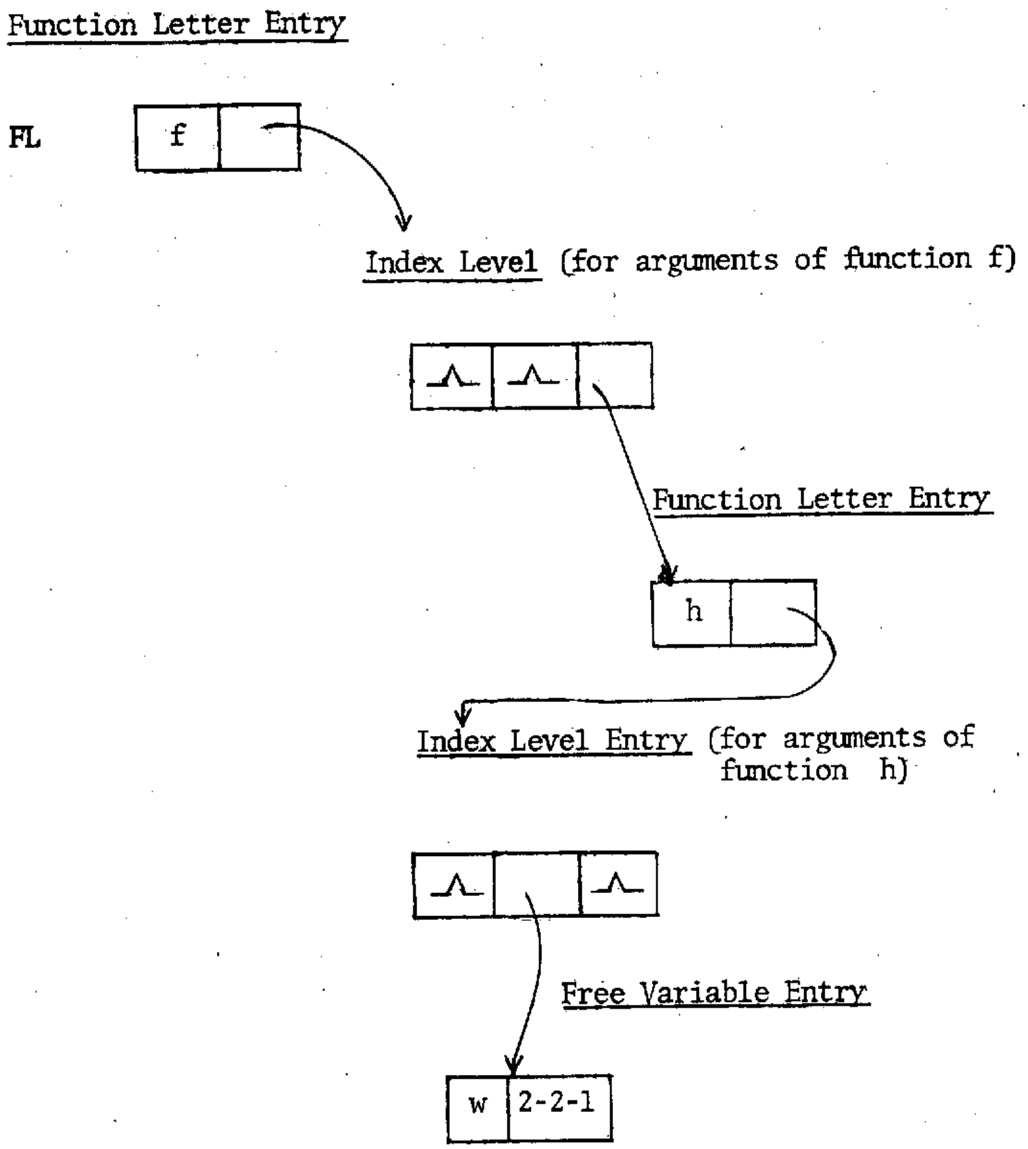
Free Variables

F_2 →

x	1-1-2

F_3 →

z	1-2-1


Function Letter Entry
FL
f
Index Level (for arguments of function f)
⋀
⋀
Function Letter Entry
h
Index Level Entry (for arguments of function h)
⋀
⋀
Free Variable Entry
w
2-2-1