THE GSS PACKAGE

by

J Strother Moore

Department of Computational Logic.

A Package of POP-2 Programs Implementing Generalized Structure Sharing for Resolution Theorem Proving Programs.

PREFACE:

This paper describes a package of POP-2 programs which allows the user to take full advantage of recent advances in data representation for resolution theorem provers. The package does not include high-level functions that perform factoring, resolution itself, etc. This is due to the fact that the package is meant to be incorporated into efficiently written programs. Packaging up high-level functions for general use inevitably slows down the user who has well-defined, task-specific operations to perform.

INTRODUCTION:

This document consists of six sections and an indefinite number of Appendices. Despite a rather general description of the Boyer-Moore clause representation scheme implemented in this package to be found at the beginning of the first section, it is felt the reader should also acquaint himself with the Department of Computational Logic's memo: "Sharing Structures in Resolution Theorem Provers."

The reader is also encouraged to study the POP-2 code of the functions herein described. Not only will a more thorough understanding of the procedures be acquired, but it is likely the individual user will find ways to recode it to his own benefit.

SECTION I -- CLAUSE REPRESENTATION

(1) General

The most basic operation in a resolution theorem prover is the formation of a clause from one or more parent clauses. This operation usually takes the form: delete a literal from each parent, concatenate the resulting strings of literals, and appy some substitution to the result. This is such a trivial procedure that one can actually store a clause as a description of how to build it, rather than actually applying the procedure. This is a very economical way to store clauses since the description of how to build it is quite independent of how long the clause would be if built. The process of building a clause given a description of its constituents will be called "reification."

As noted in the memo "Sharing Structures in Resolution Theorem Provers", such a description automatically standardizes variables apart. Furthermore, it means that after input, no constructing of new terms or literals is performed. Instead, the original terms and literals are understood to be instantiated in the appropriate ways. Substitutions play a major role in this representation.

Substitution components must specify not only the variable bound and the term to which it is bound, but the branch of the derivation the variable has come down, and the branch along which the variables in the term have become instantiated.

Whenever one encounters a variable (with some associated branch), one must ask whether it is bound along that branch. Answering this may require a search up the branch to find a substitution component which binds the

the given variable. However, if one refers to branches in the correct way, it is possible to eliminate this search while dealing with a single clause (or pair of clauses) by moving the relevant substitutions into a direct access area. Such a representation of branches should meet the requirements that branches be mapped one-to-one into integers and these integers be either consecutive or fairly even distributed over some fixed interval.

A representation of branches that meets these requirements is the generalized version discussed in the above mentioned memo. It is possible to associate with every input clause in the derivation of any clause N, a number that codes exactly one path from N to the input clause. If any clause is used more than once in the derivation in distinct places, then it will have different numbers associated with each use. Furthermore, these numbers (called "paths") are not physically stored with each clause but are dynamically computed relative to the desired N.

In every tuple that defines a clause we include two numbers. The first is the number of tips in the derivation tree of the clause (called the "base reference number"), and the second is an increment (called "delta") used to calculate the path numbers to the tips of the tree. It should be kept in mind that a given clause may have several different paths associated with it by many other clauses. Any path number attached to a clause is always relative to some other clause.

(2) Representation of Paths, Variables, Terms, and Literals

Paths: A path is a non-negative integer between 0 and 511. The limit of 2^9 is due to bit-packing considerations only. Effectively this restricts the total number of tips in any single derivation tree to 2^9.

Variables: A variable is an integer between 0 and 15. This again is limited only by packing considerations. Input clauses must be restricted to sixteen or fewer variables; no additional restriction is put on the number of variables that will occur in derived clauses.

Terms: A term is a variable or list. The HD of the list is an identifier which denotes the function (or predicate) symbol. The TL of the list contains as many elements as there are arguments for the function symbol. Each element is a term denoting an argument. Example: f(x,f(y,a)) could be denoted: [F 1 [F 2 [A]]] (depending only on the integers assigned to variables x and y). Note that constants are zero arity functions.

Literals: A literal is a list. The HD of the list is a signed number (real or integer). The sign of the literal is positive if the number is greater than zero, otherwise the sign is negative. The TL of the list is a list defining the term which is the atom of the literal. Example: $\overline{P}(g(x),y)$ could be represented as: [-1 P [G 1] 2].

Since literals are never constructed after input the list defining a literal in a derived clause is EQ to the list that the literal descended from in an input clause. If one wants to attach some item of information to literals in such a way that the information is to be inherited from ancestors, it may be put into the HD of the list defining the input literals -- that is, in the location reserved for the sign. All routines which access the sign of a literal do so through the function LITSIGN which takes a list (defining a literal) and returns TRUE if the literal is positive, and FALSE otherwise. The user may redefine this function if he wishes to replace the sign item by an item other than a number.

(3) Clauses

For the purposes of this paper it will be assumed throughout that a clause will be a five component record with the following definition:

RECORDFNS("CLAUSE", [0 0 0 0 0]) --> BINDINGS --> LENLIT12 --> PARENT2 --> PARENT1 --> BRNDELTA --> DESTCREC --> CONSCREC;

The user is free to define the clause record to contain more components for his own implementation specific information -- in fact, the user is responsible for defining a clause record in his code. However, the record class must have DATAWORD "CLAUSE" and the five selector/updater functions defined above must be given the names above.

Let DATAWORD(CL) be "CLAUSE". Then the contents of the above five components are as follows:

BRNDELTA(CL) = a packed word whose top four bits are available as marker bits to the user, whose next 9 bits specify the base reference number of the node, and whose low-order 9 bits specify the increment to be added to the nodes of the first parent.

PARENT1(CL) = a pointer to the first parent of CL. If this parent has k literals, then the first k-1 literals of CL are from this parent.

PARENT2(CL) = a pointer to the second parent of CL. If this parent has m literals, then the last m-1 literals of CL are from this parent.

LENLIT12(CL) = a packed word whose top bit is used as a marker bit for output, the next 7 bits specify the number of literals in CL (ie., k + m - 2), the next 7 bits specify the number of the

literal resolved upon (and thus to be deleted from) the first parent, and the low-order 7 bits specify the number of the literal resolved upon in the second parent.

BINDINGS(CL) = a strip of 2n full words specifying the n substitution components to be applied to the parents of CL to build CL.

Clearly three pieces of information in the above clause definition are redundant: the base reference number, delta and the number of literals in CL are all recursively computable from the two parents. As a matter of fact, so is the substitution in BINDING(CL). These items are included to speed up processing.

To free the user from packing and unpacking the BRNDELTA and LENLIT12 components of the clause, the following functions are provided:

MARKBITS: clause → integer denoting the four marker bits,

BREFNUMBER: clause → base reference number of clause,

DELTAP1: clause → delta for first parent,

CLLENGTH: clause → number of literals in clause,

MLIT1: clause → literal resolved upon (and missing) from first parent,

MLIT2: clause → literal resolved upon from second parent.

These functions select and update the proper bits in the appropriate components of the clause record concerned.

There are two special cases to be recognized: input clauses, and factors. Input clauses have NIL in PARENT1(CL) and a list of the literals in the clause in PARENT2(CL). CLLENGTH(CL) is the number of literals in the list. BINDINGS(CL) is the NILSTRIP (a full word strip of DATALENGTH zero). All other components are zero.

Factors are unique since they have only one parent, from which a literal must be deleted. In order to carry this out neatly in this frameword, a special clause exists called UNITDUMMY. This clause appears as a unit input clause with a single (dummy) literal. A factor thus has UNITDUMMY in PARENT1(CL) and the actual parent in PARENT2. MLIT1(CL) is set to 1 and MLIT2(CL) is set to the number of the literal to be deleted from the parent. The BINDINGS component contains the unifying substitution and all other components are defined as usual.

There are two function to recognize input clauses and factors, ISINPUT and ISFACTOR. Both take clauses and return truthvalues.

(4) Substitutions

It should be noted that since substitutions are never physically applied, one always binds variables to the original input terms instantiated along some branch. Since the variables occuring in these terms have come down the same branch as the term, the branches of all the variables are the same. Of course, it is possible (and quite likely) that a variable within some instantiated input term is bound to a term from some other branch.

Variables are four bit integers, and paths are nine bit integers to allow a variable and two paths to be packed into a single word. A term must occupy a full word. Since a substitution component specifies a variable, its path, a term, and its path, two words are thus sufficient.

If there are n components in a substitution then the substitution is a strip of 2n full words. Positions 1 and 2 are devoted to the first component, 3 and 4 to the next, etc. Position 1 holds a packed word with the variable bound in the top four bits, its path in the next nine bits, and the term's

path in the low-order nine. Position 2 holds a pointer to the (list defining the) term.

(5) Constructing Clauses

Since the user actually defines the clause record, this package takes no responsibility for constructing clauses. However, the following two functions are provided to compute the five components required by the representation.

CONSCOMP takes four arguments: CL1, LITNO1, CL2, LITNO2. It leaves five components on the stack (from topmost to bottommost): the current binding strip (as returned by GETBINDS -- see Section V), the packed IENLIT12 component with the CLLENGTH being CLLENGTH(CL1) + CLLENGTH(CL2) - 2, and the MLITs being LITNO1 and LITNO2 respectively, CL2, CL1, and the packed BRNDELTA component with CL2 treated as the second parent. (The marker bits of the final component are zero.) If the BREFNUMBER of the clause cannot be packed into 9 bits, error 33 occurs.

CONSINCOMP takes a list of literals and returns the five record components sufficient to define that list as an input clause. They are returned in the same order as CONSCOMP.

Thus, if CONSCREC is as defined on page five, then to build an input record for $P(x,y)\ \overline{Q}(A)$, it is sufficient to call:

```
CONSCREC(CONSINCOMP([[+1  P  1  2] [-1  Q  [A]]]));
```

To build the clause resulting from resolving literal number LITNO1 of clause CL1 with literal LITNO2 of CL2, it would be sufficient to call:

```
CONSCREC(CONSCOMP(CL1,LITNO1,CL2,LITNO2));
```

subject to the definition of GETBINDS in Section V. Factors are built if CL1 is set to UNITDUMMY and LITNO1 is set to 1.

Since the user defines the form of the clause record it is not possible for this package to automatically define UNITDUMMY. This must be done by the user's code after the package has been compiled. The following code would be sufficient if CONSCREC is defined as above:

```
CONSCREC(CONSINCOMP( [[+1   DUMMY ]])) → UNITDUMMY;
```

SECTION II -- DYNAMIC HANDLING OF CLAUSES

(1) Dynamically Determining Variable Bindings

Since each clause is also a node in the derivation tree of other clauses, we can use the words "clause" and "node" to refer to the same objects in different contexts. In order to determine if a variable, v, of some path, p, is bound as of node N, it is necessary to trace the branch specified by p through the derivation of N. The following procedure is used:

(1) if N is an input node, return FALSE, otherwise, go to step (2).

(2) if v of p is bound by the substitution in BINDINGS(N), get the term and its path from the component and go to step (3), otherwise, go to step (5).

(3) increment the term's path by the total amount of decrementing of p that has occurred since the original node N, go to step (4).

(4) return the incremented path, the term, and TRUE.

(5) if p is greater than or equal to DELTAP1(N) then go to step (6), otherwise, go to step (7).

(6) decrement p by DELTAP1(N) and replace N by PARENT1(N), then go to step (1).

(7) replace N by PARENT2(N) and go to step (1).

The following two functions are available to the user for dynamically interpreting substitutions.

BOUNDAT (Bound At) takes three arguments: a variable, a path, and a node. It returns FALSE if the variable of that path is not bound in the substitution at the node. It returns TRUE, the term to which the variable is bound, and the path of the term, if the variable of the path is bound in the substitution at the node.

DYNISBOUND (Dynamic Is Bound) takes a variable and a path as arguments. It assumes that the global DYNSNODE (Dynamic Start Node) has been set to the clause whose derivation tree is to be partially traced. (DYNSNODE is the initial value of N in the above procedure.) If the variable of the path is not bound in the tree of DYNSNODE, it returns FALSE. If it is bound, it returns TRUE, the term to which it is bound, and the path of the term (relative to DYNSNODE).

(2) Dynamically Accessing Literals in a Clause

The function DYNLITRL takes two arguments: an integer n and a clause. It returns the list defining the nth literal of the clause and the path that literal has come down (to the clause). This is done simply by determining if the nth literal has come from the first or second parent, correctly adjusting n and repeating the ancestory check until an input clause is arrived at. If n is greater than the number of literals in the clause, error 45 will occur.

(3) Example of the Use of Dynamic Functions

The function FSYMBCNT returns the number of function symbols in a given TERM of a given PATH. FSYMBLIT counts the number of function symbols in a literal (not counting the predicate letter). The functions are given as examples and are not included in this package.

```
FUNCTION FSYMBCNT TERM PATH;
VARS CNT;
LOOP:  IF ISNUMBER(TERM)
   THEN
   IF DYNISBOUND(TERM, PATH)
      THEN
       -->TERM -->PATH;
      GOTO LOOP;
      ELSE 0; CLOSE;
   ELSE
   1 --> CNT;
   APPLIST(TL(TERM),
      LAMBDA T; CNT + FSYMBCNT(T, PATH) --> CNT; END;);
   CNT;
   CLOSE;
END;

FUNCTION FSYMBLIT N CL;
VARS LIT PATH;
DYNLITRL(N, CL) --> LIT --> PATH;
CL --> DYNSNODE;
FSYMBCNT(TL(LIT), PATH) - 1;
END;
```

SECTION III -- STATIC HANDLING OF CLAUSES

(1) General

If a program must repeatedly deal with variables and literal from a given clause, it is advantageous to write the relevant variable bindings and literals into structures that allow access without requiring searching through the nodes of a tree. This implementation provides the user with the two data structures necessary. The first is an array called VALUE which holds variable bindings, and the second is a list which holds literals. Filling these structures with their proper components is called "loading."

(2) Loading and Unloading VALUE

The VALUE array is a two dimensional array of full words indexed by variables and paths. If v is a variable and p is a path, then VALUE(v,p) is a pair. The BACK of the pair is either -1 which denotes that

v of path p is unbound, or it is the term to which it is bound. If the BACK contains a term, then the FRONT is a packed word with v in the top four bits, p in the next nine, and the path of the term in the low-order nine (just as in a substitution component).

In order to load VALUE with the bindings of some clause or literal, a recursive sweep (called a "zorch") is made through the appropriate tree. Variable bindings appearing at nodes other than the lowest one must be transformed so that the paths are accurate relative to the node from which the zorch was initiated.

The following three functions can be used to load VALUE:

LOADCVIT (Load Clause Variables with Increments to Target) takes four arguments: CL, DELTA, TARGETCL, and TARGETDISP. The function returns no results. All variables bound in the derivation of clause CL are loaded into VALUE subject to the following: (i) the paths associated with variables and their bindings are uniformly incremented by DELTA (in addition to the required transformation noted in the previous paragraph), and (ii) the loading operation along a branch terminates when it arrives at clause TARGETCL with displacement TARGETDISP (ie., accumulated delta of TARGETDISP).

The DELTA argument of this function is normally used to load a clause into VALUE even though a second clause already occupies the lower portion of the array. Thus, two clauses can be loaded so that their paths appear to make them parents of a yet-to-be-formed clause, eg., the paths of the "first parent" can be incremented by the node count of the "second parent". As will be seen, it is possible to unload the "first parent" without disturbing the "second so as to allow the loading of a new "first parent".

The TARGET arguments of LOADCVIT are usefull if one knows that the VALUE array already contains a clause (TARGETCL) which is an ancestor of the one to be loaded (CL). If the paths associated with the ancestor are accurate relative to CL, then in loading CL it is possible to halt when TARGETCL is encountered. However, since a clause may be used more than once in a derivation, it is not enough to encounter TARGETCL alone; the clause that causes the loading to terminate must have been to be displaced by TARGETDISP in loading to CL). The loading of course continues on branches other than the one containing TARGETCL of TARGETDISP. The primary use of this capability is to load a factor of the clause just produced by resolution.

LOADCVI (Load Clause Variables with Increments) is the obvious special case of LOADCVIT. Its arguments are CL and DELTA as defined for LOADCVIT and it loads the entire derivation of CL, incrementing all paths by DELTA.

LOADLVI (Load Literal Variables with Increments) takes three arguments: LITNO, CL, and DELTA. It loads all bindings of all variables involved in literal number LITNO of clause CL. Paths are incremented by DELTA. No results are returned. Note that variable bindings of all depths are loaded. Thus, if x of path 1 is bound to f(z) of path 3, then the bindings of x of 1 and z of 3 are loaded.

There are three unloading commands which are the counterparts of the above functions. Unloading is accomplished by again zorching up the appropriate part of the tree, but upon finding a variable binding it removes it from the value array by assigning -1 to the BACK of the appropriate cell.

UNLDCVIT (Unload Clause Variables with Increments to Target) takes the same four arguments as LOADCVIT. For each binding in the derivation of

clause CL, it transforms the paths by DELTA and then removes the appropriate entry from VALUE. The process halts when the TARGET conditions are met as before.

UNLDCVI (Unload Clause Variables with Increments) and UNLDLVI (Unload Literal Variables with Increments) take the same type of arguments as their LOAD counterparts and perform the obvious deletion of entries.

(2) Loading and Unloading Literals

It is felt that the most natural structure into which to load literals is a list, since the only time one wishes to deal with a string of literals (rather than a single one) is during operations like factoring and subsumption where the programs must be able to cycle through pairs of literals.

LITSLOOP is a circular list of pairs. It is possible to load literals and their times into the loop at any specified location. It is circular to allow easy "appending" of one "list" of literals to another without counting literals. The following function loads LITSLOOP or any other list of pairs.

LOADCLI (Load Clause Literals with Increments) takes five arguments: LITNO1, LITNO2, CL, DELTA, and PTR. It is assumed that PTR is a (pointer into a) list of pairs. The function retrieves literal number LITNO1 through LITNO2 of clause CL and their paths, increments the paths by DELTA, and stores the literals and paths in successive pairs of the list, starting at PTR. The list defining the literal is put into the BACK of the pair, and the associated path is put into the bottom nine bits of the FRONT of the pair. The function returns a pointer to the last element of the list loaded.

Thus, after execution of:

LOADCLI(1, 4, CL, 0, LITSLOOP) --> PTR;

the first four literals of clause CL and their paths are found in the first

four elements of LITSLOOP. PTR will be the list whose HD is the fourth element of LITSLOOP and whose TL is the rest of LITSLOOP.

No mechanism is provided for unloading such lists since the user directly controls access and can overwrite old entries upon reloading.

SECTION IV -- HANDLING OF STATIC STRUCTURES

(1) Accessing and Updating VALUE

The VALUE array can be accessed to determine whether a variable, v, of path, p, is bound in the currently loaded clause or literal. Furthermore, the user can cause additional bindings to be inserted into VALUE in a way that allows sophisticated recursive coding.

The following function determines if variable v of path p is bound by accessing VALUE.

STCISBOUND (Static Is Bound) takes two arguments, v and p. It returns FALSE if BACK(VALUE(v,p))=-1. It returns TRUE, the term to which v of p is bound, and the path of the term, if VALUE(v,p) contains a term in its BACK.

In order to keep track of entries to VALUE after a load operation is completed, there is a global list of 40 elements called BINDSLIST. Associated with this list are two global pointers into it, called BINDPT1, and BINDPT2. Initially the two BINDPTs are set to BINDSLIST. There use becomes clear after the following two functions are described:

BIND takes four arguments: VAR, PATH1, TERM, and PATH2. It returns no results. It constructs a packed word out of VAR, PATH1, and PATH2, and inserts this word into FRONT(VALUE(VAR,PATH1)). The TERM is inserted into BACK(VALUE(VAR,PATH1)). In addition, the pair VALUE(VAR,PATH1) is put into the HD of BINDPT2 and the TL of BINDPT2 replaces BINDPT2.

This has the effect of binding VAR of PATH1 to TERM of PATH2 for subsequent calls of STCISBOUND. In addition, a note of this binding is put onto the BINDSLIST (at BINDPT2).

UNBIND takes no arguments and returns no results. It deletes all bindings made between the current BINDPT1 and the current BINDPT2. It then resets BINDPT2 to BINDPT1.

(2) Accessing and Updating LITSLOOP

It is often convenient to be able to temporarily delete a literal from LITSLOOP. It is also convenient to rearrange sub-lists of LITSLOOP with respect to eachother. We will deal with the second problem first.

Let the clause CL1 have the literals P, Q, and R, of paths 1, 2, and 3. Let CL2 be a clause with 7 nodes in its derivation and having literals A, B and C, of paths 1, 2, and 3. Consider the following sequence of statements:

LOADCLI(1, CLLENGTH(CL2), CL2, 0, TL(PTR1)) → PTR2;

LOADCLI(1, CLLENGTH(CL1), CL1, 7, TL(PTR2)) → PTR3;

Then if one were to call the ELEMENT function for the integers 1 through 6 on TL(PTR1), the result would be pairs of the form: (1 . P), (2 . Q), (3 . R), (8 . A), (9 . B), and (10 . C). In particular, note that the literals of CL1 follow those of CL2. Since it is usual to load the literals of the second parent first, and then repeatedly load literals of successive first parents, it would be nice to reverse the two sub-lists to get the literals of CL1 to precede those of CL2. This can be done by the statement:

TWIST(PTR1, PTR2, PTR3);

which rearranges the circular list involved in the obvious way.

The following table explains the function of TWIST:

	before TWIST	after TWIST
first lit of CL2	TL(PTR1)	TL(PTR3)
last lit of CL2	PTR2	PTR2
first lit of CL1	TL(PTR2)	TL(PTR1)
last lit of CL1	PTR3	PTR3
first free position	TL(PTR3)	TL(PTR2)
last free position	PTR1	PTR1

Thus it is possible to load a second parent, then load a first parent behind it and perform a TWIST to have the literals in LITSLOOP be in the order they are found in a resolvent of the two. Later, a new first parent could be loaded behind the second (ie., starting in TL(PTR2), the first free position) and a new TWIST performed.

In order to delete a literal from LITSLOOP temporarily the following mechanism exists. It is possible to turn on the HIGHONE bit of the FRONT of a pair defining a literal and path by calling the function DELETELIT, which takes such a pair as its single argument and returns no results. The function RESTORELIT takes a pair and turns the HIGHONE bit of the FRONT off. By ignoring pairs so marked one can effectively delete literals.

Two functions are available to make use of this mechanism.

NX (Next) takes two arguments, PTR, and NILPTR, both of which are pointers into a list of pairs (ie., LITSLOOP). It repeatedly takes the TL of PTR until it finds an unmarked pair. If such a pair is found before NILPTR is encountered, the list containing that pair as its HD is returned, otherwise NILPTR is returned. Thus NX behaves like TL except one must specify where its first argument ends (or will be considered to end).

If one uses NX in conjunction with DELETELIT and RESTORELIT, literals can easily be deleted and replaced for recursive coding.

STCLITRL (Static Literal) takes two arguments, an integer N and a list of pairs, PTR. It returns the Nth unmarked literal and its path, starting from PTR.

For example, if LITSLOOP were in the configuration left after the TWIST statement on page 16, the statement:

STCLITRL(3, TL(PTR3)) → LIT → PATH;

would set LIT to C and PATH to 10. If we then executed:

DELETELIT(HD(TL(PTR3)));

and repeated the STCLITRL statement above, LIT would be set to P and PATH to 1. RESTORELIT(HD(TL(PTR1))); would return us to the above configuration. The statement UNTWIST(PTR1,PTR2,PTR3) restores LITSLOOP to its pointer arrangement prior to the TWIST on page 16.

SECTION V -- UNIFICATION AND ASSOCIATED FUNCTIONS

The following four functions exist. The first three require that the bindings of variables be determined and call the function ISBOUND for this reason. ISBOUND is initialized by this package to be STCISBOUND. The user may redefine it a will provided that when its definition involves DYNISBOUND, the global variable DYNSNODE is appropriately set. When ISBOUND is STCISBOUND, the proper clauses or literals must be loaded into VALUE.

OCCUR takes four arguments: VAR, PATH1, TERM, and PATH2. It returns TRUE if the variable VAR of PATH1, occurs anywhere in the PATH2 instantiation of TERM. Otherwise it returns FALSE.

Also assume that CUTOFF is some global initialized to the number of nodes in the loaded clause (ie., its BREFNUMBER component, plus one). Then all variables from the loaded clause will have paths less than CUTOFF, while those from the unloaded clause will have paths greater than or equal to it. The paths of terms bound to variables from the unloaded clause will have to be displaced by CUTOFF to be accurate for further enquiries. The following function would do:

```
FUNCTION SEMISBOUND VAR PATH;
VARS TERM PATH2;
IF STCISBOUND(VAR, PATH)
   THEN TRUE;
ELSEIF PATH < CUTOFF
   THEN FALSE;
ELSEIF DYNISBOUND(VAR, PATH - CUTOFF)
   THEN
    -->TERM --> PATH2;
       PATH2 + CUTOFF;
       TERM;
       TRUE;
ELSE FALSE; CLOSE;
END;
```

Of course, DYNSNODE would be set to the clause that was not loaded, and the initial call to UNIFY to specify the path of the term from the unloaded clause incremented by CUTOFF. The unifying substitution would be available through the BINDSLIST or GETBINDS as usual, and could be removed from VALUE via the usual UNBIND.

SECTION VI -- RECURSIVE CODING

The BINDSLIST, BIND, UNBIND, DELETELIT, and RESTORELIT features allow very natural recursive coding to be written for loaded clauses. For example, assume that the user wishes to write a factor routine which recursively factors factors. Assume that the clause to be factored

is known and stored in some global, and assume its values and literals have been loaded. The factor routine then behaves in the following way: Upon entry it saves BINDPT1 as a local and redefines it to be BINDPT2. Then using the NX function, it successively tries pairs of literals until it finds a pair with the same LITSIGN which UNIFY. After forming and storing the clause using CONSCOMP and UNITDUMMY, it calls DELETELIT on the pair representing the literal to be deleted. It then recursively calls itself. All subsequent factors produced automatically respect the bindings made at this level and the literal deleted. When control is returned to this level, RESTORELIT is called on the pair above to restore the literal. UNBIND is called to remove the bindings made for the last unification at this level, and the next pair of literals is tried. When the routine finally exits from this level, VALUE and LITSLOOP will be as they were upon entry (since BINDPT1 was local when the UNBIND was called).

Similiarly slick subsumption functions can be written. In general it is felt that the sophisticated user will find coding with this implementation quite easy.

APPENDIX I -- CLAUSE STORAGE FACILITIES

A package of functions is available to implement a simple and efficient clause storage mechanism. The memory is composed of full-word strips, called BLOCKS, each of which is capable of holding 32 clause records. The BLOCKS are stored as they are created in another strip called BLOCKDOCK.

BLOCKS are created only when needed. The first clause stored causes BLOCK 1 to be built and inserted into position 1 of BLOCKDOCK. The clause is put into position 1 of BLOCK 1. When the 33rd clause is to be stored, BLOCK 2 is built and the clause is entered into position 1. BLOCKDOCK is expanded when needed (it is initialized at length 20, allowing 20 blocks, or 640 clauses, before its first expansion).

However, the above structure can be transparent to the user through the use of the function MAINCL.

MAINCL is a function which appears as a one dimensional array to the user. MAINCL(1) accesses the contents of BLOCK 1, position 1. MAINCL(33) accesses BLOCK 2, position 1, etc. MAINCL is an updater as well as a selector of course. Assignment to MAINCL(n) will automatically cause the proper BLOCK to be built, or an expansion of BLOCKDOCK, if necessary.

The function SAVE takes a clause record as an argument and stores it in the next available location in MAINCL. It returns the integer specifying which MAINCL location was filled.

The global variable MAXNOBLK contains the current size of BLOCKDOCK. The global variable BLOCKCNT contains the number of blocks currently existing. The global MAINCNT is always set to the highest position in MAINCL to which an assignment has been made. Thus, MAINCNT + 1 will always be free.

APPENDIX II -- INPUT

A package of functions is available to read items from a specified item repeater and return a list of literals in the format required to define input clauses for the GSS System. The reader is referred to SECTION I of this document for specifications of the correct format.

Items are read from a global item repeater called ITEMREP, which must be defined by the user. All functions access this repeater through the function NEXTITEM. NEXTITEM appears to be an item repeater with an accessable buffer. Thus, NEXTITEM() returns an item but also sets an input buffer (IB) to the next item it will return. Thus, if ITEMREP is reading from a stream of the form AAA BBB CCC, then when NEXTITEM() returns AAA, IB will be equal to BBB. The next call of NEXTITEM() will return BBB and set IB to CCC.

The function SKIP takes no arguments and returns no results. It advances NEXTITEM by one step.

The user must initiate this process by assigning the desired item repeater to ITEMREP, and then calling SKIP() to initialize the buffer. Thereafter, both he and this package can read from the item repeater NEXTITEM with the advantage of being able to access the buffer IB.

The basic function, READCLIST (Read Clause List) takes no arguments and returns a list of literals. It is assumed that the first item returned by NEXTITEM will be the beginning of a clause. (The user may wish to follow or precede clauses with special markers or other information. If so, he is responsible for processing it and correctly positioning the repeater before calling READCLIST. Of course, NEXTITEM, SKIP, and IB may be used in his processing.)

The stream of items defining a clause must conform to the following: Variables, constants, functions, and predicate symbols may be any POP-2 identifier. Identifiers beginning with the characters "u" through "z" are classed as variables unless they are previously declared to be non-variables (ie., functions, constants, or predicate symbols) by being made elements of the list SPCLFSYM (Special Function Symbols). An identifier not beginning with the above characters may be classed as a variable only if it is an element of the list SPCLVSYM (Special Variable Symbols). An identifier is classed as a zero arity function (ie., a constant or ground atom) if it is not a variable but not immediately followed by an open parenthesis ("("). Functions of arity greater than zero, and non-ground atoms, must be fully parenthesized. Thus, F X Y is an unacceptable notation for F(X Y), or F(X,Y). (Note: NEXTITEM ignores commas.) A symbol is classed as a predicate letter if it is not enclosed in the parentheses of another expression. Positive literals are just atoms. Negative literals are atoms preceded by the minus sign ("-"). Clauses are strings of literals, followed by a semi-colon (";").

In the lists returned by READCLIST, variables have been replaced by integers. The first variable of a clause is assigned the integer 1, the second, 2, etc. Positive literals are lists with 1 in the HD. Negative literals have 0 in the HD. Upon completing a clause, NEXTITEM will be positioned immediately after the semi-colon.

EXAMPLE: Assume NEXTITEM was positioned so as begin on the string

P (zero, alpha, f(3,y)) -- Q- - R(alpha,y); foo . . .

Also assume that SPCLFSYM = [zero] , and SPCLVSYM = [alpha] .

Then READCLIST() would return the list:

[[1 P [ZERO] 1 [F [3] 2]] [0 Q] [0 R 1 2]]

and the next call of NEXTITEM() would return "foo".

In addition, READCLIST builds three lists that may be of use to the user. They are:

PREDLIST: a list of pairs whose FRONTs are predicate symbols and whose BACKs are the arities of the symbols.

FNLIST: a list of pairs similar to the PREDLIST but containing function symbols and the order in which they were encountered.

VLIST: a list similar to the PREDLIST but containing variable identifiers and the integers which replaced them.

The VLIST is automatically set to NIL upon each call to READCLIST; if the user wishes to inspect it he must therefore do it for each clause read. The PREDLIST and FNLIST are reset only by the user (other than being initialized to NIL upon compilation of the file) and are not actually used by the input package. However, by initializing them to NIL before each new problem set is read, one may collect the relevant symbols and their arities if required.

In the above example, if the PREDLIST and FNLIST were both NIL before the call to READCLIST, the three lists would be as follows after the call:

PREDLIST = [(R . 2) (Q . 0) (P . 3)]

FNLIST = [(3 . 3) (F . 2) (ZERO . 1)]

VLIST = [(Y . 2) (ALPHA . 1)]

Readers interested should note that clauses are actually built by an extremely simple ten state finite state machine, whose states are POP-2 functions named S1 through S10. Each function inspects IB or NEXTITEM(), defines the next state (by assigning the proper function to a global RI), and leaves a single item on the stack. The list is constructed by the single statement: POPVAL(FNTOLIST(LAMBDA; RI(); END;)).