THE SHARING OF STRUCTURE IN RESOLUTION PROGRAMS.

by

Robert S. Boyer
and
J Strother Moore

1972

Metamathematics Unit
Edinburgh University
Scotland

ABSTRACT

A machine oriented representation of clauses is presented that shares substructure to the extent that no terms or literals or lists of literals are constructed after input. The representation eliminates the process of standardizing apart. The representation admits programs as fast and as natural as list representation. Striking analogies with LISP and ALGOL are described. The details of implementations of the representation for both SL and general resolution are presented.

A programmer in computational logic finds himself hard put to represent clauses in computers both naturally and compactly. He turns to lists as the most obvious solution to his problem. Lists accurately represent the tree structure of terms and literals and consequently permit efficient and elegant recursive coding. Two major disadvantages emerge however to mar the perfection of this method of representation. First, lists will typically require ten times more machine storage than the most compact character strip representation. Secondly, the rapid construction and deletion of clauses in list form demand frequent garbage collection.

We here offer a representation of clauses ten times more compact than character strips while it preserves all the virtues of lists. Suppose that C and D are clauses, K is a literal in C, -L is a literal in D, and σ is a most general substitution such that $K = L$. Then the clause

$$R = ((C - \{K\}) \cup (D - \{-L\})) \sigma$$

is a resolvent of C and D. To reify R (by applying σ and cons-ing a list of literals) in a computer may require almost as much construction as was involved in building C and D. We observe, however, that R is completely described by appropriate references to the roles of C, K, D, -L, and σ. This observation is the key to our representation.

The reader may well guess that 'appropriate references' to C, K, D, and -L are easy to program. If K is the k^{th} member of C and -L is the m^{th} member of D then the four-tuple $\langle C, k, D, m \rangle$[1] will do, for example. Treating σ is a harder problem, however, and akin to a problem elegantly solved in LISP.[2] The application of a lambda-expression to an argument was defined by Church as a substitution into the body of the lambda-expression. The LISP A-list obviates this substitution, which would be quite expensive to perform. Instead of substituting a value V, for a bound variable, say x, the LISP interpreter adds (x . V) to the A-list. If x is encountered later, its value is determined by an examination of the A-list.

---

[1] This might be programmed as a list of four things: pointers to C and D and the numbers k and m.

[2] Algol handles a similar problem in calls by name.

Many workers in computational logic have intuited that substitutions in resolution ought to be treated in a way similar to the LISP solution. In fact, J.A. Robinson[3], B. Anderson and R. Popplestone have implemented unification algorithms that do not actually perform substitutions. The main idea in their programs is that to replace a variable by a term in an expression it is sufficient to put the term into the place reserved for the value of the variable. Then if one encounters a variable, one asks: Does it have a term in its value place? If it does, one proceeds exactly as if he had encountered the term rather than the variable.

The extension of this idea to a whole theorem proving program is impeded by two problems:

(1) Whenever two clauses are resolved, their variables must be standardized apart. If this is accomplished by physically changing each variable, one will frequently need to reconstruct a whole clause.

(2) Since a clause may be resolved with many others, a variable may have different bindings on different branches of a search tree. Hence a single value cell is insufficient.

Our solution to these two problems is to expand the proposed 4-tuple

$$\langle C, k, D, m \rangle \qquad \text{(c.f. p. 1)}$$

to a five-tuple

$$\langle c, k, D, m, \sigma \rangle$$

after redefining the concept of a 'substitution component'.

At the tips of the tree defined by such five-tuple clauses are input clauses. It is only at these tips that one may encounter a 'real' literal, term, or variable. If one speaks of a variable, term, or literal, T, as being 'in' a five-tuple clause, C, then he also has in mind a single branch of clauses extending from C to an input clause in which T actually occurs. We call this the branch 'along which T occurs'. We are particularly interested in the branches associated with variables because they automatically standardize variables apart. The following syllogism makes this point clear:

---

[3] Robinson, J.A., 'Computational Logic: The Unification Computation' in Machine Intelligence VI, eds. B. Meltzer and D. Michie, University of Edinburgh Press, 1971, p→63.

(1) To standardize a variable apart, it is sufficient to associate with it the branch of the derivation along which it occurs.

(2) Inductively, if each variable in a clause is already associated with the branch along which it occurs, then in a resolvent we need merely associate with each variable the clause from which it came.

(3) If a clause is a five-tuple as above, then a variable is implicitly associated with the clause from which it comes (it is either in C or D).

(4) Therefore, no work need be done at all (besides creating the 5-tuple) to standardize variables apart.

To take advantage of this 'automatic' standardizing apart we must keep track of the branches associated with variables. To this end we define a substitution component to be a four-tuple of the form:

$$\langle \text{VARIABLE, BRANCH1, TERM, BRANCH2} \rangle$$

where BRANCH1 is the branch associated with VARIABLE and BRANCH2 is the branch associated with TERM. This four-tuple corresponds roughly to the normal substitution component TERM/VARIABLE. Since we never actually apply substitutions, we never construct new terms. Hence TERM is actually a term in an input clause. It follows that BRANCH2 is the branch not only for TERM but also for all the variables in TERM.

When, in unification, we encounter a variable v with associated branch b we ask whether v is bound to a term T with associated branch B. If it is, we proceed as if we had encountered T (of branch B) instead of v. If we wish to unify some variable v with associated branch b and a term T with associated branch B we form the substitution component

$$\langle v, b, T, B \rangle$$

We then store this component in such a way that if we ever encounter v of b again we can find that it is 'really' T of B.

In our implementation of SL-resolution[4] we actually use integers instead of branches in substitution components. This use of integers permits us to determine whether v of b is bound in one reference to an array. (This array is easily updated as

---

[4]Kowalski, R. and Kuehner, D., 'SL-Resolution', to appear in the Journal of Artificial Intelligence.

we move from one branch to another.).

In any binary resolution system one may use binary words instead of branches in substitution components. The binary word is as long as the branch and precisely describes the path of the branch through the derivation. In this case the algorithm for deciding whether v of b is bound is exceedingly interesting and delicate. We believe the reader will find some satisfaction in discovering it for himself. We have implemented such a notation for branches for general resolution. Although very pretty, it requires a significant amount of searching for the bindings of variables (much like searching up an A-list in LISP). To overcome this search we have discovered and implemented a more efficient notation for branches in the general case which uses integers no greater than the number of nodes in the derivation trees of the two clauses being resolved. This is a generalization of our SL-implementation and permits the discovery of binding in a single array access.

In Parts II and III of this paper we describe in close detail our implementation of SL and general resolution.

After we developed our representation of clauses, Pat Hayes, Bruce Anderson, Robin Popplestone and Rod Burstall pointed to its similarity to the implementations of ALGOL and LISP. We were surprised to discover the following:

(1) Standardizing apart as treated by us is remarkably similar to function entry in LISP and ALGOL. In particular, the pushing of local and formal variables resembles the introduction of new variables. Unlike recursive function evaluation however, unification frequently accesses 'values of a variable' other than the most recent value.

(2) Our definition of substitution component is strikingly similar to a generalized notion of ALGOL identifier binding described by Wegner[5]. To a lesser degree our substitution components resemble FUNARG's in LISP.

(3) The array we use in our SL implementation to speed up access to bindings along a branch resembles the ALGOL display.

(4) The search tree of derivations in our SL representation is similar to parallel processing using co-routines. The general representation however resembles a form of parallel processing in which the parallel processes actually redefine one another, simultaneously.

5

PART II

The SL-Resolution Implementation.

Readers familiar with SL resolution will find the notation in the following description somewhat different from that used by Kowalski and Kuehner. However, it is felt that this notation helps clarify our representation of clauses.

SL-resolution operates on chains. These chains are composed of cells which contain literals. We will separate cells in chains by a slash('/'). The left-most cell in a chain is called the most recent cell. A chain is thus of the form:

A B C / D E/

where A, B, C, D and E are literals in the traditional sense.

Before a chain may be resolved with an input clause a literal from the most recent cell must be selected. This selected literal must be the literal resolved upon in all resolutions of the chain with input clauses. We will denote the selected literal of a cell by underlining it, eg.

A B C / D E/

SL has three operations on chains. Extension corresponds to resolution with an input clause. Reduction is similar to factoring but also replaces ancestor resolution. Truncation is a book-keeping device for chains.

To extend upon a chain $\mathcal{C}$ with selected literal L, we must use an input clause $\mathcal{B}$ containing some literal K of sign opposite L such that the atoms of K and L unify via mgu σ. The result of extending $\mathcal{C}$ (the rearparent) by $\mathcal{B}$ (the input or far parent) is the chain $\mathcal{C}'\sigma$ where $\mathcal{C}'$ is the chain whose most recent cell is composed of all the literals in $\mathcal{B}$ except K, and the remainder of $\mathcal{C}'$ is just $\mathcal{C}$. For example, extending A B C / D E/ with input clause -B G H, yields:

G H / A B C/ D E/.

The selected literal, L, in $\mathcal{C}$ is called an A-literal (A for ancestor) in chain $\mathcal{C}'$. Note that in the above example, either G or H must be selected before the chain may be extended upon.

A chain is admissible iff no two literals in any cells of the chain have identical atoms. A chain may be extended upon only if it is admissible and the most recent cell is non-empty.

A chain, $\mathcal{C}$, may be reduced if the atom of some literal L in the most recent cell unifies with the atom of an A-literal (other than the most recent A-literal) of opposite sign, or a non-A literal (other than another most recent one) of the same sign. The result of such a reduction is the chain obtained by deleting L from $\mathcal{C}$ and applying the unifying substitution.

The first case defined above is called ancestor reduction and performs the function of ancestor resolution in other 'linear' systems. The other case is the usual factoring. Two examples of reduction are:

A B C / D B/ reduces to A C / D B/ (factoring)

A B C / D E/ F -B/ reduces to A C / D E/ F -B/ (ancestor reduction)

The final operation in SL is truncation. A chain must be truncated when the most recent cell is empty. This condition can be caused by extension with a unit clause, reduction of all of the most recent literals, or truncation. The result is simply the chain obtained by deleting the (empty) first cell and deleting the A-literal (i.e., previous selected literal) from the exposed cell (which is now the most recent). Thus,

/ A B C / D E/ truncates to A C / D E/, and

/ B / D E/ truncates to D/.

An example of an SL derivation follows.

Let (1) through (6) be ground input clauses:

(1) -A B.
(2) -B C.
(3) -C D.
(4) -D A.
(5) A C.
(6) -B -D.

Given below are two refutations of this set of clauses in (I) linear format and (II) SL-resolution.

I		II	
$\neg B \neg D$	top clause	$\neg B \neg D/$	top chain
$\neg A \neg D$	resolution with (1)	$\neg A/\neg B \neg D$	extension with (1)
$\neg D \neg D$	resolution with (4)	$\neg D/\neg A/\neg B \neg D/$	extension with (4)
$\neg D$	factoring	$/\neg A/\neg B \neg D/$	reduction
		$\neg D/$	truncation
$\neg C$	resolution with (3)	$\neg C/\neg D/$	extension with (3)
$\neg B$	resolution with (2)	$\neg B/\neg C/\neg D/$	extension with (2)
$\neg A$	resolution with (1)	$\neg A/\neg B/\neg C/\neg D/$	extension with (1)
C	resolution with (5)	$C/\neg A/\neg B/\neg C/\neg D/$	extension with (5)
$\square$	ancestor resolution	$/\neg A/\neg B/\neg C/\neg D/$	reduction
		$\square$	truncation

Of course, in general one has to apply the unifying substitutions to chains produced by extension and reduction.

Readers interested in completeness and efficiency results for SL-resolution should consult footnote 4. However, before discussing our implementation of structure sharing and automatic standardizing apart, several features of SL should be pointed out.

(1) SL derivations are truly linear since one never resolves with a clause other than an input clause. Thus, there is only one branch in an SL-derivation. This will drastically simplify the association of branches to variables.

(2) Unlike the general case, where each tuple representing a clause points to two parents which are presumably tuples themselves, SL-tuples require only an indication of which input operator was used and a pointer to the nearparent (which in general will be another tuple).

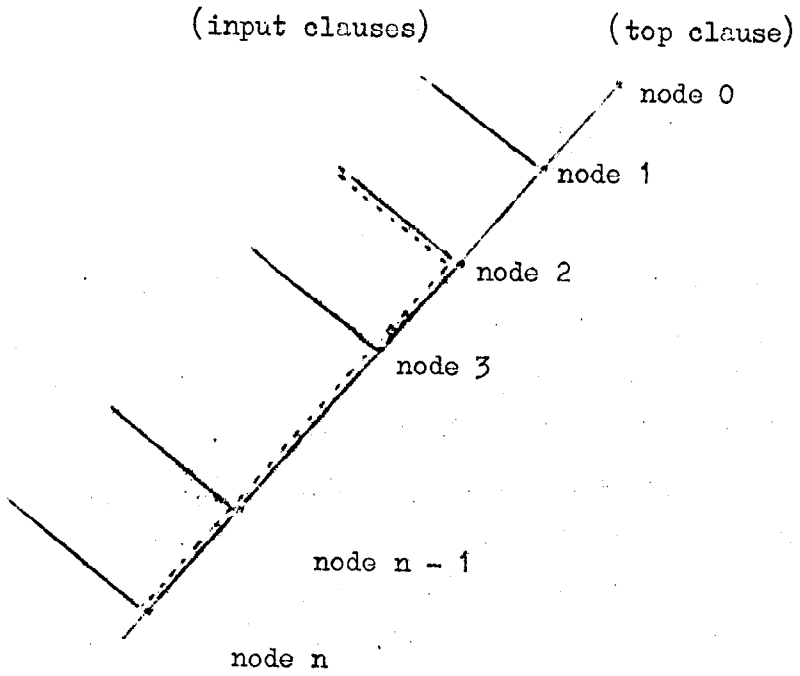
(3) In the general case we must include a reference to which literal in the nearparent was resolved upon. This is not necessary in SL since the literal resolved upon must have been the selected literal of the nearparent. Of course, since we must include the specification of which is the selected literal in every clause, nothing important has been gained.

(4) In the general case a factor must specify which literal has been deleted. Using this information will require a recursive sweep down the clause. In SL we must also include such information, but the deleted literal must have been in the most recent cell (both for reductions and truncations). Thus the literals in any cell, regardless of which operations have taken place, will be just some subset of the literals occuring in the input clause that introduced that cell. Therefore we can pack into the input parent reference a complete description of the current cell (rather than retrieve the cell recursively).

(5) Because of the importance of the cellular structure of SL chains it is convenient to regard each of the components of the SL-tuple to give information about exactly one of three different objects: the current cell, the nearparent, and the substitutions.

Recall that in order to share substructure we must manage to standardize variables apart and interpret substitutions dynamically rather than apply them when generated. In order to achieve these ends we noted that the branch a variable came down in a derivation was sufficient information to distinguish that variable from all others of the same name. Furthermore, since variables (like terms and literals) are never created but found at the tips of the tree representing each clause, the branch along which each variable occurs is implicitly stored in the tuple. We therefore only need a device that specifies which variables we are referring to in substitution components, i.e., which branches are associated with the variables mentioned in substitutions.

However, note that in the below SL-derivation involving n extensions, there are only n + 1 possible branches variables could be associated with at node n:



We have drawn a dashed line down one of the possible branches. Clearly we can label each such branch, that is, each path from some input clause to node n, by the number of the node produced by extension with that clause. (For the branch marked above, the associated number would be 2.). Note that if at node n some variable, say x, came down branch 2, then when we finally extend at node n to node n + 1, the branch associated with that same x

will still be given the name 2. That is, by referring to branches this way, branch names remain invariant under all operations. This may seem like a trivial point, however, if we had instead chosen to represent branch 2 by the list (node n, node n - 1, ... node 2) where node i is the machine pointer to the tuple representing that node, then the name of that branch at node n + 1 would be the new list, (node n + 1, node n, ... node 2).

For historical reasons, the integer that describes a branch in an SL-derivation is called the time of the branch, or node, or term, or variable. Formally, the time of a node is the number of extensions performed on the branch when the cell defined at that node was first created.

It should now be clear how branches standardize variables apart. Assume that at node n we have enough information stored to define the input literals that form the most recent cell, the bindings made, and the next node in the branch. Then any variables occuring in the most recent cell of node n will implicitly have time n. Any substitution component which binds such a variable will specify time n, and will point to some input term and specify the time of that term. Of course, variables of time other than n may be bound at node n also. Any variable not occuring in the most recent cell of node n will have a time associated with it elsewhere up the branch in an exactly similar fashion. Variables are standardized apart simply because they come from different clauses.

Readers will note that there is nothing special about our choice of assigning integers to each node in order to label branches with them. We could just as well have labelled branches with the pointer to the node. Integers have the advantage that we can pack them into halfwords.

Below is a complete specification of the tuple that defines a clause in our implementation of SL-resolution.

<time, opno, bmask, selmask, bindings, nxtbcomp >

where:

time = the time of the cell defined at the node.

opno = an 11 bit integer specifying the number of the input clause extended upon to produce the cell at this node. The list of literals defining the clause is found at position opno of an array, OPERATOR.

bmask = an 11 bit mask specifying which literals in OPERATOR (opno) are in the current cell. The nth bit is on iff the nth literal in the clause is in the cell.

selmask = an 11 bit mask specifying which literal in OPERATOR (opno) is the selected literal of this cell. If the nth literal of the input clause is the selected literal of this cell, then the nth bit of selmask is on and is the only bit in the mask which is on.

bindings = an array of length 2n (n>=0) specifying the n bindings made at this node. Variables in each input clause are textually replaced by the integers 0 through 15 on input[6]. If the binding <var, time1, term, time2> is to be represented in the array, then array position i is set to the packed word which has the variable in the top four bits, time1 in the next nine, and time2 in the low order nine. Array position i + 1 is a pointer to the term.

nxtbcomp = a pointer to the next tuple in the branch or to the word NIL signifying the end of the branch.

---

[6] Since standardizing apart is automatic, input clauses do not have to be physically standardized. In fact, as in ALGOL, the ith new variable in every clause is denoted by the integer

It should be pointed out that the selmask and the time, and the opno and the bmask components of the above tuple can be packed together. Thus, the tuple has only four components in the machine representation. Any derived clause requires exactly 7 + 2n 22-bit words, where n is the number of variables bound in the unifying substitution. Notice that this is independent of the number of literals in the clause or function nesting depth since the clause shares all the structure of its parents.

Below are three clauses (chains) in this representation. The clauses have been reified (that is, their structure has been made concrete by interpreting the information at each node and writing down the result) below the tuple representation. The bit masks have been abbreviated to only three bits, variable names are x and y instead of 0 and 1 and substitution components are given as 4-tuples of the form ⟨var, time1, term, time2⟩ for reader convenience. Readers are encouraged to examine and understand this example before proceeding.

Let OPERATORs 1 through 3 be the following lists:

(1) ((+ q (a) x)(+ p y)) denoting the clause Q(A,x)P(y)

(2) ((-p (f x))(+ r x x)) denoting the clause -P(F(x))R(x,x)

(3) ((+ q x (f (b)))(-r (a) y)) denoting the clause Q(x,F(B))-R(A,y)

Let operator 1 with selected literal P(y) be our top clause. Then the following three tuples represent the top chain, an extension of it via operator (2), and a further extension by operator (3).

*1 ⟨0; 1, 110, 010, nilarray, nil⟩

*2 ⟨1, 2, 010, 010, ⟨⟨y,0,(f x), 1⟩⟩ , * ⟩

*3 ⟨2, 3, 100, 100, ⟨⟨x,1,(a),2⟩ , ⟨y,2,(a),2⟩⟩ , * ⟩

The reified clauses are as follows:

Reified *1 Q(A,x0) <u>P(y0)</u> /

Reified *2 R(x1,x1) / Q(A,x0) <u>P(F(x1))</u> /

Reified *3 Q(x2,F(B)) / <u>R(A,A)</u> / Q(A,x0) <u>P(F(A))</u> /

Variables have been subscripted with their times.

Note that the variables introduced by the various operators have not been confused. They have been standardized apart by the

branches they came down in the derivation. That is, the 'x' we encounter in the cell at node *3 is implicitly given time 2, while that we encounter in the cell at node *2 is given time 1. The substitution components have specified which x is being bound.

It is also important to note that the terms (f x), and (a) referred to in the substitution components are not new lists, but exactly those lists in the input operators that introduced those terms into the derivation. (i.e., not only EQUAL but EQ).

The clause at node *3 can be reduced. The result will be a tuple which specifies a cell exactly like that specified by *3 except that the literal reduced upon will have its bit turned off. The substitution will bind x2 to A of time 0 and x0 to F(B) of time 2. The nxtbcomp must point back to *3 in order to refer to the substitutions found there. Thus, a reduction of *3 yields:

*4 $\langle$ 2, 3, 000, 000, $\langle \langle x,2,(a),0\rangle , \langle x,0,(f(b)),2\rangle \rangle$, *3 $\rangle$

The reader may ask how the clause denoted by *4 is reified, since its time is just that of the previous node. Since each node specifies the exact form of the cell of the time of the node, then in reifying *4 we ignore the cell specification of *3 in favor of that of *4, but must respect the bindings made at *3. The general algorithm for determining which cells are in a clause is to ignore any cell specification whose time is greater than or equal to the last cell in the clause.

Note that *4 may be truncated since the cell specified there is empty. (The bmask is all zeros). To truncate we go down the clause testing to see if when the selected literal is deleted from a cell, the cell is empty, if so we jump to the next node in the clause and repeat the test. When we find a non-empty cell, a tuple is built which specifies the same cell with its selected literal deleted; the bindings of the tuple will be the empty array, and the next branch component must point to the clause at which the truncation was initiated (to refer to substitutions made previously). Thus, *4 when truncated gives:

*5 $\langle$ 0, 1, 100, 000, nilarray, *4 $\rangle$

as the cell at *2 (which is empty when the selected literal is deleted).

Below is a complete copy of the derivation just carried out, followed by a version in reified terms.

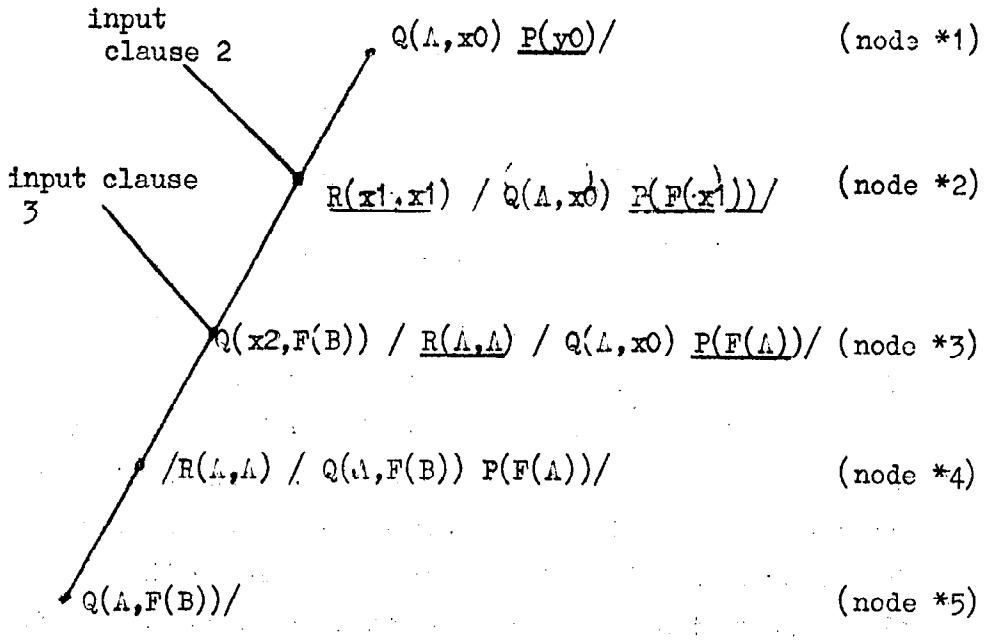
*1 ⟨0, 1, 110, 010, nilarray, nil⟩ (top)

*2 ⟨1, 2, 010, 010, ⟨⟨y,0,(f x), 1⟩⟩ , *1 ⟩ (extension)

*3 ⟨2, 3, 100, 000, ⟨⟨x,1,(a),2⟩ , ⟨y,2,(a),2⟩⟩ , *2⟩ (extension)

*4 ⟨2, 3, 000, 000, ⟨⟨x,2,(a),0⟩ , ⟨x,0,(f(b)),2⟩⟩ , *3⟩ (reduction)

*5 ⟨0, 1, 100, 000, nilarray, *4 ⟩ (truncation)



It is possible to decide what operation produced a clause simply by inspecting the time of its most recent cell, time1, and the time of the most recent cell in the previous clause, time2. The clause was produced by extension if time1 > time2. It was produced by reduction if time1 = time2 and it was produced by truncation if time1 < time2.

In order to speed up the process of determining if a variable is bound on a particular branch, we have included in our implementation of SL an array of bindings, known as the VALUE array. When a chain is to be extended upon, we make a single pass up the branch inserting into VALUE(v, t1) the

binding of the variable v of time t1 on this branch. Later to determine if v of t1 is bound, the function ISBOUND is called with the two arguments, v, and t1. It inspects VALUE (v,t1) and if it finds a term and time there it returns the truthvalue 'true' and the term and time to which the variable is bound, otherwise it returns the truthvalue 'false'.

When a unification attempt is made for the clause, the bindings produced by the UNIFY algorithm are inserted into VALUE, and a note saying the binding was made is pushed onto a binds stack. Of course, if the unification fails, the stack is popped and the bindings erased. However, if it succeeds they are left in the VALUE array and the stack is left intact. If we then **try** to reduce the clause produced by extension, its new bindings are available through ISBOUND as before. As we recursively reduce, more bindings are entered into VALUE and pushed on the stack. Upon finishing the reduction and coming out of the recursion the stack is used to restore VALUE to its configuration upon entry. Thus, while any particular clause is being extended upon and its offspring checked for reduction and admissibility, the VALUE array plays the part of the system dictionary of variable values which is recursively updated.

When a new clause is to be extended its values are swapped in. This use of the array allows our implementation to retrieve variable bindings with a single access rather than a search up the branch. It is primarily for this reason that we used integers for variables and branch codes (times) rather than identifiers and machine pointers (respectively).

Our unification algorithm is almost identical to traditional list structured implementations. This is because literals and terms are lists as usual - they just happen to be the original input lists and must be interpreted in the context (e.g. branch) in which they occur. The algorithm is given below in POP-2 code. Readers unfamiliar with POP-2 will note that ISBOUND leaves the term and its time on the users stack if it finds a binding. These are retrieved by two assignment statements → TERM1: → TIME1: If in unification we wish to bind TERM1 of TIME1 to TERM2 of TIME2 we execute BIND(TERM1, TIME1, TERM2, TIME2).

```
FUNCTION UNIFY TERM1 TIME1 TERM2 TIME2
LOOP1:
IF ISVAR (TERM1) AND ISBOUND (TERM1, TIME1)
  THEN
  -> TERM1;  -> TIME1;
  GOTO LOOP1;
  CLOSE;
LOOP2:
IF ISVAR (TERM2) AND ISBOUND (TERM2, TIME2)
  THEN
  -> TERM2;  -> TIME2;
  GOTO LOOP2;
  CLOSE;
IF TERM1 = TERM2 AND TIME1 = TIME2
  THEN TRUE;
ELSEIF ISVAR (TERM1)
  THEN
  IF OCCUR (TERM1, TIME1, TERM2, TIME2)
    THEN FALSE;
    ELSE
    BIND (TERM1, TIME1, TERM2, TIME2);
    TRUE;
    CLOSE;
ELSEIF ISVAR (TERM2)
  THEN
  IF OCCUR (TERM2, TIME2, TERM1, TIME1)
    THEN
    FALSE;
    ELSE
    BIND (TERM2, TIME2, TERM1, TIME1);
    TRUE;
    CLOSE;
ELSEIF HD(TERM1) = HD(TERM2)
  THEN
UNIFYARGS:
  TL(TERM1) -> TERM1;
  TL(TERM2) -> TERM2;
  IF TERM1 = NIL
    THEN TRUE;
    ELSEIF UNIFY (HD(TERM1), TIME1, HD(TERM2), TIME2)
      THEN GOTO UNIFYARGS;
      ELSE FALSE;  CLOSE;
ELSE FALSE;  CLOSE;
```

It should be clear that due to our ability in SL to reference bindings through an updatable array, and the fact that our implementation preserves the list structure of literals and terms, we have sacrificed almost nothing to gain a tremendous amount of shared substructure.

However, now consider how machine oriented the three SL operations are. To extend a clause we merely check that the unification succeeds; if it does we use the binds stack and VALUE to set up a permanent binding array for the new clause. We construct two bit masks, increment the time count for the branch and pack these into two words with the operator number. We then construct a four-tuple for the new clause that consists of the two packed words, the binds array, and a pointer to the nearparent.

In order to reduce we perform the unification and set up the bind array, then clobber one bit from the mask of the reduced clause and set up the new four-tuple. A truncation is necessary when the logical AND of the logical NOT of the selmask andthe bmask is zero. The result is a record whose components are taken from the tuple at which the truncation stopped and the one at which it started.

Finally note that sweeping through a clause for reduction or admissibility is just the normal cdr operation within a cell except that one leftshifts the bmask with each cdr and inspects the literal only if the highorder bit is set.

Thus, using this implementation one preserves the fast and natural recursive unification algorithm while one is able to build clauses with low-level sequences of logical ANDs, ORs, NOTs and SHIFTs (which are very efficient indeed).

It is thus clear that our implementation is extremely well suited to machines. However, we also claim it is very natural. Variables are standardized apart in the most natural way possible: they are not confused because they came from different clauses. Terms and literals are never constructed, they are just reinterpreted The cell structure of SL-resolution is perfectly represented while literals and terms retain their most natural list structure. Every clause is also its own derivation, depending upon how one chooses to interpret the node. The search tree itself is indeed

a tree that shares substructure among derivations with common pasts. All information concerning the history of a variable, term, literal, clause, or derivation, is available, including all the MGU's. Furthermore, this information is not available because it is stored in addition to the clauses produced, but because it in fact represents the clauses produced.

The debugged POP-2 code for the SL-implementation is available to interested readers.

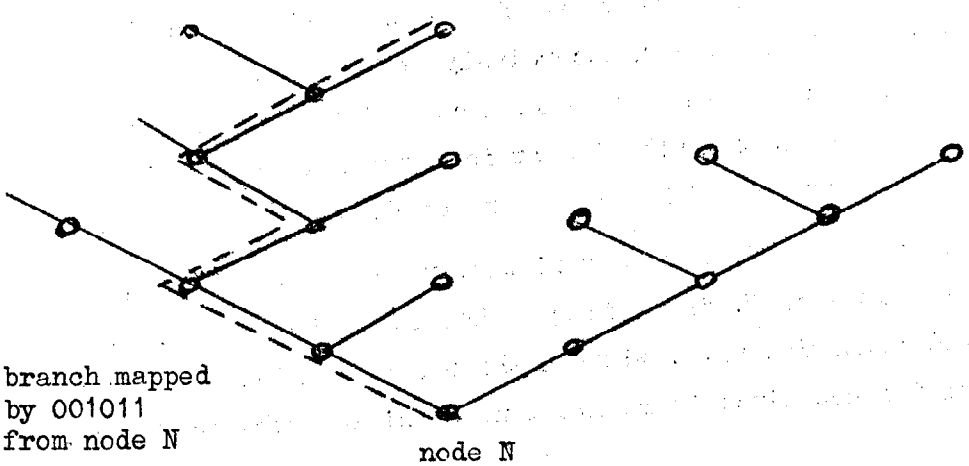
PART III

The General Resolution Implementation.

The generalized structure sharing representation has been implemented in two ways which differ only in the manner used to refer to branches. The first way is extremely elegant and is a clear demonstration of the syllogism in PART I - that is, that no work at all must be done to standardize variables apart. The second way is the straightforward generalization of the SL-implementation of PART II.

One way to refer to the branches that start at some node N at the root of a binary derivation tree is to describe the path of the branch through the tree by a bit string or logical word. We will call such a branch description a 'map'. If N is a node other than an input node, and the branch mapped by M starts at N, then the branch also contains one of the two parents of N. We will agree that the branch contains the first parent if the high order bit of M is off, and it contains the second parent if the high order bit is on. To discover which of the parent's parents is in the branch, we left shift M by 1 and use the same test again, until we arrive at an input clause.

For example, in the following tree, the branch starting at node N with map 001011 is marked by a dashed line. Note that since in tracing a map one can detect when the terminal (input) node is reached, maps do not have to specify how many of their bits are significant.



One can now define a clause to be a record of the form:

$\langle$ parent1, litno1, parent2, litno2, bindings $\rangle$ = N

where

parent1 = the first parent of the clause denoted by N

litno1 = the number of the literal in parent1 to be deleted to form N

parent2 = the second parent of N

litno2 = the number of the literal in parent2 to be deleted to form N

bindings = an array of length 4n representing the n bindings made at this node. Array position 1 contains the variable bound, position 2 contains the variable's map, position 3 the term, and position 4 the term's map. Positions 5 through 8 contain the second substitution component, etc.

In our implementation of this representation, we also store the total number of literals in a clause. This of course is recursively defined as LEN(N) = LEN(parent1) + LEN(parent2) - 2; however, its inclusion speeds up processing. We can pack the length, and two literal numbers into a word if we limit ourselves to 127 literals in the longest clause. As a result, a clause requires exactly 6 + 4n words.

The word size in our machine limits our maps to 22 bits, or derivations to a maximum depth of 22.

Input clauses are stored with NIL in one of the parents and the list of literals defining the clause in the other. Factors are stored with a dummy unit clause in one parent, which make them easily recognized but recursively identical to resolvents.

In order to determine the nth literal of such a clause, one merely asks whether it came from the first or second parent, based on the length of the first parent and the literal deleted in that parent. Then one decrements n accordingly and moves to the appropriate parent to ask the same question. When an input list is finally encountered, the nth element is retrieved as the literal and returned with the map of how it was reached.

Retrieving the bindings of a variable, v, with some map, m, in a tree whose root is N, is similar to how it is done dynamically in the SL-implementation (i.e. without the use of VALUE). However, there is a difference since there are many possible paths to follow.

One asks whether v of m is bound in the bindings at N. If not, one branches to the first or second parent depending upon the high order bit in m. Then m is leftshifted by 1 and the question is repeated. The process continues until either an input clause is encountered, which means v of the original m is unbound, or a binding substitution component with term T and map M is found at some node N'. Of course, map M is only a map from the input parent that introduced T to the node N'. It must be updated to be accurate relative to the original N. To do this, that part of the original m that led from N to N' must be 'appended' onto M. This is easily done with three logical operations. However, a more elegant, if slightly slower, method is to write the function recursively. Upon exiting from the recursion when a term is found, M should be rightshifted by 1 and the high order bit shifted out of m at that level should be inserted. When the process returns to the top, M would have been transformed appropriately. This method illustrates how our maps are like individual push down stacks for each variable or term.

The unification routine is precisely that given for the SL-implementation. The ISBOUND function always responds relative to some globally defined root node, and BIND just inserts the four word binding into a working space for UNIFY.

There is no counterpart to the VALUE array since maps are unwieldy objects to index over (hash coding will cut down the search somewhat). ISBOUND can be speeded up by ordering the maps at each node so that it only need consider maps lower than the one for which it is looking. The speed of the unification algorithm is directly proportional to the depth of the terms involved, since ISBOUND must search up the branches. Tests have shown that the implementation is from 10 to 2 times faster than character strip versions (2 times faster in the depth 22, worst-possible map orders cases).

The second representation of branches is closely related to the SL-implementation device that numbers tips in such a way that each number represents a unique path to the tip. Tips can be numbered relative to nodes lower in the tree, so that if the same input clause occurs as two different tips, it is assigned two different numbers according to the path which led to it. These numbers are called 'paths'.

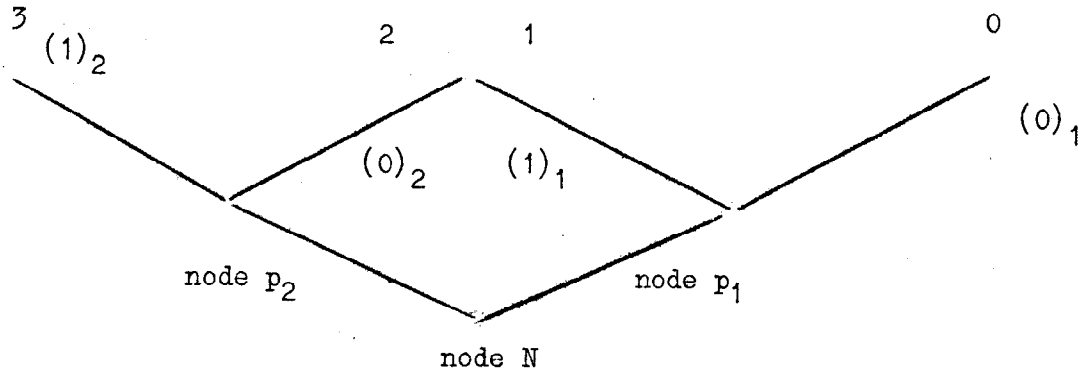
Let $T(N)$ be the set of all tips in the tree whose root is N. Let $S(N)$ be the number of nodes in the tree. Let $K(N',N)$ be the integer associated with node N' relative to node N.

If N is an input node, then let $K(N,N) = 0$. Now let N be any node such that $S(N) < n$. Assume inductively that for all $N' \in T(N)$, $0 \leq K(N',N) \leq S(N) - 1$. Let M be a node such that $S(M) = n$, and let P_1 and P_2 be the parents of M. Then, $S(P_i) < n$, for $i = 1, 2$. Thus, for all $P_i' \in T(P_i)$, $i = 1, 2$, it is the case that $0 \leq K(P_i',P_i) \leq S(P_i) - 1$. Therefore for all $M' \in T(M)$ define $K(M',M)$ as follows:

$$K(M',M) = \begin{cases} K(M', P_2) & \text{if } M' \in T(P_2) \\ K(M', P_1) + S(P_2) & \text{if } M' \in T(P_1) \end{cases}$$

which verifies the induction hypothesis.

The example below illustrates how the node numbers assigned by N are related to those assigned by P_1 and P_2.



Numbers in parentheses are relative to p_1 or p_2 according to the subscript. The unparenthesized numbers are those relative to N. Note that p_1 and p_2 share a node. The two paths from that node to N are given distinct numbers.

Using this notation, we can define a clause to be a record of the form:

⟨baseref, delta, parent1, litno1, parent2, litno2, bindings = N⟩

where:

baseref = S(N)

delta = S(parent2).

parent1 =
litno1 =
parent2 =
litno2 = } as in previous representation

bindings = an array of 2n words containing the n substitution components for this node. The components are packed as in the SL-implementation except that 'times' are replaced by 'paths'.

Again, due to packing, the actual number of words per clause is 7 + 2n, if we limit ourselves to 512 nodes in any single derivation. Note that baseref and delta are recursively computable. They are stored for faster processing.

In this representation, the algorithm for determining if a variable is bound is similar to the one outlined above. If it is not bound at the current node, one jumps to the first or second parent depending upon whether the path of the variable is greater than or equal to delta or not. If one jumps to the first parent, the path of the variable is decremented by delta. If one finds a binding the associated term's path is incremented by the total amount of decrementing that occurred, to transform it into the original node's frame of reference. The recursive version is similar; one just adds in the delta substracted from the variable's path upon exiting from each level of the recursion.

However, this version allows VALUE to be used. Bindings made in a tree can be appropriately incremented and written into VALUE(VAR,PATH) as before. Substitution components produced on the basis of the incremented paths in VALUE are of course accurate henceforth, since they are correctly interpreted later. After the VALUE array has been loaded, the unification, factoring, and other algorithms are as fast as those in the SL-implementation. Loading the array is somewhat more complex since a tree rather than a branch must be traced.

It should also be noted that the literals of a clause can be loaded into a list with their paths. Once this operation has been performed, the implementation is very similar to list structured ones, without the additional overhead of many copies of terms and literals and standardizing apart.

The primary reason this particular method of referring to branches is preferable to the logical map method is that if we limit ourselves to n bits in the item that represents a map or path, the map version limits us to a depth of n, regardless of the number of nodes in the tree, while the path version limits us only to a maximum of 2^n nodes. So while the two are equivalent for full binary trees, the path version is less restrictive otherwise, since it is sensitive to the shape of the tree. In addition, the path version assigns consecutive integers that are easy to index over.

The POP-2 code for both of these implementations is available to the interested reader.