6

The Sharing of Structure in Theorem-proving Programs

R. S. Boyer and J S. Moore
Department of Computational Logic
School of Artificial Intelligence, Edinburgh

Abstract

We describe how clauses in resolution programs can be represented and used without applying substitutions or cons-ing lists of literals. The amount of space required by our representation of a clause is independent of the number of literals in the clause and the depth of function nesting. We introduce the concept of the value of an expression in a binding environment which we use to standardize clauses apart and share the structure of parents in representing the resolvent. We present unification and resolution algorithms for our representation. Some data comparing our representation to more conventional ones is given.

1. INTRODUCTION

In this paper we are concerned with representing literals and clauses in computers. Lists provide the most obvious and natural representation of literals because lists perfectly reflect function nesting structure. A list is also a reasonable representation of a set, in particular of a clause. Lists, however, can consume large amounts of space, and cause frequent garbage collections. We shall present in this paper a representation of clauses and literals which is as natural as lists but far more compact. We achieve this economy by sharing the structure of the parents of a resolvent in our representation of the resolvent.

A clause is a set of literals; but throughout this paper we shall speak of the literals of a clause as having an order. That is, we shall speak of the first, second, etc., literal of a clause.

Suppose C and D are clauses and K is the ith literal of C and L is the jth literal of D. Suppose further that the signs of K and L are opposite. Finally, suppose that the substitution σ most generally unifies the atoms of K and L.

Under these hypotheses, we may resolve C and D on K and L to obtain

the resolvent R=((C−{K})∪(D−{L}))σ. We think of the literals in R that come from C as being 'before' the literals that come from D. (We ignore merging and factoring until section 8.)

The tuple T=⟨C, *i*, D, *j*, σ⟩ contains sufficient information to enable one to construct the resolvent R. Therefore, in some sense, T represents R. At first sight T does not appear to be a very good way to represent R; for it seems that the only way to use T is to construct a list consisting of all but the *i*th literal of C and *j*th literal of D and then to apply σ to the list.

We shall show in this paper that it is possible, in fact easy, to use a tuple like T without constructing any lists or applying any substitutions. Actually, the tuples we shall use have the form

⟨C, *i*, D, *j*, NL, MI, σ⟩.

NL is simply the number of literals in the resolvent, that is, the sum of the number of literals in C and D minus 2. MI is a number which helps us standardize clauses apart. The MI of an input clause is 1; the MI of a resolvent is the sum of the MI's of the parents. ('MI' is mnemonic for 'maximum index'.)

2. TERMS AND SUBSTITUTIONS

To understand how to avoid applying substitutions, it is first necessary to understand the concept of the value of a term in the context of a substitution. First, an example:

The value of the term

$$(P\,x\,(f\,y\,(g\,z\,x)))$$

in the context of the substitution

$$((y\,.\,(f\,v\,w))\,(z\,.\,(g\,x\,u))\,(u\,.\,(h\,x)))$$

is the term

$$(P\,x\,(f\,(f\,v\,w)\,(g\,(g\,x\,(h\,x))\,x))).$$

By a term, we mean either a variable (e.g. *x*,*y*,*z*) or a list whose first member is a symbol (e.g. *f*,*g*,*P*,*Q*) and whose other members are terms.*

By a substitution we mean a collection of pairs; the first member of each pair is a variable and the second member is a term. If (VAR . TERMB) is a member of a substitution *S*, we say that VAR is bound in *S* to TERMB.

By the value of a term T in the context of a substitution *S*, we mean the result of replacing each variable in T that is bound in *S* to a term TERMB by the value of TERMB in *S*.

Our definitions are not those standard to the theorem-proving literature. For example, we do not need to distinguish between predicate and function symbols. Furthermore, there exist substitutions *S* such that some terms have no value in the context of *S*. We take precautions never to generate such substitutions. Roughly speaking, a variable ought not be bound twice or bound to something whose value contains that variable.

It is possible to determine anything about the value of a term in the context

* We think of a term such as (*a*) as a constant.

of a substitution S without physically creating the value. The only thing one must do is:

> Whenever one encounters a variable VAR, check whether VAR is bound to some term TERMB. If VAR is bound, proceed as if one had encountered TERMB instead of VAR.

For example, suppose we wish to determine whether some variable V occurs in the value of a term TERM in the context of a substitution S. We define the recursive function OCCUR:

Definition of OCCUR(V,TERM)

If TERM is a variable, then

If TERM is bound to TERMB in S, return(OCCUR(V,TERMB))

Otherwise, if V=TERM, return(true)

Otherwise return(false)

Otherwise TERM is not a variable and has the form $(f\ T_1 \ldots T_n)$.

If any call of OCCUR(V,T_i) returns true, then return(true)

Otherwise return(false)

End of definition.

Notice that we check to see if we have encountered a variable that is bound to some term TERMB in S (S is global to OCCUR). If it is, we proceed as if we had encountered TERMB instead of the variable by returning the result of the recursive call OCCUR(V,TERMB).

By avoiding the application of substitutions to terms it is possible to achieve a dramatic saving in space, which, of course, one pays for by looking-up the bindings of variables. That this is worth while is demonstrated by the successful use of similar methods to 'substitute' values for the formal parameters in LISP and ALGOL function calls.

3. EXPRESSIONS AND BINDINGS

The key to our representation of clauses is the avoidance of physically creating the value of a term in the context of a substitution. This idea is at least as old as the first LISP. Terms and substitutions, however, are not quite sufficient for our purposes because we often need to refer to different versions of a term at one time. Therefore, we introduce the concepts of an expression, a binding environment, and the value of an expression in a binding environment. First some examples:

The value of the expression

$$(P\,x\,(f\,y\,(g\,z\,x))),10$$

in the empty binding environment is the term

$$(P\,x_{10}\,(f\,y_{10}\,(g\,z_{10}\,x_{10}))).$$

The value of the expression

$$(P\,x\,(f\,y\,(g\,z\,x))),5$$

in the empty binding environment is the term

$$(P\,x_5\,(f\,y_5\,(g\,z_5\,x_5))).$$

Notice that these two values have no variables in common.

The value of the expression

$$(P\,x\,(f\,y\,(g\,z\,x))),5$$

in the binding environment

$$((y,5\,.\,(f\,x\,y),4)\;(z,5\,.\,(g\,x\,u),5)\;(u,5\,.\,(h\,x),5))$$

is the term

$$(P\,x_5\,(f\,(f\,x_4\,y_4)\,(g\,(g\,x_5\,(h\,x_5))\,x_5))).$$

By an index we mean a positive integer. By an expression we mean a term together with an index. If we denote an expression by T,I then T is a term and I is an index.

By a binding we mean a pair (VAR,INDEX . TERMB,INDEXB) where VAR is a variable, TERMB is a term, and INDEX and INDEXB are indices.

By a binding environment we mean a collection of bindings. If (VAR, INDEX . TERMB,INDEXB) is a member of the binding environment BNDEV, we say that VAR,INDEX is bound in BNDEV to TERMB,INDEXB.

The value of an expression T,I in a binding environment BNDEV is the result of replacing each variable V in T by the value of V,I in BNDEV. If V,I is not bound in BNDEV, its value is the variable V_I (i.e. V subscript I). If V,I is bound to TERMB,INDEXB in BNDEV, then its value is the value of TERMB, INDEXB in BNDEV.

It is possible to determine anything about the value of an expression in a binding environment without physically creating the value.

Throughout this paper we shall use two procedures, ISBOUND and BIND, to facilitate the handling of binding environments. ISBOUND(VAR,INDEX, BNDEV) returns true if VAR,INDEX is bound in BNDEV, and false otherwise. If true is returned, then the global variables TERMB and INDEXB will have been so set that VAR,INDEX is bound to TERMB,INDEXB in BNDEV. BIND(V,I,T,J,BNDEV) so alters the binding environment BNDEV that V,I is then bound to T,J in BNDEV.

In the next three sections we shall display binding environments as lists of bindings. We do this to help introduce our representation of clauses intuitively. The actual structure of a binding environment is made precise in section 7. The only essential feature of a binding environment is that one can discover bindings with ISBOUND and add bindings with BIND.

Suppose we wish to determine whether some variable V_I occurs in the value of the expression TERM,J in the binding environment BNDEV. We define the recursive function OCCUR as below and call OCCUR(V,I,TERM,J):

Definition of OCCUR(V,I,TERM,J)

If TERM is a variable, then

If ISBOUND(TERM,J,BNDEV) then return(OCCUR(V,I,TERMB, INDEXB))

Otherwise if V=TERM and I=J, return(true)

Otherwise return(false)

Otherwise TERM is not a variable and has the form $(f\ T_1 \ldots T_n)$.
If any call of OCCUR(V,I,T_i,J) returns true, then return(true)
Otherwise return(false)
End of definition.

Observe the similarity between this definition and the previous definition of OCCUR. BNDEV is global to OCCUR.

4. UNIFY: OUR UNIFICATION ALGORITHM

Suppose that VAL1 is the value of the expression TERM1,INDEX1 in the binding environment BNDEV. Suppose further that VAL2 is the value of TERM2,INDEX2 in BNDEV. Finally suppose that VAL is the most general common instance of VAL1 and VAL2. If we call UNIFY(TERM1,INDEX1, TERM2,INDEX2) then BNDEV will be altered during the call so that the value of TERM1,INDEX1 in BNDEV and the value of TERM2,INDEX2 in BNDEV are both equal to VAL. If VAL1 and VAL2 have no common instance, then the call will return false. Our procedure UNIFY, like the procedure OCCUR of the previous section which UNIFY uses, applies no substitutions. We write $x=y$ if x and y are the same atom or number. By EQUAL we mean the LISP EQUAL.

Definition of UNIFY(TERM1,INDEX1,TERM2,INDEX2)
If EQUAL(TERM1,TERM2) and INDEX1 =INDEX2 then return(true)
Otherwise if TERM1 is a variable, then
If ISBOUND(TERM1,INDEX1,BNDEV) then return(UNIFY (TERMB,INDEXB,TERM2,INDEX2))
Otherwise if OCCUR(TERM1,INDEX1,TERM2,INDEX2) then return(false)
Otherwise BIND(TERM1,INDEX1,TERM2,INDEX2,BNDEV) and return(true)
Otherwise if TERM2 is a variable, then return(UNIFY(TERM2,INDEX2, TERM1,INDEX1))
Otherwise, since neither TERM1 nor TERM2 is a variable, TERM1 has the form $(f\ T_1 \ldots T_n)$ and TERM2 has the form $(g\ S_1 \ldots S_m)$.
If $f \neq g$, then return(false)
Otherwise if every call of UNIFY(T_i,INDEX1,S_i,INDEX2) returns true, return(true)
Otherwise return(false)
End of definition.

Here is an example of unification. Let TERM1 be $(P\ x\ y)$. Let TERM2 be $(P\ (g\ x)\ z)$. Let BNDEV be

$((x, 2\ .\ x, 3)$
$(y, 2\ .\ (f\ x\ y), 4)$
$(y, 4\ .\ x, 3)$
$(z, 7\ .\ (f\ x\ y), 8)$
$(x, 8\ .\ x, 7)$
$(y, 8\ .\ (g\ y), 5))$.

The value of TERM1,2 in BNDEV is $(P\ x_3\ (f\ x_4\ x_3))$.
The value of TERM2,7 in BNDEV is $(P\ (g\ x_7)\ (f\ x_7\ (g\ y_5)))$.

After a call of UNIFY(TERM1,2,TERM2,7),BNDEV is

$$\left.\begin{array}{l}((x, 4\ .\ y, 5)\\ (x, 7\ .\ x, 4)\\ (x, 3\ .\ (g\ x), 7)\end{array}\right\}\ \text{added by UNIFY}$$

$$\left.\begin{array}{l}(x, 2\ .\ x, 3)\\ (y, 2\ .\ (f\ x\ y), 4)\\ (y, 4\ .\ x, 3)\\ (z, 7\ .\ (f\ x\ y), 8)\\ (x, 8\ .\ x, 7)\\ (y, 8\ .\ (g\ y), 5)).\end{array}\right\}\ \text{the old BNDEV}$$

The value of TERM1,2 in the new BNDEV is $(P\ (g\ y_5)\ (f\ y_5\ (g\ y_5)))$.
The value of TERM2,7 in the new BNDEV is $(P\ (g\ y_5)\ (f\ y_5\ (g\ y_5)))$.

5. INCREMENTING INDICES: HOW TO STANDARDIZE EXPRESSIONS APART

Let T be the term $(Q\ (f x(a))\ (g\ y\ z))$. The value of the expression T,5 in the binding environment

BNDEV1: $((x, 5\ .\ (g\ y\ z), 5)$
$(z, 5\ .\ (f(a)\ u), 6)$
$(u, 6\ .\ x, 3))$

is $(Q\ (f(g\ y_5\ (f(a)\ x_3))\ (a))\ (g\ y_5\ (f(a)\ x_3)))$. The value of the expression T,11 in the binding environment

BNDEV2: $((x, 11\ .\ (g\ y\ z), 11)$
$(z, 11\ .\ (f(a)\ u), 12)$
$(u, 12\ .\ x, 9))$

is $(Q\ (f(g\ y_{11}\ (f(a)\ x_9))\ (a))\ (g\ y_{11}\ (f(a)\ x_9)))$.

Notice that the value of T,5 in BNDEV1 is a variant of the value of T,11 in BNDEV2; furthermore, the values have no variable in common, that is, they have been standardized apart. Notice that BNDEV2 is obtained from BNDEV1 by adding the increment 6 to every index in BNDEV1.

Suppose that T is a term, BNDEV1 is a binding environment, and BNDEV2 is obtained from BNDEV1 by adding the increment INC to every index in BNDEV1. Then the value of T,J in BNDEV1 is a variant of the value of T,J+INC in BNDEV2. If INC is greater than any index in BNDEV1 then the two values have no variable in common.

6. RESOLVING CLAUSES USING EXPRESSIONS AND BINDINGS

In this section we describe by example how expressions, incrementing indices, and our unification procedure work together in resolution. We use in this section a simple representation of clauses, namely a list of expressions in a

binding environment. After we have performed one resolution using this representation, we come to the main point of the paper.

The list

C1: $((+(Q\,y\,y)),2$
$(+(P\,x\,y)),2$
$(-(P\,x\,(f\,y\,z))),4)$

in the binding environment

B1: $((x,2\,.\,x,3)$
$(y,2\,.\,(f\,x\,y),4)$
$(y,4\,.\,x,3)$
$(z,4\,.\,(f\,x\,y),2))$

represents the clause

$\mathscr{C}1$: $(Q\,(f\,x_4\,x_3)\,(f\,x_4\,x_3))\,(P\,x_3\,(f\,x_4\,x_3))-(P\,x_4\,(f\,x_3\,(f\,x_3\,(f\,x_4\,x_3))))$.

in an obvious way.

Similarly, the list

C2: $((-(Q\,x\,y)),1$
$(-(P\,(g\,x)\,z)),3$
$(+(R\,x\,(f\,x\,y)),4)$

in the binding environment

B2: $((z,3\,.\,(f\,x\,y),4)$
$(x,4\,.\,x,3)$
$(y,4\,.\,(g\,y),1)$
$(y,2\,.\,(g\,z),3))$

represents the clause

$\mathscr{C}2$: $-(Q\,x_1\,y_1)-(P\,(g\,x_3)\,(f\,x_3\,(g\,y_1)))\,(R\,x_3\,(f\,x_3\,(g\,y_1)))$.

To resolve $\mathscr{C}1$ and $\mathscr{C}2$ on their second literals, we first standardize $\mathscr{C}1$ and $\mathscr{C}2$ apart. We do this by adding 4 to every index in C2 (we denote the result as C2′) and by adding 4 to every index in B2 (we denote the result by B2′). We add 4 because it is the maximum index in C1 or B1. C2′ in the binding environment B2′ represents the clause

$\mathscr{C}2'$: $-(Q\,x_5\,y_5)-(P\,(g\,x_7)\,(f\,x_7\,(g\,y_5)))\,(R\,x_7\,(f\,x_7\,(g\,y_5)))$.

$\mathscr{C}2'$ is a variant of $\mathscr{C}2$ and has no variables in common with $\mathscr{C}1$.

We obtain the second expression $(+(P\,x\,y)),2$ of C1 and the second expression $(-(P\,(g\,x)\,z)),7$ of C2′. We check that their signs are opposite. We then call UNIFY$((P\,x\,y),2,(P\,(g\,x)\,z),7)$. Of course UNIFY requires a binding environment. In this case BNDEV is originally set to B1∪B2′. The call to UNIFY returns true and BNDEV has been modified so that it is

$(x,4\,.\,y,5)$
$(x,7\,.\,x,4)$
$(x,3\,.\,(g\,x),7)$
} added by UNIFY

$(x,2\,.\,x,3)$
$(y,2\,.\,(f\,x\,y),4)$
$(y,4\,.\,x,3)$
$(z,4\,.\,(f\,x\,y),2)$
} B1

$$\left.\begin{array}{l}(z, 7 . (f\,x\,y), 8)\\ (x, 8 . x, 7)\\ (y, 8 . (g\,y), 5)\\ (y, 6 . (g\,z), 7)).\end{array}\right\} \text{B2}'$$

The resolvent, $\mathscr{R}$, could be represented by a list of expressions R in the binding environment BNDEV above. R is obtained by appending C1 and C2′ after removing their second literals.

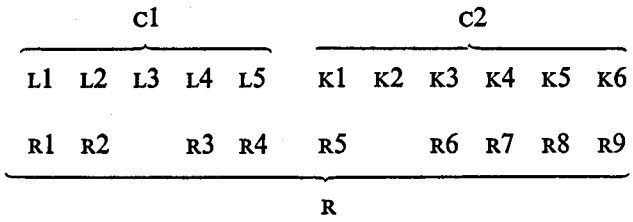
R: $((+(Q\,y\,y)), 2$
$(-(P\,x\,(f\,y\,z))), 4$
$(-(Q\,x\,y)), 5$
$(+(R\,x\,(f\,x\,y))), 8).$

R in the binding environment BNDEV represents the resolvent

$\mathscr{R}$: $(Q\,(f\,y_5\,(g\,y_5))\,(f\,y_5\,(g\,y_5)))$
$-(P\,y_5\,(f\,(g\,y_5)\,(f\,(g\,y_5)\,(f\,y_5\,(g\,y_5)))))$
$-(Q\,x_5\,y_5)$
$(R\,y_5\,(f\,y_5\,(g\,y_5))).$

We now come to the central part of this paper. It should be obvious that it is exceedingly wasteful to create physically the lists C2′ and R and the binding environments B2′ and BNDEV, given their definitions in terms of C1, B1, C2, and B2. We certainly do not physically create any of C2′, R, B2′, or BNDEV. We do represent a clause so that we can easily retrieve (1) the expression for the *n*th literal and (2) the binding of V,I (if it is bound). Under the hypothesis that we can retrieve (1) and (2) for C1 and C2, our clause record contains precisely enough information to retrieve easily (1) and (2) for R.

(1) Assume, inductively, that it is possible to retrieve the expression for the *n*th literal of either parent, C1 or C2, of a resolvent R. The expression for the *n*th literal of R is either the expression for the *j*th literal of C1 or the expression for the *j*th literal of C2 with its index incremented by the maximum index (MI) of C1. *j* depends only upon the number of literals in C1 and the numbers of the literals in C1 and C2 upon which we resolved. The following picture should make it obvious how to compute *j*. We are resolving on L3 and K2.



(2) Assume, again inductively, that it is possible to determine whether V,I is bound to TERMB,INDEXB in the binding environments of either parent. If in the representation of a resolvent we include the bindings added by the unification of the resolution, it is possible to determine if V,I is bound to

TERMB,INDEXB in the binding environment of R. In particular, V,I is bound to TERMB,INDEXB in R if and only if

I≤MI and V,I is bound to TERMB,INDEXB in the binding environment of C1, or

I>MI and V,I−MI is bound to TERMB,INDEXB−MI in the binding environment of C2, or

V,I was bound to TERMB,INDEXB in the unification made for the resolvent.

The main point of this paper is:

If we can compute the expressions and binding environments for input clauses, then we can compute them for derived clauses if we include only the following information in the record of such a resolvent:

1. the record of the left parent, C1
2. the number of the literal resolved upon in C1
3. the record of the right parent, C2
4. the number of the literal resolved upon in C2
5. the number of literals in the clause resolvent
6. the maximum index of the resolvent
7. the bindings added during the unification for the resolvent.

This is precisely the information we include in our clause representation described in detail in the next section.

7. THE DETAILS OF OUR REPRESENTATION

By a clause record we mean either an input record or a resolvent record. By an input record, we mean a list of literals. A literal has a sign which may be + or − followed by an atomic formula which is a term in the sense of section 2. Here are two input records:

((+(*P x* (*f y*))))

((−(*Q x y*)) (+(*P* (*f x*) *y*)) (+(*R* (*a*) *z*)))

If IP is an input record, then NUMBEROFLITERALS(IP)IN is the length of IP, and MAXIMUMINDEX(IP) is 1.

By a resolvent record, R, we mean a structure of 7 components:

1. a clause record which we access as LEFTPARENT(R)
2. an integer which we access as LEFTLITERALNUMBER(R)
3. a clause record which we access as RIGHTPARENT(R)
4. an integer which we access as RIGHTLITERALNUMBER(R)
5. an integer which we access as NUMBEROFLITERALSIN(R)
6. an integer which we access as MAXIMUMINDEX(R)
7. a list of bindings which we access as BINDINGS(R). The function ISBOUND accesses this component. The function BIND accesses and alters this component.

These components represent, in order, the items enumerated at the end of the previous section.

To obtain the expression for the Kth literal of a clause record CL we call

GETLIT(CL,K). GETLIT sets a global variable LITG to the input literal and a global variable INDEXG to the index such that the expression LITG,INDEXG represents the Kth literal of CL (in the binding environment of CL).

Definition of GETLIT(CL,K)

If CL is an input record, then
set LITG to the Kth member of CL
set INDEXG to 1
Otherwise if K<LEFTLITERALNUMBER(CL) then
call GETLIT(LEFTPARENT(CL),K)
Otherwise if K<NUMBEROFLITERALSIN(LEFTPARENT(CL)) then
call GETLIT(LEFTPARENT(CL),K+1)
Otherwise if K<NUMBEROFLITERALSIN(LEFTPARENT(CL))−1
+RIGHTLITERALNUMBER(CL) then
call GETLIT(RIGHTPARENT(CL),
K−NUMBEROFLITERALSIN(LEFTPARENT(CL))+1)
set INDEXG to INDEXG+MAXIMUMINDEX(LEFTPARENT(CL))
Otherwise
call GETLIT(RIGHTPARENT(CL),
K−NUMBEROFLITERALSIN(LEFTPARENT(CL))+2)
set INDEXG to INDEXG+MAXIMUMINDEX(LEFTPARENT(CL))

End of definition.

To determine the binding, if any, of a variable VAR and an index INDEX in a binding environment BNDEV we call ISBOUND(VAR,INDEX,BNDEV). The call returns true if VAR,INDEX is bound in BNDEV. Otherwise, the call returns false. If the call returns true, then ISBOUND has set a global variable TERMB to the term and a global variable INDEXB to the index such that VAR,INDEX is bound to TERMB,INDEXB in BNDEV. BNDEV is always a clause record. ISBOUND looks in the BINDINGS(BNDEV) for a binding of VAR,INDEX. If none exists, ISBOUND is called recursively on the appropriate parent of BNDEV.

Definition of ISBOUND(VAR,INDEX,BNDEV)

If BNDEV is an input clause, return(false)
Otherwise if there is some binding of the form (VAR,INDEX . T, I) in
BINDINGS(BNDEV) then
set TERMB to T
set INDEXB to I
return(true)
Otherwise if INDEX≤MAXIMUMINDEX(LEFTPARENT(BNDEV)) then
return(ISBOUND(VAR,INDEX,LEFTPARENT(BNDEV)))
Otherwise
call ISBOUND(VAR,INDEX−MAXIMUMINDEX
(LEFTPARENT(BNDEV)),RIGHTPARENT(BNDEV))
If the call returns false, then return(false)
Otherwise

set INDEXB to INDEXB+MAXIMUMINDEX(LEFTPARENT (BNDEV))
return(true)

End of definition.

To add a binding (V,I . T,J) to BNDEV we call BIND(V,I,T,J,BNDEV).

Definition of BIND(V,I,T,J,BNDEV)

set BINDINGS(BNDEV) to CONS((the binding (V,I . T,J)), BINDINGS(BNDEV))

End of definition.

To resolve two clause records CL1 and CL2 on their Ith and Jth literals respectively, we call RESOLVE(CL1,I,CL2,J). RESOLVE uses the local variables: LEFTLIT,RIGHTLIT,LEFTINDEX,RIGHTINDEX,BNDEV.

Definition of RESOLVE(CL1,I,CL2,J)

Call GETLIT(CL1,I)
Set LEFTLIT to LITG
Set LEFTINDEX to INDEXG
Call GETLIT(CL2,J)
Set RIGHTLIT to LITG
Set RIGHTINDEX to INDEXG+MAXIMUMINDEX(CL1)
Set BNDEV to the new resolvent record
⟨CL1,I,CL2,J,NUMBEROFLITERALSIN(CL1)+ NUMBEROFLITERALSIN(CL2)−2, MAXIMUMINDEX(CL1)+MAXIMUMINDEX(CL2), the empty list⟩
Check to see that the signs of RIGHTLIT and LEFTLIT are opposite.
If not, return(FAIL)
Call UNIFY(theatomof(LEFTLIT),LEFTINDEX,theatomof (RIGHTLIT),RIGHTINDEX)
If UNIFY returns true, then return(BNDEV)
Otherwise return(FAIL)

End of definition.

Note that the most time consuming function in RESOLVE is the unification step. In particular, notice that standardizing the clauses apart is accomplished entirely by incrementing indices, and that except for the unifying substitution, the work involved in the creation of the resolvent is independent of the complexity of the two clauses represented by the parents. If the unification is successful, BNDEV is the clause record of the resolvent, and is returned. Otherwise, RESOLVE returns FAIL. The functions UNIFY and OCCUR are exactly as in sections 4 and 3 (except that they now use the definitions of ISBOUND and BIND of this section).

With minor alterations one can avoid constructing the new BNDEV unless the unification succeeds. In section 8 we mention other instances in which we have sacrificed efficiency for clarity in our definitions.

Figure 1 exhibits a derivation involving four resolutions. Figure 1(a) of

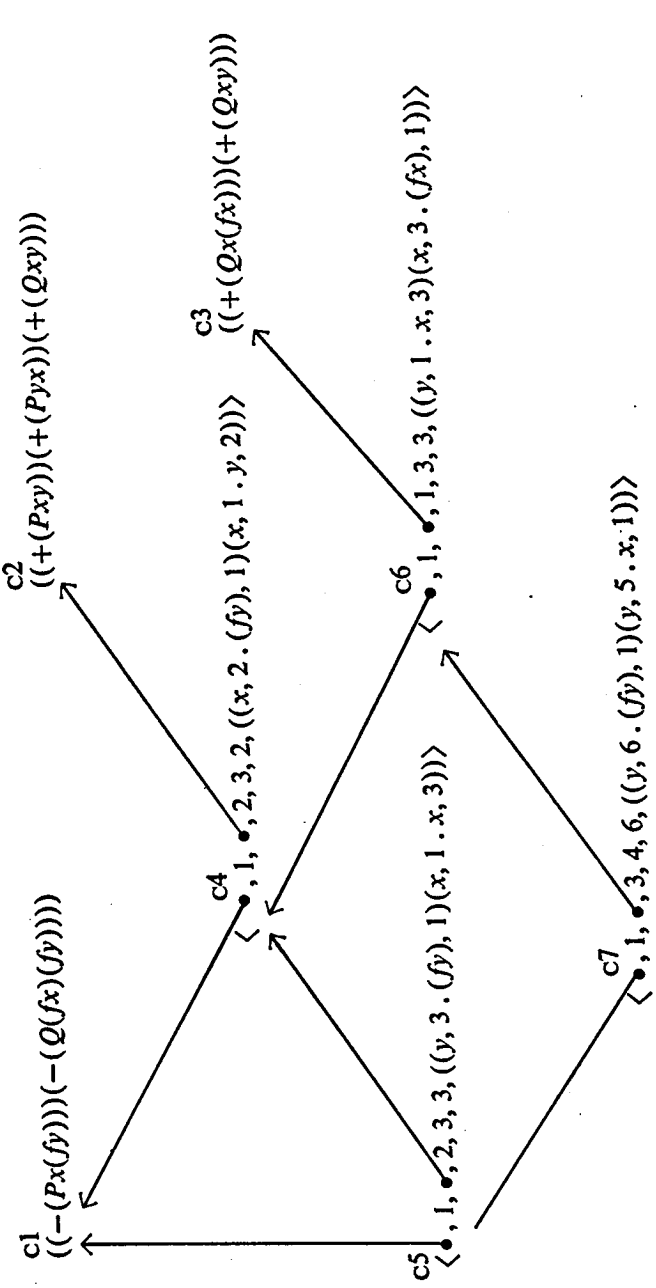

c1
((−(Px(fy)))(−(Q(fx)(fy))))
c2
((+(Pxy))(+(Pyx))(+(Qxy)))
c3
((+(Qx(fx)))(+(Qxy)))
c4
⟨, 1, •, 2, 3, 2, ((x, 2 . (fy), 1)(x, 1 . y, 2))⟩
c5
⟨• , 1, •, 2, 3, 3, ((y, 3 . (fy), 1)(x, 1 . x, 3))⟩
c6
⟨• , 1, •, 1, 3, 3, ((y, 1 . x, 3)(x, 3 . (fx), 1))⟩
c7
⟨• , 1, •, 3, 4, 6, ((y, 6 . (fy), 1)(y, 5 . x, 1))⟩

Figure 1(a).

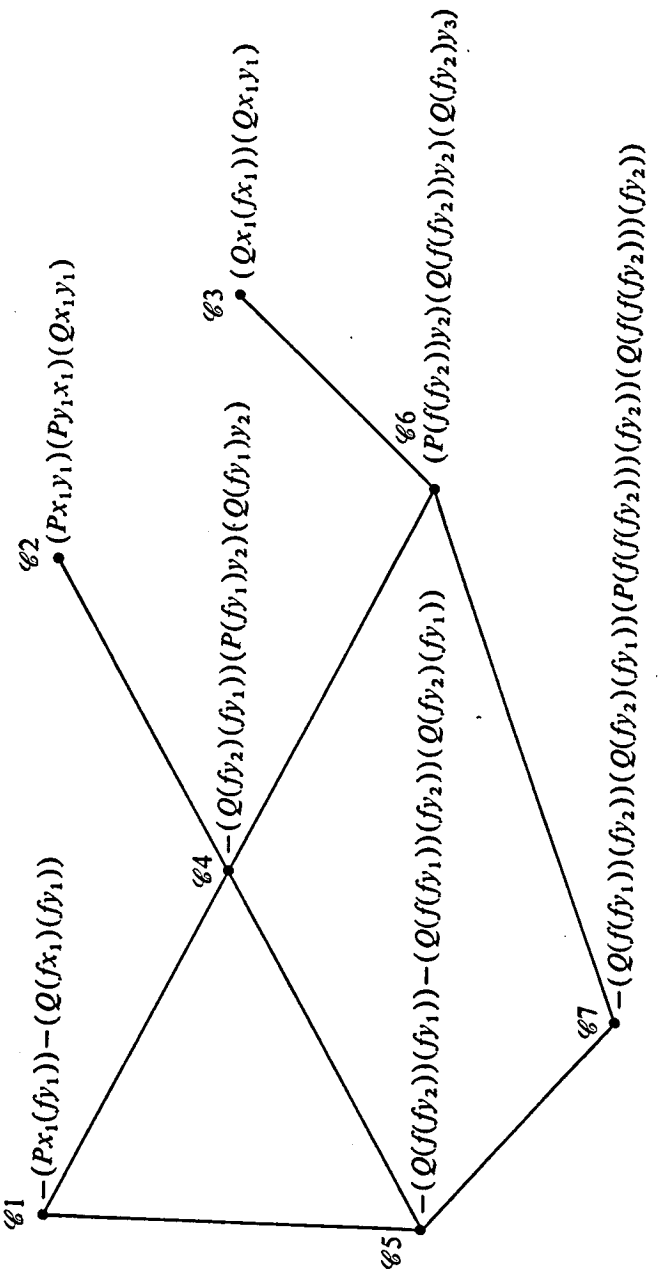

¢1
−(Px1(fy1))−(Q(fx1)(fy1))
¢2
(Px1y1)(Py1x1)(Qx1y1)
¢3
(Qx1(fx1))(Qx1y1)
¢4
−(Q(fy2)(fy1))(P(fy1)y2)(Q(fy1)y2)
¢5
−(Q(f(fy2))(fy1))−(Q(f(fy1))(fy2))(Q(fy2)(fy1))
¢6
(P(f(fy2))y2)(Q(f(fy2))y2)(Q(fy2)y3)
¢7
−(Q(f(fy1))(fy2))(Q(fy2)(fy1))(P(f(f(fy2)))(fy2))(Q(f(f(fy2)))(fy2))

Figure 1(b).

the figure exhibits the tree of clause records, while 1(b) shows the clauses represented at each node.

The clauses labeled C1, C2, and C3 are input clause records. The four remaining clause records are generated by RESOLVE as follows:

C4=RESOLVE(C1,1,C2,2)
C5=RESOLVE(C1,1,C4,2)
C6=RESOLVE(C4,1,C3,1)
C7=RESOLVE(C5,1,C6,3).

It is useful to trace the descent of the 3rd literal of C2 through the tree. In C2 it is represented by the expression T,1 where T is $(+(Q\,x\,y))$. In the binding environment C2 this expression has the value $(Q\,x_1\,y_1)$.

The descendant of this literal in clause C4 is the 3rd literal of that clause. There the expression is T,2 which has value $(Q(f\,y_1)\,y_2)$ in C4 due to the binding $(x,2\,.\,(f\,y),1)$ in BINDINGS(C4).

In C5, the term has index 3 and represents the 3rd literal of C5. T,3 in C5 has the value $(Q(f\,y_2)\;(f\,y_1))$.

Of course, the value of this expression in C5 in no way affects its value in C4. Thus, when we resolve C4 and C3 to form C6, the 3rd literal of C4 descends to become the 2nd literal of C6, where it is represented by T,2. The value of T,2 in C6 is $(Q(f(f\,y_2))\,y_2)$, due to bindings at C6 and C4.

Finally, we can trace the term, T, to C7 where it has index 3 as the 2nd literal and index 5 as the 4th literal of C7. T,3 in C7 has value $(Q(f\,y_2)\;(f\,y_1))$. T,5 in C7 has value $(Q(f(f(f\,y_2)))\;(f\,y_2))$.

Note that the double use of C4 in the tree introduces no confusion of bindings. In using both T,3 and T,5 ISBOUND finds relevant bindings at C4. However, in one case it returns to C4 via the branch through C5, and in the other via the branch through C6.

As an example of how bindings on one side of the tree can affect values of terms from the other, the reader should calculate the value of y,4 in the binding environment C7. It is found to be $(f(f\,y_2))$ after using bindings found at C6, C4, C7, and C5.

8. NOTES

We have completely ignored merging and factoring up to this point; however, they present no difficulty for our representation. All that is required to represent a merge or factor is an indication of which literal is to be deleted and the substitution used. In one of our programs we represent a merge as a resolvent in which one of the parents is a dummy (with one literal and maximum index zero), and the other is the clause containing the literal to be merged. We pretend to be resolving on that literal.

Subsumption and variant checking are also possible. Because the indices can be used to note from which clause a given variable has come, it is easy to redefine the function OCCUR in such a way that UNIFY succeeds only when one term subsumes (or is a variant of) the other. Thus the code for

UNIFY can be made to do several different jobs in this representation.

The recursion in the functions ISBOUND and GETLIT can be replaced by loops. The first recursion in OCCUR and the first two in UNIFY can also be replaced by loops. The resulting code is more efficient and opaque.

A more interesting increase in efficiency can be obtained by eliminating the search ISBOUND makes through the tree of clause records. We set up a two dimensional array we call VALUE of VARIABLES×INDICES. When we expect to use a clause record CL for any length of time (e.g., repeated resolutions, factoring, subsumption), we load VALUE with the binding environment of CL. Then to find if VAR,INDEX is bound we simply check VALUE(VAR, INDEX).

In the context of the VALUE array, BIND takes only four arguments, and it inserts TERMB,INDEXB into VALUE(VAR,INDEX). In addition, it pushes a pointer to the VALUE cell thus modified so that we can recover the substitution produced by UNIFY and later remove the bindings inserted. This allows recursive code for factoring, subsumption checking, and depth-first (backtracking) search. At each level of the recursion, VALUE contains the current BNDEV. Successful unifications add bindings to the array so that it contains the correct binding environment for the resolvent or factor produced. Upon exiting from recursion (in the search or factoring functions, for example), the stack is used to restore VALUE to its configuration upon entry (by removing the bindings inserted since entry).

It was because of the extensive use of VALUE that we chose integers as indices. Actually, all that an index must do is specify a unique branch up the binary tree of clause records. In one of our programs we use logical words treated as bit strings in place of indices. Each bit tells ISBOUND whether to branch to the right or left parent at the current node. Instead of incrementing indices, one shifts them.

Our 7-tuple representation can probably be improved for many restrictions of resolution. For example, in our implementation of SL-resolution (Kowalski and Kuehner 1971) we take advantage of the fact that in SL one parent of each resolvent is an input clause. Furthermore, the literals last to enter a clause are the first resolved upon. Thus SL-derivations have an attractive stack structure in which each stack entry is the residue of an input clause. Instead of keeping the number of literals in a clause, we keep a bit mask for each stack entry to tell us which literals from the input clause are still around. In this representation merging involves merely turning off a bit and storing a substitution.

Since many of the components of our records contain small integers, it is possible to pack these so that a record requires very few machine words. Our general non-linear implementation in POP-2 requires $(7+2n)$ 24-bit words per clause, where n is the number of bindings made. This includes the overhead for the POP-2 structures involved. Our SL implementation requires $(6+2n)$ 24-bit words. A machine code implementation of the general

structure sharing on a 36-bit word machine would require $(2+n)$ words per clause. Of course, the beauty of these expressions is that the space required to represent a clause is independent of its length or the depth of its function nesting.

We have obtained some rough statistics comparing our representation with two others, namely the most obvious list representation and the most compact character array imaginable. The latter is extremely slow to use since one spends almost all of one's time parsing. We assumed a 36-bit word machine was being used. On the basis of 3,000 randomly generated clauses, our representation is 10 times more compact than character arrays (at 5 characters per word) and 50 to 100 times more compact than lists (at 1 cons per word).

This data was generated using a general purpose implementation of structure sharing in POP-2 on an ICL 4130. Each clause was generated using structure sharing and the space required to represent it under the various schemes was then calculated. The VALUE array was not used. On terms whose average function nesting depth was 5, the program required 160 milliseconds per unification. The average longest branch in the derivations searched by ISBOUND was 8.3. For comparison purposes it should be pointed out that POP-2 on the 4130 requires 160 microseconds to execute '1+2' in a compiled function.

Our SL-resolution implementation is written as efficiently as possible in POP-2 and generates 9 clauses per second on the 4130. This includes tautology checking (but not subsumption) for each clause generated. The compiled POP-2 code requires 10K of 24-bit words and the program causes no garbage collection.

J.A.Robinson describes (1971) how unification of terms in the context of a substitution is possible without applying the substitutions. We believe that R.Yates wrote for QA3 the first such algorithm. The idea also appears in Hoffman and Veenker (1971).

Acknowledgements

Our thanks to J.A.Robinson, B.Meltzer, Pat Hayes, Robert Kowalski, and to the Science Research Council for financial support.

REFERENCES

Hoffman, G.R. & Veenker, G. (1971) The unit-clause proof procedure with equality. *Computing*, **7**, 91–105.

Kowalski, R. & Kuehner, D. (1971) Linear resolution with selection function. *Artificial Intelligence*, **2**, 227–60.

Robinson, J.A. (1971) Computational logic: the unification algorithm. *Machine Intelligence 6*, pp. 63–72 (eds Meltzer, B. & Michie, D.) Edinburgh: Edinburgh University Press.