J. Symbolic Computation (1986) **2**, 293–304

A Structure-preserving Clause Form Translation

DAVID A. PLAISTED AND STEVEN GREENBAUM†

Department of Computer Science, University of North Carolina at Chapel Hill, New West Hall 035A, Chapel Hill, North Carolina 27514, U.S.A. (on leave from University of Illinois) and

† *Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, Illinois 61801, U.S.A.*

(*Received 27 February 1986*)

Most resolution theorem provers convert a theorem into clause form before attempting to find a proof. The conventional translation of a first-order formula into clause form often obscures the structure of the formula, and may increase the length of the formula by an exponential amount in the worst case. We present a non-standard clause form translation that preserves more of the structure of the formula than the conventional translation. This new translation also avoids the exponential increase in size which may occur with the standard translation. We show how this idea may be combined with the idea of replacing predicates by their definitions before converting to clause form. We give a method of lock resolution which is appropriate for the non-standard clause form translation, and which has yielded a spectacular reduction in search space and time for one example. These techniques should increase the attractiveness of resolution theorem provers for program verification applications, since the theorems that arise in program verification are often simple but tedious for humans to prove.

1. A Structure-preserving Translation

In the traditional resolution style of theorem proving (Chang & Lee, 1973; Loveland, 1978), a theorem to be proven is first negated and then converted to clause form. The theorem is valid iff the clause form is inconsistent, and resolution theorem provers may be used to detect when a set of clauses is inconsistent. The resolution theorem proving approach has been criticised because the translation to clause form often obscures the structure of the original formula, and makes some simple theorems difficult. The non-clausal theorem proving methods of Murray (1982) attempt to reason using formulae directly, without translating them to clause form. We show how a structure-preserving translation to clause form permits many of the advantages of non-clausal theorem provers to be realised in a resolution theorem prover. This technique also may be of analytical value in proving completeness of various non-clausal strategies, since there is a relationship between resolution using the new translation, and non-clausal theorem provers. This method has been briefly described by Greenbaum *et al.* (1982), and a propositional version has been given by Tseitin (1983). Similar ideas were expressed by G. Minc in 1982 and may also be found in Eder (1984).

The basic idea of the method is to introduce new predicate symbols P_A to refer to various sub-formulae A of the original formula. Then the assertion $P_A \equiv A$ may be added to the set of formulae and A may be replaced by P_A everywhere. We actually give a refined method in which in many cases only the implication $P_A \supset A$ or $A \supset P_A$ is needed. This is

0747-7171/86/030293+12 $03.00/0

related to the concept of "polarity" of Murray (1982). This refined method is specified in the definitions of A^+ and A^- below; these formulae A^+ and A^- are conjunctions containing implications which in some sense represent the implications $P_A \supset A$ and $A \supset P_A$.

Suppose A is a formula to be translated to clause form. We define a unit clause L_A and sets (conjunctions) of formulae A^+ and A^- such that A is satisfiable iff $L_A \wedge A^+$ is satisfiable, and such that $\neg A$ is satisfiable iff $\neg L_A \wedge A^-$ is satisfiable. Each formula in A^+ and A^- is fairly simple in structure. Therefore these formulae may easily be converted to clause form. Let $Cl(A)$ be the set of clauses obtained in the standard clause form translation of A, described, for example, by Chang & Lee (1973). Let $SC(A)$ be the structure preserving clause form translation of A, which we are now defining. Then

$$SC(A) = \cup \{Cl(B) : B \in A^+\} \cup \{L_A\}.$$

The overall translation method, then, is to obtain A^+ from A and take the union of the clauses from each formula in $L_A \wedge A^+$. We need A^- since A^+ and A^- are defined recursively in terms of each other.

DEFINITION. An *atom* is a formula of the form $P(t_1, \ldots, t_n)$ where P is a predicate symbol and the t_i are terms. A *literal* is a formula of the form A or $\neg A$ where A is an atom.

We define L_A to be the literal $P_A(x_1 \cdots x_n)$ where the x_i are the free variables in A, if A is not of the form $\neg B$. If A is of the form $\neg B$, then L_A is $\neg L_B$. Intuitively, L_A is intended to be true in interpretations in which A is true, and false in interpretations in which A is false. However, this is complicated by the fact that L_A contains a predicate not appearing in A. Also, P_A is a new predicate introduced to represent the formula A, and should be chosen so that different subformulae of A correspond to different predicates P_A. We regard A^+ and A^- as conjunctions. That is, A^+ is regarded as the conjunction of its elements, for purposes of the following definitions, and similarly for A^-. We then have the following definitions: (In these definitions, free variables are assumed to be universally quantified.)

$$(A \wedge B)^+ = (L_{A \wedge B} \supset L_A \wedge L_B) \wedge A^+ \wedge B^+$$
$$(A \vee B)^+ = (L_{A \vee B} \supset L_A \vee L_B) \wedge A^+ \wedge B^+$$
$$(\neg A)^+ = A^-$$
$$(A \equiv B)^+ = (L_{A \equiv B} \supset (L_A \equiv L_B)) \wedge A^+ \wedge B^+ \wedge A^- \wedge B^-$$
$$(A \supset B)^+ = (L_{A \supset B} \supset (L_A \supset L_B)) \wedge A^- \wedge B^+$$
$$(A \wedge B)^- = ((L_A \wedge L_B) \supset L_{A \wedge B}) \wedge A^- \wedge B^-$$
$$(A \vee B)^- = ((L_A \vee L_B) \supset L_{A \vee B}) \wedge A^- \wedge B^-$$
$$(\neg A)^- = A^+$$
$$(A \equiv B)^- = ((L_A \equiv L_B) \supset L_{A \equiv B}) \wedge A^- \wedge B^- \wedge A^+ \wedge B^+$$
$$(A \supset B)^- = ((L_A \supset L_B) \supset L_{A \supset B}) \wedge A^+ \wedge B^-$$
$$(\forall\, x A(x))^+ = A(x)^+ \wedge (L_{\forall x A(x)} \supset \forall\, x L_{A(x)})$$
$$(\exists\, x A(x))^+ = A(x)^+ \wedge (L_{\exists x A(x)} \supset \exists\, x L_{A(x)})$$
$$(\forall\, x A(x))^- = A(x)^- \wedge (\forall\, x L_{A(x)} \supset L_{\forall x A(x)})$$
$$(\exists\, x A(x))^- = A(x)^- \wedge (\exists\, x L_{A(x)} \supset L_{\exists x A(x)}).$$

Finally, if A is a literal, then L_A is A, and $A^+ = A^- = \emptyset$, that is, TRUE. We note that when computing A^+ and A^- care should be taken to avoid repeated subcomputations. One way to do this is to store values in a table as they are computed, so that when A^+ and A^- are computed two or more times, the value is retrieved from the table the second and succeeding times and is not recomputed. Otherwise, in formulae containing many occurrences of the equivalence connective, the translation can take time exponential in the size of the formula. An efficient method of computing A^+ and A^- will be given in section 1.2. We assume duplicate occurrences of formulae are removed from A^+ and A^- so that the length of A^+ and A^- is polynomial in the length of A.

Examples:

(1) Suppose A is $(Q_1 \wedge Q_2) \vee (Q_3 \wedge Q_4)$. Using the definition of A^+ when A is a disjunction, we obtain that

$$A^+ = (L_A \supset (L_{Q_1 \wedge Q_2} \vee L_{Q_3 \wedge Q_4})) \wedge (Q_1 \wedge Q_2)^+ \wedge (Q_3 \wedge Q_4)^+.$$

Applying the definition for conjunctions, we have

$$(Q_1 \wedge Q_2)^+ = (L_{Q_1 \wedge Q_2} \supset (Q_1 \wedge Q_2)) \wedge Q_1^+ \wedge Q_2^+.$$

Applying the definition for atoms, we have Q_1^+ and Q_2^+ are empty, that is, TRUE. Similarly,

$$(Q_3 \wedge Q_4)^+ = (L_{Q_3 \wedge Q_4} \supset (Q_3 \wedge Q_4)) \wedge Q_3^+ \wedge Q_4^+.$$

Also, Q_3^+ and Q_4^+ are TRUE. Putting all this together, we have that A^+ is

$$(L_A \supset (L_{Q_1 \wedge Q_2} \vee L_{Q_3 \wedge Q_4})) \wedge (L_{Q_1 \wedge Q_2} \supset (Q_1 \wedge Q_2)) \wedge (L_{Q_3 \wedge Q_4} \supset (Q_3 \wedge Q_4)).$$

Assuming that P_A is R_1 and $P_{Q_1 \wedge Q_2}$ is R_2 and $P_{Q_3 \wedge Q_4}$ is R_3, we obtain that A^+ is

$$\{R_1 \supset R_2 \vee R_3,\ R_2 \supset Q_1 \wedge Q_2,\ R_3 \supset Q_3 \wedge Q_4\}.$$

From this it follows that $SC(A)$ is

$$\{R_1,\ (\neg R_1 \vee R_2 \vee R_3),\ (\neg R_2 \vee Q_1),\ (\neg R_2 \vee Q_2),\ (\neg R_3 \vee Q_3),\ (\neg R_3 \vee Q_4)\}.$$

Note that on small formulae, $SC(A)$ is often more complex than $Cl(A)$.

(2) Suppose A is $(Q_1 \wedge Q_2) \vee \neg(Q_3 \wedge Q_4)$. Then, again assuming P_A is R_1 and the predicates P_B for various subformulae B of A are chosen properly, A^+ is

$$\{R_1 \supset R_2 \vee \neg R_3,\ R_2 \supset Q_1 \wedge Q_2,\ Q_3 \wedge Q_4 \supset R_3\}.$$

(3) Suppose A is $Q_1 \supset \forall x(Q_2(x) \vee Q_3(x))$. Let A_2 be $\forall x(Q_2(x) \vee Q_3(x))$, let A_3 be $Q_2(x) \vee Q_3(x)$, and let A_1 be A. Let P_{A_i} be R_i for $i = 1, 2, 3$. Then L_{A_1} is R_1, L_{A_2} is R_2, and L_{A_3} is $R_3(x)$ since A_3 has free variable x. Also, A^+ is

$$\{R_1 \supset (\neg Q_1 \vee R_2),\ R_2 \supset \forall x R_3(x),\ R_3(x) \supset Q_2(x) \vee Q_3(x)\}.$$

Thus $SC(A)$ is

$$\{R_1,\ \neg R_1 \vee \neg Q_1 \vee R_2,\ \neg R_2 \vee R_3(x),\ \neg R_3(x) \vee Q_2(x) \vee Q_3(x)\}.$$

1.1. PROPERTIES OF THE TRANSLATION

We now show that the structure preserving translation to clause form is consistency preserving.

THEOREM 1. *For all first-order formulae W,*

$$\models (L_W \wedge W^+) \supset W \quad \textit{and} \quad \models (\neg L_W \wedge W^-) \supset \neg W.$$

PROOF. By induction on the structure of W, using case analysis on the top level connective. We need to show that the following formulae are valid:

$$L_{A \wedge B} \wedge (A \wedge B)^+ \supset (A \wedge B)$$
$$L_{A \vee B} \wedge (A \vee B)^+ \supset (A \vee B)$$
$$L_A \wedge (\neg A)^+ \supset (\neg A)$$
$$L_{A \equiv B} \wedge (A \equiv B)^+ \supset (A \equiv B)$$
$$L_{A \supset B} \wedge (A \supset B)^+ \supset (A \supset B)$$

and similar formulae for W^- and for quantifiers. We illustrate a few cases; the rest are similar. For the base case, we need to show that $\models (L_W \wedge W^+) \supset W$ and $\models (\neg L_W \wedge W^-) \supset \neg W$ if W is an atom.

First we do the base case. If W is an atom then L_W is W and W^+ and W^- are empty (that is, TRUE) so we immediately have $\models (L_W \wedge W^+) \supset W$ and $\models (\neg L_W \wedge W^-) \supset \neg W$.

Suppose W is of the form $A \wedge B$. We need to show that

$$L_{A \wedge B} \wedge (A \wedge B)^+ \supset (A \wedge B)$$

is valid. Now, $(A \wedge B)^+$ by definition is

$$(L_{A \wedge B} \supset L_A \wedge L_B) \wedge A^+ \wedge B^+.$$

Also, by induction on the size of W we know that

$$\models (L_A \wedge A^+) \supset A \quad \text{and} \quad \models (L_B \wedge B^+) \supset B.$$

Therefore

$$L_{A \wedge B} \wedge (A \wedge B)^+ \supset (A \wedge B)$$

is valid. To see this, note that $L_{A \wedge B} \wedge (A \wedge B)^+$ implies $L_A \wedge L_B \wedge A^+ \wedge B^+$. By the inductive hypotheses, we obtain $A \wedge B$.

Suppose W is of the form $A \vee B$. We need to show that

$$L_{A \vee B} \wedge (A \vee B)^+ \supset (A \vee B)$$

is valid. Now, $(A \vee B)^+$ by definition is

$$(L_{A \vee B} \supset L_A \vee L_B) \wedge A^+ \wedge B^+.$$

Also, by induction on the size of W we know that

$$\models (L_A \wedge A^+) \supset A \quad \text{and} \quad \models (L_B \wedge B^+) \supset B.$$

Therefore

$$L_{A \vee B} \wedge (A \vee B)^+ \supset (A \vee B)$$

is valid. To see this, note that $L_{A \vee B} \wedge (A \vee B)^+$ implies $(L_A \vee L_B) \wedge A^+ \wedge B^+$. By the inductive hypotheses, we obtain $A \vee B$. Similar arguments work for the other connectives.

We now look at W^-. Suppose W is of the form $A \wedge B$. We need to show that

$$\neg L_{A \wedge B} \wedge (A \wedge B)^- \supset \neg (A \wedge B)$$

is valid. Now, $(A \wedge B)^-$ by definition is

$$((L_A \wedge L_B) \supset L_{A \wedge B}) \wedge A^- \wedge B^-.$$

Also, by induction on the size of W we know that

$$\models (\neg L_A \wedge A^-) \supset \neg A \quad \text{and} \quad \models (\neg L_B \wedge B^-) \supset \neg B.$$

Therefore

$$\neg L_{A \wedge B} \wedge (A \wedge B)^- \supset \neg (A \wedge B)$$

is valid. To see this, note that $\neg L_{A \wedge B} \wedge (A \wedge B)^-$ implies $\neg (L_A \wedge L_B) \wedge A^- \wedge B^-$. By the inductive hypotheses, we obtain $\neg (A \wedge B)$. The other connectives are handled similarly.

We consider also the case in which W has a quantifier at the top level. Suppose W is of the form $\forall\, xA(x)$. We need to show that

$$L_{\forall xA(x)} \wedge (\forall\, xA(x))^+ \supset \forall\, xA(x).$$

By definition, $(\forall\, xA(x))^+$ is

$$A(x)^+ \wedge (L_{\forall xA(x)} \supset \forall\, xL_{A(x)}).$$

Also, by induction on the size of W we know that $\models (L_{A(x)} \wedge A(x)^+) \supset A(x)$. Let us assume that $L_{\forall xA(x)} \wedge (\forall\, xA(x))^+$. Using the definition of $(\forall\, xA(x))^+$ we obtain $A(x)^+ \wedge \forall\, xL_{A(x)}$. Since in the definitions, free variables are assumed to be universally quantified, we have $\forall\, x(A(x)^+) \wedge \forall\, xL_{A(x)}$. Applying the inductive hypotheses, we obtain that $\forall\, xA(x)$ is as desired. Thus we know that

$$\models L_{\forall xA(x)} \wedge (\forall\, xA(x))^+ \supset \forall\, xA(x).$$

The other cases for quantifiers are similar.

LEMMA 1. *Suppose I is a model of W. Let J be an interpretation such that for all sub-formulae V of W, $J \models (L_V \equiv V)$. Suppose J interprets V to be true exactly when $I \models V$. Then for all sub-formulae V of W, $J \models V^+$ and $J \models V^-$.*

PROOF. By induction on the size of V. For atoms V, V^+ and V^- are TRUE so the result is immediate. Suppose V is of the form $A \wedge B$. Then V^+ is

$$(L_{A \wedge B} \supset L_A \wedge L_B) \wedge A^+ \wedge B^+.$$

By induction, $J \models A^+$ and $J \models B^+$. Also, $J \models L_{A \wedge B} \equiv (A \wedge B)$ and $J \models L_A \equiv A$ and $J \models L_B \equiv B$. Therefore,

$$J \models (L_{A \wedge B} \supset L_A \wedge L_B).$$

Therefore,

$$J \models (L_{A \wedge B} \supset L_A \wedge L_B) \wedge A^+ \wedge B^+.$$

The other cases are similar.

LEMMA 2. *Suppose W is satisfiable. Then $L_W \wedge W^+$ is satisfiable. Suppose $\neg W$ is satisfiable. Then $\neg L_W \wedge W^-$ is satisfiable.*

PROOF. Let I be a model of W. Let J be as in lemma 1. Then $J \models W^+$ by lemma 1. Also, $J \models L_W$ by the way J is defined. Hence $J \models L_W \wedge W^+$, which is satisfiable. A similar argument works for $\neg L_W \wedge W^-$.

THEOREM 2. *For a first-order formula A, A is satisfiable iff $L_A \wedge A^+$ is satisfiable, and $\neg A$ is satisfiable iff $\neg L_A \wedge A^-$ is satisfiable.*

PROOF. Using theorem 1 and lemma 2 above.

COROLLARY. *A first-order formula A is satisfiable iff the set $SC(A)$ of clauses is satisfiable.*

PROOF. For any formula B, B is satisfiable iff $Cl(B)$ is satisfiable.

One of the main advantages of this translation is that it permits sub-formulae to occur naturally in proofs. For example, suppose A is $B_1 \wedge (B_1 \supset B_2) \wedge \neg B_2$ where B_1 and B_2 are large sub-formulae. Then A is obviously inconsistent and $Cl(A)$ is inconsistent, but the

shortest resolution proof of inconsistency from $Cl(A)$ may be very complicated, due to the fact that the sub-formulae B_i will be split up into many clauses which will contain pieces of both formulae. However, there will be a very short resolution proof of inconsistency from $SC(A)$ since new predicate symbols will be generated for the sub-formulae B_i. This will still be true if the two occurrences of a sub-formula, say B_1, differ in trivial ways. For example, if one occurrence is $C_1 \vee C_2$ and the other is $C_2 \vee C_1$, then there will still be a short proof of inconsistency. In fact, for resolution provers, it is sometimes difficult even to show that $B \wedge \neg B$ is inconsistent, where B is a complicated formula, if the usual clause form translation is used. This is because B is split into many clauses. Such problems have led many people to conclude that non-clausal strategies are essential. However, we feel that many of the advantages of non-clausal strategies may be obtained in a conventional resolution prover by the use of this structure-preserving clause form translation.

1.2. TIME AND SPACE COMPLEXITY

THEOREM 3. *The length of A^+ and A^- is $O(n * c(A))$, where n is the number of variables in A and $c(A)$ is the number of connectives and quantifiers in A. Also, A^+ and A^- may be computed from A in time $O(n * c(A))$.*

PROOF. Using a stack, in linear time we can traverse A and represent it as a tree structure in memory. We assign a unique integer identifier to each sub-formula of A. If i is the identifier assigned to B, then P_B is R_i. To compute A^+ we begin at the root of the tree. Now, A^+ is defined recursively in terms of B^+ and B^- for B an immediate sub-formula of A; thus B will be a son of A in the tree representation. Also, A^+ is defined in terms of an implication involving L_A and L_B for such formulae B. This implication can be output just by using the integer identifiers associated with A and sub-formulae B. Now, L_A and L_B may have up to n free variables, so their length can be proportional to n, and the time to write them can be proportional to n. For each such sub-formula, we then recursively compute whichever of B^+ and B^- are necessary; sometimes both are necessary. If both are necessary, this must be done in a way to avoid duplication of effort. The details are routine, so we omit them here.

The size of A^+ is $O(n * c(A))$ since the output per node in the tree representation of A is proportional to n, and similarly for A^-. The algorithm also takes time $O(n * c(A))$ because the effort per node in the tree is bounded by n.

Note that this bound may not be linear; consider, for example, a formula A containing n quantifiers preceding $Q(x_1, \ldots, x_n)$. Then A^+ and A^- are of length $O(n^2)$ and the algorithm above takes time $O(n^2)$ to output them.

We now consider the time to compute $SC(A)$. Let $|A|$ be the length of a formula A when expressed as a character string; let $|SC(A)|$ be the length of $SC(A)$ when written out as a character string.

THEOREM 4. *$|SC(A)|$ is $O(n * |A|)$ where n is the number of variables in A, and $SC(A)$ may be computed in time $O(n * |A|)$ from A.*

PROOF. In A^+, literals L_B occur for various sub-formulae B of A. Each such literal occurs at most a constant number of times. Let us call a sub-formula an *interior* sub-formula if it contains a logical connective, and if B is an interior sub-formula, let us call L_B an interior literal. Let us call other sub-formulae and literals *leaf* sub-formulae and leaf literals, respectively. This is a natural terminology when we think of A as a tree. Note that if B is a

leaf sub-formula of A then B is an atom. Now, interior literals will be preceded by at most one existential quantifier in their occurrences in A^+, and this quantifier will bind a variable that occurs at most once in the literal. Thus, conversion to clause form will introduce at most one Skolem function that may add at most n to their length. This gives a bound of $O(n * c(A))$ for the length of this part of $SC(A)$, since $|A^+|$ is $O(n * c(A))$. Leaf literals will be preceded by at most one existential quantifier but it may quantify a variable that appears any number of times. However, the length of the leaf literals will be multiplied by a factor of at most n by this introduction of Skolem functions. Since the sum of the lengths of the leaf literals is bounded by $|A|$, the bound for this part of $SC(A)$ is $O(n * |A|)$. Summing these, the total bound is $O(n * |A|)$. The time to compute $SC(A)$ this way is $O(n * |A|)$ also.

If we represent formulae as directed acyclic graphs, so that common subterms need be represented only once, then the above bounds for time and space reduce to $O(n * c(A))$ as before, since multiple occurrences of an existentially quantified variable will be replaced by the same term during Skolemisation. Also, if we slightly modify the definition of A^+ and A^- we can obtain these smaller time and space bounds for computing $SC(A)$ even without the directed acyclic graph representation. The idea is, for example, to define $\exists\, xP(x, x)^+$ to be

$$R_1 \supset \exists\, xR_2(x) \wedge \forall\, x(R_2(x) \supset P(x, x)),$$

where R_1 is $L_{\exists xP(x,x)}$. The effect is that multiple occurrences of a variable are never bound by an existential quantifier.

1.3. MODIFICATIONS

There are some modifications of this approach which can improve its effectiveness. First, it is probably best to translate small sub-formulae directly into clause form rather than using the structure preserving translation. One good method, suggested by a student (Lame, 1984) is to translate a sub-formula B using the standard translation if B is itself a clause, that is, a disjunction of literals. That is, if B is a clause, then we can let B^+ be simply the formula $L_B \supset B$ and we can let B^- be $B \supset L_B$. Another method is to economise for strings of quantifiers. For example, we can define $\forall\, x\, \exists\, yA(x, y)^+$ to be

$$L_{\forall x\, \exists yA(x,y)} \supset \forall\, x\, \exists\, yL_{A(x,y)} \wedge A(x, y)^+.$$

A further improvement is to economise on sub-formulae which are instances of a common, more general, formula. For example, suppose A is $B(a, x) \wedge B(y, b)$. Then we can let A^+ be

$$(L_A \supset (P(a, x) \wedge P(y, b))) \wedge B(x, y)^+$$

where P is $P_{B(x,y)}$. That is, instead of choosing different predicates to represent the sub-formulae $B(a, x)$ and $B(y, b)$, we choose one predicate P to represent a more general formula $B(x, y)$ and then represent $B(a, x)$ by $P(a, x)$ and $B(y, b)$ by $P(y, b)$. To give a formal definition, we say in this case that $L_{B(a,x)} = P_{B(x,y)}(a, x)$ instead of $L_{B(a,x)} = P_{B(a,x)}(x)$. Also, we say that $B(a, x)^+$ is defined to be $B(x, y)^+$ in this case, and $B(a, x)^-$ is defined to be $B(x, y)^-$. Similarly, we have in this case that $L_{B(y,b)} = P_{B(x,y)}(y, b)$, etc. It seems always best to use this method for sub-formulae that are variants or instances of one another. For sub-formulae which are both instances of a common, more general, formula, the advantage is that the clause form $SC(A)$ has fewer clauses, but the clauses may have more free variables.

It is easy to identify sub-formulae that are variants of one another. To do this, the

variables in a sub-formula can be renamed $x_1, \ldots, x_n$ from left to right according to the left-most occurrence of a variable. This will make sub-formulae which are variants identical, so they can be collected together using hashing or some kind of tree structure for detecting repeated strings. Identifying sub-formulae which are instances of other sub-formulae seems harder. It seems to be necessary to test each sub-formulae against all others, to see which ones it is an instance of; this testing can be done reasonably fast using a "discrimination net" data structure as described by Charniak *et al.* (1980). Our resolution prover actually uses a data structure much like a discrimination net to detect when one clause is an instance of another, and it works quite well.

2. Replacing Predicates and Functions by Their Definitions

We now show how the above idea may be combined with another modification of the clause form translation. Often a predicate will have a definition in terms of simpler predicates. For example, $x \leqslant y$ iff $\neg(y < x)$. This may also be true of functions. For example, $|x| = (\text{if } x > 0 \text{ then } x \text{ else } -x)$. Now, suppose A is a theorem to be proven. If in A all occurrences of some predicate P are replaced by their definitions, then it may be possible to eliminate all axioms about P from the statement of the theorem. Similarly, if in A all occurrences of some function symbol f are replaced by their definitions, then it may also be possible to eliminate all axioms about f from the statement of the theorem. This can be a significant help, since many irrelevant facts can be deduced from axioms during the search for a proof. We tried this idea on the theorem

$$P(A \cap B) = P(A) \cap P(B),$$

where A and B are sets, $\cap$ is set intersection, and P is the powerset operation. We used the following definitions:

$$x = y \text{ iff } x \subset y \wedge y \subset x$$
$$x \subset y \text{ iff } \forall z(z \in x \supset z \in y)$$
$$x \in P(y) \text{ iff } x \subset y$$
$$x \in y \cap z \text{ iff } x \in y \wedge x \in z.$$

These definitions permit us to reduce the theorem entirely to a statement about set membership in A and B and other sets, making the usual axioms about set intersection, subset, set equality, and powerset unnecessary. For example, using the definition of equality, the theorem is rewritten to

$$P(A \cap B) \subset (P(A) \cap P(B)) \wedge (P(A) \cap P(B)) \subset P(A \cap B).$$

Using the definition of subset, this rewrites to

$$\forall z(z \in P(A \cap B) \supset z \in (P(A) \cap P(B))) \wedge \forall z(z \in (P(A) \cap P(B)) \supset z \in P(A \cap B).$$

Note that care must be taken to avoid name conflicts when introducing bound variables, as the subset definition requires. Further rewriting yields

$$\forall z((z \subset (A \cap B)) \supset (z \in P(A) \wedge z \in P(B))) \wedge \forall z(z \in P(A) \wedge z \in P(B)) \supset (z \subset (A \cap B)).$$

Rewriting continues in this way until no more rewriting is possible. Using these techniques, followed by the usual clause form translation, we obtained the proof in about 28 seconds on a VAX 750, equivalent to about 20 seconds on a VAX 780. Our prover is implemented in Franz lisp at the University of Illinois. Without the replacement of

predicates by their definitions, we never could get this proof using any ordinary resolution strategy operating on $Cl(A)$, even with various restrictions on depth bound, term depth, etc. The prover quickly generates many irrelevant facts about sets and runs out of space.

As an example of a function definition, consider the theorem $||x|| = |x|$ where $|x|$ is the absolute value of x. We have the definition

$$|x| = (\text{if } x > 0 \text{ then } x \text{ else } -x).$$

Rewriting both occurrences of $|x|$ yields

$$|(\text{if } x > 0 \text{ then } x \text{ else } -x)| = (\text{if } x > 0 \text{ then } x \text{ else } -x).$$

Pulling if-then-else out from the left- and right-hand sides at the same time yields

$$(\text{if } x > 0 \text{ then } |x| = x \text{ else } |-x| = -x).$$

Replacing (if a then b else c) at the predicate level by $(a \wedge b) \vee (\neg a \wedge c)$ we obtain

$$(x > 0 \wedge |x| = x) \vee (x \leqslant 0 \wedge |-x| = -x).$$

Eventually all occurrences of $|\ldots|$ and if-then-else are eliminated from the theorem and we obtain a theorem entirely about inequalities. This permits us to avoid including axioms like $x > 0 \supset |x| > 0$ from the statement of the theorem. Note that such axioms can lead to irrelevant deductions. For example, given that $a > 0$ we can get $|a| > 0$ and $||a|| > 0$ and $|||a||| > 0$ etc., by unrestricted resolution. Such deductions are eliminated when the axioms about absolute value are eliminated. This rewriting approach is related to "demodulation", introduced by Wos *et al.* (1967). Rewriting in the context of natural deduction has been discussed by Bledsoe (1971, 1984).

3. Choosing Lock Indices

We next tried using the above idea, that is, replacing predicates by their definitions before translating to clause form, together with the structure-preserving clause form translation. Our initial results were very bad. However, with the proper modification, the results were quite good. Our prover uses resolution with indexing (Boyer, 1971), with indices chosen so that negative literals resolve away before positive literals. The key idea was to index literals having predicates referring to small sub-formulae, lower than literals of the same sign having predicates referring to large sub-formulae. Thus in the course of a proof, predicates referring to small sub-formulae will resolve away first. In a formula in which there are few interactions between the large sub-formulae, this will lead to fairly simple clauses about the large sub-formulae and to a fast proof. In fact, we obtained a proof in about 3 seconds on the VAX 750 in this way! When the reverse order is chosen, namely, to resolve away predicates referring to large sub-formulae first, then the resolution procedure essentially generates the standard clause form translation first and then attempts to find a proof. This degrades performance because the extra clauses generated along the way increase the search space.

4. Examples

We ran a number of set theory examples using our Franz lisp resolution theorem prover and the above techniques. The times and search spaces are as given below. These examples were run on a VAX 750. Note that the time for $P(A \cap B) = P(A) \cap P(B)$ is

slower than above; this is because the translation below was done mechanically, while the above result was obtained using a hand translation. For these examples, we used the structure-preserving clause form translation in the following way: The theorem to be proven was first negated and skolemised. Thus, for the theorems

$$\text{AXIOMS} \supset P(x \cap y) = P(x) \cap P(y)$$

we negate and obtain

$$\text{AXIOMS} \wedge \neg \exists x \exists y P(x \cap y) = P(x) \cap P(y).$$

Skolemising the theorem, we obtain

$$\text{AXIOMS} \wedge \neg P(A \cap B) = P(A) \cap P(B).$$

We then use the axioms as definitions to rewrite

$$\neg P(A \cap B) = P(A) \cap P(B)$$

into a much more complicated formula W which is unsatisfiable iff the original theorem is valid. Finally, we compute $SC(W)$ and give it to the resolution theorem prover. The advantage of this method is that W contains few free variables, which makes the search space smaller. This method of skolemising before applying the structure-preserving clause form translation, appears to be especially useful for theorems which contain only universally quantified variables, since negating and skolemising results in a ground formula. We feel that the results obtained in this way show the usefulness of rewriting before computing clause form, as well as the benefit gained on large problems by the structure-preserving clause form translation.

5. Conclusions

We feel that these experiences demonstrate the potential of these three techniques, used together: (1) The structure-preserving clause form translation. (2) Replacing predicates and functions by their definitions and eliminating axioms when possible. (3) Performing locking resolution in which predicates representing smaller sub-formulae are indexed to resolve away before predicates representing larger sub-formulae. These techniques resulted in a problem which was considerably beyond the reach of our prover, becoming a trivial problem. Note that these techniques are all quite general and apply to any problem domain in which predicates and functions have first-order definitions. In fact, techniques (1) and (3) can even be applied without (2). We found that no two of these techniques separately came close to the performance of all three together on the last example mentioned above, namely, showing that $P(A \cap B) = P(A) \cap P(B)$.

It appears that these methods should significantly increase the attractiveness of resolution theorem provers for program verification and proof checking applications. Resolution has always been appealing because of its generality, but has suffered from inefficiency and the necessity for close human interaction. The specialised decision procedures of Shostak (1977) and Nelson & Oppen (1979, 1980) are much faster when they apply but are, of course, much less general. There may be many areas in which resolution can now be used for program verification applications where specialised decision procedures do not apply. Another possible technique for such applications is non-clausal strategies. These strategies need more experimentation to determine their potential. However, we are essentially simulating a refinement of non-clausal resolution using clausal resolution, since a clause containing literals L_A may be regarded as a disjunction of sub-formulae by replacing L_A by A everywhere. In this way, resolution

Example	Cpu time, seconds, exclusive of garbage collection with all predicates replaced by definitions before translation to clause form	
	Standard clause form	Structure-preserving clause form
$A \cup A = A$	0·25	0·5
$A \cup B = B \cup A$	1·233	1·216
$(A \cup B) \cup C = A \cup (B \cup C)$	3·483	3·3
$A \cap A = A$	0·233	0·566
$A \cap B = B \cap A$	1·066	1·233
$(A \cap B) \cap C = A \cap (B \cap C)$	3·4	2·366
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	11·7	8·2
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	6	5
$(A = (A \cap B)) \equiv (A \subset B)$	29	0·65
$(A = (A \cap B)) \supset (A \subset B)$	0·8	0·866
$(A \subset B) \supset (A = (A \cap B))$	0·1	0·766
$[A-(A-B)] = [A \cap B]$	1·466	1·683
$[A \cap (B-C)] = [(A \cap B)-(A \cap C)]$	3·316	3·95
$P(A \cap B) = P(A) \cap P(B)$	30	9·9

Example	Clauses saved, with all predicates replaced by definitions before translation to clause form, as above	
	Standard clause form	Structure-preserving clause form
$A \cup A = A$	2	4
$A \cup B = B \cup A$	9	12
$(A \cup B) \cup C = A \cup (B \cup C)$	23	32
$A \cap A = A$	2	6
$A \cap B = B \cap A$	7	14
$(A \cap B) \cap C = A \cap (B \cap C)$	18	29
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	49	81
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	24	51
$(A = (A \cap B)) \equiv (A \subset B)$	62	1
$(A = (A \cap B)) \supset (A \subset B)$	8	10
$(A \subset B) \supset (A = (A \cap B))$	0	8
$[A-(A-B)] = [A \cap B]$	10	18
$[A \cap (B-C)] = [(A \cap B)-(A \cap C)]$	18	45
$P(A \cap B) = P(A) \cap P(B)$	150	76

proofs using the structure-preserving translation may be regarded as non-clausal proofs and, in fact, this may be a good format for presenting them to the user. Also, this idea may be helpful for obtaining completeness results for refinements of non-clausal resolution strategies, since many such complete refinements are known for clausal resolution. There may be some loss of efficiency due to the simulation of non-clausal strategies by ordinary resolution strategies, but the ability to change the representation without recoding the theorem prover more than compensates, in our opinion. The natural

deduction approach of Bledsoe (1983) has already been shown practical for program verification, and is another possibility.

This work was supported in part by the National Science Foundation under grant MCS 83-07755.

References

Bledsoe, W. (1971). Splitting and reduction heuristics in automatic theorem proving. *Artif. Intell.* **2,** 55–77.

Bledsoe, W. (1983). The UT natural deduction prover. University of Texas Mathematics Dept. Memo ATP-17B.

Bledsoe, W. (1984). Some automatic proofs in analysis. In: (Bledsoe, W., Loveland, D., eds) *Automated Theorem Proving: After 25 Years. Contemporary Mathematics*, Vol. 29, pp. 89–118. New York: American Mathematical Society.

Boyer, R. (1971). *Locking, a Restriction of Resolution.* Ph.D. thesis, University of Texas at Austin.

Chang, C., Lee, R. (1973). *Symbolic Logic and Mechanical Theorem Proving.* New York: Academic Press.

Charniak, E., Riesbeck, C., McDermott, D. (1980). *Artificial Intelligence Programming.* New Jersey: Erlbaum Associates.

Eder, E. (1984). An implementation of a theorem prover based on the connection method. In (Bibel, W., Petkoff, B., eds) *AIMSA '84, Artificial Intelligence—Methodology Systems Applications, Varna, Bulgaria, September 1984,* pp. 121–128. Amsterdam: North-Holland.

Greenbaum, S., Nagasaka, P., O'Rorke, P., Plaisted, D. (1982). Comparison of natural deduction and locking resolution implementations. In: (Loveland, D., ed.) Proceedings of 6th Conference on Automated Deduction, *Springer Lec. Notes Comp. Sci.* **138,** 159–171.

Lame, D. (1984). Personal communication.

Loveland, D. (1978). *Automated Theorem Proving: A Logical Basis.* New York: North-Holland.

Murray, N. (1982). Completely non-clausal theorem proving. *Artif. Intell.* **18,** 67–85.

Nelson, G., Oppen, D. (1979). Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Syst.* **1,** 245–257.

Nelson, G., Oppen, D. (1980). Fast decision procedures based on congruence closure. *J. Assoc. Comp. Mach.* **27,** 356–364.

Shostak, R. (1977). On the SUP-INF method for proving Presburger formulas. *J. Assoc. Comp. Mach.* **24,** 529–543.

Tseitin, G. (1983). On the complexity of derivations in propositional calculus. In: (Siekmann, J., Wrightson, G., eds) *Automation of Reasoning 2: Classical Papers on Computational Logic,* pp. 466–483. Berlin: Springer-Verlag.

Wos, L., Robinson, G., Carson, D., Shalla, L. (1967). The concept of demodulation in theorem proving. *J. Assoc. Comp. Mach.* **14,** 698–709.