## A Natural Proof System Based On Rewriting Techniques

**Theorem proving procedures for the propositional calculus have traditionally relied on syntactic manipulations of the formula to derive a proof. In particular, clausal theorem provers sometimes lose some of the obvious semantics present in the theorem, in the process of converting the theorem into an unnatural normal form. Most existing propositional theorem provers do not incorporate substitution of equals for equals as an inference rule. In this paper we develop a "natural" proof system for the propositional calculus, with the goal that most succinct mathematical proofs should be encodable as short formal proofs within the proof system.**

1. The substitution principle for the equivalence connective is incorporated as an inference rule.
2. A limited version of the powerful ideas of extension, originally suggested by Tseitin, are exploited. Extension allows the introduction of auxiliary variables to stand for intermediate sub-formulas in the course of a proof.
3. Formulas are standardized by converting them into a normal form, while at the same time preserving the explicit semantics inherent in the formula.
4. A generalization of the semantic tree approach is used to perform case analysis on literals as well as sub-formulas.
5. Additional enhancements such as a generalization of resolution are suggested.

We show that from a complexity theoretic viewpoint NPS is at least as powerful as the resolution procedure. We further demonstrate formulas on which NPS fares better than resolution. Finally, since proofs in NPS usually resemble manual proofs, we feel that NPS is easily amenable to an interactive theorem prover.

**KEY WORDS**
- propositional proof systems
- rewrite rules
- extension
- resolution
- semantic trees
A NATURAL PROOF SYSTEM BASED ON REWRITING TECHNIQUES
D. Kapur and B. Krishnamurthy

1. INTRODUCTION

With increased interest in theorem proving procedures, a collection of proof systems have been proposed in the literature to handle a variety of logical theories. Nevertheless, theorem proving in the propositional calculus lies at the heart of most of those procedures. Even though theoretical evidence has all but foreclosed the possibility of a uniform efficient theorem prover even for the propositional calculus, the hope remains that we can develop practical theorem provers that can cope with a large collection of "naturally occurring" theorems. Perhaps a natural and desirable class of theorems that such proof systems must be capable of handling are those for which there exist succinct mathematical proofs. It would then appear that such a proof system must incorporate the techniques that we employ when proving theorems manually. This paper deals with the development of a "natural" proof system.

We distinguish a proof system from a proof procedure. A proof system is a non-deterministic procedure that defines the notion of a proof by providing the inference rules that may be used. Any sequence of formulas that are derived using the given set of inference rules forms a valid proof of the final formula in the sequence. Non-determinism arises from the fact that the proof system does not specify any order in which the rules are to be applied. The complexity of a proof is then defined as the length of the proof.

In contrast, a proof procedure is a deterministic procedure that not only lays out the inference rules, but also incorporates specific heuristics that precisely determine the order in which the applicability of the rules are checked and subsequently applied. Consequently, given a theorem, a (complete) proof procedure generates a unique proof for the theorem. The complexity of the proof is then defined as the time/space complexity of finding the proof.

Observe that there exists polynomially bounded proof procedures for theorem proving in the propositional calculus if and only if $P = NP$. On the other hand, there exists polynomially bounded proof systems for theorem proving in the propositional calculus if and only if NP is closed under complementation (see [4]). It is commonly believed that neither $P = NP$ nor NP is closed under complementation. Thus, we should expect neither a polynomially bounded proof procedure nor such a proof system for the propositional calculus. Nevertheless, we could ask if most mathematical proofs that we normally encounter, can be encoded within a given proof system without a significant increase in the length of the encoded proof. For, if certain mathematical proofs have no efficient encoding within a given proof system, then there is no hope of efficiently finding a proof of the theorem using any proof procedure based on that proof system.

Manuscript received March 1, 1984
In this paper we argue that most existing proof systems have certain weaknesses that prevent succinct encodings of certain types of mathematical arguments. We also point out that many of these popular proof systems require the formula to be presented in a normal form that is unnatural. This makes it difficult to translate intuitive heuristics that we normally use while proving theorems manually, into a proof procedure based on those proof systems. We propose a new method for theorem proving in the propositional calculus, that we call as "Natural Proof System," wherein we synthesize various attractive features from a variety of proof systems.

In the remainder of this section we develop the necessary terminology and notation. In Section 2 we comment on existing proof systems and their strengths and weaknesses. We deliberately go into some detail to point out the attractive features of these proof systems and the difficulties encountered by them. This is intended to motivate the techniques employed in the proposed proof system. In Section 3 we present an initial version of the proposed theorem proving technique, followed by a number of enhancements to the technique in Section 4. Finally, we conclude in Section 5 with remarks on transforming the proof system to a proof procedure and its implementation.

We call a propositional variable simply as a variable and distinguish it from a literal, which is a variable together with a parity- positive or negative. By a formula we mean a well formed propositional formula using any of the 16 binary connectives. The connectives of primary interest are: AND (∧), OR (∨), NOT (¬), IMPLIES (⇒), EQUIVALENCE (≡) and EXCLUSIVE-OR (+). The constants TRUE and FALSE will be represented by 1 and 0, respectively.

2. CURRENT PROOF SYSTEMS

2.1 Resolution

The elegance of the resolution proof system, originally suggested by Robinson ([12]), is, in part, due to its simplicity. It is based on a single rule of inference, called the resolution principle. This makes the conversion from a proof system to a proof procedure relatively easy, since the heuristics need not be complicated and messy. The only non-determinism that needs to be resolved is the decision of where to apply the rule of inference. Furthermore, this technique can be extended to the predicate calculus efficiently, through the process of unification.

In spite of its simplicity, the resolution proof system compares favorably in its power with most other existing proof system, i.e., the ability to encode succinct proofs. It has been shown [4] that resolution can polynomially simulate most other "reasonable" proof systems. (A proof system Δ₁ can polynomially simulate Δ₂ if for every proof in Δ₂ there is a proof of the same theorem in Δ₁ with at most a polynomial increase in the length of the proof.)

However, there are two main drawbacks of resolution. The requirement that the theorem be represented in conjunctive normal form (CNF) is both restrictive and unnatural. It prevents us from incorporating intuitive heuristics based on the structure of the original formula. The
second and more serious drawback is its inability to handle the logical equivalence connective well. Consider the following set of equations:

\[ a + b + c = 1 \]
\[ a + d + e = 0 \]
\[ b + d + f = 0 \]
\[ c + e + f = 0 \]

where, \( a, b, c, d, e \) and \( f \) are variables. Observe that every variable occurs twice in this set of equations. Hence, the sum of the left-hand sides of the equations, which is 0, is not equal to the sum of the right-hand sides. Consequently, if we encode each equation in CNF then the conjunction of the collection of clauses representing the above equations would be unsatisfiable. However, a resolution proof of this statement seems to be unnecessarily lengthy. The reason is that there is no short representation for a chain of Boolean sums in CNF. Using this fact, Tseitin [13] has demonstrated that theorems of the type illustrated in the above example, have no polynomially bounded proofs in certain restricted forms of resolution. While a similar result for unrestricted resolution has not been theoretically proved, there is ample evidence to believe that unrestricted resolution will fare no better.

2.2 Natural Deduction Systems (Gentzen Systems)

Natural deduction systems incorporate the deduction theorem as a rule of inference. One of the earliest natural deduction system was proposed by Gentzen [7]. Gentzen systems do not require the theorem to be represented in any normal form. They operate on each of the binary connectives in a natural way. Gentzen systems develop a tree in which the root is the theorem and the leaves are axioms. This makes the proof structure more comprehensible, and this could prove a useful feature in an interactive theorem prover. A unification based extension of Gentzen systems to the predicate calculus has been recently suggested by Abdali and Musser [1].

On the negative side, the variety of inference rules needed to handle all of the binary connectives makes it more difficult to design a deterministic proof procedure based on Gentzen systems. A more serious difficulty is the fact that a proof tree should really be viewed as a dag (directed acyclic graph) to avoid repeated proofs of the same sub-formulas. Finally, Gentzen systems are also and-or based, and hence can not handle the logical equivalence connective well.

2.3 Semantic Trees

Semantic trees were originally suggested in [8] (also by Davis and Putnam, see [3]). They represent a proof of the theorem based on a case analysis on the variables occurring in the theorem. A tree is developed in which the root is labelled with the theorem, the leaves are labelled with the constant 1, and the sons of a node labelled \( F \) are labelled \( F_1 \) and \( F_2 \) where, \( F_1 \) and \( F_2 \) are obtained by choosing a variable \( x \) in \( F \) and evaluating \( F \) with \( x = 0 \) and \( x = 1 \). An extension of this technique has been suggested by Monien and Speckenmyer [11] where, instead of a variable, a clause is used for splitting the original problem. Yet another generalization, suggested by Bibel [2], calls for splitting over a set of variables instead of a single variable.
The simplicity of this technique is appealing. Further, if the tree is viewed as a dag, this proof system is not known to be any less powerful than resolution.

However, a deterministic proof procedure based on this proof system is sensitive to the heuristics used in choosing the variables for instantiation. In addition, efficient extensions of this technique to the predicate calculus have not been investigated. Finally, this too is an and-or based proof system and our earlier comment on the logical equivalence connective prevails.

2.4 Rewrite Rules Technique

Recently, Hsiang [5,6] has reported a proof system based on rewrite rules. The negation of the theorem is specified as a conjunction of a set of equations. Each equation is a Boolean-sum of products equated to a constant. These equations are manipulated using the usual rules of equality to obtain a contradiction. While this is a more general version than that reported in [5,6] it is very much in the spirit of Hsiang’s technique. The most attractive feature of this proof system is the ability to substitute equals for equals (Hsiang does so in a limited way though)—a principle, fundamental to any manual theorem proving process. The substitution principle is precisely what is needed to efficiently handle the exclusive-or connective.

While the normal form used in this technique is tailored to support a chain of exclusive-or’s, it has its own weaknesses. For example, it does not support an efficient representation for a chain of disjunctions. Further, if the normal form is derived from a CNF representation of the formula, as the author suggests, the advantages of using the exclusive-or connective could well be completely lost. In fact, if the equations in Tseitin’s examples are converted from CNF to the normal form used here, the form of the equations changes so drastically that this proof system no longer admits short proofs for the same theorems when presented in this manner!

Another drawback of this proof system is that it relies on substitution alone for proving the theorem. Equations are expanded by unifying one term in one equation with another term in another equation, so that the terms may be substituted for each other. This expansion process often increases the length of the equations (not necessarily the number of terms in the equations). It appears that it would be preferable to replace the expansion process with some other technique.

2.5 Extension

Tseitin [13] introduced a technique called extension which allows the introduction of auxiliary variables to stand for arbitrary functions of existing variables. He demonstrated that by repeated applications of the principle of extension, one can efficiently manipulate complex sub-formulas by merely operating on the auxiliary variables that stand for the sub-formulas. Cook and Reckhow [4] have shown that extension is so powerful that it overshadows the inference rules of the proof system. That is, whenever two reasonable proof systems — with different inference rules—are enriched with the power of extension, the two resulting proof systems are equally powerful. Krishnamurthy [9] has shown that a variety of mathematical arguments can be succinctly encoded using the principle of extension. Thus, this simple feature of extension can significantly reduce proof lengths.
However, the difficulty is in a deterministic implementation of the principle of extension. How can a proof procedure recognize the need for auxiliary variables and determine how they should be defined? In the proof system suggested in the sequel we provide a technique for a deterministic implementation of a limited form of extension.

3. A NATURAL PROOF SYSTEM

In this section we present the main ideas of the proposed proof system, which we shall call NPS (Natural Proof System). It is a natural deduction system in the sense that (i) the proof preserves the structure of the theorem, and (ii) the deduction theorem is assumed. Theorems are proved by assuming the negation and arriving at a contradiction. Thus, to prove $\Phi \vdash F$ where $\Phi$ is a set of formulas and $F$ is a formula which is a logical consequence of $\Phi$, we assume $(\Phi \land \neg F)$ and derive a contradiction.

3.1 Equational Normal Form

The observations about normal forms made in the previous section indicate that while normal forms are desirable to avoid handling every binary connective and to give some structure to the formulas that are manipulated within the proof system, they should preserve the structure of the original formula. In particular, the distributive properties of the Boolean operators should not be used in deriving the normal form. With this in mind we develop a normal form purely for standardization of the formula, called equational normal form (ENF). For the sake of convenience, in an interactive implementation of a proof procedure, we might include the remaining connectives as well. However, for clarity, we limit ourselves here to $\lor$, $+$, and $\neg$.

ENF is a representation for not just a formula, but for a statement of the form "$F = X$" where $F$ is a formula and $X$ is either a literal or a Boolean constant. ENF is a set of equations of two types: $d$-type (disjunction) and $s$-type (summation). A $d$-type equation is of the form $A_1 \lor A_2 \lor \cdots \lor A_n = B$ where, $A_1, A_2, \cdots, A_n$ are literals and $B$ is either a literal or the constant 1. An $s$-type equation is of the form $A_1 + A_2 + \cdots + A_n = B$ where, $A_1, A_2, \cdots, A_n$ are positive literals and $B$ is a Boolean constant.

We sketch below a procedure for converting "$F = X$" into ENF and follow it up with an example. We begin with the expression tree (also know as the parse tree) for $F$. Recall that the leaves of the expression tree are labelled with variables and the internal nodes are labelled with Boolean operators. Using the usual rules of the Boolean operators, other than the distributive laws, we reduce all operators to $\neg, \lor$ and $\lor$. We also flatten the parse tree by using the associative properties of $\lor$ and $\lor$. Finally, convert the tree into a dag by identifying common sub-expressions. Now associate a new variable name with each $\lor$-node and with each $\lor$-node of the dag, $X$ being assigned to the root of the dag. Write an equation for each such node in the obvious way — a $d$-type equation for a node labelled with $\lor$ and an $s$-type equation for a node labelled with $\lor$. The collection of these equations is the ENF.

Example 1: Consider the formula

$$[(a \land \neg b \land \neg c) \land d) \lor ((c + a) \Rightarrow (b \equiv \neg c))] \land [(a \land \neg b) \Rightarrow (c + a)]$$
The corresponding equation tree and the modified dag are shown in Figure 1.

Figure 1

The ENF equivalent is shown below:

\[ \sim a \lor b \lor c \lor \sim d = Z_1 \]

\[ c + a = Z_2 \]

\[ b + c = Z_3 \]

\[ \sim Z_1 \lor \sim Z_2 \lor Z_3 = Z_4 \]

\[ Z_2 \lor \sim a \lor b = Z_5 \]

\[ \sim Z_4 \lor \sim Z_5 = \sim X \]

3.2 Simplification Rules for Equations

A set of equations can be simplified using the following rules. Recall that the order of the literals appearing on the left-hand sides of the equations is immaterial, since + and \lor are associative and commutative. Thus, the literals on the left-hand sides of the equations can be rearranged in any manner.
1. **Substitution**: We can substitute equals for equals. Thus from the two equations, \( A + X = 0 \) and \( A + Y = 0 \) we can obtain, by substituting the value of \( A \) from the first equation into the second equation, the equation \( X + Y = 0 \). Here \( X \) and \( Y \) can be an arbitrarily long chain of Boolean sums. Observe that \( X + Y = 0 \) will automatically be in the normal form. We can require that equations of the form \( A = B \) are eliminated by replacing all occurrences of \( A \) with \( B \) (or, vice versa). We must also point out that in the substitution process \( \neg 0 \) should be treated as a 1 and \( \neg 1 \) should be treated as a 0.

2. **Reduction of equations**: The following rule can be used to replace equations by simpler equations:

\[
\begin{align*}
R1: & \quad A \vee X = 0 \rightarrow A = 0 ; X = 0 ; \\
R2: & \quad A \vee X = A \rightarrow A = 1 ; X = 1 ;
\end{align*}
\]

Note that \( X \) can be a chain of disjunction.

3. **Reduction of formulas**: The following rules can be used to simplify the left-hand sides of equations to simpler forms:

\[
\begin{align*}
R3: & \quad X \vee X \rightarrow X ; \\
R4: & \quad 1 \vee X \rightarrow 1 ; \\
R5: & \quad 0 \vee X \rightarrow X ; \\
R6: & \quad \neg A + X \rightarrow 1 + A + X ; \\
R7: & \quad X + X \rightarrow 0 ; \\
R8: & \quad 0 + X \rightarrow X ;
\end{align*}
\]

### 3.3 The Proof System

The main core of the proof system described herein is based on a *generalization of the semantic tree approach*. We begin with a formula that we wish to prove to be a theorem or that it is inconsistent. To prove it to be a theorem, we transform the formula into a set of equations asserting that the formula has a value 0. Similarly, to prove it to be inconsistent, we transform the formula into a set of equations asserting that the value of the formula is 1. We then proceed to derive a contradiction. To this end, we construct a proof tree by repeated applications of the simplification process described above and the *splitting* process described below.

We first simplify the set of equations using the rules given in Section 3.2. At any step, if the simplification results in an inconsistent equation \( 0 = 1 \) or \( x = \neg x \) in the set, then the set
of equations is contradictory and we are done. Otherwise, we choose a sub-expression of the
left-hand side of one of the equations. The sub-expression may (and, often will) simply be a
literal. We then invoke the splitting rule and create two sub-problems: one in which the
sub-expression is equated to 0 and another in which it is equated to 1. These two sub-problems
are represented by two sets of equations obtained by adding each of the two equations men-
tioned above to the original set of equations. The original set of equations is contradictory if
and only if each of these new sets of equations is contradictory. We then recursively apply this
process. We illustrate the technique on an example in Figure 2.

Example: Theorem: \( b + c + (b \lor c) + (\neg b \lor \neg c) \)

\[ b + c + (b \lor c) + (\neg b \lor \neg c) = 1 \]

1. \( b \lor c = Z_1 \)
2. \( \neg b \lor \neg c = Z_2 \)
3. \( b + c + Z_1 + Z_2 = 0 \)

Split on \( b + Z_2 \)

\[ b + Z_2 = 0 \]

\[ b + Z_2 = 1 \]

4. \( b + Z_2 = 0 \)
5. \( \neg b \lor \neg c = b \) (subst. 4 in 2)
6. \( b = 1 \) (R2 on 5)
7. \( c = 0 \) (R2 on 5)
8. \( Z_1 = 1 \) (subst. 6,7 in 1)
9. \( Z_2 = 1 \) (subst. 6 in 4)
10. \( 1 = 0 \) (subst.6-9 in 3)

11. \( b + Z_2 = 1 \)
12. \( c + Z_1 = 1 \) (subst. 11 in 3)
13. \( b \lor c = \neg c \) (subst. 12 in 1)
14. \( c = 0 \) (R2 on 13)
15. \( b = 1 \) (R2 on 13)
16. \( Z_1 = 1 \) (subst. 14 in 12)
17. \( Z_2 = 1 \) (subst. 14,15 in 2)
18. \( 0 = 1 \) (subst. 15,17 in 11)

Figure 2

3.4 Completeness

The soundness of this proof system is evident, for, each of the simplification rules can be
verified and the soundness of the splitting rule is inherited from the semantic tree approach.
We show below that this proof system is also complete.

Theorem 1: NPS is complete for the propositional calculus.

Proof: Given a theorem \( T \) in the propositional calculus, we write the assertion "\( T = 0 \)" in
ENF. We need to show that by repeated applications of the simplification process and the split-
ting process we will necessarily arrive at a contradiction. If we restricted ourselves to the split-
ting process alone, we will eventually eliminate all the variables. If this does not result in a
contradiction along every path in the tree, then by the soundness of the proof system, we would have produced a satisfying truth assignment for the negation of the theorem. This would contradict the hypothesis that T is a theorem. Thus, all that remains to be shown is that the simplification process terminates. In particular, we need to show that the substitution process cannot be applied indefinitely. The number of new variables introduced in transforming T into ENF is bounded by the number of internal nodes in the expression tree for the formula T. This bounds the total number of variables and consequently, the number of possible equations. Hence, the simplification process must terminate. Q.E.D.

4. ENHANCEMENTS

Equations whose left-hand sides are disjunction of literals can be simplified only by splitting on a formula. We would like to avoid the splitting of a problem into subproblems as much as possible in order to avoid an exponential growth in the complexity of the proof. In this section, we discuss two enhancements to the proof system to handle disjunctions—generalized resolution and compaction. Generalized resolution is a generalization of standard resolution [12]; it can be used to simulate resolution proofs. Even in cases when resolution is not applicable, it generates information which might be useful in subsequently establishing a contradiction. Compaction allows us to recognize a set of equations whose left-hand sides are disjunctions and which have a succinct representation using the boolean sum notation. Later in the section, we discuss ways to identify similar nodes in the proof tree which can be used to avoid repeated independent derivations of identical sets of equations.

4.1 Generalized Resolution

Even though simplification is a powerful tool, it is not sufficient to efficiently handle certain types of arguments. For example, consider the following set of d-equations:

\[ \{ X \lor Z = 1; \ \neg X \lor Y = 1; \ \neg Y \lor Z = 1; \ \neg Z_1 \lor \neg Z_2 = \neg Z \} \]

It is easy to deduce from the above equations that \( Z = 1 \) thus making \( Z_1 = 1 \) and \( Z_2 = 1 \). However, we cannot make any substitutions, as the left-hand sides are disjunctions. On the other hand, if we used the semantic tree approach, we could branch on \( X \) and in each of the two sub-trees we would be able to derive \( Z_1 = 1 \) and \( Z_2 = 1 \). But then all subsequent arguments that use \( Z_1 = 1 \) and \( Z_2 = 1 \) would have to be duplicated in the two subtrees. Instead, we can derive the necessary conclusion without branching, using resolution. If we viewed each of the left-hand sides as a clause and the equations as asserting the conjunction of those clauses, then through two applications of the resolution principle, we can derive \( Z = 1 \). This suggests that a generalization of resolution applicable to a set of equations would be a useful tool.

*Theorem 2*: Let \( E \) be a set of equations including \( A \lor X = Z \) and \( \neg A \lor Y = Z' \). Let \( Z_1 \) and \( Z_2 \) be variables not occurring in \( E \). We can add \( Z \lor Z' = 1 \), \( Z_1 \lor Z_2 = 1 \), \( X + Z + 1 = Z_1 \), and \( Y + Z' + 1 = Z_2 \) without affecting the satisfiability of \( E \).
Proof: Since either A or \( \neg A \) is always true, at least one of Z and Z' must be true. Thus, we can conclude \( Z \lor Z' = 1 \). For the remaining 3 equations, consider two cases based on the value of A. If A is 0 then \( X = Z \) and \( Z_1 = 1 \). If A is 1 then \( Y = Z' \) and \( Z_2 = 1 \). In either case \( Z_1 \lor Z_2 = 1 \). Q.E.D.

The simplification rule corresponding to generalized resolution is:

\[
GR: \quad A \lor X = Z; \quad \neg A \lor Y = Z' \quad \rightarrow \quad A \lor X = Z; \quad \neg A \lor Y = Z';
\]

\[
Z \lor Z' = 1; \quad Z_1 \lor Z_2 = 1;
\]

\[
X + Z + 1 = Z_1; \quad Y + Z' + 1 = Z_2.
\]

When \( Z = Z' = 1 \), new variables \( Z_1 \) and \( Z_2 \) are not introduced because \( X = Z_1 \) and \( Y = Z_2 \) in that case; instead, of 4 additional equations, only one new equation, \( X \lor Y = 1 \) is added.

For example, consider the following set of equations:

1. \( P \lor Q = 1 \)
2. \( Q \lor R = 1 \)
3. \( R \lor W = 1 \)
4. \( \neg P \lor \neg R = 1 \)
5. \( \neg Q \lor \neg W = 1 \)
6. \( \neg Q \lor \neg R = 1 \)

The derivation is as follows:

7. \( \neg P \lor Q = 1 \) \hspace{1cm} GR (2, 4)
8. \( Q = 1 \) \hspace{1cm} GR (1, 7)

When 1 is substituted for Q in the above equations, equations 1, 2 and 7 are discarded; equations 5 and 6 simplify to:

5: \( \neg W = 1 \) \hspace{1cm} subs. (8)
6. \( \neg R = 1 \) \hspace{1cm} subs. (8)

Using them in 3 gives an inconsistent equation \( 0 = 1 \).
Theorem 3: NPS with GR can simulate the resolution proof system without an increase in complexity.

Proof: Consider a formula F in CNF which has a proof in the resolution proof system. From F, we get a set of equations such that there is an equation corresponding to each clause and nothing else. The equation corresponding to a clause C is \( C = 1 \). For every step in a proof of F in the resolution system in which a variable x is resolved, we take the equations corresponding to the two clauses involved in the resolution and apply the generalized resolution rule on them using x. Following the resolution proof tree which gives the empty clause, we would obtain \( 1 = 0 \), an inconsistent equation. Q.E.D.

4.2 Compaction

Consider the following set of equations:

\[
\neg A \lor B \lor C = Z; \quad A \lor \neg B \lor C = Z; \quad A \lor B \lor \neg C = Z; \quad \neg A \lor \neg B \lor \neg C = Z
\]

If these equations have to be transformed into a Boolean sum of products as required by Hsiang's technique, it will take a considerable effort to get to the equivalent form using +, which is:

\[
A + B + C = 0; \quad Z = 1
\]

Note that if \( Z = 0 \), then we immediately get an inconsistent equation from the original 4 equations. Using the above two equations, it is possible to substitute for any of the variables the remaining part of the equation; such a substitution was not possible from the original set of equations. Further, the new equations provide more insight into the semantics of the original set of equations than the original set does. It is often useful to recognize formulas with such structures and transform them to a more succinct form.

Given 3 variables a, b, c, 8 clauses can be expressed using them; so the possible number of subsets of such clauses is \( 2^8 \), which is 256. It turns out that there are only 5 relevant cases as other cases either simplify easily using the rules of inference discussed so far or can be obtained using symmetry (i.e., by renaming the variables) from the 5 cases. Without any loss of generality, we can assume that the clause \((a \lor b \lor c)\) is present in all 5 cases, as any clause can be transformed to this clause by symmetry. We can construct the following five subsets of equations and the corresponding reductions on these subsets are also given:

1. \( a \lor b \lor c = z \), which is kept as it is.

2. \( a \lor b \lor c = z \) and \( \neg a \lor \neg b \lor \neg c = z \).

This is a hard case; most hard tautologies, including Ramsey formulas [10], can be expressed using this case. Further, the satisfiability problem remains NP-complete even if all clauses in a formula are of these forms. These two equations are kept as they are; in addition, we can derive \( z = 1 \), as otherwise we get an inconsistent equation.
3. $a \lor b \lor c = z$ and $\neg a \lor \neg b \lor c = z$.

This case is equivalent to the following set of equations:

$\neg c \lor z_1 = 1; a + b = z_1; z = 1$.

4. $a \lor b \lor c = z; \neg a \lor \neg b \lor c = z; \neg \neg a \lor b \lor \neg c = z$.

$\rightarrow a + b + c = z_1; a \lor \neg b \lor \neg c = z_1; z = 1$.

5. And, finally,

$\rightarrow a \lor b \lor c = z; \neg a \lor \neg b \lor c = z; \neg \neg a \lor b \lor \neg c = z; a \lor \neg b \lor \neg c = z$.

$\rightarrow a + b + c = 1; z = 1$.

The utility of the above rules can be easily demonstrated on the CNF representation of formulas used by Tseitin [13]; for an instance, see example 3.6 in Bibel [2]. Using the last rule (case 5), we get formulas identical to those discussed in subsection 2.1 (except that different variable names are used in example 3.6). Adding those four equations immediately gives an inconsistent equation.

4.3 Identification

A proof tree in any proof system can in general have nodes labelled with identical formulas. Two different paths from the root—the formula being proved or disproved—may lead to identical subproblems (formulas) after splitting and simplification. We can avoid repeated independent derivations of sub-trees corresponding to identical nodes in a proof tree by maintaining a hash table for the formulas associated with those nodes of the proof tree that have been completely explored. Prior to exploring a new node, we would use the hash table to ensure that the formula associated with that node has not been encountered earlier.

In the proposed proof system, a node in a proof tree has a set of d-type and s-type equations associated with it. Before hashing the set, we canonicalize the set so as to obtain a unique representation for a set of equations using associative and commutative properties of $\land$, $\lor$, and $+$. One way to achieve this is by using an ordering on variable names and constants 0 and 1, parity, sorting arguments of $\lor$ and $+$ in ascending or descending order and deciding whether d-type equations follow s-type equations or vice versa. This would lead to a check for equality of two sets of equations using hashing.

Notice that the above scheme only checks whether two sets of equations (or two formulas) are identical. Since the inconsistency of a set of equations (formula) does not depend upon the particular names used for variables in the equations, two sets of equations that are identical up to variable renaming (in fact, up to literal renaming) characterize the same Boolean function. If
a node has a label E which is identical up to variable renaming to the label E' of a known dead node, then E is also a dead node. (A dead node is one whose proof tree has been completely developed.) A variation of the above canonicalization and hashing scheme can be used to check for this symmetry property of formulas (see Reference 9). This substantially reduces the complexity of proofs of a class of hard tautologies. For handling symmetry, the canonicalization of a set of equations (formula) also involves standardizing the variable names used in the set of equations. The hashing is done on the standardized canonicalized set of equations. Plaisted [personal communication] has implemented the above two schemes on a resolution based proof procedure and reported a substantial improvement in the performance of the theorem prover as a result of these schemes.

As should be evident, what is really needed is a way to identify whether a node labelled with a set E of equations contains as its subset (up to variable renaming), E', the set of equations associated with a known dead node. This is a hard problem and we do not know how to implement this kind of identification check yet (in resolution and semantic tree based proof systems, such a check would amount to doing subsumption check with symmetry). Even if we sacrifice symmetry, the subsumption check has been found hard to implement; incorporating symmetry makes it harder because standardization of variable names does not quite work.

5. CONCLUSION

We have proposed a proof system in which formulas at any intermediate stage are closely related to the structure of the original formula being proved or disproved. It extensively uses the powerful technique of extension for introducing new variables to stand for formulas in a proof, substitution and simplification derived from expressing formulas in Boolean sum notation and generalized semantic tree approach in which split could be done on a variable in the original formula or the one introduced via extension.

A crucial step involved in transforming a nondeterministic proof system to a deterministic proof procedure is the development of a set of heuristics that determine the order in which various rules of inference in a proof system are selected for possible application. Obviously, many proof procedures can be developed from a proof system by choosing different sets of heuristics. The challenge here is to come up with a proof procedure that makes the right choice in selecting the inference rules in most cases, and results in a minimal proof that could be obtained in a proof system. The overhead involved in implementing the heuristics and selecting it is usually justifiable only if it makes the right choices in most cases. One must then establish a completeness result for the proof procedure with respect to a set of heuristics, namely, that if a proof exists, the proof procedure finds it.

A proof procedure is considered good for a set of formulas if the complexity of a proof found by the proof procedure is at most a polynomial function of the complexity of a minimal proof in its proof system.
We believe that the heuristics are often dependent and derived from the application area. Although there may exist a small subset of general purpose heuristics based on the syntactic structure of formulas, the rules of inference and the semantics of various logical connectives, powerful heuristics often make use of the knowledge of the domain of the formulas. We believe that a "natural" proof procedure ought to provide facilities for a user to incrementally introduce heuristics and specify how the built-in and the user specified heuristics interact. This is a hard issue and falls outside the scope of this paper; however, we believe that the proposed proof system can incorporate some useful heuristics based on extension such as the user suggesting an arbitrary formula for a case analysis or a lemma to prove the original formula, etc. We plan to implement a proof procedure based on NPS using various heuristics and analyze their performance on different sets of formulas.

We believe that standard techniques suggested in Kowalski and Hayes [8] and Abdali and Musser [1] can be used to lift the proposed proof system to the first-order predicate calculus. However, an efficient generalization of the proposed proof system to the first-order predicate calculus such as that achieved by unification in the resolution procedure, requires further investigation.

6. REFERENCES


