AN IMPROVED THEOREM PROVER BY USING THE SEMANTICS OF STRUCTURE

by

DONALD GORDON JOHNSON II

B.S., Kansas State University, 1982

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by:

Major Professor

CHAPTER 1

Introduction

Automated theorem proving (ATP) is a central theme in Artificial Intelligence. It has direct application in the building of inference engines, explanation generation, program verification, and other applications requiring some form of mechanical reasoning.

There are some large problems currently in ATP. The first problem is that the search space tends to explode combinatorially. Alan Bundy in [2] said, "An unguided theorem prover can prove trivial theorems by sheer brute strength, but collapses under the weight of anything the least bit difficult." He found that current guidance heuristics cut the search space roughly in half (in his example, from 10^{21} elements to 10^{11} elements.) The other major problem is that the prover has no way of knowing that it may be searching down an infinitely long path. (These paths are known as "black-holes," because once the program goes into one it never comes out.)

Bibel, in [3], has suggested that these guidance heuristics are just substitutions for structural insight. In this thesis, a procedure is set forth that gives a machine some understanding of structure. Semantics of structure, as discussed in Chapter 3, appear at first to be part of syntax. This is because all of the symbols denoting structure have fixed interpretations. It is not these fixed interpretations, but the combination of them that gives the statement its structure and meaning. Each statement combines these fixed interpretations differently. By using the semantics of structure, a procedure may be generated that has a search space like that of a logician and, as we will see, is able to tell us when it is investigating a potentially infinite path. If it finds an end to this potentially infinite path, it can inform us of that as well. The use of this structural knowledge will allow the new procedure to reason out more difficult proofs, instead of out-running it with "brute strength."

This thesis contains a total of seven chapters. In the second chapter, the notations of the thesis are defined. Chapter 3 contains a discussion of semantics of structure. In Chapter 4, current proof procedures for both logician and machine are discussed. The new proof procedure is introduced, as well as contrasted with current procedures in Chapter 5. The remaining two chapters give details of the implementation of the new proof procedure and the conclusions of this thesis.

CHAPTER 2

Notation

The purpose of this chapter is to define briefly the three calculi with which this thesis is concerned. These claculi are: (1) notation of classical predicate calculus used by logicians; (2) the calculus used to communicate formulas to a machine; and (3) the notation used predominantly in this thesis. The second and the third are syntactic variants of the first.

In all of these calculi, the symbols used for predicates, constants, and variables will be the same.

Predicates --> P, Q, R, S ...
Constants --> a, b, c, d ...
Variables --> w, x, y, z ...

It is assumed that these symbols are countably infinite in number.

2.1 Classical Predicate Calculus

The classical predicate calculus referred to here is the same as that defined in the first section of the second chapter of [1]. This calculus uses the infix connectives $\sim, \wedge, \vee, \supset$, and $\equiv$ as symbols for the negation, conjunction, disjunction conditional and biconditional operations respectively. The quantifiers used are the existential ($\exists$) and the universal ($\forall$). Parentheses or brackets may be used as grouping symbols. In reading the examples, assume that P stands for "is a Person" and Q stands for "Loves," also "a" stands for John and "b" stands for Sue. Some examples of the classical predicate calculus and their meaning are given in Figure 1.

	Classical Predicate Calculus	English
1.	$\sim Pb$	Sue is not a Person
2.	Pa	John is a Person
3.	$\forall xPx$	For all x, x is a Person
4.	Qab	John Loves Sue
5.	$\forall x \exists y(Px \supset Qxy)$	For all x, there exists a y, such that if x is a person, then x Loves y. (Every person Loves someone)
6.	$\forall xyz[(Qxy \wedge Qyz) \supset Qxz]$	For all x, y, and z, if x Loves y and y Loves z then x Loves z. (Axiom of transitivity for the predicate Loves.)

Figure 1

These examples are all well-formed formulas (WFF's) of the calculus, but the truth of these statements is not explicitly stated in the calculus.

2.2 A Predicate Calculus for a Machine

This calculus differs only slightly from the one discussed in 2.1. The changes were made in order to be able to use a standard typeface and to aid the machine in parsing the calculus.

The typographical changes are the following ~, +, -, IF, IFF, $, * are used for $\sim, \wedge, \vee, \supset, \equiv, \forall, \exists$ respectively[1].

There were three changes made to the standard calculus to make it easier for the machine to parse the input. The first change is that the statement must be fully parenthesized, for example, the statement:

$$Pa \wedge Qa \wedge Ra$$

will be written as:

$$((Pa \wedge Qa) \wedge Ra) \quad \text{or} \quad (Pa \wedge (Qa \wedge Ra))$$

(whichever is more natural). The grouping here is not important. The second of these changes is that the operators are in prefix

---

1 The author chose $ as the universal quantifier. After all, "Money is the universal language."

form, not infix. There is no change in the meaning of the operators, just location. The last change is that quantifiers may quantify only one variable at a time. This difference is shown in Example 6 in Figure 2. Also note in Figure 2 that the predicates are parenthesized.

	Classical Predicate Calculus	Machine
1.	$\sim Pb$	(~ (P(b)))
2.	Pa	(P(a))
3.	$\forall x\ Px$	($x (P(x)))
4.	Qab	(Q(a b))
5.	$\forall x \exists y\ (Px \supset Qxy)$	($x (*y (IF (P(x)) (Q(x y)))))
6.	$\forall xyz\ ((Qxy \wedge Qyz) \supset Qxz)$	($x ($y ($z (IF (+ (Q(x y)) (Q(y z))) (Q(x z))))))

Figure 2

2.3 The Notation Used In This Thesis

The notation used in this thesis is a hybrid of the notations in Sections 2.2 and 2.3. It is the classical notation with the substitutions made to allow for a standard typeface. This means it is in infix form and not necessarily fully parenthesized. (See Figure 3)

	Classical Predicate Calculus	Thesis
1.	~Pb	~Pb
2.	Pa	Pa
3.	$\forall$ x Px	$x Px
4.	Qab	Qab
5.	$\forall x \exists y$ (Px $\supset$ Qxy)	$x *y (Px IF Qxy)
6.	$\forall$ xyz ((Qxy $\wedge$ Qyz) $\supset$ Qxz)	$xyz ((Qxy + Qyz) IF Qxz)

Figure 3

2.4 Final Comments on Notation

It is easy to see that these three calculi have equivalent expressive power. It is also quite easy to move from one notation to another. This was done intentionally to make it easy to take formulas from a book and present them to the machine for analysis without having to make an unnecessary transition to a normal form.

CHAPTER 3

Semantics

There are two types of semantics in predicate calculus. They are semantics of "interpretation" and "structure." The semantics of interpretation has to do with the meaning of predicates and determining the truth-value of atomic statements. This information is domain dependent and not of particular interest in this thesis. The semantics of structure is set forth below in Section 3.1.

The rest of this chapter is devoted to a discussion of the semantics of structure, their relation to proof procedures, and the consideration of semantics of structure in the choice of syntactic constructs. A brief discussion of normal forms is also included.

3.1 Semantics of Structure

The semantics of structure provide the meanings of the elements of the calculus that allow one to create compound statements. In the calculi of Chapter 2, the structural elements were the five connectives and two quantifiers, +, - ,IF, IFF, ~, *, and $. Semantics of structure is also involved with the relationship of the structure of a statement and its meaning with respect to the meaning of its substatements.

These, however, are not the only structural elements one could use in creating a calculus. Other first order structural elements would include functions and equality.

Some notations show their structural features better than those we have seen so far. These include the graphical notation of Frege and the box notation of Peirce.

All these notations can to express the concepts in a first order logic. They are all, in this sense, equivalent. This is to say, these notations can have the same semantics of interpretation.

The semantics of the structure of these notations are, however, different[2]. This difference allows for a shift of emphasis in the meaning of statements of the calculus. This difference may be used to make it easier to read or easier to prove theorems. An example of this in English is the structural change in a sentence when the voice is changed from active to passive. For example:

Mary ate the cake.

implies that Mary performed the action of eating the cake, as opposed to:

The cake was eaten by Mary.

which implies that the cake was eaten and that it was Mary who ate it. In the first sentence, the emphasis is that Mary was doing something and in the second, the emphasis is that the cake was having something done to it.

The next section looks at how the semantics of structure is related to proof procedures.

---

2 Note: All of the notations presented in this thesis have the same semantics of structure.

3.2 The Relation of Semantics of Structure to Proof Procedures

It is rare for a logician to discuss semantics of structure. Most of the time, a logician will simply define a notation which is convenient to his purposes. Semantics of structure is often viewed as part of syntax. This is because the semantics of the structure of syntax has a fixed interpretation, for example, the "or" symbol always means disjunction. The part of semantics of structure that is interesting is not the fixed interpretation of syntax, but the combination of these fixed interpretations in a theorem and how this affects proof procedures.

In all proof procedures, the semantics of structure must be used. Some of this information is used explicitly, such as statement-substatement relationships. Some information is used implicitly in the form of rules which guard the soundness of the algorithm.

The role of semantics of structure can be more clearly identified in a specific example. The example chosen here is a method of building consistency trees (a form of proof by contradiction) from [1]. The proof procedure is laid out in some detail in the three parts of Figure 4.

In this procedure, considerations for the semantics of structure are made in two ways. The first involves the statement-substatement relationships. These relations are the connective rules of Figure 4C and General Rule B in Figure 4A. These rules guarantee that statements are properly decomposed into their substatements, and the substatements are added to all branches of the proof to which it is appropriate.

The notion of checked and unchecked statements keeps one from decomposing the same statement over and over. It should be noted that the universally quantified statement is never checked, meaning the procedure may run forever. This notion of checked and unchecked statements is also tied to the other way in which the meaning of the symbols is considered. This is quantifier scope.

A. A branch must be closed as soon as both a statement and its negation appear on it.

B. Any statement that is not universally quantified must be checked off when decomposed by the appropriate rule, and the statements into which it decomposes must be added to the bottom of all the branches which may be traced back to it.

C. When decomposing an (unnegated) existential quantification or negated universal one, "instantiate" it on each open branch through the statement by means of the alphabetically earliest term foreign to that branch (or, if preferred, by means of the alphabetically earliest term foreign to every branch on which it is to be "instantiated").

D. When decomposing an (unnegated) universal quantification or negated existential one, "instantiate" it on each open branch through the statement by means of all (and only) the terms that have occurred so far along that branch, omitting the "instances" of the statement that already appear on the branch. If no term has yet occurred along a given branch, choose a new one.

Figure 4A
General Rules for the LeBlanc and Wisdom Procedure

1. Write down the negation of the theorem. (This is the "Root" branch.)

2. Is there an open branch on which all the statements are checked?

   No: Go to step 3.
   Yes: The theorem is unprovable.

3. Is there an unchecked statement on an open branch to which a connective rule applies?

   No: Go to step 4.
   Yes: Apply the relevant rule to it.
   Go to step 2.

4. Is there an unchecked statement on an open branch of the sort *x A or ~ $x A?

   No: Go to step 5.
   Yes: Apply the relevant rule to it.
   Go to step 2.

5. Are there any statements on an open branch of the sort $x A or ~ *x A?

   No: The theorem is provable.
   Yes: Apply the relevant rule to every one of them.

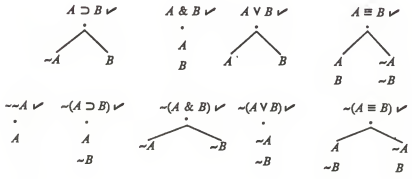
   5.a. Is there an open branch to which no new line was added?

      No: Go to step 3.
      Yes: The theorem is unprovable.

Figure 4B
Proof Procedure from LeBlanc and Wisdom

These rules are just for the decomposition of statements and should be applied using Rule B of the general rules.

The rules are best shown graphically:



INTERPRETATION:

* The conjunction rule should be read as: add both conjuncts to the end of every open branch which it is on.

* The conditional as: fork the end of each open branch and add the negation of the antecedent to one side and the consequence to the others.

Figure 4C
Connective Rules

The considerations made for quantifier scope are quite important. The violation of quantifier scoping rules is a common fallacy in predicate calculus. The fallacy can cause not only the generation of incorrect proofs of true statements, but also seemingly good proofs of false statements.

The considerations for scope made in this proof procedure are in both rules and the structure of the procedure itself. The rules involved are C and D from Figure 4A. These state that existential quantifiers may only be instantiated for new constants on a given branch and that universals must be instantiated for all constants on a branch for which the universal has yet to be instantiated. The proof procedure insures that the outermost quantifiers are instantiated before the inner ones.

These two considerations insure that when an existential quantifier is inside the scope of a universal, as in Statement 5 of Figure 1, the quantifiers are not instantiated to the same constant in any single instance of the statement. These considerations also insure that statements are properly quantified before they are broken into substatements.

Many of these considerations will become more important in later chapters in the discussion of decidability, completeness, and soundness.

3.3 Semantics and the Choice of Syntax

As shown in the previous section, a proof procedure is tied both implicitly and explicitly to the structural semantics of a calculus. In Section 3.1 it was shown how different structural elements can shift the characteristics of a

calculus. It is important for the purpose of this thesis to choose a syntax whose structural semantics facilitate the machine generation of proofs.

There are four basic types of structural elements[3] of a calculus. They are connectives, quantifiers, equalities, and functions. A minimal calculus usually contains two connectives (negation and conjunction), and one quantifier (the universal). To this minimal system, many add the other connectives (such as the ones in Chapter 2), and the existential quantifier. This is done to aid the logician in the expression of complex statements.

The semantics of structure of these other elements is very similar to the three elements of the minimal system above. Because of this, a procedure is not complicated significantly by adding these elements. Note that by removing these extra connectives, the routine outlined in Figure 4 would not be simplified. There would be fewer rules to use from Figure 4C. These elements have been added to give a richer set of connectives to a logician.

The equalities are notions such as equality, greater than, less than, and so on. These elements, unlike the extra connectives, have semantics that are not similar to anything in the minimal system. They are expressed in the minimal system as special predicates and some statements (axioms) that express the concepts of transitivity, reflexivity,

---

3 For ease of parsing and simplicity, only written notations are considered here for machine use. An interface to graphical notions could be applied.

associativity, and so on for each predicate. By not including the equalities, the logician may lose a little ease of expression with these concepts, but not the ability to express them. This also allows a procedure to concern itself with one fewer type of structural semantics.

The same argument may be made for functions, except that the equivalence to the minimal system is less obvious. For this equivalence, see the work of Skolem. There is also another argument to consider with functionals.

This argument is presented by Bundy in [2]. It says basically that for a function (f(x) = y), the existence of a suitable y for a given x and the uniqueness of that y must be guaranteed. This further complicates the consideration of the structural semantics of functionals.

3.4 Comments on Normal Forms[4]

Traditionally, automated theorem provers have used conjunctive normal form for input. This choice was made in order to make it easy for the machine to read input expressions.

Conjunctive normal form does have some quite distinct disadvantages. One major problem is that the move to conjunctive normal form involves skolemization to eliminate exsistential quantifier. Bibel in [3] has shown that skolemizing can increase the length of a formula quadratically. He also points out that this increase is

---

4 A normal form, in the words of John Robinson [8], is just "a standard way of saying things," or a way of saying what something says in a fixed format.

significant in all but trivial theorems, and has a serious effect by increasing the time and space required to process the theorem.

Bibel also points out that conjunctive normal form (CNF) is not good for interactive theorem provers because it is hard to read [3]. Bundy shows many places in [2], how semantic information can be used to limit the search space. In the move to conjunctive normal form, it is necessary to use the semantics of structure in order to remove the natural structure of the theorem. This move to conjunctive normal form causes quantifiers to be moved outward. This is called prenex normal form (all the quantifiers in front). Bibel demonstrates that moving quantifiers inward into the formula, instead of outward as in CNF, leads not only to a quicker proof, but also a simpler one. He has proven in [9] that the proof can never be made more complex by this inward move. This process of moving quantifiers inward he calls antiprenexing. Prenexing is done by using a set of structural transformation equivalences. Since these are equivalences, they may be reversed and the same set is used for antiprenexing.

From this we can infer that the more highly structured the theorem, the faster a proof may be found, and the simpler it will be.

Another blow against normal forms comes from cognitive psychology. This is the fact that humans tend naturally to group related items by relevant factors. This grouping would tend to make proofs shorter and simpler again because related items are found together. The move to a normal form, again destroys this natural grouping.

Normal forms, although they are easier for a machine to read,

have some very serious drawbacks. These drawbacks warrant not using them.

In Chapter 4 current proof procedures are examined. Although not mentioned explicitly, the reader should notice the role of semantics in the procedures examined.

CHAPTER 4

Proof Procedures

In 1900, at the International Congress of Mathematicians in Paris, David Hilbert presented a list of unsolved problems in mathematics. The 23rd problem on this list was to discover a procedure that will determine whether any arbitrary statement of the classical predicate calculus is true or false.

The purpose of this chapter is to discuss proof procedures, for both humans and machines. In the first section, a basis for proof procedures is presented that is formed mainly from definitions of terms and the works of Church and others.

4.1 Soundness, Completeness and Decidability

If a theorem is valid, a "complete" proof procedure will produce a proof for it. If a proof procedure is "sound", it will produce proofs only for valid theorems. All of the procedures presented here have these properties. These properties are of great importance since they establish the credibility and reliability of the procedures. For further information about the soundness and completeness of a particular proof procedure, refer to the references for that procedure .

Whether or not a proof procedure for the predicate calculus may be "decidable" is Hilbert's 23rd problem. Many people worked on this problem, most notably Church, Skolem, and Peter. Skolem tried to produce a decision procedure through the use of a series of structural transformations. (one of which was seen in Chapter 3 as Skolemization). His attempt failed. Church in his 1936 thesis conjectured that there can

not be a mechanical decision procedure for predicate calculus. This has never been proved. Peter, along with some other Hungarian mathematicians, have objected to Church's thesis. Probably the greatest argument against Church is that made by Kalmar [6]. Kalmar has proved that if Church's thesis is true, then there exists a simple proposition which we know is true, but cannot prove in any way. This issue is a difficult one and has been a sore point with theorem provers for many years.

The proof procedures presented in this paper, while they are sound and complete, are not decidable. They are called semidecidable. Semidecidable procedures give correct proof for valid theorems and can also provide counter-examples for invalid ones when it stops. The problem is that many invalid theorems cause non-terminations in proof procedures. This phenomenon of non-termination is often referred to as a black-hole. It is called a black-hole because, when the procedure falls in, it never comes out again. It is caught in an endless cycle of some sort.

4.2 Human Proof Procedures

The proof procedure presented here is that described in Chapter 3 (Figure 4). This procedure is from LeBlanc and Wisdom [1]. It's roots, as for most human proof procedures, is semantic tableau from Beth [7]. Additional discussion may be found in Jeffries [6].

LeBlanc and Wisdom define four categories into which a statement's proof tree may fall. A discussion of each category follows.

4.2.1 CATEGORY 1: A Closed Tree.

The first case occurs when all branches of a proof tree contain a closure, i.e., a statement and its negation. When this happens, the theorem is valid. It has been shown, by a closure on each branch, that there is no possible instance for which the statement's negation can be true. This is checked in Figure 4B, Step 5.

4.2.2 CATEGORY 2: An Open Branch With All of the Statements Checked.

This case represents an instance in which the statement's negation is consistent, meaning the statement is not valid. A counter-example may be generated from the truth values of the atomic statements on the branch. This is checked in Figure 4B, Step 2.

4.2.3 CATEGORY 3: New Instances Could be Added, but It Would Be Pointless.

This case is checked in Figure 4B, Step 5A. Category 2 occurs when the only unchecked statements are universally quantified statements that produce no new constants when instantiated, for example, the statement $x Px. The point is that no information is gained by further instantiations. In this case, an infinitely long open branch has been found. As in Case 2, the theorem is invalid and a counter example may be generated from the branch.

4.2.4 CATEGORY 4: Black-Holes

Category 4 occurs when a statement of the general form ($x ... (*y ...)) is found that doesn't produce a closure. When this happens the procedure cycles endlessly through Steps 4 and 5, generating new constants. To complicate matters, Steps 3 and 2 may be used many times in each cycle. Another possibility is that the cycles between Steps 4 and 5 may be nested as well. These complications and the fact that one doesn't know if the statement will produce a closure makes it very difficult, in general, to decide if the procedure runs forever or just for a very a long time.

Most logicians eventually recognize that they are repeating themselves. Due to the difference in memory structures and means of recognition, it is highly unlikely, at this time, that a machine do this. The method simply does not tell you that you are cycling within a black-hole.

4.3 Machine Proof Procedure

Resolution, which was chosen primarily for its simplicity, has dominated automated theorem proving since 1969. Since Robinson's early work [11] many enhancements have been made. These improvements were contributed mainly by Bledsoe [12], Boyer and Moore [13], Loveland [14], and Wos [15].

The simplicity of resolution is that it is based on a single rule of inference, the cut rule. The cut rule is also known as the complex dilemma and is a generalization of modus ponens. An example of the cut rule is: if given the formulas ~A v B and A v C, one may infer from these and the cut rule the formula B v C. The idea of resolution is to end up with an empty resolvent. This means that none of the original

alternatives could possibly be true, and again we have a form of proof by contradiction.

Resolution, however, has a few problems. The first is the input must be in conjunctive normal form. This, as shown in Chapter 3, destroys much useful information, besides making the theorem harder to read. Also, since conjunctive normal form requires skolemization, there are "skolem" functions or predicates now in the theorem that have a somewhat unclear meaning.

The second problem with the resolution technique is that the search space grows exponentially. This growth occurs because the resolvent clause is added to the set of clauses and then resolution is performed again. An excellent example of this is give by Bundy in Section 7.2 of [2]. In this example, Bundy gives a problem from group theory. The problem requires a minimum of 42 resolutions to solve and has an average fanout of three. This means the search space contains about 1021 elements. By using a heuristic technique called paramodulation, the search space was curbed to about 1011 elements. What this tells us is that we cannot overpower problems.

The third major problem is that resolution falls into black-holes, just as the LeBlanc and Wisdom procedure did (see Figure 5). Resolution also does not know when or if it has fallen into a black-hole.

4.4 The Need for New Proof Procedures

As shown above, when a proof procedure drops into a black-hole, there is no warning and the only way to stop is for a human to realize something has gone wrong. In the case of a

manual procedure, it is the realization of, "Haven't I just done this?" and then looking for the repeating pattern. In the case of an automated procedure, it is a guessing game, "Is the tree infinite or just very large?" This analysis is hindered because that the search space is very large and we can't overpower it, even with heuristics.

There is a need for a new type of automated proof procedure. The procedure should use the structure of the theorem to limit the search space, much as the human procedures do. This new procedure shold be able to recognize and "build a fence" around potential black-holes. This would allow it to give information to its user, which in turn would allow the user to make a more informed decision. The procedure should inform the user when it has the potential of falling into a black-hole.

When the procedure is finished working around the black-hole (found a closure) and no longer has the possibility of running infinitely, it should inform the user of this.

It should be noted that the tighter this fence is around the black-hole, the more accurate the information to the user. A tighter fence also means the less likely it is that the procedure will venture inside the fence.

These problems are, as discussed in Chapter 5, related to the semantics of structure.

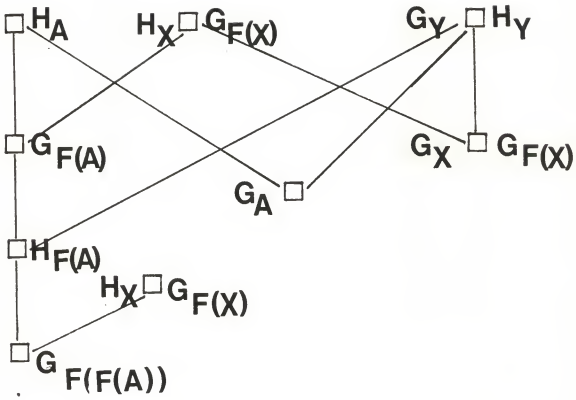


Figure 5

Infinite Clause Generation
In Resolution
(Using Robinson's Notation)

CHAPTER 5

A New Proof Procedure

In this chapter a new proof procedure is presented. Although it is intended for machine use, it is presented at the level of the procedure in Figure 4. This is to avoid getting bogged down in the details of a particular implementation.

In the first section the new proof procedure and some definitions are set forth. The second section contains a discussion aimed at providing the reader with an understanding of the new procedure. The final section is a comparison of this new procedure with ones currently in use.

5.1 The Procedure

The proof procedure is presented in Figure 6. Before looking at the procedure the reader should become aware of structures used in it. They are as follows:

STACK - This is a list of statements to be processed by the procedure. It may be ordered to increase efficiency.

SCOPE LIST - This is an ordered list that is attached or associated with a statement. Its purpose is to keep track of the quantifiers that scope this statement.

BRANCH - This is the branch of the proof tree currently being investigated. It may be represented as a simple list of the statements on the branch.

OPEN - This is a return value of the routine meaning an open branch has been found.

The sub-procedures PROVE, COMBINE, UNIFY, and COLLECT are functions. These functions all return a set of bindings. A binding tells us what a quantifier should be instantiated to, in order to produce closure in part of the tree. The overall plan is that the procedure in Figures 6A-6D directs the unifier (UNIFY) to produce all of the closures possible between atomic statements. Then, COLLECT and COMBINE take these closures for atomic statements and produce closures for parts of the tree. A closure for the whole tree means that proof by contradiction can be made by instantiating the quantifiers for the values in the closure.

1. Push the negation of the theorem onto the STACK.(Be sure its SCOPE LIST is empty).

2. Start with BRANCH empty.

3. PROVE (STACK, BRANCH)

Figure 6A

The Augmentation Step of the Proof Procedure

```
PROVE (STACK, BRANCH):

   1. Is the stack empty?
         NO:  Pop a statement off STACK
              Add the statement to the branch.
              GO TO STEP 2
         YES: RETURN "OPEN"

   2. Is the statement atomic?
         NO:  GO TO STEP 3
         YES: RETURN COLLECT (UNIFY (STATEMENT,BRANCH),
                              PROVE (STACK, BRANCH))

   3. Is the statement quantified?
         NO:  GO TO STEP 4
         YES: Add the quantifier to the statement's
                   SCOPE LIST
              RETURN PROVE (STACK, BRANCH)

   4. Is the statement's main connective a branching one?
      (See Figure 6D)
         NO:  Apply the relevant non-branching rule
              RETURN PROVE (STACK, BRANCH)
         YES: Apply the relevant branching rule, producing
              two stacks.
              RETURN COMBINE (PROVE(STACK1, BRANCH),
                              PROVE(STACK2, BRANCH))
```

Figure 6B
New Proof Procedure

COLLECT is literally a union operation with "OPEN" representing the empty set. What COLLECT does is collect all of the sets of bindings produced by a single branch.

COMBINE, on the other hand, takes the bindings from two branches and combines them to form a single set of bindings to be returned.

UNIFY is the routine that produces these sets of bindings. A binding represents the instantiations required to make the current (atomic) STATEMENT form a closure with another statement on this BRANCH. The set of bindings produced by UNIFY is the set of all such instantiations that form closure. If this set is empty, "OPEN" is returned.

Figure 6C
Explanation of Auxiliary Procedures

NON-BRANCHING RULES

STATEMENT	RULE
~ ~ A	add A to the STACK
A + B	add A and B to the STACK
~(A - B)	add ~A and ~B to the STACK
~(A IF B)	add A and ~B to the STACK

BRANCHING RULES

A - B	add A to stack, producing STACK 1 add B to stack, producing STACK 2
A IF B	add ~A to stack, producing STACK 1 add B to stack, producing STACK 2
A IFF B	add A and B to stack, producing STACK 1 add ~A and ~B to stack, producing STACK 2
~(A + B)	add ~A to stack, producing STACK 1 add ~B to stack, producing STACK 2
~(A IFF B)	add ~A and B to stack, producing STACK 1 add A and ~B to stack, producing STACK 2

Figure 6D
Connective Rules

5.2 A Closer View

At a high level, this new proof procedure looks almost like the procedure of Figure 4. The difference between the two are described below. From this high level, one of the most outstanding differences is the way in which quantifiers are handled.

5.2.1 The Handling of Quantifiers

Quantifiers, in this procedure, are never actually instantiated. The reason for this is that black-holes are caused by instantiation cycles. By not instantiating, the proof procedure is able to work with patterns of statements and control information about the variables of the pattern. The semantics of structure provide the control information in the form of Scope Lists. By using this control information and the pattern of a statement, UNIFY can identify the instantiations that produce closures with other statements (patterns) on this BRANCH.

5.2.2 An "Intelligent" Unifier

Once these patterns of statements and scope lists have been created, a unifier is needed that is capable of handling this information. The unifier must be able to utilize this information to tell when two instantiations of a statement are needed. The unifier accepts two patterns and their related scope lists. The unifier then attempts to find a binding that when instantiated makes the patterns literally equal. If the unifier fails, the patterns may not be unified and the "OPEN" value is returned. The job of UNIFY in this procedure is to call the unifier to find a set of closures between the current

(atomic) STATEMENT and each of the (atomic) statements on the current BRANCH.

The first thing done by the unifier is to be sure the predicates are the same. This is done by an equality operation, and if it fails, so does the unification. The next problem facing the unifier is to unify the variables and constants to produce the binding. The rule used to do this is: A variable may be bound to any constant that is not inside its scope (i.e., following it on the scope list). The exception to this rule is when the variable and constant are from different statements (meaning the two statements being unified are substatements of the same superstatement). When this happens, the statements together may be bound, but it must be noted that the variable is from the second instantiation of the pattern. When the case of two instantiations occurs, the unification of variables and constants must start again, only this time the unifier knows there are two instantiations.

As the unification proceeds the scoping rules become stricter. This is the result of the binding process. When, for example, the variable X is bound to the constant A, this means that X is inside the scope of A or, in terms of the proof tree, X must be instantiated after A is instantiated. This is not very constraining in itself, but constraint comes from the fact that everything that was inside X is now forced to be inside everything which is outside of A. If, at any time, a variable cannot be bound to a constant by these rules, or two constants are not equal, the unification fails.

The point here is that an "intelligent" unifier using the semantical notion of scope is capable of making all possible closures without having to instantiate any of the statements involved.

5.2.3 The need for Collect and Combine

By not instantiating quantifiers, a problem with soundness and completeness has been caused. The soundness problem is, in the procedure described so far, that there is no control to insure that a quantified variable on different branches is instantiated to the same constant. The completeness problem occurs when a universally quantified variable on different branches is instantiated to the same constant. Then, when universally quantified "or" branches (the branching rules of Figure 6D) require more than one instantiation, a tree is formed. The outside branches have been looked at but the interior ones have not. These problems both occur at the place where two branches meet. COMBINE is the routine that handles these problems where they occur.

COLLECT is needed to put together two sets of bindings on the same branch into a single set of bindings. COMBINE and COLLECT are very closely related to the operations of intersection and union. This duality is directly related to the duality of "and" and "or."

5.2.4 The COLLECT Routine

The COLLECT routine is literally a union operation, as stated in Figure 6C. The union operation is sufficient for this presentation, but some improvement could be made. The most obvious is to remove specific bindings when a more general one exists. This removes redundancy by taking the most general binding.

5.2.5 The COMBINE Routine

The COMBINE routine is related to the intersection operation. COMBINE is more complicated than intersection. COMBINE's intersection process must consider the case when the binding from one set is more general than the one from the other. In this case, the binding resulting from "intersection" reflects the most specific binding that can be made from the two. This routine must also be mindful of multiple instantiations, scoping rules, and the particular scope lists, as was our unifier. This "intersection" step in COMBINE restores the soundness property to our procedure. It insures that the resultant bindings produce closure for both branches that come from the branching connective.

The second step in COMBINE is to insure completeness. The loss of completeness come when a branching connective is instantiated multiple times. The routine PROVE has only looked (through patterns) at the possible closures for instantiations of only the left hand branch and only the right hand branch, the exterior branches of the tree. In this case, there are also interior branches that have not been looked at yet. In order to make the procedure complete, we can instantiate the branching statement for the values in the binding and add them to the STACK and ask PROVE to tell us if it closes. But, we must also add the pattern for the branching statement. The reason for this is that the combination of left and right hand sides of a branching statement may create new closures. This makes the procedure complete, but, unfortunately, also makes it semidecidable because the pattern may be copied from the second pattern of the statement to make a third, and so on.

Improvements can be made to eliminate some of the need for

this second step. If one of the bindings in the first step doesn't require any instantiations (just that variables bound to the same thing), then the second step is not needed. It can already be deduced that the tree closes. Other improvements can be made, but are not necessary.

This procedure has limited the semidecidability to just branching connectives with multiple instantiations. This allows the procedure to inform the user when it is and is not in danger of hitting black-hole areas. Processing in potential black-hole areas is a last resort, to be done only when COMBINEing and COLLECTing other bindings fail to produce a closure for the proof tree.

5.3 A Comparison with Existing Proof Procedures

The new procedure presented in this chapter is, as are the procedures from Chapter 4, sound, complete, and semidecidable. As we have seen, the new procedure achieves this in a manner different from the other procedures. Let us see how this different approach affects some other properties of this new procedure.

5.3.1 Search Space

The search space of resolution was found to be exploding combinatorially. The LeBlanc and Wisdom procedure doesn't seem to have this problem. The new procedure has an even better search space than that of LeBlanc and Wisdom because it uses patterns of statements instead of actual instantiations of them. This means that the new procedure examines all the possible instantiations at once, instead of one at a time. This means the new procedure can find multiple bindings at

once (possibly infinite if a binding is found only between variables). This means the new proof procedure's search space is smaller than that of LeBlanc and Wisdom and very much smaller than that of Resolution.

5.3.2 Can a Fence Be Built Around Black-Holes?

Previously, no proof procedure has been able to establish a reasonable fence around black-holes and inform the user when it is necessary to venture inside the fence. As we have seen, this new procedure, because of the way soundness and completeness are checked, can build a reasonable fence around black-holes.

The problem of not being able to spot black-holes originated in the soundness constraints of human proof procedures. The problem was then, unknowingly, propagated into machine proof procedures.

5.3.3 Complexity

Resolution is the simplest procedure; it has only a single rule of inference. LeBlanc and Wisdom's procedure is next. The new procedure seems to be definitely the most complex, which means it will be harder to implement.

The reason for the difference in complexity is the amount of explicit use of semantics of structure. This increased use of semantics gives a smaller search space. Bibel in [3], also notes that routines with smaller search spaces are more complex.

CHAPTER 6

Implementation

The realization and implementation of this new proof procedure is the result of four and a half years of study and experimentation. In this chapter the motivation, details of implementation, and some results of this procedure are described.

6.1 Motivation

In the Spring of 1979, I was approached by two professors, George Georgacarakos and William Schank-Hamlin, with the problem of building consistency trees for first order predicate calculus statements. A consistency tree generator can be used as a theorem prover by negating the theorem. Professors Georgacarakos and Schank-Hamlin needed a consistency tree generator to show some theorems consistent which were too large to be done by hand in any reasonable length of time.

6.2 Details of Implementation

In order to understand the problem better, the initial task of writing a program to build trees for propositional calculus was performed. This program builds trees in the same way logicians do. In the end, this initial program resembled the early prover of Change & Lee. It is a decision procedure and runs in such a negligible amount of time that its speed was never benchmarked. This first program took approximately eight months to write.

The next step was to add the existential and universal quantifiers to the proof procedures. Again, I tried to look at how logicians build trees that involve quantifiers and to emulate them. This version of the program looks somewhat like the procedure in Figure 4. This is probably due to the fact that this was where I learned how to build consistency trees. The new procedure is different from that in figure 4, because I noticed that while that procedure works that is not in fact how I build consistency trees.

I also noticed that I seem to have very little problem with "black-holes." I cut them off by the realization of "having done this before." I found that I would look ahead at the ramifications of instantiating a variable, before I actually created a new instance of a statement. This helps greatly in controlling the proof.

This new proof procedure is an attempt to document the manner in which how I prove theorems. Some things were done slightly differently because of the knowledge that the user of the new procedure would be a machine. This is because the machine is much better at clerical things than most humans.

6.3 Results

Test problems, for the proof procedure in this thesis and previous "experimental" versions, were derived from many sources. The author generated approximately forty formulas for testing specific mechanisms in the implemented versions. The next set of formulas is all the formulas found in the second chapter of LeBlanc and Wisdom. About twenty formulas were taken from the twelfth chapter of Church's "Introduction to Mathematical Reasoning." Some more difficult formulas were provided by G. N. Geosgacarakos and the problem known as

"Challenge Problem 1" (CP1), presented to the Fourth Workshop on Automated Deduction by Dr. Peter Andrews was also used.

This proof procedure was implemented at Kansas State University on the equivalent of an IBM 4341, running LISP atop VM/370. The program ran all formulas correctly except the last class of formulas, in between zero to twenty-three hundredths of a cpu seconds. For some reason, CP1 and the other "difficult" problems ran much more slowly (approximately 43 cpu seconds). On the surface, CP1 does not appear to be two orders of magnitude larger or more complex than any of the other problems, and it is not. The problem arises in this proof procedure, because the procedure attempts to find all the possible bindings that will yield a proof. CP1 has more than 16 million of these bindings. This is a problem with this procedure. All other procedures known to this author find only one answer, as opposed to all possible answers.

CHAPTER 7

Conclusions

The use of semantics of structure allows a proof procedure to be created with an improved search space and the capability to "fence" black-holes. This procedure was arrived at by examining how logicians really prove theorems. First, the logician looks down a branch or two and finds closures. Then he uses the procedure in Figure 4 to verify that one of these closures is correct. By examining what rules the logician applied when he looked ahead down that branch led to the discovery and explicit use of semantics of structure in the new proof procedure.

7.1 Further Research in ATP

Some other semantic techniques have just recently been applied to resolution theorem provers. With some work these could be useful additions to the procedure above. These techniques include Bibel's work with connection graphs in [3] and Walther's work with the typing of variables and constants in [10]. Both of these have shown advances in ATP. Possibly, by combining one of these with the structural semantics presented here, a decision procedure may be found.

Work should also be done toward the goal of making this procedure seek out only one answer, instead of many. This may be done through the use of advanced control structures. These structures should use the semantical information of structure of the formula for a proof (possibly optimal). The notation is akin to the view in software engineering that the structure of the inputs and outputs determine the structure of the program.

7.2 Further Research Involving the Application of Structural Semantics

Semantics of structure may be used in related areas in AI, expert systems, non-monotonic reasoning, and natural language systems. Structural semantics may benefit automated programming efforts as in the previous section. It could also be applied to many search problems. The search may benefit from understanding the structure of the search space, rather than the structure of an individual element of the space.

BIBLIOGRAPHY

[1] LeBlanc M., W. A. Wisdom, Deductive Logic, Allyn and Bacon, Inc. 1976.

[2] Bundy, A., The Computer Modelling of Mathematical Reasoning, Academic Press, 1983.

[3] Bibel, W., Automated Theorem Proving, Friedr. Vieweg and Sohn, 1983.

[4] Frege, G., Begriffsschrift, Halle, 1879.

[5] Sowe, J.F., Conceptual Structures, Addison-Wesley, 1984.

[6] Jeffery, R. C., Formal Logic: Its Scope and Limits, McGraw-Hill, 1962.

[7] Beth, E. W., Formal Methods, D. Riedel, 1962.

[8] Robinson, J. A., Logic: Form and Function, Edinburgh University Press, 1979.

[9] Bibel, W., "An Approach to a Systematic Theorem-Proving procedure in First-Order Logic," Computing 12, pp.43-55, 1974.

[10] Walther, C., "A Mechanical Solution of Schubert's Steamroller by Many Sorted Resolution," Proceedings of the National Conference on Artificial Intelligence, 1984, pp.330-334.

[11] Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle," JACM 12, 1965, pp.23-41.

[12] Bledsoe, W. W., "Non-Resolution Theorem Proving," Artificial Intelligence 9, 1977, pp.1-35.

[13] Boyer, R.S., J. S. Moore, A Computational Logic, Academic Press, 1979.

[14] Loveland, D. W., Automated Theorem Proving, North-Holland, 1978.

[15] Wos, L., R. Overbeek, L. Henschen, "Hyperparamodulation: A Refinement of Paramodulation," 5th Conference on Automated Deduction, 1980, pp.208-219.

AN IMPROVED THEOREM PROVER BY USING THE SEMANTICS OF STRUCTURE

by

DONALD GORDON JOHNSON II

B.S., Kansas State University, 1982

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

ABSTRACT

Automated theorem proving is a central theme in Artificial Intelligence. Theorem provers today have two major problems. The first is described by Alan Bundy, "Resolution collapses under the weight of anything the least bit difficult." This problem is a result of the search space exploding combinatorially. Bundy points out that guidance heuristics, in the general case, do little to curb this explosion. The second problem is that when a theorem prover runs infinitely it has no way of detecting this. It cannot build a fence around this "Black-Hole", in order to inform the user when it might be running infinitely.

These problems are addressed, and, in some sense, solved in this thesis. The search space is improved to be similar to that used by a human and a fence can now be built around Black-Holes. The way this has been done is by the use of the semantics of structure. By using these semantics, an "intelligent" unifier may be built, which can create all the possible variable bindings for a statement, instead of one at a time, as with current unifiers.

The use of semantics of structure and an "intelligent" unifier have allowed a new type of proof procedure to be developed. The procedure is described in detail.