CONSTRUCTING PROGRAMS AUTOMATICALLY USING THEOREM PROVING

By

Richard J. Waldinger

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania
May, 1969

Submitted to the Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-67-C-0058) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited. (AD 697 041)
ACKNOWLEDGEMENTS

I am deeply indebted to Prof. Herbert Simon, who posed the problem of program writing to me and who contributed ideas and encouragement during all phases of this work.

I also owe thanks to Dr. Richard C. T. Lee, who collaborated with me on the implementation and who helped develop the conceptualization.

Dr. James R. Slagle guided our summer work at the Heuristics Lab at MIT, and helped relate program writing and question answering.

A draft of this work was read in very crude form by Profs. Peter Andrews, Donald Loveland and J. Alan Meyer, who, along with Prof. Simon, served on my committee. Many improvements have resulted from their suggestions. Prof. Andrews introduced me to mathematical logic, Prof. Loveland lead a seminar on mechanical theorem proving, and Prof. Meyer introduced me to recursive function theory.

I have benefitted from conversations with many others; I learned much from discussing program verification with Jim King, and Prof. Robert Floyd. I have taken advantage of encounters with Prof. Zohar Manna and Cordell Green. Prof. Alan J. Perlis and Dr. C. L. Chang have offered suggestions. Dr. David Luckham and I collaborated briefly in hand-simulating the construction of a sort program. Tom Frank helped me reconcile the program writer to the 360 LISP
The thesis was produced on a text editing system called LEAD (List, Edit and Display.), originally developed by Dr. Robert Balzer, and modified by Mary Shaw, Peter Freeman, Richard Young, and others. Mary Shaw made herself quite useful in explaining the quirks of the system.
ABSTRACT

The paper describes a method by which programs may be constructed mechanically. The problem of writing a program is transformed into a theorem proving task. The specifications for the program are used to construct a theorem, the theorem is proved, and the program is derived from the proof of the theorem. The specifications for the program are described as a relation between the input and output variables expressed in predicate calculus. Mechanical theorem proving techniques are used to prove the existence of output variables satisfying the specifications. Existence is proven constructively, so that embedded in the proof is a method to compute the desired output values. A program is extracted from the proof.

Restrictions to Robinson's resolution principle are proposed so that only constructive proofs are produced.

A proof of the soundness of the method is presented. In other words, it is shown that the programs written by the program writer do indeed satisfy the specifications.

It is also shown that programs for the entire class of recursive functions may be written by automatic program writing. Thus nothing is lost by restricting application of the resolution principle.

An implementation of the method which writes LISP programs is described.
# TABLE OF CONTENTS

1. Introduction
   - Purpose 7
   - Approach 8
   - A short history of programs that write programs 9
   - The method 12
   - Fertile and sterile clauses 20
   - Primitivity 22
   - Primitive proofs 26
   - The algorithm 28
   - LISP 31

2. Soundness
   - Notation 34
   - Partial assignments 34
   - Truth 38
   - Primitive proofs 49
   - Main theorem 60

3. Loops
   - Induction 67
   - The least-number principle 71
   - List induction 75

4. Completeness
   - Limitations 89
5. Implementation

6. Needed Improvements

7. Bibliography
CHAPTER 1. INTRODUCTION

PURPOSE

The main goal of the work described in this paper is to construct a program that writes programs. How do you describe a program to such a system without writing the program yourself? One way is to describe the input, the output, and the relation between them. Then the problem of programming is expressed this way: given a relation between the input and output variables expressed in first-order predicate calculus, construct a program that, given values for input variables, will find values for the output variables bearing the desired relation to the input values.

Example: Suppose we wish the system to construct a program to subtract two integers. The relation we would give to the system might be

\[ x + y = z, \]

where \( x \) and \( z \) are designated as input variables, and \( y \) as an output variable. (Note: For readability we will often write functions between their arguments rather than in front; thus we write \( x + y = z \) instead of something like \( \text{P1}((\text{f1}(x,y)),z) \), except in the chapter on SOUNDNESS.) We assume that the system somehow knows the properties of the integers, as well as the constructs available in the programming language in which the program is to be written. The output of the system should be a program that performs the subtraction. Thus the
user of the system need not construct an algorithm for the
program, nor need he know the language in which the program
is to be written. The system does more than make the
predicate calculus a higher level language: it takes the
problem of programming, or algorithm design, completely out
of the hands of the user. All he must do is provide the
specifications for the program; the system does the rest.

Of course it might be difficult to express the relation
between the input and the output variables; it could be
easier to write the program itself than to express the
relation. In such cases the system would be of little use.

Program verification is not necessary in the system to
be described once the program is produced, it is guaranteed
to be correct.

APPROACH

The approach we use may be described as follows: We wish
a program to find values of some variables $y_1, \ldots, y_n$ given
values of some variables $x_1, \ldots, x_n$, such that a certain
relation holds between the $x_i$ and the $y_j$.

To do this we prove that, for all $x_i$, such $y_j$ exist. If
we are careful how we prove the theorem, then we can tell
from the proof how to find the values of the $y_j$. When we say
we must be careful, what we mean is that we must, in a
certain sense, be constructive. That is, the only method we
shall accept for proving that a $y_j$ exists is one that tells
us its value in all circumstances. That description is extracted from the proof, giving a program.

Example: In order to write a subtraction program, we give the program the relation $x_1 + y_1 = x_2$. The program proves from its axioms that, for all values for $x_1$ and $x_2$, a value for $y_1$ exists satisfying that relation. In proving the theorem it suggests that the difference between $x_2$ and $x_1$ always satisfies the relation. It then recommends the computation of that difference as the program.

The system is given a set of axioms; these axioms describe "facts" about the area in which the program is being written (for example, facts about number theory if integer arithmetic is being used, and facts about lists if a list processing language is being used); the axioms also describe the properties of the constructs available in the programming language. Which language the program is written in depends essentially on what axioms the system has been given. These axioms need not be supplied by the user; they may be stored in the system in advance.

A SHORT HISTORY OF PROGRAMS THAT WRITE PROGRAMS

We shall outline here some earlier attempts to construct program-writing programs; there was a pioneering effort by Friedberg to use stimulus-response techniques in teaching a program to write programs, but we shall limit discussion to programs that used symbol-manipulating techniques, rather
than inductive or experimental methods in program writing.

H. A. Simon constructed a "Heuristic Compiler" which wrote IPL-V programs. Actually there were two parts to the Heuristic Compiler, a "state-description" and a "process-description" component. The state-description part is closer in purpose to our present work. The process-description component translates imperative sentences in a subset of English into IPL code. Thus it really is a compiler rather than a program writer, in our terminology. However, the state-description component actually writes programs, given a "snapshot" of the input and the output. This task is quite similar to the one we posed earlier; however, the snapshot language is too restrictive; predicate calculus gives us a much richer vocabulary, allowing us to express concepts not easily represented pictorially.

J. R. Slagle successfully applied his question-answering program "DEDUCOM" to one of the tasks which Simon assigned the Heuristic Compiler. Slagle's approach is closer to the spirit of our own work, in that the relation between input and output is expressed in the language of the predicate calculus. The technique used in deriving the program is similar to the present technique: a theorem is proven, and the program is written by keeping track of the substitutions made for certain crucial variables in the course of the proof. The current program extends the technique so that branches and loops can be written.
C. C. Green, working with E. Raphael reports that he is applying their question-answering programs QA2 and QA3 to the task of program writing. The techniques used are generally similar to ours. Green's program writing work has not been published at the time of this writing. In personal communication we have observed several differences between our approaches:

1. Green proves that a program exists; we prove that an output variable exists.

2. Green's program writes branches in its programs by including two explicit axioms for the conditional expression; our branches arise from the structure of the proof itself, with no special axioms to introduce them.

3. Green's theorem prover is not restricted to producing only constructive proofs. However he has not encountered any example in which uncomputable programs were written.

4. Green's system produces a program as soon as the proof is complete. The present system requires further processing after the proof is found in order to find the program. However, the time our system saves by not doing program writing during the proof-finding phase is greater than the time required by the program-writing phase.

The theoretical foundations for automatic program writing go back to S. C. Kleene, who did the original work relating recursive function theory to intuitionist logic. Kleene proved that if the existence of a number satisfying
certain properties can be proven in a formal intuitionist system, then the definition of the function computing that number can be extracted from the proof.

Program writing is of course closely related to program verification. Our statement of the programming problem is based on R. Floyd and J. King's use of mathematical logic to describe the meaning of programs and prove their correctness. Also Z. Manna's employment of logical techniques to prove the correctness and termination of algorithms and the equivalence of pairs of algorithms parallels our own attempt to use logic to construct algorithms.

THE METHOD

The procedure the system uses to construct a program may be described briefly as follows: The system is given a relation

\[ M(x_1, \ldots, x_m, y_1, \ldots, y_n), \]

in predicate calculus, where \( x_1, \ldots, x_m \) are designated as input variables and \( y_1, \ldots, y_n \) as output variables. Assume \( A \) is the list of axioms describing the mathematical system and the programming language. It then tries to prove the theorem

\[ A \Rightarrow (x_1) \ldots (x_m)(\exists y_1) \ldots (\exists y_n)M(x_1, \ldots, x_m, y_1, \ldots, y_n), \]

using Robinson's "resolution principle" as a rule of inference. Then it observes what substitutions were made for the variables \( y_j \) in the course of the proof, and from these it constructs a program.
Before describing this method in more detail, we will give some examples of its application.

Example: Suppose we wish to write a program that always gives 1 as the value of its output variable, regardless of the value of its input variable. The relation is \( y_1 = 1 \), where \( y_1 \) is the only output variable and \( x_1 \), which does not occur in the relation at all, is the input variable. The only axiom we will need to prove the theorem is

\[(z)z = z.\]

For simplicity we will assume that this is the only axiom on the list. We can write the theorem

\[(z)z = z \rightarrow (x_1)(3y_1) y_1 = 1.\]

In order to use the resolution principle we must first translate the negation of the theorem into the quantifier-free format which Robinson calls LDP (The Language of Davis and Putnam.) In LDP, the negation of the theorem is

1. \( (z = z) \)
2. \( \neg(y_1 = 1) \)

These clauses can be resolved, giving the empty clause \( \varnothing \). In performing the deduction, the variable \( y_1 \) was replaced by the constant 1. Consequently, the program we derive from the proof assigns the variable \( y_1 \) the value 1.

Example: We wish to write a LISP program that finds the second element of a list. In order to simplify the exposition, we will assume that no LISP S-expressions are atoms. Then we can include among our axioms the clause
1. \( \{ z = \text{cons}(\text{car}(z), \text{cdr}(z)) \} \)

The relation between the input and output is then

\( (\exists u)(\exists v)(\exists w)(x_1 = \text{cons}(u, v) \land v = \text{cons}(y_1, w)) \),

where \( x_1 \) is the input variable and \( y_1 \) is the output variable. This relation is not the most natural, but it will lead to a theorem whose proof is easy to follow. The theorem derived from the relation is

\( (x_1)(\exists y_1)(\exists u)(\exists v)(\exists w)(x_1 = \text{cons}(u, v) \land v = \text{cons}(y_1, w)) \).

The negation of the theorem, in LDP, is then

2. \( \{ \neg(a = \text{cons}(u, v)), \neg(v = \text{cons}(y_1, w)) \} \)

The input variable \( x_1 \) has been replaced by the input constant \( a \). Clauses (1) and (2) may be resolved: \( v \) is substituted for \( z \), \( \text{car}(v) \) for \( y_1 \) and \( \text{cdr}(v) \) for \( w \), and the second literal of (2) is deleted, giving

3. \( \{ \neg(a = \text{cons}(u, v)) \} \)

Now clauses (1) and (3) can be resolved, giving the empty clause.

Suppose we write the proof in tree form, in which each clause becomes a node and each node is connected by arrows to the two nodes from which it was derived:
Next we mark each arc with the substitution which was made for the variables in the upper node in deriving the lower node. Substitutions will be written as strings of assignment statements.

Suppose we ignore the arcs leading to node (1). (Later we will discuss which arcs we should ignore.) Let us treat the figure as a program, whose initial node is (4) and whose halt node is (2). The substitutions will be executed as assignment statements. Then at the end the value of the output variable y1 will be the second element of the list a.

Notice that the substitutions made for the variables u and w were irrelevant; they did not affect the value of the
output variable, and hence could have been omitted. If an assignment to a variable in a node affects the value of an output variable, the former variable will be called a hot variable of the clause. Thus, in the above example v is hot in (3) whereas u is cold.

Example: Suppose we wish to write the characteristic function of the predicate Atom, that is, the function whose value is 1 if its argument is a LISP atom, and 0 otherwise. The relation may be expressed as

\[(\text{Atom}(x1) \land y1=1) \lor (\neg \text{Atom}(x1) \land y1=0),\]

where \(x1\) is the input variable and \(y1\) the output variable.

The only member of the set of axioms we need is

\[(z) \ z=z\]

The theorem then becomes

\[(z) z=z \Rightarrow (x1)(\exists y1). \text{Atom}(x1) \land y1=1 \lor \neg \text{Atom}(x1) \land y1=0.\]

The negation of the theorem, in Robinson's format, is then

1. \(\{z=z\}\)
2. \(\{\neg \text{Atom}(a), \neg (y1=1)\}\)
3. \(\{\text{Atom}(a), \neg (y1=0)\}\)

The proof proceeds as follows:

4. \(\{\neg \text{Atom}(a)\},\)

from (1) and (2). The output variable \(y1\) was replaced by 1 in making the deduction.

5. \(\{\text{Atom}(a)\},\)

from (1) and (3). The output variable \(y1\) was replaced by 0.
from (5) and (6). Let us construct a tree, as before, this time only writing down those substitutions whose execution is necessary in computing the value of the output variable:

![Diagram of decision tree](image)

Beginning at node (6), we see that the program contains a branch, and that the path it chooses determines what value is assigned to the output variable $y_1$. We must give the program some criterion for choosing a path.

Let us place near the tail end of each arrow the complement of the literal that has been deleted from the upper clause in deriving the lower, all substitutions being made in the literal. For example, in deriving (4), the literal $\neg(y_1=1)$ was deleted from literal (2). The constant 1 was substituted for $y_1$. Therefore, near the tail of the arrow between arcs (4) and (2) we place the literal $(y_1=1)$, which is the complement of $\neg(y_1=1)$ after the substitution has been
made. The completed tree is then

The new literals near the bases of the arrows serve as tests. If the literal near the base of an arc is true, that arc may be traversed; otherwise the other arc must be the true path, since the literals are complements.

The test for the arc between (4) and (1) is \( \neg (1=1) \). This expression will never be true, and hence the arc will never be traversed. Similarly the arc between (5) and (1), whose test is \( \neg (0=0) \), will never be traversed. Consequently node (1), and the arcs leading into it, may be deleted from the program, since they will always be bypassed.
Note that the program satisfies the specifications. In the beginning, control is at node (6); if the input \( a \) is an atom, the condition \( \text{Atom}(a) \) is satisfied, and control passes to node (4); then the assignment \( y_1 \leftarrow 1 \) is executed, control passes to node (2), and the program halts. The final value of \( y_1 \) is 1. Similarly, if \( a \) is not an atom, the program will eventually halt at node (3), with \( y_1 \) equal to 0.

A short discussion of the rationale for this algorithm may make it seem less arbitrary: The reader may note that as control passes through a node in the program, the clause corresponding to the node is false, if the assignments are made to the variables of the clause. That is, all the literals in the clause are false. For instance, the initial node (6) corresponds to the empty clause, which is always false. If we reach node (4), we know from the test that \( \text{Atom}(a) \) was true, so clause (4) is falsified. We consequently assign \( y_1 \) the value 1, and pass to node (2). But then node (2) is falsified too, since we have assigned \( y_1 \) the value 1. Now, if one of the initial clauses is falsified, we know that
the original relation is true, because the initial clauses correspond to the negation of the theorem. This explanation is essentially correct, but it must be made more precise; in particular, the notion of truth must be clarified to cover the case in which we know the values of some of the variables and constants but not those of others; to do this the concept of "partial assignment" will be introduced, and a more formal proof of the correctness of the algorithm will be given in the section entitled SOUNDBNESS.

The clauses correspond to the negation of the assertions used by Floyd [1967].

The next section will discuss precisely which nodes and arcs may be eliminated from the proof in deriving the program.

STERILE AND FERTILE CLAUSES

In writing the program for the characteristic function of the Atom predicate, we could delete node (1), and both arcs leading up to it, from the tree, because the tests leading into it were all conditions which could never be satisfied.

Clause (1) was the only axiom in the proof. Among the initial clauses in any proof, there will be some which may be said to be true. The axioms will always be true. Some of the hypotheses may be true; we will discuss the truth of a clause more formally later. True clauses will be called sterile.
Clauses deduced from two sterile clauses will also be called sterile. Other clauses will be called fertile. It is proven in the section on SOUNDNESS that nodes corresponding to sterile clauses, and the arcs leading into them, may always be deleted, since the tests leading into them will never be true when the program is being executed.

Furthermore, if a clause is derived from one fertile and one sterile clause, one of the arcs leading from the node is deleted. Hence the test on the remaining arc is redundant, since it will always be true; that test too may be deleted.

It is a task of the user to decide which of the hypotheses is to be sterile, since in general he has a choice. For instance, if we wish to write a program to do division, the relation might be

\[ \neg(x_1=0) \Rightarrow x_2 = y_1 \times x_1, \]

where \( x_1 \) and \( x_2 \) are input variables and \( y_1 \) is the output variable. The list of clauses would contain

\[ \neg(a_1=0), \]

where \( a_1 \) is the constant which replaced \( x_1 \). This clause could be true or false. If the user wishes to interpret the relation as saying that the number \( x_1 \) is always non-zero, then the clause is true. The program may then assume that the input data are such that the divisor is always non-zero. However, another interpretation of the relation is possible: the divisor may in fact be zero, but the relation \( x_2 = y_1 \times x_1 \) must be satisfied only if the divisor is not zero. Thus in
the case that the divisor is zero. In this latter case, the clause

\{(a1=0)\}

is not sterile. It is up to the user to decide if the clause is fertile or not. In the first case, the program is simply to perform the division; in the second case, it is to test the divisor, give out an arbitrary answer if the divisor is zero, and perform the division otherwise.

Sterile clauses will by definition never contain output variables or hot variables (variables whose values affect output variables.)

PRIMITIVITY

In an earlier example, the predicate Atom was used in the statement of the theorem; it also appeared in the LISP program. Fortunately, an Atom predicate is defined in LISP; a LISP interpreter or compiler can understand it. A symbol in the vocabulary of the programming language will be called a primitive symbol. Are we then limited to primitive symbols in the statement of our axioms and theorems? Such a limitation would be quite constricting for several reasons:

1. It is often convenient to describe programs using concepts that are not defined in the language;

2. In particular, we might wish to use the name of the function being written in describing it.

3. In changing the negation of the theorem into LDP,
variables that were universally quantified in the theorem are replaced by terms involving function symbols in the clause form. Except for constants resulting from input variables and a few other symbols, these functions are not primitive; they are not to appear in the program. Yet the theorem cannot be stated without using them.

Let us examine an example in which non-primitive symbols occur in a program. The program is to be given a list as input. If the list consists entirely of atoms, the output is to be \( T \); otherwise it is to be \( NIL \). Assume that the predicate \( \text{Member}(x,1) \), which tests whether \( x \) is an element of the list \( 1 \), is in the system; that is, the symbol "Member" is primitive. Then we might describe the desired relation by

\[
y_1 = T \land (u)(\text{Member}(u,x_1) \Rightarrow \text{Atom}(u)) \\
\lor y_1 = NIL \land (\exists v)(\text{Member}(v,x) \land \neg \text{Atom}(v))
\]

where \( x_1 \) is the input and \( y_1 \) is the output variable. We assume the axiom list to include \( z=z \). The theorem would then be

\[
(x_1)(\exists y_1). \ (z)z=z \Rightarrow \\
y_1=T \land (u)(\text{Member}(u,x_1) \Rightarrow \text{Atom}(u)) \\
\lor y_1=NIL \land (\exists v)(\text{Member}(v,x_1) \land \neg \text{Atom}(v)).
\]

Put into clause form, the negation of the theorem might look like

1. \( \{z=z\} \)
2. \( \{\neg(y_1=T), \text{Member}(g(y_1),a1)\} \)
3. \( \{\neg(y_1=T), \neg \text{Atom}(g(y_1))\} \)
4. \( \{\neg(y_1=NIL), \neg \text{Member}(v,a1), \text{Atm}(v)\} \)
The constant a₁ replaced the input variable x₁; since we assume the value of the input variables to be available to the program, a₁ is primitive. However, the term g(y₁) replaced the variable u, and g is not primitive. The intuitive meaning of g(y₁) is "that element of a₁ which is not an atom, if there is one; otherwise an arbitrary S-expression." Such a construct is not defined in LISP.

We will write down a proof, using the notation "n₃ (n₁, n₂) C" to mean "the clause C is numbered n₃ and was derived from clauses (n₁) and (n₂)":

5. (1,2) {Member(g(T),a₁)}
6. (1,3) {¬Atom(g(T))}
7. (1,u) {¬Member(v,a₁), Atom(v)}
8. (6,7) {¬Member(g(T),a₁)}
9. (5,8) φ

A tree may be constructed in the same way as before. The sterile node (1) will be eliminated:
This flowchart would be correct if a value were assigned to \( g(T) \) by the LISP system, just as a value is assigned to the input constant \( a_1 \); since it is not, the program is not executable. The proof above is not sufficiently constructive; we must modify our theorem prover so that such proofs are not produced. The proof must be so restricted that only primitive symbols occur in the derived program. Such proofs will be called "primitive" themselves.

There are three ways in which a symbol in a proof may occur in a program:

1. It may be an output variable. In LISP, it then occurs as an argument of "return".

2. It may occur in a term which has been substituted for a hot variable: In that case it appears in an assignment or "SETQ" statement.

3. It may occur in a literal which has been deleted when
the resolution rule has been applied to two fertile clauses. It then occurs in a test or condition in the program. A primitive proof, then, is one in which non-primitive symbols do not occur in the above contexts. The theorem prover, in order to produce only primitive proofs, will block any application of the rule of inference that will lead to a non-primitive proof.

PRIMITIVE PROOFS

This section will review a bit more concisely what restrictions on resolution are necessary for a proof to be primitive. Many of the definitions will be repeated; more formal definitions of the same concepts will be given in the section on soundness.

Let A be the theorem under consideration. The "halt" clauses are the initial clauses of the proof.

The "sterile" halt clauses are clauses that are true for all values of the input variables, and that do not contain output variables.

The sterile clauses are the sterile halt clauses and any clause deduced from two sterile clauses.

The "fertile" clauses are those which are not sterile.

The "hot" variables of a clause are defined inductively:
1. The output variables of a halt clause are hot.
2. Let B be a resolvent of C and D in a proof. Then by
the definition of the resolution rule (Robinson [1965]) there exist substitutions \( \sigma_C \) and \( \sigma_D \) and subsets \( L \) of \( C \) and \( M \) of \( D \) such that \( B = (C \setminus L)\sigma_C \cup (L \setminus M)\sigma_D \), where \( L, \sigma_C \) and \( M, \sigma_D \) are singletons and complements. Then if \( w_1 \) is hot in \( C \), those variables which occur in both \( w_1, \sigma_C \) and \( B \) are hot in \( B \). Similarly, if \( w_2 \) is hot in \( D \), those variables which occur in both \( w_2, \sigma_D \) and \( B \) are hot in \( B \).

Let "Prim" be the set of symbols that contains all the variables and all the constants that correspond to input variables in the theorem \( A \), and all the constant, function and predicate symbols defined in the programming language under consideration.

An expression will be said to be "primitive" if all its symbols belong to Prim.

A proof of \( A \) will be said to be primitive if

1. whenever \( B \) is a resolvent of \( C \) and \( D \) in the proof as before, all the terms \( w_1, \sigma_C \) and \( w_2, \sigma_D \) are primitive, where \( w_1 \) is hot in \( C \) and \( w_2 \) is hot in \( C \).

2. whenever \( B \) is a resolvent of two fertile clauses \( C \) and \( D \), the literal \( L, \sigma_C \) (or equivalently \( M, \sigma_D \)) is primitive.

The above definition has been so constructed that if a proof is primitive all the symbols which appear in the program derived from the proof will be primitive.

For instance, in the preceding example [Ex1] the
program contains the symbol q which is not primitive. However, the proof itself was not primitive because clause (8), for example, was derived from clauses (6) and (7); \texttt{L.sigmaC} in that case was \texttt{-Atom(q(T))}, which contains the non-primitive symbol q. Hence the derivation of clause (8) violates condition (2) of the definition. Similarly, the derivation of clause (9) violates the same condition. If the symbol \texttt{NIL} were not primitive in \texttt{LISP}, the derivation of clause (7) would not be primitive either, because the variable \texttt{y1} is hot in clause (4), and the non-primitive \texttt{NIL} is \texttt{y1.sigmaC = y1.(NIL/y1)}, violating condition (1) of the definition of primitivity. Thus, in order to write programs, the theorem prover must be equipped with an attachment that prevents non-primitive steps from being made in the proof. The theorem in the example can be proved, but the proof given is inadequate.

THE ALGORITHM

This section summarizes the method for deriving a program from a primitive proof of a theorem. For the sake of definiteness, the program derived is constructed in \texttt{LISP}; actually any programming language could be used, and the section on \texttt{SOUNDNESS} presents a language-independent treatment.

Let \texttt{sigma} be a substitution and \texttt{H} a set of variables. then define \texttt{proqsub(sigma,H)} to be a sequence of \texttt{"SETQ"} instructions defined inductively as follows:
1. If \( \text{sigma} \) is the empty substitution, 
\[
\text{progsup}(\text{sigma}, H) = \text{the empty sequence.}
\]
2. \[
\text{progsup}(\tau U \{t1/v1\}, H)
\]
\[
= (\text{SETQ } v1 \ T1) \text{progsup}(\tau, H)
\]
if \( v1 \) belongs to \( H \),
\[
= \text{progsup}(\tau, H) \text{otherwise.}
\]

Let \( B \) be a fertile clause in the proof. "progc(B)" is defined inductively as follows:

1. If \( B \) is a halt clause, and if \( y1, ..., yn \) is the complete list of output variables in the program being written, then
\[
\text{progc}(B) = (\text{RETURN } (\text{LIST } y1 ... yn)).
\]

2. If \( B \) is a resolvent of clause \( C \) and \( D \) in the proof, where \( E = (C-L)\text{sigma}\ C U (D-M)\text{sigma}\ D \), and
   a. \( C \) is fertile but \( D \) is not, let \( HC \) be the hot variables of \( C \). Then
   \[
   \text{progc}(B) = \text{progsup}(\text{sigma}C, HC) \text{progc}(C).
   \]
   An analogous definition covers the case that \( D \) is fertile and \( C \) is not.
   b. If both \( C \) and \( D \) are fertile, let \( HC \) and \( HD \) be the hot variables of \( C \) and \( D \) respectively. Then
   \[
   \text{progc}(B) =
   \]
   \[
   (\text{COND } (\text{L.sigmac} \ (\text{GC} \ g1)))
   \]
   \[
   \text{progsup}(\text{sigmac}, HC) \text{progc}(C)
   \]
   \[
   \]
   \[
   g1 \text{progsup}(\text{sigmad}, HD) \text{progc}(D)
   \]
   where the negation sign, if any, in \( \text{L.sigmac} \) has been replaced by the LISP function \( \text{NCT} \), and the rest of \( \text{L.sigmac} \)
is in LISP notation.

Thus a program has been associated with every fertile clause. In particular, a program is associated with the clause NIL. Let \( a_1, \ldots, a_n \) be the complete list of constants which originate from input variables during the translation into LDP. Let \( u_1, \ldots, u_k \) be the complete list of variables which occur in \( \text{progC}(\text{NIL}) \). Then the program produced by the system is

\[
(\text{LAMBDA} (a_1 \ldots a_n) \\
(\text{FRCG} (u_1 \ldots u_k) \text{progC}(\text{NIL}))).
\]

Example: Let us rewrite the proof of the theorem associated with the characteristic function of the predicate \( \text{Atom} \), [Ex 1] writing next to each clause the ancestry of the clause, an indication of whether the clause is fertile, the list of hot variables in the clause, and the two substitutions \( \sigma_C \) and \( \sigma_D \) used to derive the clause if it is fertile:

<table>
<thead>
<tr>
<th>Clause</th>
<th>Ancestry</th>
<th>Fertility</th>
<th>Hot vars.</th>
<th>( \sigma_C )</th>
<th>( \sigma_D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( { z = z } )</td>
<td>Halt</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 2. \( \{ \neg \text{Atom}(a), y_1 
eq 1 \} \) Halt | Yes | \( y_1 \) | | (\( y_1 \)) | | |
| 3. \( \{ \text{Atcm}(a), y_1 \neq 0 \} \) Halt | Yes | \( y_1 \) | | (\( y_1 \)) | | |
| 4. \( \{ \neg \text{Atom}(a) \} \) | (1,2) | Yes | \( 1/z \) | \( 1/y_1 \) | |
| 5. \( \{ \text{Atom}(a) \} \) | (1,3) | Yes | \( 0/z \) | \( 0/y_1 \) | |
| 6. \( \circ \) | (4,5) | Yes | | | |

Then

\[
\text{progC}((2)) = (\text{RETURN} \ (\text{LIST} \ y_1))
\]

\[
\text{progC}((3))
\]
\[ \text{progc}(4) = \text{progsub}(\{1/y\}, (y1)) \text{ progc}(2) \]
\[ = (\text{setq} \; y1 \; 1) \; (\text{return} \; (\text{list} \; y1)) \]
\[ \text{progc}(5) = (\text{setq} \; y1 \; 0) \; (\text{return} \; (\text{list} \; y1)) \]

and

\[ \text{progc}(6) = (\text{cond} \; ((\text{not} \; (\text{atom} \; a)) \; (\text{go} \; 1)) \]
\[ \; (\text{setq} \; y1 \; 1) \]
\[ \; (\text{return} \; (\text{list} \; y1)) \]
\[ 1 \; (\text{setq} \; y1 \; 0) \]
\[ \; (\text{return} \; (\text{list} \; y1)) \]

Then the program produced by the system is

\[ (\text{lambd}a \; (a) \]
\[ (\text{progc} \; (y1) \]
\[ <\text{progc}(6)>)) \].
\( (\text{Equal}(x, y) \equiv \text{Equal}(\text{car}(x), \text{car}(y)) \land \text{Equal}(\text{cdr}(x), \text{cdr}(y))) \)

\( \text{Null}(x) \equiv \text{Equal}(x, \text{NIL}) \)

\( \text{Atom}(\text{NIL}) \)

There are also a number of axioms for \( \text{Equal} \), which we omit. A version of the induction axiom for lists is described under LOOPS.

Now let us examine some more examples of LISP program writing. Suppose we wish to write a program to reverse the \text{car} and \text{cdr} of a list: in other words the relation between input \( a \) and output \( x \) is

\[ \text{Equal}(\text{cdr}(a), \text{car}(x)) \land \text{Equal}(\text{car}(a), \text{cdr}(x)). \]

The theorem is

\[ (a)(\exists x). \text{Equal}(\text{cdr}(a), \text{car}(x)) \land \text{Equal}(\text{car}(a), \text{cdr}(x)). \]

The negation of the theorem, in clause form, is

1. \{\neg \text{Equal}(\text{cdr}(a), \text{car}(x)), \neg \text{Equal}(\text{car}(a), \text{cdr}(x))\}.

The relevant axioms from the list are

2. \{\text{Equal}(x, \text{car}(\text{cons}(x, y)))\}

3. \{\text{Equal}(y, \text{cdr}(\text{cons}(x, y)))\}

The proof proceeds as follows:

4. \{\neg \text{Equal}(\text{car}(a), \text{cdr}(\text{cons}(\text{cdr}(a), y)))\}

from (1) and (2), deleting the first literal of (1).

5. \circ

from (3) and (4).
Let us write the proof in tree form, deleting the sterile clauses (2) and (3), and marking down the substitutions for the hot variables:

(1) \{\neg \text{Equal}(\text{cdr}(a), \text{car}(x)), \neg \text{Equal}(\text{car}(a), \text{cdr}(x))\}

| x \cdot \text{cons}(\text{cdr}(a), y)

(4) \{\neg \text{Equal}(\text{car}(a), \text{cdr}(\text{cons}(\text{cdr}(a), y)))\}

| y \cdot \text{car}(a)

(5) o

The program may be seen to satisfy the specifications.
CHAPTER 2. SOUNDNESS

This section proves that the program writing algorithm is correct, that the program written actually does satisfy the specifications. It is designed to be read independently, although it will appear better motivated if the preceding sections are read first. However, all new terms are redefined, sometimes more formally. The section can be skipped by those not interested in the mathematical justification of the correctness of the algorithm.

NOTATION

We shall generally follow the notation of Robinson [1965]. The set of function and predicate symbols will be called the constants. The individual constants are function symbols of degree 0.

We shall often confuse singleton sets with their unique elements. This, we believe, is a common confusion. The variables of clauses shall be considered to be free variables. Bracketed numbers within the text refer back to the section with the corresponding number above it on the right margin.

PARTIAL ASSIGNMENTS

Let $S$ be any set of closed well-formed formulas in a first order language with function symbols, or of clauses in the sense of Robinson. Let $BD$ be a set which we will assume to be fixed for purposes of this discussion.

Def: A "partial assignment" $\phi$ is a function whose
domain is a subset of the set of symbols in our alphabet, such that

1. \( \phi \) maps variables and individual constants into \( \mathcal{D} \).
2. \( \phi \) maps function symbols of degree \( n \) into functions \( h: \mathcal{D}^n \rightarrow \mathcal{D} \).
3. \( \phi \) maps predicate symbols of degree \( n \) into \( n \)-place relations over \( \mathcal{D} \).

Considering \( \phi \) as a function, we shall call its domain "\( \text{dom}(\phi) \)". If \( x \) belongs to \( \text{dom}(\phi) \) then we shall also say \( \phi \) is "defined on" \( x \).

Partial assignments shall be written to the right of their arguments. This is to make our notation compatible with that for substitutions, which are also written to the right of their arguments.

Def: Let \( \phi \) and \( \xi \) be partial assignments, and let \( T \) be a set of symbols. We say that \( \phi \) and \( \xi \) "agree on" \( T \) if \( T \) is contained in \( \text{dom}(\phi) \) and \( \text{dom}(\xi) \) and if \( t.\phi = t.\xi \) for all \( t \) belonging to \( T \).

Def: Let \( \phi \) and \( \xi \) be partial assignments. We say that \( \xi \) is an "extension" of \( \phi \), or that \( \phi \) is a "restriction" of \( \xi \), if \( \text{dom}(\phi) \) is contained in \( \text{dom}(\xi) \), and if \( \phi \) and \( \xi \) agree on \( \text{dom}(\phi) \).

Def: Let \( S \) be any set of wffs or clauses. We say that a partial assignment \( \phi \) is complete on \( S \) if all the symbols of \( S \) belong to \( \text{dom}(\phi) \), except perhaps for variables that do not occur free in members of \( S \).

Def. Let \( S \) be a set of wffs or clauses. Let \( \phi \) be a
partial assignment. We say that phi* is an "expansion" of phi on S if

1. phi* is an extension of phi.
2. dom(phi*) consists precisely of dom(phi) and the constants which occur in elements of S. Recall that the set of constants includes the function and predicate symbols as well as the individual constants.

Def: We say that phi' is a "completion" of phi on S if

1. phi' is an extension of phi.
2. dom(phi') consists of dom(phi) and the set of symbols which occur in elements of S, except for variables that do not occur free in any element of S. Note that a completion of phi on S is complete on S.

Example: Let S be the set consisting of the following clauses:

1. \{P(b,a1)\}
2. \{¬P(x1,y1), ¬P(y1,z1), P(x1,z1)\}
3. \{P(f(x2),x2)\}

Let DD be the set of integers. Let phi be the partial assignment defined by

P.phi = the > relation
b.phi = 1
a1.phi = 0
x1.phi = 2

Then the partial assignment phi* that agrees with phi:

P.phi* = the > relation
b.phi* = 1
a1. phi* = 0
x1. phi* = 2

and such that
f. phi* = the function x+1

is an expansion of phi on S.

The partial assignment phi' that agrees with phi*:
P. phi' = the > relation
b. phi' = 1
a1. phi' = 0
x1. phi' = 2
f. phi' = the function x+1

and such that
y1. phi' = 3
z1. phi' = 4
x2. phi' = 5

is a completion of phi on S.

Example: Let S be the set of wffs
1. (x)(3y)P(x,y)
2. Q(x)

Let DD be the integers again, and let phi be defined by
P. phi = the equality relation
f. phi = the function x+1

Then phi is a partial assignment on S, even though there exists an element of dom(phi) which does not occur in any element of S.
Let \( \phi^* \) be defined by

\[
\begin{align*}
P \cdot \phi^* &= \text{the equality relation} \\
f \cdot \phi^* &= \text{the function } x+1 \\
Q \cdot \phi^* &= \text{the relation } x > 0
\end{align*}
\]

Then \( \phi^* \) is an expansion of \( \phi \) on \( S \).

On the other hand, let \( \phi^\dagger \) be defined by

\[
\begin{align*}
P \cdot \phi^\dagger &= \text{the equality relation} \\
f \cdot \phi^\dagger &= \text{the function } x+1 \\
Q \cdot \phi^\dagger &= \text{the relation } x > 0 \\
x \cdot \phi^\dagger &= 0 \\
y \cdot \phi^\dagger &= 1
\end{align*}
\]

Then \( \phi^\dagger \) is not a completion of \( \phi \) on \( S \), although it is indeed complete on \( S \). The variable \( y \) does not occur free in any element of \( S \) so it should not belong to \( \text{dom}(\phi^\dagger) \) if \( \phi^\dagger \) is to be a completion.

**TRUTH**

The time has come to introduce the notion of truth under a complete partial assignment.

**Def:** Let \( C \) be a term, clause, or wff, and let \( \phi \) be a partial assignment which is complete on \( C \). We define \( C \cdot \text{Val}(\phi) \) inductively as follows:

1. If \( C \) is a constant or variable,

\[
C \cdot \text{Val}(\phi) = C \cdot \phi
\]
2. If \( C \) is a term \( g(x_1, \ldots, x_n) \),
\[
\text{C.Val(\phi)} = (g.\text{phi})(x_1.\text{Val(\phi)}, \ldots, x_n.\text{Val(\phi)})
\]

3. If \( C \) is the empty clause NIL, then
\[
\text{C.Val(\phi)} = F.
\]
(We then say that \( C \) is "false" under \( \phi \).)

4. If \( C \) is an atomic formula \( P(x_1, \ldots, x_n) \), then
\[
\begin{align*}
\text{C.Val(\phi)} &= T \text{ if } (P.\text{phi})(x_1.\text{Val(\phi)}, \ldots, x_n.\text{Val(\phi)}) \\
&= F \text{ otherwise.}
\end{align*}
\]
(When \( \text{C.Val(\phi)} = T \) we say \( C \) is "true" under \( \phi \).)

5. If \( C \) is a clause \( (C_1 \lor C_2) \) or a wff \( (C_1 \lor C_2) \), then
\[
\begin{align*}
\text{C.Val(\phi)} &= T \text{ if } C_1.\text{Val(\phi)} = T \text{ or } C_2.\text{Val(\phi)} = T \\
&= F \text{ otherwise.}
\end{align*}
\]

6. If \( C \) is \( \neg B \), then
\[
\begin{align*}
\text{C.Val(\phi)} &= T \text{ if } B.\text{Val(\phi)} = F \\
&= F \text{ otherwise.}
\end{align*}
\]

7. If \( C \) is \( C_1 \land C_2 \), then
\[
\begin{align*}
\text{C.Val(\phi)} &= T \text{ if } C_1.\text{Val(\phi)} = T \text{ and } C_2.\text{Val(\phi)} = T \\
&= F \text{ otherwise.}
\end{align*}
\]

8. If \( C \) is \( (x)B \) then
\[
\begin{align*}
\text{C.Val(\phi)} &= T \text{ if } B.\text{Val}(x_i) = T \\
&\text{for every partial assignment } x_i \text{ complete on } B \\
&\text{that agrees with } \phi \text{ on } \text{dom}(\phi) \setminus \{x\} \\
&= F \text{ otherwise.}
\end{align*}
\]

9. If \( C \) is \( (\exists x)B \) then
\[ \text{C.Val(\phi)} = \begin{cases} T & \text{if B.Val(\phi)} = T \\ \text{for some partial assignment } x_i \text{ complete on } B \\ \text{that agrees with } \phi \text{ on } \text{dom}(\phi) - \{x\} \\ F & \text{otherwise.} \end{cases} \]

The dot may be eliminated when readability is not impaired.

**Note:** It is intuitively clear, and may be shown by induction, that if \( \phi \) and \( x_i \) are two partial assignments complete on \( C \) that differ only on symbols that do not occur in \( C \), then

\[ \text{C.Val(\phi)} = \text{C.Val(x_i)}. \]

**Def:** Let \( S \) be a set of closed wffs or clauses and let \( \phi \) be a partial assignment. \( S \) is "\( \phi \)-valid" if, for every completion \( \phi' \) of \( \phi \) on \( S \) and for every \( C \) belonging to \( S \),

\[ \text{C.Val(\phi')} = T. \]

[6]

\( S \) is "\( \phi \)-satisfiable" if there exists an expansion \( \phi^* \) of \( \phi \) on \( S \) such that for every completion \( \phi' \) of \( \phi^* \) on \( S \), and for every \( C \) belonging to \( S \),

\[ \text{C.Val(\phi')} = T. \]

In this case, we shall also say that \( \phi^* \) "\( \phi \)-satisfies" \( S \).

\( S \) is "\( \phi \)-unsatisfiable" if it is not \( \phi \)-satisfiable.

If \( \phi \) is the empty substitution, then \( \phi \)-satisfiability coincides with the common notion of satisfiability, providing that the variables in each clause are considered to be universally quantified.

**Def:** On the other hand, if \( \text{dom}(\phi) \) contains all the constants that occur in elements of \( S \), and if \( S \) is
phi-satisfiable, then $S$ is in fact phi-valid, since the only expansion of phi on $S$ is phi itself. In that case we say that phi is a "model" for $S$.

Example: Let DD be the positive integers, and let phi be a partial assignment such that $x_1, \phi = 1$. Suppose $P_1$ does not belong to $\text{dom}(\phi)$. Then the clause $C = \{P_1(x_1, x_1)\}$ is phi-satisfiable. For instance, the expansion phi*, which maps $P_1$ into the equality relation, phi-satisfies C. However, C is not phi-valid; for instance the completion of phi that maps $P_1$ into the "less than" relation gives $C$ the value $F$. If $x_i$ is the partial assignment that maps $P_1$ into the equality relation, then $C$ is xi-valid.

Let $D$ be the clause $\{P_1(f_1(x_1), x_1)\}$. Let phi be a partial assignment that maps $P_1$ into the "less than" relation. Then if $f_1$ does not belong to $\text{dom}(\phi)$, $D$ is phi-satisfiable. For instance, let phi* be that expansion of phi on $D$ that maps $f_1$ into the function $x_1^1$. Then any completion of phi* on $D$ gives $D$ the value $T$. On the other hand, $D$ is not phi-valid. The completion phi* of phi on $D$ that maps $f_1$ into the identity function and $x_1$ into $1$ gives $D$ the value $F$.

Theorem: Let $A$ be a sentence of the form

$$(x)(\exists y_1) \ldots (\exists y_n)M(x, y_1, \ldots, y_n),$$

where $M(x, y_1, \ldots, y_n)$ is itself in prenex normal form. Let $A'$ be the transformation of $\neg A$ into Robinson's format. Suppose the variable $x$ is transformed into the constant $a$. Let phi be
a partial assignment complete on \( M(a, y_1, \ldots, y_n) \) containing no symbols in \( \text{dom}(\phi) \) which do not occur free in \( M \). Assume \( A' \) is \( \phi \)-unsatisfiable. Then \( M(a, y_1, \ldots, y_n).\text{Val}(\phi) = T \).

Proof: Before giving the detailed proof, we shall outline it. Suppose

\[
M.\text{Val}(\phi) = F.
\]

Then

\[
\neg M.\text{Val}(\phi) = T \quad [4].
\]

Then by conventional techniques (Davis and Putnam [1960]), we can remove the existential quantifiers from \( \neg M \), replacing them with terms involving new function symbols, and we can successively extend \( \phi \) so that \( \text{dom}(\phi) \) contains all the new function symbols, and so that the extended partial assignment \( \phi \)-satisfies the transformed formula, contradicting our hypothesis.

Now we shall fill in the details of the proof. Assume inductively that at some stage in the process \( \neg M \) has been transformed into \( M' \) of form

\[
(u_1) \ldots (u_k)(\exists v)N,
\]

where \( M' \) is in prenex form and contains \( q \) existential quantifiers, and assume further that

\[
(M')\text{Val}(\phi^*) = T,
\]

where \( \phi^* \) is an extension of \( \phi \) complete on \( M' \). Then we know that

\[
((\exists v)N)\text{Val}(x_i) = T,
\]

where \( x_i \) is any completion of \( \phi^* \) on \( (\exists v)N \). [5] (Note that none of \( u_1, \ldots, u_k \) belongs to \( \text{dom}(\phi^*) \).)
Hence for each such partial assignment \( x_i \) there exists a completion \( x_i' \) of \( x_i \) on \( N \) such that

\[
N.\text{Val}(x_i') = T.
\]

Assume that if \( x_i \) maps \( u_1, \ldots, u_k \) into members \( e_1, \ldots, e_k \) of \( DD \) then \( x_i' \) maps \( v \) into \( q(e_1, \ldots, e_k) \), another member of \( DD \). Let \( f \) be a function symbol of degree \( k+n \) which does not occur in \( N \). Let \( N' \) be \( N \) with each occurrence of \( v \) replaced by the term \( f(y_1, \ldots, y_n, u_1, \ldots, u_k) \). Suppose that \( \phi \) maps \( y_1, \ldots, y_n \) into \( d_1, \ldots, d_n \) respectively, and let \( f' \) be any function mapping \( DD \) into \( DD \) such that

\[
f'(d_1, \ldots, d_n, e_1, \ldots, e_k) = q(e_1, \ldots, e_k).
\]

Let \( \phi* \) be the extension of \( \phi \) which maps \( f \) into \( f' \). Then

\[
N'.\text{Val}(\phi*) = T
\]

for any completion \( \phi' \) of \( \phi* \) on \( N' \). Let \( M'' \) be

\[
(u_1) \ldots (u_k)N';
\]

then

\[
(M'')\text{Val}(\phi*) = T
\]

and \( M'' \) contains \( q-1 \) existential quantifiers. We can conclude by induction that the process will produce a sentence \( M* \) and a partial assignment \( \phi** \) such that \( M* \) contains no existential quantifiers and

\[
(M*)\text{Val}(\phi*) = T.
\]

Then \( M* \) is put into Robinson's format by deleting the universal quantifiers and putting the matrix into conjunctive normal form, yielding \( M' \). Now \( \phi** \) is \( \phi \) extended so that its domain includes all the function symbols introduced by
the process. Thus phi** is an expansion of phi on A'. Let phi'' be any completion of phi** on A'. Then

\[(A')Val(phi'') = T [5].\]

We conclude that phi** phi-satisfies A' [6], contradicting our assumption. Hence M is phi-valid.

Example: Let A be

\[(x)(\exists y_1)(\exists z). P1(z, z) \rightarrow P2(x) \land P1(y_1, a_1) \lor \neg P2(x) \land P1(y_1, a_2)\]

Let A' be the set of clauses

1. \{P1(z, z)\}
2. \{\neg P2(a), \neg P1(y_1, a_1)\}
3. \{P2(a), \neg P1(y_1, a_2)\}

Let DD be the integers, and let phi be defined by

P1.phi = the equality relation
P2.phi = the relation x>y
a_1.phi = 1
a_2.phi = 0
a.phi = 5
y_1.phi = 1

Now A' is phi-unsatisfiable, since clause (2) is false under phi. The theorem implies that

\[(\exists z). P1(z, z) \rightarrow P2(a) \land P1(y_1, a_1) \lor \neg P2(a) \land P1(y_1, a_2)\]

is true under phi.

Def: Let C and D be two clauses. A clause B is a resolvent of C and D if there exist subsets L of C and M of D and substitutions sigmaC and sigmaD, such that
1. \(L \cdot \sigma\text{C}\) and \(M \cdot \sigma\text{D}\) are singletons and the elements of \(L \cdot \sigma\text{C}\) and \(M \cdot \sigma\text{D}\) are complements.

2. \(B = (C \cdot \overline{L}) \cdot \sigma\text{C} \cup (D \cdot \overline{M}) \cdot \sigma\text{D}\). In the text, if we mention that a resolvent is of form (2), it will be assumed that condition (1) holds.

Example: Let \(C\) be

\[\{P(x,a), Q(x,y)\}\]

and let \(D\) be

\[\{\overline{P}(b,x), \overline{P}(y,a), Q(z,x,y)\}\]

Let \(\sigma\text{C}\) be

\[\{b/x, x1/y\}\]

and \(\sigma\text{D}\) be

\[\{a/x, h/y, y1/z\}\]

Then \(L \cdot \sigma\text{C}\) is

\[\{P(h,a)\}\]

\(M \cdot \sigma\text{D}\) is

\[\{\overline{P}(h,a)\}\]

and

\[E = (C \cdot \overline{L}) \cdot \sigma\text{C} \cup (D \cdot \overline{M}) \cdot \sigma\text{D} = \{Q(b,x1), Q(y1,a,b)\}\]

Lemma: Let \(C\) be a clause, \(\sigma\) a substitution, and \(\phi\) a partial assignment that is complete on \(C \cdot \sigma\). Define a partial assignment \((\sigma \cdot \phi)\) such that

1. \((\sigma \cdot \phi)\) agrees with \(\phi\) on constants which belong to \(\text{dom}(\phi)\).

2. If \(w\) is a variable in \(C\),
\[ w((\sigma)\phi) = (w.\sigma)\text{Val}(\phi) \]

[12]

Then \((\sigma)\phi\) is complete on \(C\), and

\[ C.\text{Val}((\sigma)\phi) = (C.\sigma).\text{Val}(\phi). \]

Example: Let \(C\) be \(P(x,y)\) and let \(\sigma\) be 

\( \{f(a,b)/x, x/y\} \).

Then \(C.\sigma\) is

\[ P(f(a,b),x). \]

Let \(DD\) be the natural numbers and let \(\phi\) be the partial assignment defined by:

\[ P.\phi = \text{the } > \text{ relation} \]
\[ f.\phi = \text{the } + \text{ function} \]
\[ a.\phi = 2 \]
\[ b.\phi = 1 \]
\[ x.\phi = 4 \]

Then \((\sigma)\phi\) is the partial assignment defined by

\[ P.\sigma(\phi) = \text{the } > \text{ relation} \]
\[ f.\phi = \text{the } + \text{ function} \]
\[ a.\sigma(\phi) = 2 \]
\[ b.\phi = 1 \]
\[ x.\sigma(\phi) = 1 \]
\[ y.\sigma(\phi) = 4 \]

Note the change in the assignment to \(x\).

Proof (of Lemma [10]): Clearly \((\sigma)\phi\) is complete on \(C\). If \(b\) is an individual constant in \(C\),

\[ b.\text{Val}((\sigma)\phi) = b.(\sigma)\phi \]

[1]
= b.\text{phi}

since \text{sigma} has only variables in its domain. Thus

\[ b.\text{Val}((\text{sigma})\text{phi}) = b.\text{Val}(\text{phi}) \quad [1] \]

\[ = (b.\text{sigma}).\text{Val}(\text{phi}) \]

since \( b.\text{sigma} = b \).

On the other hand, if \( w \) is a variable occurring in \( C \), then

\[ w.\text{Val}((\text{sigma})\text{phi}) = w((\text{sigma})\text{phi}) \quad [1] \]

\[ = (w.\text{sigma}).\text{Val}(\text{phi}) [11] \]

Suppose \( t \) is a term which is not an individual constant or variable. Assume as an inductive hypothesis that for all terms \( t' \) shorter than \( t \),

\[ (t').\text{Val}((\text{sigma})\text{phi}) = ((t').\text{sigma})\text{Val}(\text{phi}) \]

Suppose \( t \) is of form \( f(t_1, \ldots, t_n) \). Then

\[ t.\text{Val}((\text{sigma})\text{phi}) = f(t_1, \ldots, t_n).\text{Val}((\text{sigma})\text{phi}) \]

\[ = (f.((\text{sigma})\text{phi})(t_1.\text{Val}((\text{sigma})\text{phi}), \ldots \]

\[ \ldots , t_n.\text{Val}((\text{sigma})\text{phi})) \]

by inductive hypothesis,

\[ = (f.\text{phi})(t_1.\text{sigma})\text{Val}(\text{phi}), \ldots , (t_n.\text{sigma})\text{Val}(\text{phi})] \]

\[ = f(t_1.\text{sigma}, \ldots , t_n.\text{sigma}).\text{Val}(\text{phi}) \quad [2] \]

\[ = (f(t_1, \ldots , t_n).\text{sigma}).\text{Val}(\text{phi}) \]

Thus the theorem holds for all terms; the proof that it holds for atomic formulas is similar, and that it holds for clauses then follows.

[13]

\textbf{Lemma:} Suppose \( C \) and \( D \) are two clauses, and that \text{phi} is a model for \( C \) and \( D \), where \text{dom}(\text{phi}) contains no variables. Let \( B \) be a resolvent of \( C \) and \( D \), \( B = \)
(C-L) \cdot \Sigma_C U (D-M) \cdot \Sigma_D$. Then $\phi$ is also a model for $B$.

Proof: Let $\phi'$ be any completion of $\phi$ on $C \cdot \Sigma_C U D \cdot \Sigma_D$. (Hence $\phi'$ is complete on $B$). Say that

$$(L \cdot \Sigma_C) \text{Val}(\phi') = F.$$ 

By the preceding lemma, [10],

$$C \cdot \text{Val}(\Sigma_C \cdot \phi') = (C \cdot \Sigma_C) \text{Val}(\phi').$$

But $(\Sigma_C) \phi'$ is a completion of $\phi$ on $C$, and $\phi$ is a model for $C$. Therefore

$$C \cdot \text{Val}(\Sigma_C \cdot \phi') = T,$$

and hence

$$(C \cdot \Sigma_C) \text{Val}(\phi) = T.$$ 

But we have assumed

$$(L \cdot \Sigma_C) \text{Val}(\phi') = F.$$ 

Therefore

$$(C-L) \cdot \Sigma_C \text{Val}(\phi') = T \ [3].$$

But then

$$B \cdot \text{Val}(\phi') = T \ [3].$$

Thus $\phi$ is a model for $B$. The case in which

$$(L \cdot \Sigma_C) \text{Val}(\phi'') = T$$

is treated similarly.

Example: Let $C$ and $D$ be the axioms of symmetry and transitivity respectively, that is, $C$ is

$$\{\neg P(x,y), P(y,x)\},$$

and $D$ is

$$\{\neg P(x,y), \neg P(y,z), P(x,z)\}.$$ 

Then $C$ and $D$ are both satisfied by that assignment which maps $P$ into the equality relation. Let $E$ be a resolvent of $C$ and $D$ obtained by deleting the first literal of $C$ and the third of $D$: $E$ is
\{P(x_1,x_2), \neg P(x_2,x_3), \neg P(x_3,x_1)\}

Then \(E\) is also satisfied by the assignment which maps \(P\) into the equality relation.

**PRIMITIVE PROOFS**

[14]

**Def:** Let \(A\) be a sentence of form

\[(x)(\exists y_1) \ldots (\exists y_n) M(x,y_1, \ldots, y_n),\]

where \(M\) is itself in prenex normal form. (What follows readily generalizes to include sentences of form

\[(x_1) \ldots (x_m)(\exists y_1) \ldots (\exists y_n) M(x_1, \ldots, x_m, y_1, \ldots, y_n),\]

but we use the special case to simplify notation.) Translate \(\neg A\) into Robinson's format for the resolution method. Let \(H\) be the resulting set of clauses. Call \(H\) the set of halt clauses. Suppose that in the course of translating into Robinson's format, the variable \(x\) is replaced by the constant \(a\). Call \(a\) the input constant. Those halt clauses which contain some of the output variables will be called output clauses.

**Def:** Let \(\phi\) be a partial assignment, and let \(\dot{d}\) belong to \(\mathcal{D}\). Then \(\phi \cdot \dot{d}\) is a partial assignment which assigns \(\dot{d}\) to the particular individual constant \(a\) and which agrees with \(\phi\) elsewhere.

**Def:** Let \(H\) be a set of halt clauses for some sentence \(A\), and let \(\mathcal{D}_I\) be a subset of \(\mathcal{D}\), called the input set. Let \(\phi\) be a partial assignment complete on \(A\) which contains no variables in its domain, let \(I\) be such that for all \(\dot{d}\) in \(\mathcal{D}_I\), \(\phi \cdot \dot{d}\) is a model for \(I\). Assume that no output clause belongs to \(I\). Let \(P\) be a resolution proof of \(A\) which takes the set \(H\)
of halt clauses as its initial set. The fertile clauses of \( P \) with respect to \( \phi, I \) and \( DI \) (simply known as the fertile clauses when \( P, I, \phi, \) and \( DI \) are fixed) are defined inductively as follows:

1. If \( B \) is a halt clause, then \( B \) is fertile if and only if it is not a member of \( I \).

2. If \( B \) is a clause of \( P \) which is a resolvent of clauses \( C \) and \( D \) of \( P \), then \( B \) is fertile if and only if \( C \) is fertile or \( D \) is fertile, or both.

Def: The sterile clauses of \( P \) are those clauses which are not fertile.

[15]

Ccr: For each member \( d \) of \( DI \), \( \phi \circ d \) is a model for the sterile clauses of \( P \).

The proof is by induction on the length of the derivation of the clause in \( P \). By definition, \( \phi \circ d \) is a model for the sterile halt clauses of \( P \).

Say \( B \) is sterile but not a halt clause. Then \( B \) is derived from two other sterile clauses \( C \) and \( D \) which are satisfied by \( \phi \circ d \) according to the inductive hypothesis. Then by the lemma, [13], \( B \) is satisfied by \( \phi \circ d \) too. This concludes the proof of the corollary.

Let us assume that a particular fixed proof \( P \) has been chosen for discussion, and that \( DD, \phi, \) and \( I \) are also fixed.

Def: The hot variables of the clauses of \( P \) shall be defined inductively as follows.
1. The hot variables of a halt clause are those output variables, if any, which appear in that clause.

2. Let $B$ be derived from $C$ and $D$ in $P$. Say $B = (C\wedge L)\cdot \sigma_{\text{m}}C \cup (D\wedge M)\cdot \sigma_{\text{m}}D$.

   a. If $w_1$ is a hot variable in $C$, then those variables which occur in both $w_1\cdot \sigma_{\text{m}}C$ and $B$ are hot in $B$. Similarly if $w_2$ is hot in $D$, then those variables which occur in both $w_2\cdot \sigma_{\text{m}}D$ and $B$ are hot in $B$.

   b. If furthermore both $C$ and $D$ are fertile, then those variables which occur in both $L\cdot \sigma_{\text{m}}C$ (or, equivalently, $M\cdot \sigma_{\text{m}}D$) and $B$ are hot in $B$.

Example: Let the theorem $A$ be

$$(x)(\exists y_1)(\exists z) \cdot P_1(z,z) \wedge P_1(y_1,a_1) \lor \neg P_2(x) \wedge P_1(y_1,a_2)$$

where $y_1$ is the only output variable. In Robinson's format, this translates into three clauses

1. $\{P_1(z,z)\}$
2. $\{\neg P_2(a), \neg P_1(y_1,a_1)\}$
3. $\{P_2(a), \neg P_1(y_1,a_2)\}$

These clauses make up the set $H$. Let $DD$ be all the integers, and let $\phi$ be the partial assignment defined by

$P_1, \phi =$ the equality relation

$P_2, \phi =$ the predicate $x > 0$.

$a_1, \phi = 0$

$a_2, \phi = 1$. 
Now let \( P \) be the proof which consists of \( H \) and the clauses

4. \( \neg P1(y1,a1), \neg P2(y2,a2) \) from (2) and (3).
5. \( \neg P1(y2,a2) \) from (1) and (4).
6. \( a \) from (1) and (5).

Let the set \( I \) consist of clause (1). Then all the other clauses in \( P \) are fertile. The hot variables of clauses (2) and (3) are both \( y1 \), since \( y1 \) is the output variable. Both \( y1 \) and \( y2 \) are hot variables in clause (4), while \( y2 \) is the hot variable in clause (5). Clause (6) has no hot variables. Of course clause (1) has no hot variables either; only fertile clauses can have hot variables.

Example: Suppose in some proof, the clause

\[
\{Q(x)\}
\]

has been derived from the two fertile clauses

\[
\{P(x), Q(x)\}
\]

and

\[
\{\neg P(x)\},
\]

where neither clause has any hot variables. Then \( x \) is hot in

\[
\{Q(x)\}
\]

neverless, because the two clauses from which it was derived are both fertile.

[19]

Def: Let \( \text{Prim} \) be any set of symbols which contains all the variables, and the constant \( a \). Let \( \phi \) be a partial assignment which is defined on all the constants in \( \text{Prim} \), other than \( a \), and none of the variables. The elements of \( \text{Prim} \) are said to be primitive. A term or literal is said to be
primitive if all the symbols which occur in it are primitive. Let \( P, DI \) and \( I \) be as before. The proof \( P \) is said to be primitive (with respect to \( Prim \) and \( I \)) if and only if it obeys the following restrictions

1. Let \( B \) be derived from \( C \) and \( D \) (\( B = (C\neg L).\sigma \alpha C \cup (D\neg M).\sigma \alpha D \)). Then if \( w \) is hot in \( C \), \( w.\sigma \alpha C \) must be primitive. If \( w \) is hot in \( D \), \( w.\sigma \alpha D \) must be primitive.

2. If \( C \) and \( D \) are both fertile, \( L.\sigma \alpha C \) must be primitive.

Example: In the preceding example if \( Prim \) includes \( P_2, a_1, a_2 \) and \( a \), the proof \( P \) is primitive. However, if \( Prim \) does not include \( a_1 \), the proof is not primitive, because the derivation of clause (5) involves substituting \( a_1 \) for \( y_1 \) in (4), and \( y_1 \) is hot in (4), but \( a_1 \) is not primitive. This violates condition (1). On the other hand, if \( Prim \) does not contain \( P_2 \), the proof \( P \) is not primitive, because the derivation of clause (4) from clauses (2) and (3) is illegal; clauses (2) and (3) are both fertile, and \( L.\sigma \alpha C \) is \( \{\neg P_2(a)\} \), but \( P_2 \) is not primitive. This violates condition (2) above [21]. Let us assume that the fixed proof \( P \) is primitive for the balance of this chapter.
Def: Let nil be a particular element of DD, and let B be derived from C and D in P.

Let \( x_i \) be an extension of \( \phi_i d \) to the hot variables of B. Then \( x_i \cap \text{nil} \) is that extension of \( x_i \) which maps each variable which occurs in L.\( \sigma \text{mac} \) but not in B into nil.

Def: Let B be derived from C and D in P, \( B = (C \cap L).\sigma \text{mac} \cup (D \cap M).\sigma \text{mac} D) \),

and suppose \( x_i \) is an extension of \( \phi_i d \) to the hot variables of B. Then \( \{\sigma \text{mac} x_i\} \) is the extension of \( \phi_i d \) to the hot variables of C defined by

\[ w.\{\sigma \text{mac} x_i\} = (w.\sigma \text{mac}) \text{Val}(x_i \cap \text{nil}) \]

for each \( w \) hot in C.

Example: Let \( B = a \) in the example above [18], and let \( d = 2 \). Let \( x_i = \phi_i d \). Say C is clause (5),

\[ \{\neg P1(y2, a2)\} \]

and D is clause (1),

\[ \{P1(z, z)\} \]

Then \( \sigma \text{mac} C \) is \( \{a2/y2\} \), and \( \{\sigma \text{mac} x_i\} \) is defined by

P1.\( \{\sigma \text{mac} x_i\} \) = the equality relation
P2.\( \{\sigma \text{mac} x_i\} \) = the relation \( x > 0 \).

\( a1.\{\sigma \text{mac} x_i\} = 0 \)

\( a2.\{\sigma \text{mac} x_i\} = 1 \)

\( a.\{\sigma \text{mac} x_i\} = 2 \)

\( y2.\{\sigma \text{mac} x_i\} = 1 \)

Lemma: In the terminology of the above definition,
(\sigma_{\text{C.xi}}) is defined

Lemma: \( (\sigma_{\text{C.xi}}) \) is defined on each of the hot variables of C:

1. The constants of \( w.\sigma_{\text{C}} \) belong to \( \text{Prim} \), and hence to \( \text{dom}(\phi_{i+d}) \) [19], by definition of primitive proof. [20]

2. The variables which occur in both \( w.\sigma_{\text{C}} \) and \( B \) are hot in \( B \), by definition of hot variable [16], and hence belong to \( \text{dom}(\xi_i) \) [24].

3. The variables which occur in \( w.\sigma_{\text{C}} \) but not in \( B \) belong to \( \text{dom}(\xi_i\text{nil}) \) by definition of \( \xi_i\text{nil} \) [23].

Lemma: Let \( B \) be derived from C and D in P, \( (B = (C-L).\sigma_{\text{C}} U (D-M).\sigma_{\text{D}}) \) and assume both \( C \) and \( D \) are fertile.

[26]

Suppose \( \xi_i \) is an extension of \( \phi_{i+d} \) to the hot variables of \( B \) such that \( B \) is \( \xi_i\text{-unsatisfiable} \), and assume

[27]

\((L.\sigma_{\text{C}})_{\text{Val}}(\xi_i\text{nil}) = F.\)

Then \( C \) is \( (\sigma_{\text{C.xi}})\text{-unsatisfiable} \).

[28]

Proof: First note that the symbols of \( L.\sigma_{\text{C}} \) all belong to \( \text{dom}(\xi_i\text{nil}) \)

1. The constants of \( L.\sigma_{\text{C}} \) are all in \( \text{Prim} \), and hence in \( \text{dom}(\phi_{i+d}) \) and \( \text{dom}(\xi_i\text{nil}) \), by definition of primitive proof [21].

2. The variables of \( L.\sigma_{\text{C}} \) hot in \( P \) are in \( \text{dom}(\xi_i) \) by
hypothesis [28].

3. The variables of \( L.\sigma C \) not hot in \( B \) do not occur in \( B \) at all, by definition of hot variable [17]. Hence they belong to \( \text{dom}(x_i\uparrow\text{nil}) \) by definition of \( x_i\uparrow\text{nil} \) [23].

Now suppose \( C \) is \( [\sigma C.x_i] \)-satisfied by some expansion \([\sigma C.x_i]^*\). Let \( x_i^* \) be any expansion of \( x_i \) on \( B \) which agrees with \([\sigma C.x_i]^*\) on those constants in \( \text{dom}([\sigma C.x_i]^*) \).

Then by the \( x_i \)-unsatisfiability of \( B \), there is some completion \( x_i' \) of \( x_i^* \) on \( B \) such that

\[
B.\text{Val}(x_i') = T.
\]

Then \( (x_i')\uparrow\text{nil} \) is complete on \( C.\sigma C \), since \( C.\sigma C \) is contained in \( B \cup L.\sigma C \), and \( (x_i')\uparrow\text{nil} \) is complete on \( L.\sigma C \) [30], as well as on \( B \). The partial assignment \( \sigma C((x_i')\uparrow\text{nil}) \) [10] is a completion of \([\sigma C.x_i]^*\) on \( C \). Thus, by our supposition,

\[
C.\text{Val}(\sigma C((x_i')\uparrow\text{nil})) = T
\]

[31]. But

\[
C.\text{Val}(\sigma C((x_i')\uparrow\text{nil})) = (C.\sigma C)\text{Val}((x_i')\uparrow\text{nil})
\]

[12]; therefore

\[
(C.\sigma C)\text{Val}((x_i')\uparrow\text{nil}) = T.
\]

As a consequence of a hypothesis to our theorem [29], we know that

\[
(L.\sigma C)\text{Val}((x_i')\uparrow\text{nil}) = T,
\]
since $x_i'$ is an extension of $x_i$. Hence
\[ ((C \land \sigma) \cdot \text{Val}((x_i')^\ast \text{nil})) = T \] [3],

and therefore
\[ ((x_i')^\ast \text{nil}) \cdot \text{Val}(B) = T. \]

But since $(x_i')^\ast \text{nil}$ and $x_i'$ differ only on variables which do not occur in $B$,
\[ B \cdot \text{Val}(x_i') = T. \]

This contradicts the definition of $x_i'$; [32] hence $C$ is not $[\sigma \cdot x_i]$-satisfiable, and this concludes the proof.

Example: In the example we have been examining all along [181], clause (4),
\[ \{ \neg P_1(y_1, a_1), \neg P_1(y_2, a_2) \}, \]
was derived from the two fertile clauses (2),
\[ \{ \neg P_2(a), \neg P_1(y_1, a_1) \} \]
and (3),
\[ \{ P_2(a), \neg P_1(y_1, a_2) \}. \]

Let $d = 2$, and let $x_i$ be that extension of $\phi_i$ such that
\[ y_1. x_i = a_1 \]
and \[ y_2. x_i = a_2. \]

Then, since $P_1.x_i$ is the equality relation, clause (4) is $x_i$-unsatisfiable. We know $\neg P_2(a)$ is false under $x_i$, since $P_2$ maps into the $> 0$ relation and $a$ into 2. Let $C$ be clause (2),
\[ \{ \neg P_2(a), \neg P_1(y_1, a_1) \}, \]
and $\sigma$ the empty substitution. Then $[\sigma \cdot x_i]$ is the extension of $\phi_i$ such that $y_1[\sigma \cdot x_i] = 0$. Then clause (2) is false under $[\sigma \cdot x_i]$, and hence is
[signatureC.xi]-unsatisfiable. On the other hand, let \( d = -2 \) but define \( x_i \) the same way as before. Then \( \{P_2(a)\} \) is false under \( x_i \), so let \( C \) be clause (3),
\[
\{P_2(a), \neg P_1(y_1, a_2)\}.
\]
This time \( \text{signatureC} \) is \( \{a_2/y_1\} \), and so \( [\text{signatureC}, \text{xi}] \) is that extension of \( \text{phi} \) \( d \) such that \( y_1.[\text{signatureC}, \text{xi}] = 1 \). Again, \( C \) is \( [\text{signatureC}, \text{xi}] \)-unsatisfiable.

[33]

Lemma: Let \( B \) be derived from \( C \) and \( D \) in \( \mathcal{P} \)
\[
B = (C\neg L).\text{signatureC} \cup (D\neg M).\text{signatureD},
\]
and assume that \( C \) is fertile and \( D \) is sterile. Let \( x_i \) be an extension of \( \text{phi} \) \( d \) to the hot variables of \( B \) such that \( B \) is \( x_i \)-unsatisfiable. Then \( C \) is \( [\text{signatureC}, \text{xi}] \)-unsatisfiable.

Proof: Assume \( [\text{signatureC}, \text{xi}] \ast [\text{signatureC}, \text{xi}] \)-satisfies \( C \). Let \( x_i \ast \) be any expansion of \( x_i \) on \( B \) such that \( x_i \ast \) agrees with \( [\text{signatureC}, \text{xi}] \) on the constants which belong to \( \text{dom}([\text{signatureC}, \text{xi}]) \). By the \( x_i \)-unsatisfiability of \( B \), there exists a completion \( x_i \) of \( x_i \ast \) on \( B \) such that \( B.\text{Val}(x_i) = F \).

[34]

So for any completion \( x_i \) of \( x_i \) on \( C.\text{signatureC} \cup D.\text{signatureD} \),
\[
B.\text{Val}(x_i) = F.
\]
The partial assignment \( \text{signatureC}((x_i)\uparrow \text{nil}) \) is a completion of \( [\text{signatureC}, \text{xi}] \ast \) on \( C \) such that
\[
C.\text{Val}(\text{signatureC}((x_i)\uparrow \text{nil})) = (C.\text{signatureC})\text{Val}((x_i)\uparrow \text{nil}).
\]

[35]

But
\[
C.\text{Val}(\text{signatureC}((x_i)\uparrow \text{nil})) = 1,
\]
because of our supposition that \([\sigma_{C . x i}]\) satisfies \(C\). Therefore
\[
(C . \sigma_{C}) \text{Val}((x'i)\cdot\text{nil}) = T.
\]
Since \(x'i\) and \((x'i)\cdot\text{nil}\) differ only on variables which do not occur in \(B\),
\[
B . \text{Val}((x'i)\cdot\text{nil}) = F \quad [35]. \tag{36}
\]
But \(B = (C\cdot L) . \sigma_{C} U (D\cdot M) . \sigma_{D}\). Hence
\[
((x'i)\cdot\text{nil}) \text{Val}((C\cdot L) . \sigma_{C}) = F \text{ too} \ [3].
\]
But then it must be the case that
\[
(L . \sigma_{C}) \text{Val}((x'i)\cdot\text{nil}) = T, \quad [37]
\]
\[
(M . \sigma_{D}) \text{Val}((x'i)\cdot\text{nil}) = F \quad [38]
\]
[9]. We know that \(\phi_{F D}\) is a model for \(D\) \([15]\). The partial assignment \(\sigma_{D}((x'i)\cdot\text{nil})\) is a completion of \(\phi_{F D}\) on \(D\),
and therefore
\[
D . \text{Val}(\sigma_{D}((x'i)\cdot\text{nil})) = T.
\]
But \(\sigma_{D}((x'i)\cdot\text{nil})\) is so constructed that
\[
L . \text{Val}(\sigma_{D}((x'i)\cdot\text{nil})) = (D . \sigma_{D}) \text{Val}((x'i)\cdot\text{nil}) \quad [12], \quad [39]
\]
and hence
\[
(D . \sigma_{C}) \text{Val}((x'i)\cdot\text{nil}) = T.
\]
Now we have shown that
\[
(M . \sigma_{D}) \text{Val}((x'i)\cdot\text{nil}) = F. \quad [38].
\]
Therefore
\[
((D\cdot M) . \sigma_{D}) \text{Val}((x'i)\cdot\text{nil}) = T. \quad [39, 3]
\]
But since \(B = (C\cdot L) . \sigma_{C} U (D\cdot M) . \sigma_{D}\), we have
\[
B . \text{Val}((x'i)\cdot\text{nil}) = T,
\]
contradicting [37]. Hence our assumption is false, and C is \( \{\sigma C, x_1\} \)-unsatisfiable.

**Example:** In the proof above, [18], clause (5) is derived from clauses (4) and (1), where (4) is fertile and (1) is sterile. Let \( \phi \) be as before, and let \( d \) be an arbitrary integer. Let \( x_1 \) be that extension of \( \phi \odot d \) which maps \( y_2 \) into 1. Since \( P_1 \) is mapped into the equality relation and \( a_2 \) maps into 1, clause (5) is false under \( x_1 \), and hence \( x_1 \)-unsatisfiable. Now \( \sigma C \) is the substitution \( \{a_1/y_1\} \), and so \( \{\sigma C, x_1\} \) is that extension of \( \phi \odot d \) which maps \( y_1 \) into 0 and \( y_2 \) into 1. Then (4) is false under \( \{\sigma C, x_1\} \).

**MAIN THEOREM**

**Def:** For any \( d \) in \( DI \), the input set, define the \( d \)-execution to be a set of pairs \((E(i), \text{par}(i))\), \( i = 1, \ldots, k_d \), where \( E(i) \) is a fertile clause and \( \text{par}(i) \) is an extension of \( \phi \odot d \) to the hot variables of \( E(i) \) defined inductively as follows:

1. \( E(\emptyset) = \sigma; \text{par}(\emptyset) = \phi \odot d. \)

(Note that \( \sigma \) must be fertile, because \( \phi \odot d \) is a model for all the sterile clauses, but \( \sigma \) has no model.)

[40]

2. Suppose \( E(i) \) is derived from two fertile clauses \( C \) and \( D \) in \( P \),

\[
E(i) = (C \odot L).\sigma C \cup (D \odot M).\sigma C.
\]

That \( \text{par}(i) \) is complete on \( L.\sigma C \) is proven in a lemma. [37] Say
(L.\text{sigma}C)\text{Val}(\text{par}(i)\text{\texttt{nil}}) = F.

Then $E(i+1) = C$ and $\text{par}(i+1) = [\text{sigma}C.\text{par}(i)]$. On the other hand, if

$$(L.\text{sigma}C)\text{Val}(\text{par}(i)\text{\texttt{nil}}) = T,$$

then $E(i+1) = D$ and $\text{par}(i+1) = [\text{sigma}D.\text{par}(i)]$. We know $\text{par}(i+1)$ is defined on all the hot variables of $E(i+1)$ [26].

Suppose $E(i)$ is derived from two clauses $C$ and $D$ in $P$, but that only one of them, say $C$, is fertile. Then define $E(i+1) = C$ and $\text{par}(i+1) = [\text{sigma}C.\text{par}(i)]$.

[41]

Since, for each $i$, $E(i+1)$ is earlier in $P$ than $E(i)$, the process must terminate, and it does so when $E(i)$ is a halt clause.

[42]

Then if $E(kd)$ is this last clause in the d-execution, define $\text{fn}(d) = \text{par}(kd)$.

Example: In our proof P [18], let $d$ be any positive integer. Then the $d$-execution will be

$$
((6), \phi_1d),

((5), \text{that extension of } \phi_1d \text{ which maps } y_2 \text{ into } 1),

((4), \text{that extension of } \phi_1d \text{ which maps } y_2 \text{ into } 1 \text{ and } y_1 \text{ into } 0),

((2), \text{that extension of } \phi_1d \text{ which maps } y_1 \text{ into } 0).
$$

Then $\text{fn}(d)$ is that extension of $\phi_1d$ which maps $y_1$ into 0.

If $d$ is less than or equal to $C$, the $d$-execution is identical
to the case in which d is positive, except for the last pair:

((3), That extension of phi \&
which maps y1 into 1).

So fn(d) is that extension of phi \& which maps y1 into 1.

In determining the final member of the sequence, recall that clause (4),

\{-P1(y1,a1), ~P1(y2,a2)\}

was derived from clauses (2),

\{-P2(a), ~P1(y1,a2)\},

and (3),

\{P2(a), ~P1(y1,a2)\}.

The literal ~P2(a) takes the place of L.sigmac. Then
(L.sigmac).par(2) is ~(d>0). This statement is false in the case d>0; hence E(3) = clause (2); ~(d>0) is true when d is non-positive; hence E(3) = clause (3).

Lemma: For each i, i = 0, ..., kd, E(i) is par(i)-unsatisfiable.

Proof: By induction on i.

1. For i = 0, we know NIL is tau-unsatisfiable for any tau.

2. Assume E(i) is par(i)-unsatisfiable. Suppose E(i) is derived from two clauses E(i+1) and D in P. Say

E(i) = (E(i+1) \&L).sigmas U (D\&M).sigmad.

2.1 Suppose E(i+1) and D are both fertile. Then by the definition of \&-execution,

(L.sigmac)Val(par(i)\&nil) = F. [40]

Therefore, by a lemma [27], E(i+1) is par(i+1)-unsatisfiable.
2.2 Suppose $E(i+1)$ is fertile but $D$ is not. Then immediately by another lemma, $E(i+1)$ is $\text{par}(i+1)$-unsatisfiable. [34].

[43]

Cor: Let $H$ be the set of halt clauses of $P$. Then $H$ is $\text{fn}(d)$-unsatisfiable.

Proof: By the lemma, $E(kd)$ is $\text{par}(kd)$-unsatisfiable. But $E(kd)$ is a halt clause [41]; therefore $H$ is $\text{par}(kd)$-unsatisfiable, i. e. $\text{fn}(d)$-unsatisfiable.[42]

Theorem: Preserve the notation at the beginning of the section on PRIMITIVE PROOFS [14]. If there is a primitive proof $P$ for $A$, define $\text{fn}$ as above [42]; Then $M(a, y_1, \ldots, y_n). \text{Val}(\text{fn}(d)) = T$.

Proof: The halt clauses of $P$ are $\text{fn}(d)$-unsatisfiable. [43] Therefore, by a theorem on the translation into Robinson's format [7],

$M(a, y_1, \ldots, y_n). \text{Val}(\text{fn}(d)) = T.$

Example: In the preceding example, $n = 1$ and $M(a, y_1)$ is

$(\exists z). P(z, z) \rightarrow P2(a) \land P1(y_1, a) \lor \neg P2(a) \land P1(y_1, a_2).$

Let $d$ be positive. Then $M(a, y). \text{fn}(d)$ is

$(\exists z). (z = z \rightarrow d > 0 \land 0 = 6 \lor \neg (d > 0) \land 0 = 1).$

This statement is true. If $d$ is less than or equal to 0,

$M(a, y). \text{fn}(d)$ is

$(\exists z). (z = z \rightarrow d > 0 \land 0 = 1 \lor \neg (d > 0) \land 1 = 1).$

This statement is true too. Thus

$M(a, y_1). \text{Val}(\text{fn}(d)) = T$

in both cases.
Cor: Introduce new function symbols $q_1, \ldots, q_n$, and let $\phi^*$ be an extension of $\phi$ such that $q_i.\phi^*$ is a function mapping $DD$ into $DD$ such that

$$q_i.\phi^*(b) = y_i.fn(b) \quad [44]$$

for all $b$ in $DD$. Then $M(x,q_1(x),\ldots,q_n(x))$ is $\phi^*$-valid.

Proof: Let $\phi^*$ be any completion of $\phi$ on $M(x,q_1(x),\ldots,q_n(x))$, and suppose $x.\phi^* = d$, where $d$ is in $DD$. We know

$$M(a,y_1,\ldots,y_n).Val(fn(d)) = T$$

by the theorem. But

$$x.\phi^* = d = a.fn(d) \quad [45]$$

and $q_i(x).Val(\phi^*) = (q_i.\phi^*)(x.\phi^*) \quad [2]$

$$= (q_i.\phi^*)(d) \quad [45]$$

$$= y_i.(fn(d)) \quad [44]$$

Consequently

$$M(x,q_1(x),\ldots,q_n(x)).Val(\phi^*)$$

$$= M(a,y_1,\ldots,y_n).Val(fn(d)) = T,$$

so we may conclude $M(x,q_1(x),\ldots,q_n(x))$ is $\phi^*$-valid.

Example: in the example which we have been following all along, \[18\],

$$q_1.\phi^*(b) = y_i.fn(b) = 0 \text{ if } b > 0$$

$$= 1 \text{ if } \sim(b > 0)$$

$M(x,q_1(x)) \phi^*$ stands for

$$\exists z(z = z \Rightarrow x > 0 \land [\text{if } x > 0 \text{ then } 0 \text{ else } 1] = 0$$

$$\lor \sim(x > 0) \land [\text{if } x > 0 \text{ then } 0 \text{ else } 1] = 1),$$

which is true.

The functions $q_i$ make up the program. The definition of
d-execution gives us explicit rules for computing these functions $g_i$, assuming that a processor exists capable of computing those constants in the set Prim.
CHAPTER 3. LOOPS

All the programs we have been able to generate so far have been loop-free. This is not surprising: all proofs are loop-free; they have a characteristic tree-like structure. However, nearly all interesting problems require programs with loops to solve them. Clearly the methods outlined do not suffice for actual program writing.

As it turns out, a simple extension to the technique allows us to write iterative and recursive programs. That method is the employment of some rules of inference corresponding to the axiom of mathematical induction. Suppose we are writing programs whose variables represent natural numbers. Then the theorems we must prove are from number theory. Formulations of number theory involve some version of the axiom schema of mathematical induction. This presents a problem because the schema must be considered either as a second order axiom or as an infinite set of first order axioms. The resolution principle applies only to first order theories, but it is impossible to include an infinite set of axioms in the list of clauses. The solution we adopt must be regarded as a compromise, and it is hoped that as new ways of treating higher-order theories or set theory arise they may be adapted to write programs.

We choose to apply the mathematical induction scheme as a new rule of inference rather than as an axiom. We will
treat the least-number principle in the same way. These rules will represent only a subset of the instances of the corresponding axiom schemata. However, we will see that in a certain sense these rather limited rules are sufficient.

For the discussion we will introduce those rules necessary for writing programs with one input variable. These rules immediately generalize when it is desired to write functions of several variables.

**INDUCTION**

The mathematical induction schema usually has the form

$$P(0) \land (a)(P(a) \Rightarrow P(a+1)) \Rightarrow (x)P(x).$$

In our treatment the expression $P(a)$ will be of form

$$(\exists y)R(c,a,y).$$

We shall actually apply the contrapositive of the axiom: we will show that

$$(\exists y)R(c,a,y)$$

contradicts our axioms by first refuting

$$(\exists y)R(c,0,y)$$

and then

$$(a). (\exists b)R(c,a,b) \Rightarrow (\exists y)R(c,a+1,y).$$

Thus induction is applied as a rule of inference as follows: Suppose a fertile clause $C$ of form:

$$\{ \neg R(x,a,y), \neg \neg A(y) \}$$

is among the list of clauses, where $R(x,a,y)$ is an atom, where $A(y)$ is a disjunction of literals, perhaps empty, and $x$
does not occur in \( A(y) \). Also assume that \( y \) is a hot variable. Then the theorem prover tries to find primitive refutations of the following two subsidiary sets.

1. The sterile clauses of the main refutation and the fertile clause

\[ \{ \neg R(c, 0, y) \} \]

where \( c \) is an input constant and \( y \) is the output variable.

2. The sterile clauses of the main refutation, the additional sterile clause

\[ \{ R(c, a, b) \} \]

and the fertile clause

\[ \{ \neg R(c, a+1, y) \} \]

where \( c, a, \) and \( b \) are input constants, \( b \) does not occur elsewhere in the set of clauses, and \( y \) is the output variable.

In refuting the two sets the program writer constructs two programs \( q(c) \) and \( h(c, a, t) \). If both subsidiary refutations are completed, we derive the clause

\[ \{ A(f(x, a)) \} \]

in the main refutation, where \( f \) is a new primitive function symbol representing the function defined by

\[ f(z, 0) = q(z) \]
\[ f(z, x+1) = h(z, x, f(x)). \]

The rationale for this rule of inference is as follows: \( f \) is such that \( R(c, a, f(c, a)) \) is true, and hence \( \neg R(c, a, f(c, a)) \) is false, and thus \( \neg F(x, a, y) \) may be dropped
from the disjunction \( C \) after the substitution of \( f(x,a) \) for \( y \) in \( A(y) \) is made. More precisely, we show that \( \neg R(x,a,f(x,a)) \) is inconsistent with the sterile clauses in the proof by induction on \( a \). Hence from clause \( C \) and the other clauses we can infer

\[ \{ A(f(x,a)) \} \]

In the case \( a=0 \), we have \( f(x,a) = f(x,0) = g(x) \). But \( g \) has been constructed from the refutation (1); it has the property that \( \neg R(x,0,g(x)) \) is false for any \( x \) in any model which satisfies the sterile clauses. Hence \( \neg R(x,0,g(x)) \) (i.e. \( \neg R(x,a,f(x,a)) \)) is inconsistent with the sterile clauses. Now suppose for some \( a \) and \( x \) that \( \neg R(x,a,f(x,a)) \) is inconsistent with the sterile clauses; then, if \( M \) is any model which satisfies the sterile clauses, \( M \) satisfies \( R(x,a,f(x,a)) \). But the function \( h \) is such that any model which satisfies the sterile clauses and \( R(x,a,b) \) cannot be extended to falsify \( \neg R(x,a+1,h(x,a,b)) \). Thus any completion of \( M \) falsifies \( \neg R(x,a+1,h(x,a,f(x,a))) \), that is \( \neg R(x,a+1,f(x,a+1)) \), and the latter is inconsistent with the sterile clauses, Thus the induction step is proven.

The program associated with the new clause \( \{ A(f(x,a)) \} \) consists of the assignment

\[ y \leftarrow f(x,a) \]

followed by the program associated with the old clause \( C \).

For example, let us consider how the factorial function would be written. Let us use the predicate "Fact" of two
variables such that \( \text{Fact}(x,y) \) is to mean that \( y = x! \). Then a set of axioms describing "Fact" might be

1. \( \{\text{Fact}(0,1)\} \)

2. \( \{\neg\text{Fact}(x,y), \text{Fact}(x+1, (x+1)!y)\} \)

Clause (2) corresponds to

\[ \text{Fact}(x,y) = \text{Fact}(x+1, (x+1)!y) \]

Clauses (1) and (2) are sterile. The theorem to be proven is

\[ (a)(3y)\text{Fact}(a,y) \]

The negation of the theorem, in LDP, is the fertile clause

3. \( \{\neg\text{Fact}(a,y)\} \).

Let us apply the induction scheme to clause (3); the variable \( x \) does not occur in this case, and \( \Lambda(y) \) is the empty set. The system generates two subproblems; the first is to refute the set consisting of the sterile clauses (1) and (2) and the fertile clause

4. \( \{\neg\text{Fact}(0,y)\} \).

Clause (4) clearly contradicts clause (1), giving a program with no input variables which always produces output 1:

\[ q() = 1. \]

The second set of clauses consists of the sterile clauses (1) and (2), an additional sterile clause

5. \( \{\text{Fact}(a,b)\} \),

and a new fertile clause

6. \( \{\neg\text{Fact}(a+1,y)\} \).

Clauses (5) and (2) may be resolved, deleting the first
literal of (2), giving

7. \( \{ \text{Fact}(a+1, (a+1)\times b) \} \).

Then clauses (6) and (7) are contradictory. The program derived from the proof is then

\[ h(x, u) = (x+1)\times u. \]

Since both refutations have been completed, we may conclude

8. \( \{ A(f(x, a)) \} \),

but since \( A(y) \) was the empty set, (8) is the empty clause, and the refutation is complete.

The program derived from this proof is

\[ y = f(a) \]

where

\[ f(0) = 1 \]
\[ f(x+1) = (x+1)\times f(x), \]

which is a program for the factorial function. Since the definition of \( f \) is recursive, we have introduced a recursive loop into the program.

Note that the theorem prover in executing a proof should not devote all its time to refuting subsidiary sets: if a subsidiary set is in fact not primitively unsatisfiable, the theorem prover may run forever trying to refute it, thus ignoring another, possibly successful avenue of refutation.

THE LEAST-NUMBER PRINCIPLE

In ordinary arithmetic, the least-number principle
follows from the induction axiom. However, using primitive resolution there are programs that may be written assuming the least-number principle that may not be written without it, as will be shown later. Therefore we must assume the principle as an axiom or a rule of inference.

The least-number principle has the form
\[(\exists y) P(y) \Rightarrow (\exists u), P(u) \land (h), P(h) \Rightarrow h \geq a.\]

In our case we assume \(P(y)\) has form
\[R(x, y, c).\]

We use the principle as a rule of inference; we try to refute the consequent by refuting the antecedent. Suppose two clauses of the following forms belong to our list of fertile clauses:

\[C_1: \{\neg R(x, u, c), R(x, h(u), c), A_1(x, u)\}\]
\[C_2: \{\neg R(y, u, c), h(u) < u, A_2(y, u)\}\]

where \(P\) is a predicate letter, \(x\) and \(y\) are either both variables or both the same constant, \(h\) is a new primitive function symbol and \(A_1(x, u)\) and \(A_2(y, u)\) are disjunctions of literals.

In order to apply the rule, we first try to prove
\[(x) (\exists y) R(x, y, c)\]
from our axioms and hypotheses. Then we try to establish that \(R(x, u, c)\) is a recursive predicate in \(x\) and \(y\). We must refute two sets:

1. The first set consists of the sterile clauses of the main refutation and the clause \(\neg R(c, y, c)\). The proof need not
be primitive; it is merely the existence of \( y \) we need to establish.

2. The second set consists of the sterile clauses of the main refutation, which are considered sterile in the new set too, and two new fertile clauses

\[
\{ \neg R(a, b, 0), \neg (y = 0) \}, \text{ and}
\{ F(a, b, 0), \neg (y = 1) \},
\]

where \( a \) and \( b \) are input variables and \( y \) is the output variable. This corresponds to the theorem

\[
(a) (b) (\exists y) . R(a, b, 0) \land y = 0 \lor \neg R(a, b, 0) \land y = 1.
\]

The proof must be primitive; the theorem then guarantees that \( R(x, y, 0) \) is a recursive predicate. In fact for any model \( M \) which satisfies the sterile clauses, we derive the definition of a function \( g \) with the property that \( g(a, b) = 0 \) if and only if the predicate \( R(a, b, 0) \) is true in \( M \).

If both these refutations are obtained, then the rule allows us to derive the clause

\[
C: \{ A1(x, f(x)), A2(x, f(x)) \},
\]

where \( f \) is a new primitive function symbol. This conclusion actually is sound, for let \( M \) be any model for number theory and the clauses of the main proof, including \( C1 \) and \( C2 \), such that \( f \) is assigned the function \( \text{min} y \ [g(x, y) = 0] \). We know that the function is defined, since we have proven \( (x)(\exists y)R(x, y, 0) \) from the sterile clauses, and we have defined \( g \) so that \( g(a, b) = 0 \) if and only if \( R(a, b, 0) \) is true in \( M \). Suppose \( C \) is not \( M \)-valid. Then we attempt to show that \( M \) does
not satisfy C1 or C2, contradicting our assumptions. We know M does not satisfy \( \{A_1(x,u)\} \) or \( \{A_2(y,u)\} \) since it can be extended to make the same assignment to \( x \) and \( y \), and to assign the value of \( f(x) \) to \( u \). We also know that the same extension of \( M \) falsifies \( \sim R(x,f(x),0) \), and hence \( \sim R(x,u,0) \) and \( \sim R(y,u,0) \). Since we have assigned to \( u \) the least number such that \( R(x,u,0) \) is true in the extension of \( M \), we know that for any number assigned to \( h(u) \) either \( R(x,h(u),0) \) is false in the assignment, in which case C1 is falsified, or \( h(u) \geq u \) is true, in which case C2 is falsified. This contradicts our hypothesis, and so the rule of inference is sound.

We must also describe the program associated with the new node. Assume prog1 is associated with C1, and prog2 is associated with C2. Then the program associated with C is
This function is recursive because we have required that $h(u)$ be a primitive term, and hence recursive.

**LIST INDUCTION**

In order to write LISP programs (or to prove theorems about LISP functions) it is necessary to have a version of the induction axiom that applies to lists. There are a number of such axioms; we will speak of one of them.

Suppose a fertile clause $C$ is among the list of clauses, and is of form

$$\{ \neg R(x, a, y), \Lambda(y) \},$$

where $R(c, a, y)$ is an atom, and $\Lambda(y)$ is a disjunction of literals, perhaps empty. Assume $y$ is a hot variable. Then the theorem prover tries to find primitive refutations of the
following two subsidiary sets:

1. The sterile clauses of the main refutation and the fertile clause

\[ \neg \mathcal{R}(c, \text{NIL}, y) \]

where \( c \) is the input constant, \( \text{NIL} \) is the empty list of LISP, and \( y \) is the output variable.

2. The sterile clauses of the main refutation, the additional sterile clause

\[ \{ \mathcal{F}(c,a,b) \} \]

and the fertile clause

\[ \neg \mathcal{R}(c,\text{cons}(d,a),y) \]

where \( c, a, b, \) and \( d \) are input constants, and \( y \) is the output variable. In refuting the two sets, the program writer constructs two programs \( g(c) \) and \( h(c,a,d,b) \). If both subsidiary refutations are completed, we derive the clause \( \{ \mathcal{A}(f(x,a)) \} \) in the main refutation, where \( f \) is a new function symbol defined by

\[
f(z,x) = g(z) \text{ if } x = \text{NIL} \]

\[
= h(z, \text{cdr}(x), \text{car}(x), f(z, \text{cdr}(x))) \text{ otherwise}
\]

The rationale for this rule of inference is analogous to that for the number theoretic rule, except that it relies on the following principle:

If \( P(x) \) is a property such that \( P(\text{NIL}) \) and such that \( P(x) \) implies \( P(\text{cons}(y,x)) \) for all \( x \) and \( y \), then \( P(x) \) for all \( x \).

This version of the list induction axiom is used in
writing such functions as the MEMBER function.

This completes the discussion of the introduction of the mathematical induction principle and the least-number principle as rules of inference. We will assume henceforth that the theorem prover somehow includes these rules among its inferential machinery.
CHAPTER 4. COMPLETENESS

In restricting ourselves to primitive resolution we have weakened the power of our theorem prover. There are some valid sentences that we will not be able to prove. For instance, if \( P \) is not a primitive predicate,

\[
(a)(\exists y). P(a) \land y=1 \lor \neg P(a) \land y=0,
\]

is not primitively provable, where \( a \) is an input variable and \( y \) is an output variable. Thus, a primitive theorem prover is not complete. However, the usual formulation of completeness is not appropriate to a program-writing program: what we are really interested in is the class of programming problems the system can solve, not the class of theorems its theorem prover can prove.

We would like to be able to prove the following assertion: For any correct specification of any recursive function, the program writer will be able to construct a program that computes that function. This theorem is unfortunately too strong: it is impossible to write such a program writer for theoretic reasons. In the section on LIMITATIONS we will show that it is possible for the user to write down a perfectly good description of a constant function, and for the program writer to be unable to solve the problem, regardless of how much time and space it is given, even though another statement of the same problem is solved easily. Furthermore, this sensitivity to the representation of the problem is characteristic of all
program writers, regardless of the techniques they use. On the other hand, the only examples we have found of such phenomena are extremely pathological. We have not found any examples of problems that a user would be likely to pose and that lead to theorems that were not primitively provable.

We are able to prove that for any recursive function, there exists some way of specifying it so that the theorem prover will be able to prove the relevant theorem primitively, and thereby write a program to compute the function. This property we will call completeness with respect to program writing.

As usual, we will speak about functions of one variable even though what is said generalizes easily to functions of several variables. We will also confuse numbers with the symbols which represent them in the language. This does not seem to cause too much disorder in this application.

THE COMPLETENESS THEOREM

Before stating the theorem we must introduce a few terms.

Def: We say that a clause C is primitively inconsistent with a set S of clauses if there exists a primitive refutation of the set whose insipid clauses are the elements of S and whose sole vital clause is C.

In speaking of primitivity in conjunction with
number-theoretic proofs, we will assume that the constant \( \lambda \), the successor function, the equality predicate and the predicate \( \geq \) are in the set of primitives.

Def: Let \( f \) be any function. We say that \( f \) is primitively representable if there exist a binary predicate \( F \) and a set of clauses \( CF \) such that

1. if \( a \) is a constant not occurring in \( CF \), and \( y \) any variable, the clause \( \{\neg F(a,y)\} \) is primitively inconsistent with \( CF \) when \( a \) is the input constant and \( y \) the output variable,

2. The function \( f' \) defined from the refutation is equal to \( f \) (we call \( f' \) the program for \( f \)), and

3. If \( M \) is any model for number theory and for \( CF \) which includes \( \{a\} \) in its domain, and if \( \phi \) is a completion of \( M \) such that

\[
\{\neg F(a,y)\} \text{Val}(\phi) = T,
\]

then

\[
y.\phi = f(a,\text{Val}(\phi)).
\]

We then say that \( F \) and \( CF \) primitively represent \( f \), and we call \( F \) the predicate for \( f \), and \( CF \) the hypotheses for \( F \) or \( f \).

Condition (3) insures that it is impossible to find a refutation of \( \{\neg F(a,y)\} \) from which may be derived an incorrect program for \( f \).

Theorem (Completeness): Every recursive function is
primitively representable.

Proof: The proof is by induction on the "length" of the definition of \( f \). The recursive functions are generated in the following manner:

The four initial functions are recursive

1. \( n(x) = x + 1 \).
2. \( t(x, y) = x \cdot y \), the product of \( x \) and \( y \).
3. \( m(x, y) = x - y \), the difference between \( x \) and \( y \). (= 0 if \( y > x \)).
4. The functions \( \text{uni}(x_1, \ldots, x_n) = x_i \), the \( i \)-th projection of \( n \) variables.

Furthermore,

5. If \( g(y) \) and \( h(x) \) are recursive then so is the function \( f(x) = g(h(x)) \) obtained by substitution.

6. If \( g(x, y) \) is recursive and if for all \( x \) there exists a \( y \) such that \( g(x, y) = 0 \), then the function \( f(x) = \min y \) \([g(x, y) = 0]\) is recursive, and we say that \( f \) is obtained from \( g \) by minimization.

Note: Our primitive basis follows that of Davis [Davis], except for the elimination of the redundant addition function \( x+y \) (which equals \((x+1) \cdot (y+1) - ((x \cdot y) + 1)\).) The substitution should be stated for \( g \) and \( h \) of several variables, but the proof in that case is the same.

It is easy to show that the initial functions are primitively representable. The successor function is
represented by the predicate $N(x, y)$ and the hypotheses

1. $\{N(x, n(x))\}$
2. $\{\neg N(x, y), y=n(x)\}$

which correspond to the sentence

$$(x), N(x, n(x)) \land (y), N(x, y) \Rightarrow y=n(x)$$

The clause

3. $\{\neg N(a, y)\}$

primitively contradicts the first hypothesis. The function derived from this refutation is the successor function.

Suppose $M$ is a model for number theory which satisfies (1) and (2). Let $\phi$ be a completion of $M$ such that

$$\{(N(a, y))\} Val(\phi) = T.$$ We know that $M$ satisfies (2). Therefore we know

$$\{(\neg N(a, y), y=n(a))\} Val(\phi) = T,$$

because the clause is an instance of (2). But since

$$\{(\neg N(a, y))\} Val(\phi) = F,$$

we have

$$\{y=n(a)\} Val(\phi) = T.$$ Thus

$$y, \phi = a, \phi + 1.$$ The product function and the difference function are handled in the analogous way, since they are both functions which we know to be computable. The projection functions are represented by the predicates $\text{Uni}$ and the sets of hypotheses

1. $\{\text{Uni}(x_1, \ldots, x_n, x_i)\}$
2. $\{\neg \text{Uni}(x_1, \ldots, x_n, y), y=x_i\}$
The proof is again similar to that for the zero function.

Suppose \( g(x) \) and \( h(y) \) are recursive and primitively represented by predicates \( G(x,y) \) and \( H(y,z) \) respectively, and
\[
f(x) = h(g(x))
\]
is obtained by substitution. Then \( f \) is represented by the predicate \( F(x,z) \) and hypotheses consisting of those for \( G \) and \( H \) and also the clauses
\[
1. \{ \neg G(x,y), \neg H(y,z), F(x,z) \}
2. \{ \neg F(x,z), G(x,k(x,z)) \}
3. \{ \neg F(x,z), H(k(x,z),z) \}
\]
which correspond to the sentence
\[
(x)(z), (\exists y)(G(x,y) \land H(y,z)) \equiv F(x,z)
\]
The refutation of the clause
\[
4. \{ \neg F(a,z) \}
\]
from the hypotheses proceeds as follows:
\[
5. \{ \neg G(a,y), \neg H(y,z) \} \text{ from (1) and (4)}.
\]
Now we know that the clause \( \{ \neg H(t,z) \} \) is primitively inconsistent with the set \( CH \), the hypotheses for \( h \), and hence there is a primitive refutation of
\[
CH \cup \{ \neg H(t,z) \}.
\]
Clause (5) contains the literal \( \neg H(y,z) \). Suppose we carry out the refutation using (5) instead of \( \{ \neg H(t,z) \} \). Then each clause in the new proof will contain all the literals of the corresponding clause of the old proof, perhaps with variables instead of occurrences of \( t \), but replacing a constant with a
variable will not interfere with any inference. However, each new clause may have one or more variants of \( \neg \exists a, y \) among its literals. Consequently the clause in the new proof corresponding to the empty clause in the old proof will consist of a number of variants of \( \neg G(a, y) \). By factoring, the unit clause

6. \{\neg G(a, y)\}

is derived. The program derived from the old proof is \( h(b) \). Thus the program associated with clause (6) is \( h(y) \). Now (6) is primitively inconsistent with \( CG \), and from that refutation will be derived a program which sets \( y \) to \( g(a) \). Thus the program associated with the entire refutation of \{\neg F(a, z)\} sets \( z \) to \( h(g(a)) \), just as we had hoped.

It is necessary to verify condition (3) of the definition of primitive representation. Let \( M \) be a partial assignment which satisfies \( CF \), which is a model for number theory, and which includes \( a \) in its domain, and let \( \phi \) be a completion of \( M \) which thus includes \( y \) in its domain. Assume

\[
\{F(a, z)\} Val(\phi) = T.
\]

We know that \( M \) is a model for (2),

\[
\{\neg F(x, z), G(x, k(x, z))\}.
\]

Hence

\[
\{\neg F(a, z), G(a, k(a, z))\} Val(\phi) = T.
\]

But then

\[
\{G(a, k(a, z))\} Val(\phi) = T,
\]

and so by condition (3) for \( g \),
(k(a,z)) Val(phi) = g(a, Val(phi)).

Since M satisfies (3),

\(\{\neg F(a,z), H(k(a,z), z)\}\) Val(phi) = T,

and thus

\(\{H(k(a,z), z)\}\) Val(phi) = T,

and hence

\(z \cdot \text{Val}(\phi) = h((k(a,z))\text{Val}(\phi))\).

Combining our results, we obtain

\(z \cdot \text{Val}(\phi) = h(q(a, \text{Val}(\phi))) = f(a, \text{Val}(\phi))\),

which shows that condition (3) is satisfied here.

Suppose \(f\) is defined by the min operator:

\(f(x) = \min y \{g(x,y) = 0\}\),

where \(g\) is a recursive function such that for each \(x\) there exists a \(y\) such that \(g(x,y) = 0\). We wish to prove that there exists a primitive representation for \(f\). By inductive hypothesis there exists a primitive representation \(G\) for \(g\), with hypotheses CG. Let our hypotheses for \(F\) be those for \(G\) with additional clauses:

1. \(\{\neg G(x,y,0), G(x, l(x,y), 0), F(x,y)\}\)
2. \(\{\neg G(x,y,0), l(x,y)<y, F(x,y)\}\)
3. \(\{\neg F(x,y), G(x,y,0)\}\)
4. \(\{\neg F(x,y), \neg G(x,u,0), \neg (u<y)\}\)
5. \(\{G(x, k(x), 0)\}\)
6. \(\{\neg G(x,y,z1), \neg G(x,y,z2), z2=z1\}\)
7. \(\{\neg G(x,y,z1), G(x,y,z2), \neg (z1=z2)\}\)
8. \(\{x=x\}\)
Clauses (1) through (4) correspond to the sentence

\[(x)(y) \cdot P(x,y) \equiv G(x,y,C) \land (u).G(x,u,0) \Rightarrow \neg (u<y)\]

Clause 5 corresponds to

\[(x)(\forall y)G(x,y,y).\]

Clause (6) states that \(G\) has the functional property:

\[(x)(y)(z1)(z2).G(x,y,z1) \land G(x,y,z2) \Rightarrow z2=z1\]

Clause (7) asserts one instance of the substitutivity of equality:

\[(x)(y)(z1)(z2).z1=z2 \land G(x,y,z1) \Rightarrow G(x,y,z2)\]

Clause 8 states the reflexivity of equality.

The function \(l\) is primitive. These hypotheses and \(P\) then primitively represent \(f\). Taking clause

9. \(\{\neg P(a,z)\}\)

as the vital clause, with input constant \(a\) and output variable \(z\), we can derive

10. \(\{\neg G(a,z,0), G(a,l(a,z),0)\}\) from (1) and (9),

11. \(\{\neg G(a,z,0), l(a,z)<0\}\) from (2) and (9).

These clauses are in the proper form for application of the least-number principle. In applying the rule, the theorem prover tries to refute two sets:

a. The hypotheses and the clause

12. \(\{\neg G(c,y,0)\}\)

The proof need not be primitive. But clause (12) contradicts clause (5), concluding the refutation.

b. The hypotheses and the vital clauses
13. \{\neg G(a, b, c), \neg (y=0)\}
14. \{G(a, b, c), \neg (y=1)\}.

Derive
15. \{\neg G(a, b, c)\} from (8) and (13), and
16. \{G(a, b, c)\} from (8) and (14).

Notice that a contradiction may not be derived from (15) and (16), since they are both vital clauses and \( G \) is not primitive.

17. \{\neg G(a, b, z2), z2=0\} from (6) and (16), deleting the first literal of (6).
18. \{\neg G(a, b, z1), \neg (z1=0)\} from (7) and (15), deleting the second literal of (7).
19. \{\neg G(a, b, z2), \neg G(a, b, z1)\} from (17) and (18), deleting the second literal of each.
20. \{\neg G(a, b, z1)\} from (19) by factoring.

Now we have assumed that \( g \) is recursive and hence, by our inductive assumption, primitively representable. We have further assumed that \( CG \), the hypotheses for \( G \), are among our hypotheses for \( P \). Hence we can derive a contradiction from (20) and \( CG \).

Therefore the least-number principle may be applied to (11) and (11), and the derived clause is empty. Thus we have completed the main refutation. In executing the derived program, we may assume the function \( 1 \) to be any primitive function, say the zero function. The resulting program is the
correct but somewhat redundant:

We have yet to verify condition (3) of the definition of primitive representation.

Suppose $M$ is a model for number theory and for the hypotheses for $f$ which includes $a$ in its domain, and let $\phi$ be a completion of $M$ such that

$$(\{F(a,z)\})\text{Val}(\phi) = T.$$ 

We wish to prove that

$z.\text{Val}(\phi) = a.\text{Val}(\phi).$

We know $M$ satisfies clause (3). Consequently, since $\phi$ falsifies $\neg F(a,z)$, $\phi$ satisfies $G(a,z,*).$ By condition (3) applied to $q,$ we have

$$(q(a,z))\text{Val}(\phi) = 0.$$ 

Furthermore we know $M$ satisfies clause (4). Hence

$$((\neg G(a,u,*), \neg (u<z))\text{Val}(\phi) = T.$$ 

Thus, for any value of $u.\text{Val}(\phi),$ either $q(a,u,\phi) = 0$ or $u.\phi \geq z.\phi.$ Since $u.\phi$ can be any natural number, we conclude that $z.\phi$ is the least number $w$ such that $g(a,w) = 0,$ i.e. $e.$
\[ z.\phi = (f(a))Val(\phi). \]

This concludes the proof of the final case of the completeness theorem.

LIMITATIONS

As we mentioned above, the completeness theorem is not so strong as one might hope. It claims that for every recursive function \( g \) there exists a primitive representation. It does not state the following:

Non-Theorem: Let \( f \) be any recursive function, and let \( CF \) be a set of clauses and \( P \) be a predicate of two variables such that if

1. \( M \) is a model for number theory and for \( CF \) with \( a \) in its domain, and
2. \( \phi \) is any completion of \( M \) on all the clauses,

then

3. \( y.\text{Val}(\phi) = (f(a))\text{Val}(\phi) \) if and only if \( ((P(a,y))\text{Val}(\phi) = T \) for any terms \( a \) and \( y \).

Then there exists a primitive refutation of \( \neg F(a,y) \) such that the program \( f' \) derived from it is equal to \( f \).

What the non-theorem asserts is that any correct set of hypotheses describing a function will allow the system to write a program computing that function. This statement, regretably, is false; it is possible to give the theorem prover a correct description of the function, and to give it
axioms describing enough operators to compute the function, and still have it fail to find a proof. We can show that such failure is characteristic of all program writers.

For example, both the zero function $z(x)=0$ and the one function $w(x)=1$ are recursive. It is possible to write axioms $CH$ representing the halting predicate for Turing machines, i.e.:

$$H(x) = x \text{ is the Gödel number of a Turing machine which halts on the empty tape.}$$

Let $n$ be any numeral. Let the hypotheses for $f$ be $CH$, the axioms for number theory, and the clauses

$$\{-H(n), F(x,?)\}, \text{ and}$$
$$\{H(n), F(x,1)\}.$$  

The desired relation between input $a$ and output $y$ is $F(a,y)$. Note that $n$ is not an input variable: it is used here to represent a specific numeral. The function $F$ is to represent is either the constant function zero or the constant function 1, depending on which $n$ is given. We have thus described an infinite class of programming problems, one corresponding to each $n$. But no program writer can solve all these problems without solving the halting problem for Turing machines: otherwise, to determine whether machine $n$ halts, submit the corresponding problem to the program writer, and evaluate the resulting program at an arbitrary argument. Machine $n$ halts if and only if the program gives output 0.
A similar argument shows us that no general program writer will be efficient. In fact we can show that any program writer will exceed any computable estimate of its running time infinitely often, even restricting it to problems it can solve. For, consider the following programming problem: Assume we have the same hypotheses CH for the halting predicate as before, as well as the axioms for number theory. The input constant is a; the output variable is y, and the relation between input and output is \( H(n) \), where \( n \) is an numeral as before. Notice that the input and output do not appear in the relation. Thus the programming problem is solved by any program if \( n \) is the Godel number of a Turing machine which halts on the empty tape. On the other hand, if the machine represented by \( n \) does not halt, then no program will solve the problem. If a time estimate existed, then we could solve the halting problem in all but a finite number of cases as follows: Allow the program writer to run for the estimated time. If it has written a program, then the machine halts; otherwise it does not. Thus we have found a procedure to determine whether a Turing machine halts which is wrong only finitely many times; it may be shown that this is impossible, and hence the computable time estimate does not exist.

We found it necessary to include the least-number principle instead of the mathematical induction schema, in spite of the fact that they are equivalent in classical
number theory. It is our insistence that proofs be primitive which forces us to include the least-number principle.

Remark: There exist sets which can be refuted using primitive resolution, the induction rule of inference and the least-number principle which cannot be refuted without the least-number principle.

Proof: Let \( f \) be any function which is recursive but not primitively recursive. We know that it is primitively representable by the completeness theorem. Suppose there exist hypotheses for \( f \) such that its set can be refuted without resorting to the least-number principle. Then there exists a program \( f' \) computing \( f \) that does not use the min operator, since the only way the min operator is introduced into a program is by means of the least-number principle. But this contradicts the assumption that \( f \) was not primitive recursive.
CHAPTER 5. IMPLEMENTATION

In the summer of 1968, a program writing program was implemented in collaboration with R. C. T. Lee at J. R. Slaqle's Heuristics Laboratory at NIH. The implementation was more a pilot model than an attempt to write a feasible program writer; however, it did succeed at writing a number of simple programs.

The program would accept a set of axioms and a theorem as input. It would prove the theorem, and the proof would be passed to a "post-proof processor," which would construct a program from the proof. The program was written in LISP, and the programs produced were in LISP. The implementation was on the Q32 machine at SDC.

The theorem prover used two strategies, the bridging strategy and the fewest-component preference strategy, which will not be described here. It was modified to make it suitable for program writing. The user declared which of the initial clauses were insipid and which were vital, which of the symbols were primitive, and which constants were input constants and which variables were output variables. Whenever a clause was being generated, it was determined whether or not that derivation was primitive. If it violated any of the rules for primitivity, the clause was rejected.

If a clause was accepted, it was computed whether or not the clause was vital, and what its hot variables were.
The post-proof processor retraced through the proof, generating a program for each vital clause. The final program was constructed from the program associated with the empty clause. Because of the limited space available, the largest number of clauses it ever generated was about 30. Timing information is not available. The longest program it wrote was the program to switch the contents of two registers: a machine language program simulated in LISP. The program writer was transferred to the IBM 360/67, where it was extended to include a version of the induction axiom. The induction actually implemented is quite limited: it assumes that the theorem is a unit clause, and that induction is to be done on that clause; furthermore it must be told which variable to do induction on. Needless to say, we do not consider this the final solution to the induction problem. The 360 version of the program was slowed by the unavailability of a compiler in the LISP system being used.
CHAPTER 6. NEEDED IMPROVEMENTS

Program writing depends on theorem proving. Machines have had an undistinguished record in proving theorems. The theorems which have been obtained have been simple, by human standards. The theorems necessary for writing the MEMBER and the EQUAL functions in LISP required about thirty and fifty clauses to prove. These proofs were done by hand by R. C. T. Lee, but they were beyond the capacity of our theorem prover. Were the theorem prover efficiently programmed in assembly code, its power would improve somewhat.

Furthermore, the theorem prover does not know that it is trying to write a program. That is, it has no heuristics which are especially adapted to program writing. Indeed, we have little idea as yet what heuristics are suitable for program writing. One intuition is that non-primitive symbols are bad and should be removed from the vital clauses. Thus one might seek to delete non-primitive predicate and function symbols from the list of vital clauses.

One rather frustrating weakness was the theorem prover's inability to use the power of the LISP evaluation function in proving its theorems. For example, if a clause

\[ \{2=0\} \]

is derived, one might hope that the theorem prover would be able to recognize that a contradiction has been derived.
Instead, it must construct several more clauses before it can complete the refutation. If the theorem prover had the ability to derive conclusions by evaluating primitive terms, it would have a greater theorem proving power.

In actual practice a large body of axioms describing the programming language must be stored if the program is to write programs in that language. As the set of axioms grows larger, the problem of theorem proving becomes more and more the art of knowing which axioms to ignore. This difficulty is precisely that faced by workers in the field of deductive question answering with a large body of stored facts. Our present work pays no attention to this problem.

The theorem prover is especially weak in proving theorems strongly relying on the equality relation. This is short-coming shared with resolution type theorem provers in general. Since most interesting programming problems require the properties of equality in their proofs, this disability has serious repercussions in automatic program writing. There is some work, that of Wos's group for example, which tries to give special treatment to the equality predicate: it contains a special rule of inference which derives the conclusions resulting from the properties of equality. This rule shortens the proofs of most theorems involving the equality predicate, and it also makes the search for a proof more efficient. If is expected that if a program writer using such a rule in its theorem prover were constructed, its performance would be
greatly improved.

The version of the least-number principle discussed suffices to prove the completeness theorem; however, in practice we would hope that the theorem prover would be able to find other instances of the induction axiom and the least-number principle. In human theorem proving, choosing the correct instance of the induction axiom to prove a given theorem often requires a great deal of ingenuity; therefore we expect that this problem will not be solved easily by a computer program.

The first-order predicate calculus was chosen as an input language not because it was particularly well suited for stating programming problems, but because some theorem proving methods in this language were available. There are no existing theorem proving programs for higher order theory at the time of this writing, to our knowledge. In higher order theory the problems involving the equality relation and the induction axiom are greatly simplified. Robinson is now investigating theorem proving methods for higher order theories. If progress is made in this area, it can be expected to have repercussions in efforts at automatic program writing.

Another possible language extension is that of allowing predicates and functions with a variable number of arguments. For instance, a term such as list(x_1, \ldots, x_n) is not permitted in ordinary predicate calculus. We propose that a
new type of variable be introduced called the "sequence" variable, which we will represent here by double letters. For example, if \( xx \) were a sequence variable, \( \text{list}(xx) \) would represent \( \text{list}(x_1, \ldots, x_n) \), where \( n \) could be 0. Sequence constants would also be introduced. The theorem for the program to find the second element on a list might then be

\[(a)(\exists y)(\exists w)(\exists zz) \text{Equal}(a, \text{list}(w, y, zz)),\]

where \( a \) is the input constant, \( y \) is the output variable, and \( zz \) is a sequence variable, and where we assume that all lists are sufficiently long. The function \( \text{member}(a, b) \) might be represented by

\[(a)(b)(\exists y) . (\exists xx)(\exists zz) \text{Equal}(b, \text{list}(xx, a, zz) \land y = T \lor (cc)(dd) \neg\text{Equal}(b, \text{list}(cc, a, dd) \land y = \text{NIL}).\]

In proving theorems involving sequence variables, the unification algorithm would be augmented with a COMIT-SNOBOL type matching algorithm, so that in unifying a set of literals, sequence variables could be matched with sequences of terms, rather than with just a single term, the way individual variables are. Furthermore, only a sequence variable or a sequence constant could match a sequence constant; an individual variable could not match a sequence constant. Sequence functions, whose value is a sequence, might also be admitted. The logical properties of this system have yet to be investigated.

No provision has been made for controlling the efficiency of a machine-written program. There are many
possible programs computing the same function; the one the program writer finds is the one corresponding to the first proof it comes across; only by a great stroke of luck will that be a good program in any sense. We have paid little attention to this problem. One possible line of attack is this: suppose the method of evaluation of a program depends exclusively on its running time and its storage space. The program writer, in proving its theorem can make estimates of the time and space requirements of the segment of program corresponding to each clause. It can take these estimates into consideration in deciding which clause to focus its attention on. When it completes a proof, it can evaluate the entire program and continue looking for other proofs, using the evaluation of the completed program as a cutoff point, so that it discards clauses whose evaluation is not less than that of the completed program.

In stating the programming problem as a relation in predicate calculus, one has already solved a portion of the problem: the task of representation. In general there will be many different formulations of the same problem. Which description the user chooses often determines whether or not the program writer will succeed. Of course one may provide axioms which enable the program writer to switch representations, but the theorem prover will be biased in favor of the representation it is originally given. The task of choosing a representation seems difficult, and we have nothing to say about it.
BIBLIOGRAPHY


Beth, E. [1962], Formal Methods, Reidel, Dordrecht


Blum, M. [1967] On the size of machines, Information and Control, v. 1)


Davis, M. Axioms for number theory, Rensselaer Polytechnic Institute, Troy, N. Y.


Freeman, P. and Shaw, M. Helpful hints to would-be users of list, edit and display, Carnegie-Mellon University, Pittsburgh, Pa.


Kowalski, R. An exposition of paramodulation with refinements, University of Edinburgh, Edinburgh.


Manna, Z. [1963] Termination of algorithms, Ph.D.


Robinson, A. [1957]. Proving theorems, as done by man, machine and logician, Proceedings of the 1957 Summer School in Logic, Cornell, Ithaca, N. Y.


Freeman, P. and Shaw, M. Helpful hints to would-be users of list, edit and display, Carnegie-Mellon University, Pittsburgh, Pa.


Weissman, C. [1967] Lisp 1.5 Primer, Dickenson, Belmont, Calif.


CONSTRUCTING PROGRAMS AUTOMATICALLY USING THEOREM PROVING

Richard J. Waldinger

May 1969

F44620-67-C-0058

9718

6154501R

681304

1. This document has been approved for public release and sale; its distribution is unlimited.

12. SPONSORING MILITARY ACTIVITY
Air Force Office of Scientific Research (SRMA)
1400 Wilson Boulevard
Arlington, Virginia 22209

13. ABSTRACT

The paper describes a method by which programs may be constructed mechanically. The problem of writing a program is transformed into a theorem proving task. The specifications for the program are used to construct a theorem, the theorem is proved, and the program is derived from the proof of the theorem. The specifications for the program are described as a relation between the input and output variables expressed in predicate calculus. Mechanical theorem proving techniques are used to prove the existence of output variables satisfying the specifications. Existence is proven constructively, so that embedded in the proof is a method to compute the desired output values. A program is extracted from the proof.

Restrictions to Robinson's resolution principle are proposed so that only constructive proofs are produced.

A proof of the soundness of the method is presented. In other words, it is shown that the programs written by the program writer do indeed satisfy the specifications.

It is also shown that programs for the entire class of recursive functions may be written by automatic program writing. Thus nothing is lost by restricting application of the resolution principle.

An implementation of the method which writes LISP programs is described.