Proving programs to be correct.

Debra Fish
PROVING PROGRAMS TO BE CORRECT

by

Debra Fish

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of

Master of Science
in
Computer Science

Lehigh University
1977
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

May 6, 1977
(date)

Professor in Charge

Chairman of Department
Acknowledgements

The author wishes to thank her advisor, Professor Gerhard Rayna, for his continued advice and guidance during the preparation of this thesis.
Table of Contents

Abstract .................................................. 1
Introduction .............................................. 2
Description of the Programming Language .................. 4
Symbolic Execution ..................................... 6
Symbolic Execution Tree ................................ 8
Examples of Procedures and Proofs ....................... 10
  1: SORTING ............................................. 10
  2: MIN AND MAX ....................................... 14
  3: SQUARE ROOT ...................................... 17
  4: FACTORIAL ......................................... 21
  5: POWER .............................................. 24
  6: FIBONACCI .......................................... 26
Problems with the Programming Language ................ 28
Problems with Proofs ..................................... 30
Current State ............................................ 33
Conclusions ............................................. 34
Appendix I ................................................ 35
  1: SORTING TREE ...................................... 36
  2: MIN AND MAX TREE .................................. 41
  3: SQUARE ROOT TREE .................................. 44
  4: FACTORIAL TREE .................................... 47
Abstract

This paper is concerned with the notion of proving computer programs to be correct. Several examples of correct programs with their corresponding proofs are presented.

Several methods of proving the correctness of programs are presented with special emphasis on techniques known as Symbolic Execution and Symbolic Execution Trees. A separate symbolic execution tree is presented for each of the sample programs.

The notion of symbolic execution is such that instead of supplying normal inputs to a program, the programmer supplies symbols to represent arbitrary values. This often leads to a case analysis in the method and infinite loops in the symbolic execution tree.

The proofs of the sample programs are through the use of inductive assertions. Several problems arising with this method and with the method of symbolic execution, are discussed.

Much research is now being done in proving programs, including methods of having the computer itself prove the programs. Several experimental interactive program verifiers are already in existence and have proven to be quite useful, especially in aiding symbolic execution.

The language used to write the programs follows a PL/I style with some variations from PASCAL. The use of a language written in this way helps simplify the proof.
Introduction

In recent years a considerable amount of time and effort has been put into finding methods of proving computer programs to be correct. Although a program may produce correct results for many different sets of input values, the program may not be correct for all. There would be no way to test the infinitely many possible sets of input values that could be associated with the variables in the program. But methods have been developed to help solve this problem.

This paper describes some of the work that has been done in proving programs correct. It also presents several examples of procedures with their corresponding proofs. The procedures are written in a language which follows a PL/I style such as that used by King [4]. The only control structures are IF...THEN...ELSE,DO WHILE...;...END, and procedure calls.

The proof of these procedures is accomplished by proving inductive assertions to be correct. For example, the DO WHILE statement presents a certain condition. Any assertion within the DO WHILE loop will repeat that condition, and it will be true (by the DO WHILE hypothesis), unless some operation within the loop has affected it. This is because in order to enter the body of the loop, the condition must have been true.

In the proofs we will often make use of a simple fact about integer values. If I and N are both integers, we
can assert the following: if $I < N$, and we set $I := I + 1$, and as a result $I \neq N$, then $I = N$. This will be known as the integer inequality property.

We will make use of a prime notation for the input variables. An input variable with a prime is considered to contain the original value that the variable contained when the call was made to the procedure.
Description of the Programming Language

The language used to write the procedures included in this paper contains simple assignment statements and arithmetic operations. The variables used represent signed integer values and arrays of such.

The following is a partial list of the statements used in the language:

"name : PROCEDURE (p_1, p_2, p_3, ... p_n);

<statement list>

END;

where name is the procedure name and p_1, p_2, p_3, ... p_n are procedure parameters."

"DECLARE variable_1, variable_2,

... variable_n : INTEGER;

which creates integer valued variables named variable_1, variable_2, ... variable_n whose values are known only within the procedure." The same form also is used to specify the types of the actual parameters of the procedure.

"IF <Boolean> THEN statement_1

ELSE statement_2"


2. Ibid.
where \( \text{statement}_1 \) and \( \text{statement}_2 \) are statements or compound statements and either \( \text{statement}_1 \) or \( \text{statement}_2 \) is executed depending on the truth of the \( <\text{Boolean}> \).\(^3\)

\[
\text{"DO WHILE } <\text{Boolean}>; \\
<\text{statement list}> \\
\text{END;}
\]

When control reaches the DO WHILE statement, if the value of the \( <\text{Boolean}> \) is true, the statement list is executed and control is returned to the DO WHILE statement. If the \( <\text{Boolean}> \) is false, control passes immediately to the statement following the END statement.\(^4\)

Procedures will be invoked by a CALL statement in the main program. The procedure head will contain the input parameters, and the symbol VAR followed by any variables whose values will be passed back to the main program. Return to the main program is achieved by a RETURN statement.

\(^3\)Ibid., p. 333.

\(^4\)Ibid.
Symbolic Execution

One of the methods developed for proving programs to be correct makes use of a technique known as symbolic execution. This method is used by James King [4], who follows the ideas presented by Robert Floyd [1].

This technique requires that you use symbols to represent the input variables and prove the program through the use of those symbols. If the execution flow of the program is completely independent of the inputs, then only one symbolic execution is necessary for the proof. If, however, the program is dependent on the inputs (for example, because of an IF statement) then a case analysis is required for the proof.

In order to prove a program by using symbolic execution, we need to introduce three more statements to our language. The first is an ASSUME statement and will appear immediately after the procedure name. This has the form ASSUME (<Boolean>); and describes what is "given" about the input parameters.

The next statement has the form PROVE (<Boolean>) and asserts what is required to be true of the output parameters of the procedure. This statement appears immediately before the RETURN statement.

The third statement has the form ASSERT (<Boolean>); and may appear several times within the procedure body. It makes
inductive assertions regarding the values of the variables at that point in the procedure.

"A procedure is said to be correct (with respect to its input and output assertions) if the truth of its input assertion upon procedure entry insures the truth of its output assertion upon procedure exit."\(^5\)

By using ASSERT statements we reduce the complexity of the proof. We know the input assertion is true so we need to deduce from it the PROVE assertion; the ASSERT statements serve as lemmas.

The ASSUME statement is said to start a "cut." This cut terminates at a PROVE or ASSERT statement. The ASSERT statement thus serves two purposes. First, it can act as a prove statement for the previous cut. At the same time, it acts as an assume statement for the next cut. This reduces the complexity of proving a procedure which contains loops, such as a DO WHILE statement. We now just need to prove that the assertion at each cut is true, thereby eliminating any problems regarding an infinite loop.

\(^5\)Ibid., p. 334.
Symbolic Execution Tree

The complete symbolic execution of a procedure can also be represented by a symbolic execution tree. This is very similar to a flowchart and depicts the entire procedure. Due to the existence of loops, resulting from the DO WHILE statement, infinite trees will have each cut represent the beginning of a tree.

Each statement of the procedure is given a number. Within a symbolic execution tree, this number represents a node. The first statement is placed at the top of the tree and the tree is traversed from top to bottom. Should a decision be required, resulting from an IF or DO WHILE statement, then an arc is used. Any conditional statement will have more than one arc leaving its node. Nonexecutable statements are not represented in the tree.

The symbolic execution tree is traversed by using a "path condition" (abbreviated pc). It is set to true upon entry to the tree. Each time a new case is considered, the path condition is updated.

A tree is structured so that every possible case and execution is considered. The tree proves that the procedure is correct if the word "verified" appears at the final node of the tree.
A symbolic execution tree for each of the procedures presented in this paper appears in the appendix.
Examples of Procedures
and Proofs

1 : Sorting

1. SORT :

PROCEDURE (X,N);

2. cut 2 ASSUME (N > 0);

3. DECLARE N, I, J, TEMP : INTEGER;
   X(1..N) : ARRAY OF INTEGER;

4. I := 1;

5. DO WHILE (I < N);

6. cut 6 ASSERT (I < N, HEAP X UNCHANGED,
   X(I)....X(I-1) IN INCREASING ORDER);

7. J := I + 1;

8. DO WHILE (J < N);

9. cut 9 ASSERT (I < N, J < N, J > I, HEAP X
   UNCHANGED);

10. IF X(I) > X(J) THEN

11. BEGIN

12. TEMP := X(I);

13. X(I) := X(J);

14. X(J) := TEMP;

15. cut 15 ASSERT (I < N, J < N, J > I, HEAP X
   UNCHANGED, X(I)....X(I-1) IN
   INCREASING ORDER, X(I) IS THE
   SMALLEST OF X(I)....X(J));
16. END;
17. cut 17 ASSERT (I < N, J > I, HEAP X UNCHANGED, X(I) IS THE SMALLEST OF X(I) ... X(J));
18. J := J + 1;
19. cut 19 ASSERT (I < N, HEAP X UNCHANGED, X(1) ... X(I-1) IN INCREASING ORD.
X(I) IS THE SMALLEST OF X(I) ... X(N));
20. END;
21. I := I + 1;
22. END;
23. cut 23 PROVE (X(1) ... X(N) IN ASCENDING ORDER, HEAP X UNCHANGED);
24. RETURN;
25. END;
Procedure SORT is a simple exchange sort in which we are sorting an array of $N$ integer values in ascending order. The proof of this procedure requires the proof of each cut.

Cut 2 is the input assumption that $N > 0$ insuring at least one element exists in the array.

Cut 6 can be reached from cut 2 and lines 3, 4, and 5. The assertion is true because of the DO WHILE hypothesis; because none of the members in the array have changed; and $I$ is one so $X(1) ... X(0)$ are the zero smallest in increasing order. Cut 6 can also be reached from cut 19 and lines 20, 21, 22, and 5. The assertion is true because of the DO WHILE hypothesis; the array has not changed; and $I$ has been incremented by one so the first $I-1$ elements are sorted.

Cut 9 can be reached from cut 6 and lines 7 and 8. The assertion is true because the previous assertion was true; because of the DO WHILE hypothesis; and because $J$ is $I + 1$. Cut 9 can also be reached from cut 19 and lines 20 and 8. The assertion is true because of the DO WHILE hypothesis and cut 19 (nothing has been changed).

Cut 15 is reached from cut 9 and lines 10 through 14 inclusive. The assertion is true because the previous assertion is true and if $X(I)$ was not $\leq X(J)$ it was made to be by the interchange controlled by the IF statement.
Cut 17 is reached from cut 15 and line 16. The assertion is true because it follows directly from the previous assertion. Cut 17 is also reached from cut 9 and line 10 if FALSE. In this case, the assertion is true because of the DO WHILE hypothesis; the values of I and J have not changed; the values of the array have not changed; and if the test $X(I) > X(J)$ failed then $X(I) \leq X(J)$.

Cut 19 is reached from cut 17 and line 18. The assertion is true because J is now $N + 1$ (integer inequality property!), therefore the previous value of J was N so this assertion follows immediately from the assertion at cut 17.

Cut 23 is reached by the failure of the condition of the test in line 5. Due to the integer inequality property, I is now equal to $N + 1$, therefore the assertion follows from the previous assertion.
MIN and MAX:

PROCEDURE (X,N, VAR MIN, MAX);

ASSUME (N > 0);

DECLARE MIN, MAX, I, N : INTEGER;
X(1..N) : ARRAY OF INTEGER;

MIN := X(1);
MAX := MIN;
I := 2;
DO WHILE (I ≤ N);

ASSERT (I < N, X = X', MIN IS SMALLEST OF X(1)...X(I-1), MAX IS LARGEST OF X(1)...X(I-1));

IF X(I) < MIN THEN
    MIN := X(I);
    IF X(I) > MAX THEN
        MAX := X(I);
        ASSERT (I < N, X = X', MIN IS SMALLEST OF X(1)...X(I), MAX IS LARGEST OF X(1)...X(I));
    END;
    I := I + 1;
END;

PROVE (MIN IS SMALLEST OF X(1)...X(N), MAX IS LARGEST OF X(1)...X(N), X = X');
RETURN;
END;
Procedure MINMAX is a procedure to find the minimum and maximum elements in an array of integer values. The proof of this procedure requires the proof of each cut.

Cut 2 is the input assumption that \( N \geq 0 \) insuring that there is at least one element in the array.

Cut 8 can be reached from cut 2 and lines 3, 4, 5, 6, and 7. The assertion is true because of the DO WHILE hypothesis and because nothing has been done to any members of the array. I at this point is two, therefore MIN is the smallest of \( X(1) \ldots X(I) \) and MAX is the largest of \( X(1) \ldots X(I) \). Cut 8 can also be reached from cut 13 and lines 14, 15, and 7. The assertion is true because of the DO WHILE hypothesis; because the set has not changed; and, from cut 13 which mentions \( X(I) \), we get the corresponding statement using \( X(I-1) \) since \( I \) has been incremented by one.

Cut 13 is reached from cut 8 and lines 9 through 12 inclusive. This assertion is true because of the DO WHILE hypothesis; the array has not changed; and the tests of the IF conditions placed the smallest value in MIN and the largest value in MAX.

Cut 16 is reached by the failure of the test of the condition in line 7. The truth of the PROVE statement follows directly from the truth of the assertion at cut 13, together
with the fact (integer inequality property!) that I is now equal to \( N + 1 \), therefore the previous value of I was \( N \); and, finally, that the array has not been altered.
3: SQUARE ROOT

1. SQRROOT:

   PROCEDURE (X, VAR TRIAL);

2. cut 2   ASSUME (X ≥ 0);

3.      DECLARE X, TRIAL, INCR : INTEGER;

4.      TRIAL := 0;

5.      INCR := 1000;

6. cut 6   ASSERT (SQR(TRIAL) ≤ X, INCR is a power of 10);

7.      DO WHILE (INCR ≥ 1);

8. cut 8   ASSERT (INCR > 1, X = X', INCR is a power
           of 10);

9.      DO WHILE (SQR(TRIAL) ≤ X);

10. cut 10  ASSERT (INCR > 1, X = X', SQR(TRIAL)
             ≤ X, INCR is a power of 10);

11.      TRIAL := TRIAL + INCR;

12.      END;

13. cut 13  ASSERT (INCR > 1, X = X', √X < TRIAL ≤ √X
           + INCR, INCR is a power of 10);

14.      TRIAL := TRIAL - INCR;

15. cut 14  ASSERT (INCR > 1, X = X', √X - INCR <
             TRIAL ≤ √X, INCR is a power of 10);

16.      INCR := INCR/10;

17.      END;

18. cut 18  PROVE ( √X - 1 < TRIAL ≤ √X, X = X');

17
19. RETURN;
20. END;
Procedure SQRROOT is a procedure to find the largest integer not exceeding the square root of a positive integer. To prove this procedure to be correct, we must prove each cut.

Cut 2 is the input assumption that \( X > 0 \) and is therefore true.

Cut 6 is reached from cut 2 and lines 3, 4, and 5. The assertion is true because \( X > 0 \) and TRIAL = 0; and because \( \text{INCR} = 10^3 \).

Cut 8 can be reached from cut 6 and line 7. The assertion is true because of the DO WHILE hypothesis and the value stored in \( X \) has not changed. Cut 8 can also be reached from cut 15 and lines 16, 17, and 7 in which case the assertion is true because of cut 15; the DO WHILE hypothesis; \( X \) has not changed; and a power of 10, divided by 10, yields either a power of 10 or (the integer result) 0.

Cut 10 can be reached from cut 8 and line 9. The assertion is true because the previous assertion is true and because of the DO WHILE hypothesis. Cut 10 can also be reached from cut 10 and lines 11, 12, and 9. In this case, the assertion is true because the value in \( X \) and INCR have not changed and because of the DO WHILE hypothesis.

Cut 13 is reached from cut 10, lines 11 and 12, and the failure of the test of the condition in line 9. The assertion
is true because the value in INCR and the value in X have not changed and because now $\text{SQR(TRIAL)} > X$ while $\text{SQR(TRIAL)} - \text{INCR} \leq X$ therefore $\sqrt{X} < \text{TRIAL} \leq \sqrt{X} + \text{INCR}$.

Cut 15 is reached from cut 13 and line 14. The assertion is true because the previous assertion is true and because line 14 decremented TRIAL by the value stored in INCR.

Cut 18 is reached from cut 15, lines 16 and 17, and the failure of the test of the condition in line 7. In this procedure, we are assuming that when INCR has the value of $10^o = 1$ and line 16 is executed, the result will be zero. The truth of the PROVE statement follows directly from the previous assertion at cut 15 and from the fact that the value stored in X has not changed.
4: FACTORIAL

1. FACT:

   PROCEDURE(N, VAR X);

2. cut 2      ASSUME (N > 0);

3. DECLARE N, X, I : INTEGER;

4. X := N;

5. I := 1;

6. cut 6      ASSERT (N = N', X = N!/(N-1)!);

7. DO WHILE (I < N);

8. cut 8      ASSERT (I < N, N = N', X = N!/(N-I)!);

9. X := X * (N-I);

10. cut 10    ASSERT (I < N, N = N', X = N!/(N-I-1)!);

11. I := I + 1;

12. cut 12    ASSERT (I < N, N = N', X = N!/(N-I)!);

13. END;

14. cut 14    PROVE (X = N!, N = N');

15. RETURN;

16. END;
Procedure FACT is a procedure to compute \( N \) factorial where \( N \) is a positive integer. The proof of this procedure is accomplished through the proof of each cut.

Cut 2 is the input assumption that \( N \geq 0 \) and is therefore true.

Cut 6 is reached from cut 2 and lines 3, 4, and 5. This assertion is true because the value of \( N \) has not been changed and \( N!/(N-1)! = N \) which was placed in \( X \) in line 4.

Cut 8 is reached from cut 6 and line 7. In this case, the assertion is true by the DO WHILE hypothesis and because the previous assertion is true. Cut 8 can also be reached from cut 12, line 13, and line 7. The assertion is true because of the DO WHILE hypothesis and the assertion at cut 12.

Cut 10 is reached from cut 8 and line 9. The assertion is true because the previous assertion is true and because \( X \) has now been replaced with \( X \times (N-I) \).

Cut 12 is reached from cut 10 and line 11. The assertion is true because the previous assertion is true and because \( I \) has been incremented by one.

Cut 14 is reached by the failure of the test of the condition of line 7. Due to the integer inequality property, \( I = N \). Therefore, from the assertion at cut 12 we get
\[
X = N!/(N-I)! = N!/(N-N)! = N!/0! = N! \text{ because } 0! \text{ is considered}
\]
equal to one. Thus, the PROVE statement is true.
5: POWER

1. POWER:

   PROCEDURE (X,Y, VAR A);

2. cut 2  ASSUME (Y >= 0);

3. DECLARE X, Y, A, I : INTEGER;

4.   A := 1;

5.   I := 0;

6.   DO WHILE (Y > I);

7. cut 7  ASSERT (Y > I, A = X^I, X = X', Y = Y');

8.   A := A * X;

9.   I := I + 1;

10. cut 10  ASSERT (Y > I, A = X^I, X = X', Y = Y');

11. END;

12. cut 12  PROVE (A = X^Y, X = X', Y = Y');

13. RETURN;

14. END;
Procedure POWER is a procedure to raise X to the Y power where Y is a positive integer. The proof of this procedure requires the proof of each cut.

Cut 2 is the input assumption that $Y \geq 0$ and is therefore true.

Cut 7 can be reached from cut 2 and lines 3, 4, 5, and 6. The assertion is true because of the DO WHILE hypothesis and because $I = 0$, therefore $A = X^I = X^0 = 1$ which is true from line 4. (We are interpreting $0^0$ as 1 if $X = 0$). Cut 7 can also be reached from cut 10 and lines 11 and 6. In this case, the assertion is true because of cut 10 the DO WHILE hypothesis.

Cut 10 is reached from cut 7 and lines 8 and 9. It is true because the assertion at cut 7 is true; I has been incremented by one, so I may be equal to Y; and because $X^{I+1} = X^I \times X$.

Cut 12 is reached from cut 10, line 11, and the failure of the test of the condition in line 6. Since $Y \geq I$ but not $Y > I$, we have $Y = I$ therefore the truth of the PROVE statement follows directly.
6: FIBONACCI

1. FIBONACCI:

   PROCEDURE (N, VAR X);

2. cut 2
   ASSUME (N > 2);

3. DECLARE N, I : INTEGER;
   X(1..N) : ARRAY OF INTEGER;

4. X(1) := 1;

5. X(2) := 1;

6. I := 3;

7. DO WHILE (I < N);

8. cut 8
   ASSERT (I < N, N = N', X(1) ... X(I-1) ARE THE FIRST I-1 FIBONACCI NUMBERS);

9. X(I) := X(I-1) + X(I-2);

10. I := I + 1;

11. END;

12. cut 12
   PROVE (X(1) ... X(N) ARE THE FIRST N FIBONACCI NUMBERS, N = N');

13. RETURN;

14. END;
Procedure FIBONACCI is a procedure to create an array containing the first \( N \) Fibonacci numbers where \( N \) is a positive integer. By definition, each number in the series is to be the sum of the two numbers preceding it, starting with 1, 1, ... The proof of this procedure requires the proof of each cut.

Cut 2 is the input assumption that \( N \geq 2 \) and is therefore true.

Cut 8 can be reached from cut 2 and lines 3, 4, 5, 6, and 7. The assertion is true because of the DO WHILE hypothesis; the fact that the value of \( N \) has not changed; and \( X(1) = 1, X(2) = 1 \) are the first 2 Fibonacci numbers. Cut 8 can also be reached by cut 8 and lines 9, 10, 11, and 7. The assertion is still true by the DO WHILE hypothesis; the fact that \( N \) has not changed; and \( X(I) \) has been replaced by the sum of the two previous values and \( I \) has been incremented by one so \( X(I-1) = X(I-2) + X(I-3) \), are the \((I-1)st\) Fibonacci number.

Cut 12 is reached from cut 8, lines 9, 10, 11, and line 7 when the condition \( I < N \) is false. \( I \) is now equal to \( N+1 \) because we are dealing with integer values so the PROVE statement is true.
Problems with the Programming Language

Several problems arise when using the programming language suggested by King [4].

The variables which are passed as the procedure parameters are declared in the DECLARE statement. This does not make sense because the ASSUME statement makes an assertion regarding these variables and the ASSUME statement appears before the DECLARE statement. It would have been better to declare the procedure parameters in the procedure head.

No means is provided within a procedure to check for overriding the bounds of an array. Suppose an array is dimensioned in the main program as \( X(10) \). A procedure may do the following:

\[
I := 1;
\]
\[
\text{DO WHILE (} I \leq N\text{)};
\]
\[
\text{ASSERT (} I \leq N\text{);} \\
X(I+2) := X(I+1) + X(I); \\
I := I + 1;
\]
\[
END;
\]

where \( N \) is the upper bound of the array. This loop will continue until \( I = 9 \) in which case the assignment \( X(9+2) := X(9+1) + X(9) \) will be attempted, placing a value in \( X(11) \) which does not exist since the upper bound on the
array subscript is 10. There is no method provided in the proof of this procedure to show that there is an error.

Another problem occurs with the RETURN statement. According to the language as presented by King, the procedure head contains only the parameters being passed from the main program. However, a procedure may create something using those parameters and the result may be stored in another variable which must be returned to the main program. According to King's language, the RETURN statement simply returns the new variable. But where does it go in the main program? The actual call to the procedure contained only the input parameters. A means, such as that used by PASCAL, should be provided to declare in the procedure head that a variable will be passed back to the main program.
Problems with Proofs

There are several problems with regard to proving programs and procedures to be correct.

The definition of a correct procedure presented by King, and quoted in this paper, discusses correctness only with respect to input and output assertions. Consider the following example:

EXCHANGE:

PROCEDURE (VAR X,Y);
    ASSUME (X > 0, Y > 0);
    DECLARE I,X,Y, TEMP : INTEGER;
    I := 1;
    DO WHILE (I = 1);
        ASSERT (I > 1, X = X', Y = Y');
        TEMP := X;
        X := Y;
        Y := TEMP;
        I := I + 1;
    END;
    PROVE (X = Y', Y = X');
    RETURN;
END;
This is a simple procedure which exchanges the values of $X$ and $Y$. According to the definition, this procedure is correct because the truth of the input assertion does insure the truth of the output assertion. However, nothing is said about the intermediate assertions within the program. The definition should also require the assertions within the procedure to follow from the input assertion.

One of the biggest problems encountered in proofs is that inductive assertions are often very difficult to formulate. In the previous example, the assertion is obviously false but this is not always easy to see. Therefore we may be attempting to prove an assertion when, in fact, it is incorrect. This is, of course, no different from the situation in other areas of Mathematics: to find valid lemmas and intermediate steps useful to proving a desired theorem.

Another problem is the problem of termination. Consider the following example:

```
SUM:

PROCEDURE (X, Y, VAR Z);

ASSUME (X > 0, Y > 0);

DECLARE I, J, X, Y, Z : INTEGER;

I := 2;

J := 1;
```
DO WHILE (I > J);
    ASSERT (I > J, X = X', Y = Y');
    I := I;
    J := J;
END;
Z := X + Y;
PROVE (Z = X+Y, X = X', Y = Y');
RETURN;
END;

This is a simple procedure to compute the sum of two positive integers. However, once the DO WHILE loop is entered, J will never be greater than or equal to I and this loop will never terminate. According to the definition, this is a correct procedure. The truth of the input assertion does lead to the truth of the output assertion and the inductive assertion is true, but the procedure will never execute past the DO WHILE loop.
Current State

Through the efforts of several people, various computer programs have been proven to be correct. The method most widely used to accomplish this task is through the use of inductive assertions. Other methods also used include structural induction, standard mathematics, case analysis, truncation induction, computational induction, and recursion induction.

The following are examples of proven correct programs:

"TREESORT 3, FIND, McCarthy-Painter compiler, lisp compilers, Early's recognizer, THE operating system, readers/writers, operating system kernel, numerical calculations, interval arithmetic, floating point arithmetic, Todd-Coxeter algorithm, and Fischer-Galler algorithm." 6

Numerous computer systems have also been implemented for demonstrating correctness of programs. With these systems, the user usually supplies the assertions.

Conclusions

Several programs with their corresponding proofs have been presented along with a brief look at one of the methods of proving programs correct. Also, several problems were cited regarding this method and proving programs in general.

The idea of proving programs to be correct is not new. However, it is one of the most exciting ideas in the field of computer science.

Clearly, a proof for most programs, by any combinations of techniques, still remains a challenge. But knowing that a program is correct and reliable is one of the most important concepts of today.

Of course, program testing will reassure a programmer that his program is correct for sample test runs because he can check the results by hand. However, if the program is formally proven to be correct, the programmer can be assured that his program will always execute properly.

The research being done in the area of attempting to get the computer itself to construct or assist in the construction of proofs should prove very useful in the near future. Eventually, proving programs will become a very basic feature which will require little or no effort, but will assure reliable large-scale computer programs.
APPENDIX I
1: SORTING TREE

cut 2

\[ pc: \text{true}, X: \alpha, N: \beta \]

\[ \beta > 0 \]

\[ I : 1 \]

\[ I > \beta \]

\[ I \leq \beta \]

cut 23 verified \( \alpha(1) \ldots \alpha(n) \) in ascending order, heap \( \alpha \) unchanged

cut 6 verified \( I \leq \beta \), heap \( \alpha \) unchanged, \( \alpha(I-1) \) in increasing order
cut 6

$\text{pc: true, } X : \alpha, N : \beta$

$\text{pc: } I \leq \beta, \text{ heap } \alpha \text{ unchanged, }$

$\alpha(1) \ldots \alpha(I-1) \text{ in increasing order}$

$J : I + 1$

$J > \beta$

$J \leq \beta$

cut 19 verified
$\text{I } \leq \beta, \text{ heap } \alpha \text{ unchanged, } \alpha(I) \text{ smallest of } \alpha(1) \ldots \alpha(\beta)$

cut 9 verified $I \leq \beta$,
$J < \beta, J > I, \text{ heap } \alpha \text{ unchanged}$
cut 9

pc : true, X : α, N : β

9

pc: I < β, J < β, J > I, heap α unchanged

10

α(I) ≤ α(J)

α(I) > α(J)

17 verified
I < β, J > I, heap α unchanged, α(I) smallest of α(I)...α(J)

12

TEMP : α(I)

13

α(I) : α(J)

14

α(J) : TEMP

15

cut 15 verified
I < β, J < β, J > I, heap α unchanged,
α(I)...α(I-1) in increasing order α(I) smallest of α(I)...α(J)

38
cut 15

pc : true, X : α, N : β

15

pc : I < β, J < β, J > I, heap α unchanged, α(I) ... α(I-1) in increasing order, α(I) smallest of α(I) ... α(J)

17

cut 17 verified I < β, J > I, heap α unchanged, α(I) smallest of α(I) ... α(J)

cut 17

pc : true, X : α, N : β

17

pc : I < β, J > I, heap α unchanged, α(I) smallest of α(I) ... α(J)

18

J : J + 1

19

cut 19 verified I < β, heap α unchanged, α(I) ... α(I-1) in increasing order, α(I) smallest of α(I) ... α(N)
cut 19

\[ \text{pc : true, } X : \alpha, N : \beta \]

\[ \text{pc : } I \leq J, \text{ heap } \alpha \text{ unchanged, } \alpha(1) \ldots \alpha(I-1) \text{ in increasing order, } \alpha(I) \text{ smallest of } \alpha(I) \ldots \alpha(\beta) \]

\[ \text{cut 19 verified } I \leq J, \text{ heap } \alpha \text{ unchanged, } \alpha(I) \text{ smallest of } \alpha(I) \ldots \alpha(\beta) \]

\[ \text{cut 9 verified } I \leq J, J \leq I, \text{ heap } \alpha \text{ unchanged} \]

cut 23

\[ \text{pc : true, } X : \alpha, N : \beta \]

\[ \text{pc : } \alpha(1) \ldots \alpha(N) \text{ in ascending order, heap } \alpha \text{ unchanged} \]

\[ \text{verified return } X \]
2 : MIN AND MAX TREE

MIN AND MAX TREE

I > 8

cut 2

pc : true, X : α, N : β

β > 0

MIN : α(1)

MAX : MIN

I : 2

I > β

cut 16 verified MIN
smallest of α(1)...α(N), MAX largest of α(1)...α(N), α=α'

I ≤ β

cut 8 verified I<β, α=α',
MIN smallest of α(1)...α(I-1), MAX largest of α(1)...α(I-1)

41
cut 8

pc : true, X : α, N : β

pc : I<β, α = α', MIN smallest of α(1)...α(I-1), MAX largest of α(1)...α(I-1)

α(I) ≥ MIN

α(I) ≤ MAX

α(I) > MAX

MIN : α(I)

cut 13 verified I<β, α = α', MIN smallest of α(1)...α(I), MAX largest of α(1)...α(I)

MIN : α(I)

cut 13 verified I<β, α = α', MIN smallest of α(1)...α(I), MAX largest of α(1)...α(I)
1. **Cut 13**
   - \( pc : true, X : \alpha, N : \beta \)

2. **Cut 16**
   - \( pc : I \leq \beta, \alpha = \alpha', MIN \) smallest of \( \alpha(I) \ldots \alpha(N) \), \( MAX \) largest of \( \alpha(1) \ldots \alpha(I) \)
   - \( I = I + 1 \)

3. **Cut 8**
   - \( pc : MIN \) smallest of \( \alpha(1) \ldots \alpha(N) \), \( MAX \) largest of \( \alpha(I) \ldots \alpha(I-1) \)

4. **Cut 16**
   - \( pc : true, X : \alpha, N : \beta \)
   - \( pc : MIN \) smallest of \( \alpha(1) \ldots \alpha(N) \), \( MAX \) largest of \( \alpha(1) \ldots \alpha(N) \), \( \alpha = \alpha' \)

5. **Cut 17**
   - verified return \( MIN, MAX \)
3. SQUARE ROOT TREE

\[ \alpha \geq 0 \]

\[ \text{TRIAL} : 0 \]

\[ \text{INCR} : 1000 \]

\[ \text{cut 6 verified} \]

\[ \text{SQR(TRIAL)} < \alpha, \text{INCR is a power of } 10 \]

\[ \text{pc : true, } X : \alpha \]

\[ \text{INCR} < 1 \]

\[ \text{cut 18 verified} \]

\[ \sqrt{\alpha} < \text{TRIAL} \leq \sqrt{\alpha}, \quad \alpha = \alpha' \]

\[ \text{INCR} \geq 1 \]

\[ \text{cut 8 verified} \]

\[ \text{INCR} > 1, \alpha = \alpha', \text{INCR is a power of } 10 \]
\[
\text{cut 8: } pc : \text{true, } X : \alpha
\]

\[
\text{cut 10 verified INCR } \geq 1, \alpha = \alpha', \text{ INCR is a power of 10}
\]

\[
\text{SQR(TRIAL) } \leq \alpha
\]

\[
\text{cut 13 verified INCR } \geq 1, \alpha = \alpha', \text{ INCR is a power of 10}
\]

\[
\sqrt{\alpha} < \text{TRIAL} < \sqrt{\alpha} + \text{INCR}, \text{ INCR is a power of 10}
\]

\[
\text{cut 13: } pc : \text{true, } X : \alpha
\]

\[
\text{cut 14: } TRIAL : TRIAL - \text{INCR}
\]

\[
\text{cut 15 verified INCR } \geq 1, \alpha = \alpha', \sqrt{\alpha} - \text{INCR} < \text{TRIAL}
\]

\[
\leq \sqrt{\alpha}, \text{ INCR is a power of 10}
\]
cut 15

pc : true, X : a

15

pc : INCR > 1, \( \alpha = \alpha' \),
\( \sqrt{\alpha} - \text{INCR} < \text{TRIAL} < \sqrt{\alpha} \),
INCR is a power of 10

16

INCR : INCR/10

7

INCR < 1

cut 18 verified

18

\( \sqrt{\alpha} - 1 \leq \text{TRIAL} \leq \sqrt{\alpha} \),
\( \alpha = \alpha' \)

8

cut 8 verified

\( \sqrt{\alpha} > 1 \), \( \alpha = \alpha' \),
INCR is a power of 10

19

verified return TRIAL

pc : true, X : a

18

pc : \( \sqrt{\alpha} - 1 < \text{TRIAL} \leq \sqrt{\alpha} \)
\( \alpha = \alpha' \)
4. FACTORIAL TREE

\[ X = a! \quad a = a' \]

**Cut 2**
- \( pc : true, N : a \)
  - \( \alpha \geq 0 \)

**Cut 6**
- \( pc : true, N : a \)
  - \( X = a!/(a-1)! \)
  - \( \alpha = \alpha', X = \alpha!/(\alpha-1)! \)

**Cut 14**
- \( \alpha = \alpha', X = \alpha!/(\alpha-1)! \)
  - \( I < \alpha \)
  - \( I \geq \alpha \)
cut 8

\[ \text{pc : true, } N : \alpha \]

\[ \text{pc : } I < \alpha, \quad \alpha = \alpha', \quad X = \alpha'/((\alpha-I)!) \]

8

\[ X : X \times (\alpha-I) \]

9

cut 10 verified \( I < \alpha, \alpha = \alpha', X = \alpha'/((\alpha-I-1)!) \)

10

cut 10

\[ \text{pc : true, } N : \]

\[ \text{pc : } I < \alpha, \quad \alpha = \alpha', \quad X = \alpha'/((\alpha-I)!) \]

10

\[ I : I + 1 \]

11

cut 12 verified \( I \leq \alpha, \alpha = \alpha', X = \alpha'/((\alpha-I)!) \)

12
I > α

cut 14 verified
X = α', α = α'

cut 12
pc : true, N : α

pc : I ≤ α, α = α', X = α!/ (α-1)!

I ≤ α

cut 8 verified I ≤ α,
X = α', X = α!/ (α-1)!

I > α

cut 14
pc : true, N : α

pc : X = α', α = α'

verified return X

49
5. POWER TREE

\[ \begin{align*}
\text{cut 2} & \\
\text{pc : true, } X: \alpha, Y: \beta \\
\text{\( \beta \geq 0 \)} & \\
\text{4} & \\
\text{A : 1} & \\
\text{5} & \\
\text{I : 0} & \\
\text{\( \beta < I \)} & \\
\text{6} & \\
\text{\( \beta > I \)} & \\
\text{cut 12 verified} & \\
\text{\( \beta \geq 0 \), \( A=\alpha' \), \( \alpha = \alpha' \), \( \beta = \beta' \)} & \\
\text{cut 7 verified} & \\
\text{\( \beta > I \), \( A=\alpha I \), \( \alpha = \alpha' \), \( \beta = \beta' \)} &
\end{align*} \]
cut 7

\[ \text{pc : true, } X : \alpha, Y : \beta \]

7

\[ \text{pc : } \beta > I, \ A = \alpha I, \ \alpha = \alpha', \beta = \beta' \]

8

\[ A : A * \alpha \]

9

\[ I : I + 1 \]

cut 10 verified \( \beta > I \),
\[ A = \alpha I, \ \alpha = \alpha', \beta = \beta' \]
cut 10
\[ \text{pc: true, } X : \alpha, Y : \beta \]
\[ \text{pc: } \beta \geq I, A = \alpha^I, \alpha = \alpha', \beta = \beta' \]

\[ \text{cut 7 verified} \]
\[ \text{cut 12 verified} \]
\[ A = \alpha^\beta, \alpha = \alpha', \beta = \beta' \]

cut 12
\[ \text{pc: true, } X : \alpha, Y : \beta \]
\[ \text{pc: } A = \alpha^\beta, \alpha = \alpha', \beta = \beta' \]

\[ \text{verified return } A \]
6. FIBONACCI TREE

\[ \text{cut 2} \]

\[ \text{pc : true, N : } \alpha \]

\[ \alpha > 2 \]

\[ \alpha(1) : 1 \]

\[ \alpha(2) : 1 \]

\[ I : 3 \]

\[ I > \alpha \]

\[ I \leq \alpha \]

\[ \text{cut verified } I \leq \alpha, \ a = \alpha', \ X(1) \ldots X(I-1) \text{ are the first } I-1 \text{ Fibonacci numbers} \]

\[ \text{cut verified } I > \alpha, \ X(1) \ldots X(N) \text{ are the first } N \text{ Fibonacci numbers} \]
I > α

I < α

cut 12 verified
X(1)...X(N) are
the first N Fibonacci numbers, α=α'

cut 8 verified I < α,
α=α', X(1)...X(I-1)
are the first I-1 Fibonacci numbers

pc : I ≤ α, α=α', X(1)...X(I-1) are the first N Fibonacci numbers
X(I) : X(I-1) + X(I-2)

I : I + 1
pc : true, N : α

pc : X(1)...X(N) are the first N Fibonacci numbers

verified return X
Bibliography


Vita

Debra Fish, daughter of Dr. and Mrs. Henry Fish, was born in Scranton, Pennsylvania, on November 2, 1953. She graduated from Central High School, Scranton, Pennsylvania, in 1971. In 1975, she received a Bachelor of Arts Degree in Mathematics from Beaver College, Glenside, Pennsylvania. While at Beaver, she served as President of the Mathematical Honorary Society, was on the Dean's Distinguished Honor List, and received departmental honors. In September 1975, she began work toward the Master of Science Degree in Computer Science at Lehigh University, Bethlehem, Pennsylvania. During that time she was a teaching assistant in the Department of Mathematics.