SRI International

Technical Note 86

October 1973 (Revised March 1974)

REASONING ABOUT PROGRAMS

By: R. J. Waldinger
Artificial Intelligence Center

K. N. Levitt
Computer Science Group

Artificial Intelligence Center

SRI Project 2245

The research reported herein was primarily sponsored by the National Science Foundation under Grant GJ-36146. Additional support was provided by the National Aeronautics Space Administration under Contract NASW-2086 and by the Advanced Research Projects Agency under Contract DAHC04-72-C-0008.

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

ABSTRACT

This paper describes a theorem prover that embodies knowledge about programming constructs, such as numbers, arrays, lists, and expressions. The program can reason about these concepts and is used as part of a program verification system that uses the Floyd-Naur explication of program semantics. It is implemented in the QA4 language; the QA4 system allows many pieces of strategic knowledge, each expressed as a small program, to be coordinated so that a program stands forward when it is relevant to the problem at hand. The language allows clear, concise representation of this sort of knowledge. The QA4 system also has special facilities for dealing with commutative functions, ordering relations, and equivalence relations; these features are heavily used in this deductive system. The program interrogates the user and asks his advice in the course of a proof. Verifications have been found for Hoare's FIND program, a real-number division algorithm, and some sort programs, as well as for many simpler algorithms. Additional theorems have been proved about a pattern matcher and a version of Robinson's unification algorithm. The appendix contains a complete, annotated listing of the deductive system and annotated traces of several of the deductions performed by the system.

CONTENTS

ILLUSTRATIONS

TABLES

ACKNOWLEDGMENTS

The work on program verification was done in close collaboration with Bernie Elspas. The work on QA4 was done with Jeff Rulifson and Jan Derksen. Irene Greif wrote the first version of the simplifier and participated in the conceptualization of the pattern matcher and unification proofs. Jeff Rulifson encouraged us to write this paper and suggested its format. Rich Fikes has helped with design modification and debugging of QA4. Bert Raphael read the manuscript and suggested many improvements. This work has benefitted from our conversations with Cordell Green, Peter Neumann, Earl Sacerdoti, René Reboh, Mark Stickel, and Steve Crocker. Many members of the Artificial Intelligence Center and the Computer Science Group at SRI helped with support and criticism.

A shorter version of this paper appears in Proceedings SIGACT/ SIGPLAN Symposium on Principles of Programming Languages, October 1-3, 1973.

Problems worthy
of attack
Prove their worth
by hitting back.

Piet Hein

I INTRODUCTION AND BACKGROUND

This paper describes a computer program that proves theorems about programs. Proving theorems about programs is of practical importance because it helps certify that they are correct. Instead of testing a program on test cases, which may allow some bugs to remain, we can try to prove mathematically that it behaves as we expect. We hope future systems that reason about programs and understand how they work will help us to write and change programs.

Many programs have done this sort of reasoning. James King [1969]* developed a program verifier that could prove theorems about programs; his program proved an interesting class of theorems and was very fast. Peter Deutsch [1973] has recently written a system for interactive program writing that can also prove things about programs. It is perhaps not as fast as King's system, but it can prove more interesting theorems. S. Igarashi, R. London, and D. Luckham [1973] have recently applied a resolution theorem prover to program verification, and their results are impressive also. They can verify such programs as Hoare's [1961] FIND. Their system does little actual resolution and a lot of simplification and reasoning about equality. A program devised by Boyer and Moore [1973] can prove difficult theorems about LISP programs.

Thus, there is no shortage of interesting work related to our own. The special characteristic of our own system is that it is markedly concise, readable, and easy to change and apply to new subject areas.

*References are listed alphabetically at the end of the paper.

Our program verifier consists of a theorem prover (or deductive system) and a "verification condition generator." The verification condition takes an annotated program as input and constructs a list of theorems as output. The truth of the constructed theorems implies the correctness of the program. The task of the deductive system is to prove these theorems. The verification condition generator [Elspas et al., 1973] is written in BBN-LISP [Teitelman et al., 1971], and the deductive system is written in QA4 [Rulifson et al., 1972]. This paper focuses on the deductive system but, to be complete, gives examples of verification condition generation as well.

In writing our deductive system, we were motivated by several goals. First, the system should be able to find proofs; it should have enough deductive power to prove, within a comfortable time and space, the theorems being considered. Also, these proofs should be at the level of an informal demonstration in a mathematical textbook. This means that the difficulty in following one line to the next in any proof should be small enough that the proof is understandable, yet large enough not to be trivial. In any practical program verifier, the user will wish to follow the steps in a deduction. Who would believe a program verifier that only printed out "true" in the course of pursuing a proof? Furthermore, the strategies the system uses in searching for a proof should be strategies that we find natural. Not only should the tactics that eventually lead to the proof be ones we might use in proving the statement by hand, but also the false starts the system makes should be ones we might make ourselves. We do not want the system to rely on blind search; the trace of an attempted solution should make interesting reading.

In addition to the requirement that proofs be readable, the rules the system uses in going from one line to the next should be easy to read and understand. We should be able to look at a rule and see what it does. Also, it should be easy to change old rules and to add new rules. The user of a program verifier is likely to introduce new concepts, such as operators or data structures. We want to be able to tell the deductive system how these structures behave and to have the system reason effectively using the new symbols. Giving the system new information should be possible without knowing how the system works, and certainly without reprogramming the system. Furthermore, the addition of new information should not prohibitively degrade the performance of the system.

The system is intended to evolve with use. As we apply the system to new problems, we are forced to give the system new information and, perhaps, to generalize some old information. These changes are incorporated into the system, which may then be better able to solve new problems.

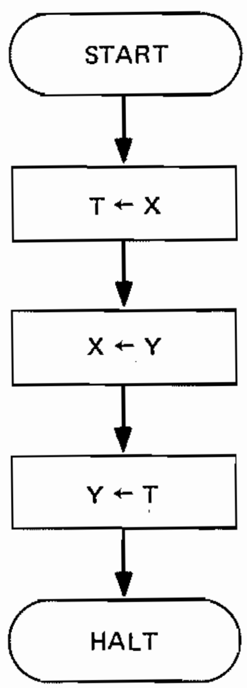
Since the system is easy to extend and generalize, we do not worry about the completeness or generality of any particular version of the system. It is powerful enough to solve the sort of problem on which it has been trained, and it can be easily changed when necessary.

These considerations played a part in the design of the programming system called QA4, as well as in the construction of our deductive system, which is written in the QA4 language. Some of the techniques described below are embedded in the QA4 system itself; others are expressed as parts of the deductive system.

II THE FLOYD-NAUR METHOD

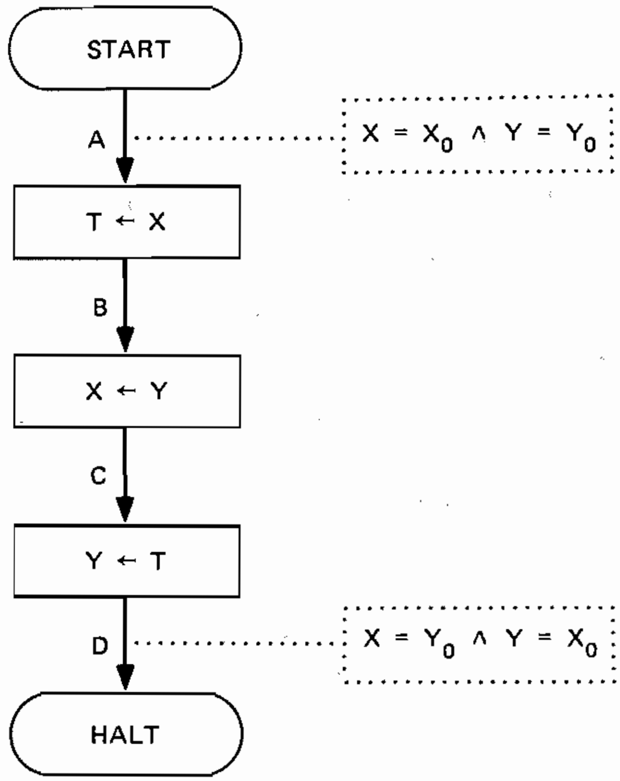
Perhaps not all readers are familiar with the method of proving statements about programs that we have followed in our work. Our method is a natural technique introduced independently by Floyd [1967] and Naur [1966] and formalized by Hoare [1969]. Knuth [1968] traces the germ of the idea back to von Neumann and Goldstine [1963] in the paper that introduced the concept of the flow chart. Although we cannot give a thorough introduction to that subject here, we provide below an example of its application to convey the flavor of the approach.

Consider a simple program that exchanges the values of two variables:



TA-740522-3

We assume that before the program is executed, X and Y have some initial values X_0 and Y_0. Suppose we want to prove that after the program is executed, $X = Y_0$ and $Y = X_0$. We offer these <u>input</u> and <u>output</u> <u>assertions</u> as comments in our program:

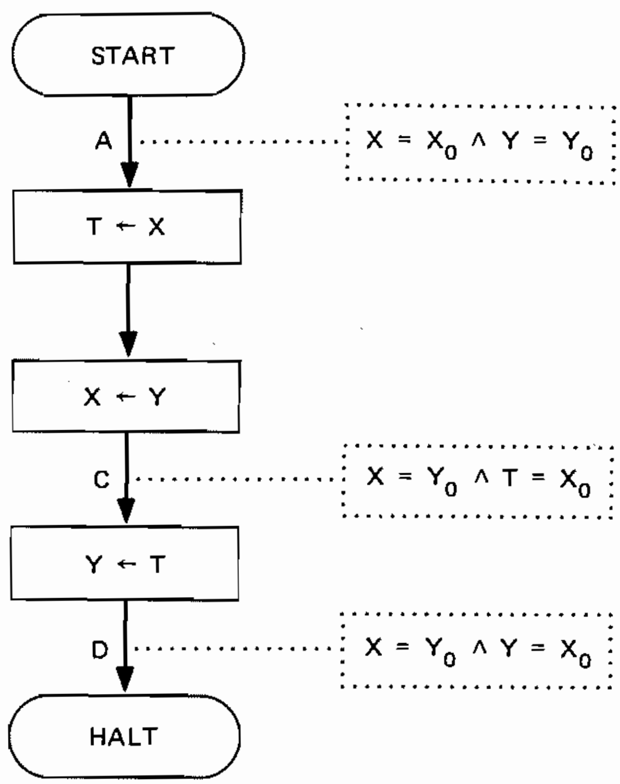


TA-740522-4

These assertions are not to be executed by the program in any way, but they tell us something about the way the programmer expects his program to behave. He expects the assertion at A to be true when control passes through A, and the assertion at D to be true when control passes through D.

The essence of the Floyd-Naur approach is to generate from a commented program like the one above a set of statements called the verification conditions. If these statements are true, then the assertions the programmer has put in his program are correct. Whereas the programmer's assertions are correct only when control passes through the appropriate point, the verification conditions are true in general, and they can be proved by a deductive system that knows nothing about sequential processing, loops, recursion, or other concepts about the flow of control and nothing else about the particular program.

To generate the verification condition for our sample program, we pass the output assertion back toward the input assertion. As we pass it back, we change it to reflect the changing state of the system. In particular, if any assignments are made within the program, then the corresponding substitution should be made in the assertion. Passing the assignment at D back to point C changes it to $X = Y_0$ and $T = X_0$:



TA-740522-5

We can argue that if the assertion at C is true when control passes through C, then the assertion at D will be true when control passes through D. In particular, if $T = X_0$ is true at C, and we execute $Y \leftarrow T$, then $Y = X_0$ will be true at D.

Passing the assertion all the way back to A in this manner gives the assertion $Y = Y_0 \wedge X = X_0$. If this assertion is true at A, then the final assertion will be true at D. However, we are already given the initial assertion $X = X_0 \wedge Y = Y_0$. The truth of the assertion at D then depends on the truth of the implication $X = X_0 \wedge Y = Y_0 \supset Y = Y_0 \wedge X = X_0$. This statement is the verification condition for this program. It can be proved by a deductive system independently of any knowledge about this program.

Constructing verification conditions by this method is an algorithmic process, not a heuristic one. Although the design of a language for expressing assertions remains an important and challenging problem (we introduce below some constructs for such a language), it is not a problem in the artificial intelligence domain. On the other hand, there is no cut and dried algorithm for proving verification conditions, and this is thus a fit subject for artificial intelligence research. Although we have no general algorithm for proving verification conditions, this somewhat restricted domain is more tractable than the general theorem-proving problem.

III A PROGRAM THAT FINDS THE LARGEST ELEMENT OF AN ARRAY

Before we explain how the system is structured or implemented, let us first look at a sample of some deductions performed by our system. This example will give a better idea of the subject domain of the inference system and of the sort of reasoning we have to do. It will also give a better picture of the process of generating a verification condition (Floyd [1967]).

Suppose we are given the annotated program shown in Figure 1 to compute the largest element in an array and its location. This program searches through the array, keeping track of the largest element it has seen so far and the location of this element. The intermediate assertion at C* says that MAX is the largest element in the array between 0 and I and that LOC is the index for MAX. Although our assertion language does not permit the ellipsis notation ("..."), we have introduced some suitable analogues, which are discussed later.

To prove assertions about a complex program, the system decomposes it into simple paths. This program can be decomposed into four simple paths:

- The path from B to C.
- The path from C to D.
- The path from C around the loop and back to C through point E.
- The path from C around the loop and back to C through point F.

*In this program, and in examples throughout the paper, when we list several statements in an assertion, we mean the implicit conjunction of those statements. We will often also refer to each conjunct as an "assertion."

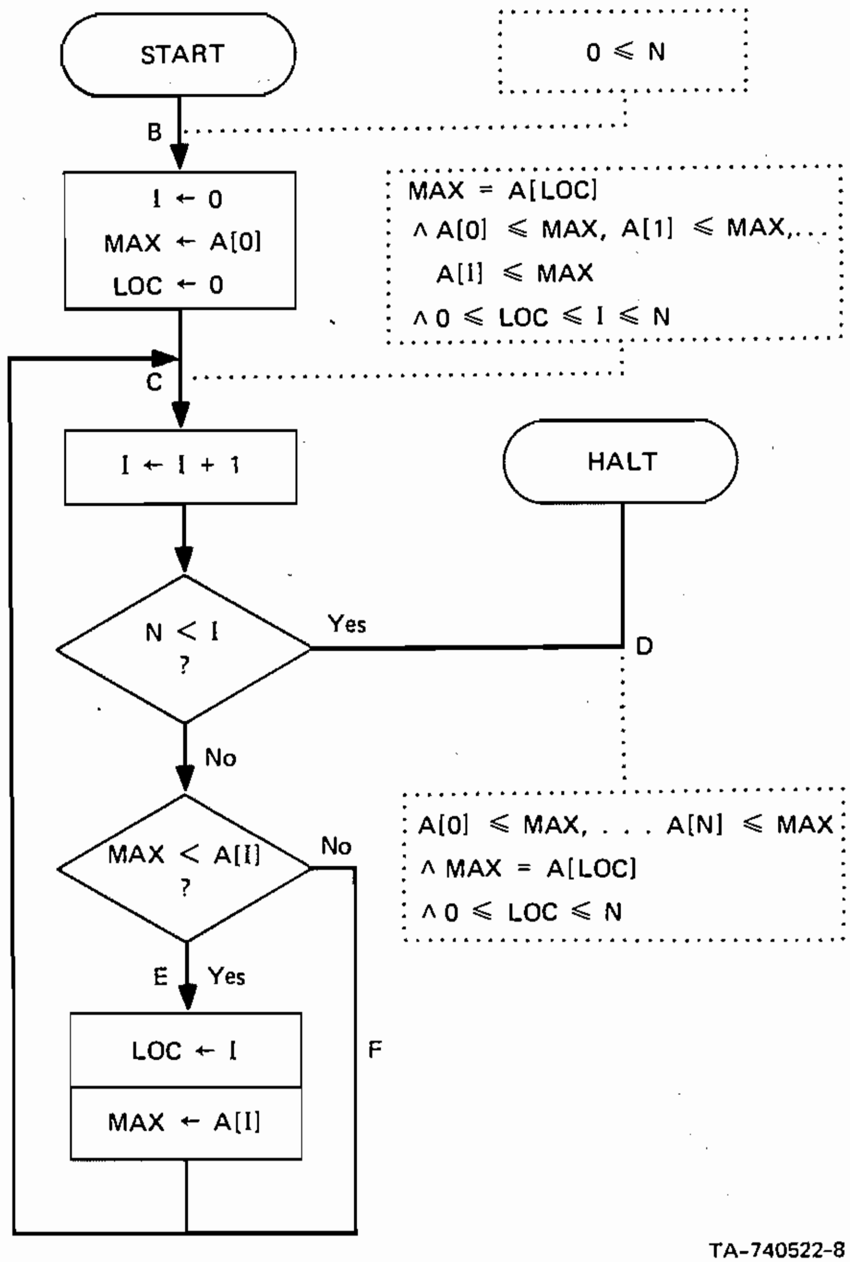


FIGURE 1 FINDING THE MAXIMUM OF AN ARRAY

Notice that the author of this program has put assertions not only at the START and HALT nodes of the program, but also at the intermediate point C. He has done this so that the proof of the program can be reduced to proving straight-line paths in the same way that the simple program of the previous section was verified. For instance, the path that begins at C, travels around the loop through E, and returns to C can be regarded as a simple, straight-line program with the assertion at C as both its start assertion and its halt assertion. The assertion at C has been cleverly chosen to be true when the loop is entered, to remain true whenever control travels around the loop and returns to C, and to be strong enough to allow the assertion at D to be proved when control leaves the loop and the program halts. (The choice of suitable internal assertions can be an intellectually exacting task; some heuristic methods have been proposed that will work in this and many other examples (Elspas et al. [1972], Wegbreit [1973], Katz and Manna [1973]).

If all the straight-line paths of the program are shown to be correctly described by the given assertions, and if the program can be shown to terminate (this must be done separately), then we can conclude that the program is indeed correct, at least with respect to the programmer's final assertion.

Although there are many paths in the decomposition of a program, typically most of the paths are easy to verify. For this program, we examine two of the paths.

First, suppose we want to demonstrate that if the assertion at point C is true when control passes through C, then the assertion at C will still be true if control passes around the loop and returns again to C. We will restrict our attention to the more interesting case, in which the test MAX < A[I]? is true; in this case, control passes through E. Furthermore, we will try to prove only that the second conjunct of the assertion at C remains true. Our verification condition generator gives us the following statement to prove:

$$\text{MAX} = A[\text{LOC}] \wedge \qquad (1)$$

$$A[0] \le \text{MAX}, \ldots, A[I] \le \text{MAX} \wedge \qquad (2)$$

$$0 \le \text{LOC} \le I \le N \wedge \qquad (3)$$

$$\neg (N < I+1) \wedge \qquad (4)$$

$$\text{MAX} < A[I+1] \supset \qquad (5)$$

$$A[0] \le A[I+1], \ldots, A[I+1] \le A[I+1] . \qquad (6)$$

This statement is actually represented as five separate hypotheses and a goal to be deduced from these hypotheses. Lines (1) through (3) come from the assertion at C, and lines (4) and (5) come from the tests along the path. Line (6) comes from the assertion at C again. How the above statement is derived from the program is shown in detail in Appendix C.

The behavior of the deductive system in this problem is typical of its approach to many problems. The goal, (6), is broken into two subgoals:

$$A[0] \le A[I+1] \wedge \ldots A[I] \le A[I+1] \qquad (7)$$

and

$$A[I+1] \le A[I+1] \quad . \qquad (8)$$

The second subgoal, (8), is immediately seen to be true. The first subgoal, (7), is easily derived from (2) and (5).

Now let us look at the path from C to D. We will assume the assertion at C is true and will prove the assertion at D. We will look at the first conjunct of the assertion at D. Our verification condition generator gives us the following statement to prove:

$$\text{MAX} = A[\text{LOC}] \wedge \qquad (9)$$

$$A[0] \le \text{MAX} \wedge \ldots \wedge A[I] \le \text{MAX} \wedge \qquad (10)$$

$0 \le LOC \le I \le N \land$ (11)

$N < I+1 \supset$ (12)

$A[0] \le MAX \land \ldots \land A[N] \le MAX$. (13)

The reasoning required for this proof is a little more subtle than the previous deduction. When the system learns that $N < I+1$ (12), it immediately concludes that $N+1 \le I+1$, since N and I are integers. It further deduces that $N \le I$. Since it already knows that $I \le N$ (11), it concludes that $N = I$. Using hypothesis (10), the system reduces the goal (13) to proving that $I = N$, which it now knows.

This deduction involves a lot of reasoning forward from assumptions and not much reasoning backward from goals. Both of these proofs are typical of the behavior of the system at large because of their strong use of the properties of equality and the ordering relations.

The QA4 system incorporates enough of the common techniques of theorem proving and problem solving that our inference system needs no general problem-solving knowledge, but only some knowledge about numbers, arrays, and other structures. The following sections show how the QA4 language allows that knowledge to be represented.

IV THE QA4 LANGUAGE

A. Pattern Matching and the Goal Mechanism

The deductive system is made up of many rules expressed as small functions or programs. Each of these programs knows one fact and the use for that fact. The QA4 programming language is designed so that all these programs can be coordinated; when a problem is presented to the system, the functions that are relevant to the problem "stand forward" in the sense explained below.

A program has the form

(LAMBDA ⟨pattern⟩⟨body⟩) .

Part of the knowledge of what the program can be used for is expressed in the pattern. When a function is applied to an argument, the pattern is matched against that argument. If the argument turns out to be an instance of the pattern, the match is said to be successful. The unbound variables in the pattern are then bound to the appropriate subexpressions of the argument, and the body of the program is evaluated with respect to those new bindings.

For example, the program

REVTUP = (LAMBDA (TUPLE ←X ←Y) (TUPLE $Y $X))

has pattern (TUPLE ←X ←Y) and body (TUPLE $Y $X). The prefix "←" means that the variable is to be given a new binding. The prefix "$" means that the variable's old binding is to be used. When REVTUP is applied to (TUPLE A B), the pattern (TUPLE ←X ←Y) is matched against (TUPLE A B). The match is seen to be successful, the variable X to be bound to A and the variable Y to be bound to B. The body (TUPLE $Y $X) is evaluated with respect to these bindings, giving (TUPLE B A).

On the other hand, if a function is applied to an argument and the pattern of that function does not match the argument, a condition known as failure occurs. At many points in the execution of a program, the system makes an arbitrary choice between alternatives. Failure initiates a backing up to the most recent choice and the selection of another alternative. The mismatching of patterns is only one of the ways in which failure can occur in a program.

We have yet to explain how a program stands forward when it is relevant. In the above example, the function was called by name, much as it is in a conventional programming language. But it is also possible to make an argument available to any applicable program in a specified class. This is done by means of the goal mechanism.

When we say (GOAL ⟨goalclass⟩⟨argument⟩), we assume that the goal class is a tuple of names of functions. We are making that argument available to the entire class of functions. The pattern of each of those functions is matched in turn against the argument. If the match is successful, the function is applied to that argument. If the function returns a value, that value is returned as the value of the goal statement. On the other hand, if a failure occurs in evaluating the function, backtracking occurs, the next function in the goal class is tried, and the process is repeated. If none of the functions in the goal class succeed, the entire goal statement fails.

For example, in our deductive system, one of the goal classes is called EQRULES, the rules used for proving equalities. One of these rules is

```
EQTIMESDIVIDE = (LAMBDA (EQ ←W (TIMES (DIVIDE ←X ←Y) ←Z))
                   (GOAL $EQRULES
                      (EQ (TIMES $Y $W)(TIMES $X $Z))))   .
```

This rule states that to prove W = (X/Y)*Z, we should try to prove Y*W = X*Z. (The actual EQTIMESDIVIDE, shown in Appendix A, is more general than this.) The rule has the pattern

(EQ ←W (TIMES (DIVIDES ←X ←Y) ←Z)) .

If we execute (GOAL $EQRULES (EQ A (TIMES (DIVIDES B C) D))) [i.e., we want to prove A = (B/C)*D], the system will try all the applicable EQRULES in turn. If none of the previous rules succeed, the system will eventually reach EQTIMESDIVIDE. It will find that the pattern of EQTIMESDIVIDE matches this argument, binding W to A, X to B, Y to C, and Z to D. Then it will evaluate the body of this function; i.e., it will try

(GOAL $EQRULES (EQ (TIMES A C) (TIMES B D))) .

If it fails to prove (EQ (TIMES A C)(TIMES B D)), it will try to apply the remaining EQRULES to the original argument, (EQ A (TIMES (DIVIDES B C) D)). The goal statement is an example of the pattern-directed function invocation introduced by Hewitt in PLANNER [1971].

The net effect of this mechanism is that it enables the user to write his programs in terms of what he wants done, without needing to specify how he wants to do it. Furthermore, at any point, he can add new rules to EQRULES or any other goal class, thus increasing the power of the system with little effort.

B. Some Sample Rules

The deductive system is a collection of rules represented as small programs. One rule was given in the preceding section; two more rules are presented here. The complete deductive system is included in Appendix A.

The first rule, EQSIMP, attempts to prove an equality by simplifying its arguments:

```
EQSIMP = (LAMBDA (EQ ←X ←Y)
            (PROG (DECLARE)
                  (SETQ ←X ($SIMPONE $X))
                  (GOAL $EQRULES (EQ $X $Y)))
            BACKTRACK)    .
```

This rule says that to prove terms A and B are equal, simplify A and then prove that the simplified A is equal to B. This rule, a member of EQRULES, has the pattern (EQ ←X ←Y). SIMPONE, the simplifier, will fail if its argument cannot be simplified. In that case, EQSIMP will also fail. EQSIMP can actually simplify the right side of an equality, as well as the left, as explained in the Appendix A.

The second rule is

```
FSUBTRACTI = (LAMBDA (←F (SUBTRACT ←X ←Y) ←Z)
                     (GOAL $INEQUALITIES
                        ($F $X (PLUS $Y $Z))))    .
```

This rule says that to prove $X-Y \leq Z$, try to prove $X \leq Y+Z$. It belongs to the goal class INEQUALITIES and is thus used not only for the predicate LTQ, but also for LT, GT, and GTQ. The variable F is bound to the appropriate predicate symbol.

C. Demons

The goal mechanism is used for reasoning backward from a goal. However, sometimes we want to reason forward from a statement. For example, suppose that whenever an assertion of the form $X \geq Y$ is asserted, we want to assert $Y \leq X$ as well. We do this by a QA4 mechanism known as the demon.

A demon is imagined to be a spirit that inhabits a hiding place, waiting until some specified event occurs, at which time it appears, performs some action, and vanishes again. We have put several demons in the system, each watching for a different condition. For instance, one demon watches for statements of the form X ≥ Y and makes the statement Y ≤ X. The user of the system can create his own demons. Demons are a tool for reasoning forward from an antecedent. In particular, we use demons to drive antecedents into a canonical form. For example, we drive all inequality expressions with integer arguments into an assertion of the form X ≤ Y.

D. Representations

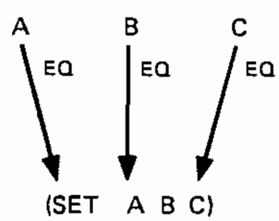
To as great an extent as possible, we have chosen representations that model the semantics of the concepts we use so as to make our deductions shorter and easier. For example, our language has data structures expecially intended to eliminate the need for certain inferences. In addition to tuples, which are like the familiar lists of the list-processing languages, we have the finite sets of conventional mathematics and bags, which are unordered tuples or, equivalently, sets that may have multiple occurrences of the same element. (Bags are called multisets by Knuth [1969], who outlines many of their properties.) Furthermore, we allow arbitrary expressions to have property lists in the same way that atoms can have property lists in LISP [McCarthy et al., 1962].

These data structures are useful in the modeling of equivalence relations, ordering relations, and arithmetic functions. For instance, if the addition of numbers and the multiplication of numbers are each rerepresented by a function of two arguments, then it becomes necessary to use numerous applications of the commutative and associative laws to prove anything about the number system. However, in QA4 all functions take only one argument, but this argument can be a tuple, set, or bag, as well as any other expression. Functions of multiple arguments can be represented by a function defined on tuples. However, a function that is commutative and associative, such as PLUS, is defined on bags. The expression (PLUS A 2 B) really means (PLUS (BAG A 2 B)). Recall that bags are unordered; the system cannot distinguish between (BAG A 2 B) and (BAG 2 A B). Consequently, the expressions (PLUS A 2 B) and (PLUS 2 A B) are identically equal in our system. This makes the commutative law for addition redundant and, in fact, inexpressible in the language. Most needs for the associative law are also avoided.

The logical function AND has the property that, for instance, (AND A A B) = (AND A B). The number of occurrences of an argument does not affect its value. Consequently, AND takes a set as its argument. Since (SET A A B) and (SET A B) are indistinguishable, (AND A A B) and (AND A B) are identical, and a statement of their equality is unnecessary. Some functions that take sets as arguments are AND, OR, EQ, and GCD (greatest common divisor).

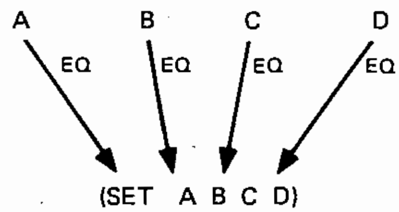
When a new fact is asserted to our system, the value TRUE is placed on the property list of that fact. If at some later time we want to know if that fact is true, we simply look on its property list.

However, certain facts are given special handling in addition. For example, if we tell the system that certain expressions are equal, we form a set of those expressions. On the property list of each expression, we place a pointer to that set. For instance, if we assert (EQ A B C), the system stores the following:



TA-740522-6

If we subsequently discover any of these expressions to be equal to still another expression, the system adds the new expression to the previously formed set and puts the set on the property list of the new expression as well. For instance, if we assert (EQ B D), our structure is changed to the following:



TA-740522-7

The transitivity, symmetry, and reflexivity of equality are thus implicit in our representation. If we ask whether A and D are equal, the system knows immediately by looking at the property list of A or D.

Ordering relations are stored using the property list mechanism. If we know that some expression A is less than B, we place a pointer to B on the property list of A:

$$A \xrightarrow{LT} B \quad .$$

If we learn that B is less than C, we put a pointer to C on the property list of B:

$$A \xrightarrow{LT} B \xrightarrow{LT} C \quad .$$

If we then ask the system if A is less than C, it will search along the pointers in the appropriate way to answer affirmatively. The transitive law is built into this representation.

The system knows about LT (<), GT (>), LTQ ($\leq$), GTQ ($\geq$), EQ (=), NEQ ($\neq$), and how these relations interact. For example, if we assert

$X \geq Y$, $Y \geq Z$, and $X \leq Z$, the system will know $X = Y = Z$ and that (F X A) = (F Y A). Or if we assert $X \geq Y$ and $X \neq Y$, the system will know $X > Y$.

E. Contexts

When we are trying to prove an implication of the form $A \supset B$, it is natural to want to prove B under the hypothesis that A is true. Our assumption of the truth of A holds only as long as we are trying to prove B; after the proof of B is complete, we want to forget that we have assumed A. For this and other reasons, the QA4 language contains a context mechanism. All assertions are made with respect to a context, either implicitly or explicitly. For any context, we can create an arbitrary number of lower contexts.

A query made with respect to a context will have access to all assertions made with respect to higher contexts but not to any assertions made with respect to any other contexts. For instance, suppose we are trying to prove $i < j \supset i + 1 \leq j$ with respect to some context C_0. We may have already made some assertions in context C_0. We establish a lower context, C_1, and assert $i < j$ with respect to C_1. Then we try to prove $i + 1 \leq j$ with respect to C_1. When proving $i + 1 \leq j$, we know $i < j$, as well as all the assertions we knew previously in C_0. When the proof of B is complete, we may have other statements to prove in C_0. In doing these proofs, we will know all the assertions in C_0 and also, perhaps the assertion $i < j \supset i + 1 \leq j$, buy we will not know $i < j$ because it was asserted with respect to a lower context.

F. User Interaction

Sometimes our rules ask us whether they should continue or fail. This allows us to cut off lines of reasoning that we know in advance to be fruitless. If we make a mistake in answering the question, we may cause the system to fail when it could have succeeded. However, we never cause the system to find a false or erroneous proof.

In addition to these mechanisms, which are built into the language processor, we have developed some notations that make it easier to discuss programming constructs; these notations are a part of our assertion language and are interpreted by the deductive system.

V NOTATIONS

In speaking about the program to find the maximum element of an array, we found it convenient to use the ellipsis notation ("..."). We have not introduced this notation into our language; however, we have found ways of getting around its absence.

A. TUPA, SETA, BAGA

Let A be a one-dimensional array and I and J be integers. Then (TUPA A I J) is the tuple

(TUPLE A[I], A[I+1], ..., A[J]) .

If I > J, then (TUPA A I J) is the empty tuple.

(SETA A I J) and (BAGA A I J) are the corresponding bag and set. To state that an array is sorted between 0 and N, we assert

(LTQ (TUPA A 0 N)) .

To state that an array A is the same in contents between 0 and N as the initial array A_0, although these contents may have been permuted, we assert

(EQ (BAGA A 0 N) (BAGA A_0 0 N)) .

B. The STRIP Operator

Let X be a set or bag, X = (SET X_1, ..., X_n), or X = (BAG X_1, ..., X_n). Then (LTQ (STRIP X) Y) means $X_1 \leq Y$ and ... $X_n \leq Y$. For instance, to state that MAX is greater than or equal to any element in an array A between I and J, we assert

(LTQ (STRIP (BAGA A I J)) MAX) .

This is perhaps not quite as clear as

$$A[I] \leq MAX, \ A[I+1] \leq MAX, \ \ldots, \ A[J] \leq MAX \quad ,$$

but we prefer it to

$$(\forall u) \ [I \leq u \wedge u \leq J) \supset A[u] \leq MAX] \quad .$$

The STRIP operator is also used to remove parentheses from expressions:

(BAG A (STRIP (BAG B C D)))

is

(BAG A B C D) .

We will eventually need two distinct operators, one to act as a quantifier and one to remove parentheses, but the single operator STRIP has played both roles so far.

C. ACCESS and CHANGE

Arrays cannot be treated as functions because their contents can be changed, wheras functions do not change their definitions. Thus, while f(x) is likely to mean the same thing for the same value of x at different times, A[x] is not. We overcome this difficulty by adopting McCarthy and Painter's [1967] functions ACCESS and CHANGE in our explication of the array concept:

- (ACCESS A I) means A[I].
- (CHANGE A I T) means the array A after the assignment statement A[I] ← T has been executed.

We do not propose that ACCESS and CHANGE be used in writing programs or assertions; we do find that they make reasoning about arrays simpler, as

King suspected they would.

The next sections show examples of some fairly difficult proofs produced by the deductive system. The actual traces for some of these are included in Appendix B.

VI THE REAL NUMBER QUOTIENT ALGORITHM

Very little work has been done to prove properties of programs that work on the real numbers or the floating point numbers, although there is no reason to believe such proofs could not be done. Figure 2 shows, for instance, a program (Wensley [1958], Elspas et al. [1972]) to compute an approximate quotient Y of real numbers P and Q, where $0 \leq P < Q$. This is an interesting and computationally plausible algorithm. It uses only addition, subtraction, and division by two, and it computes a new signigicant bit of the quotient with each iteration.

The algorithm can be understood in the following way. At the beginning of each iteration, P/Q belongs to the half-open interval [Y,Y+D). It is determined whether P/Q belongs to the left half or the right half of the interval. Y and D are adjusted so that in the new iteration, the half-interval to which Y belongs plays the role of the interval [Y, Y+D). Thus Y becomes a better and better approximation for P/Q.

We will consider here only one path through this program, i.e., the path around the loop that follows the right branch of the test P < A+B. We will prove only one loop assertion: P < Y*Q + D*Q. Our verification condition generator supplies us with the following hypotheses:

$$0 \leq P \quad , \tag{14}$$

$$P < Q \quad , \tag{15}$$

$$A = Q*Y \quad , \tag{16}$$

$$2*B = Q*D \quad , \tag{17}$$

$$P < Y*Q + D*Q \quad , \tag{18}$$

$$Y*Q \leq P \quad , \tag{19}$$

$$\neg (D < E) \quad , \tag{20}$$

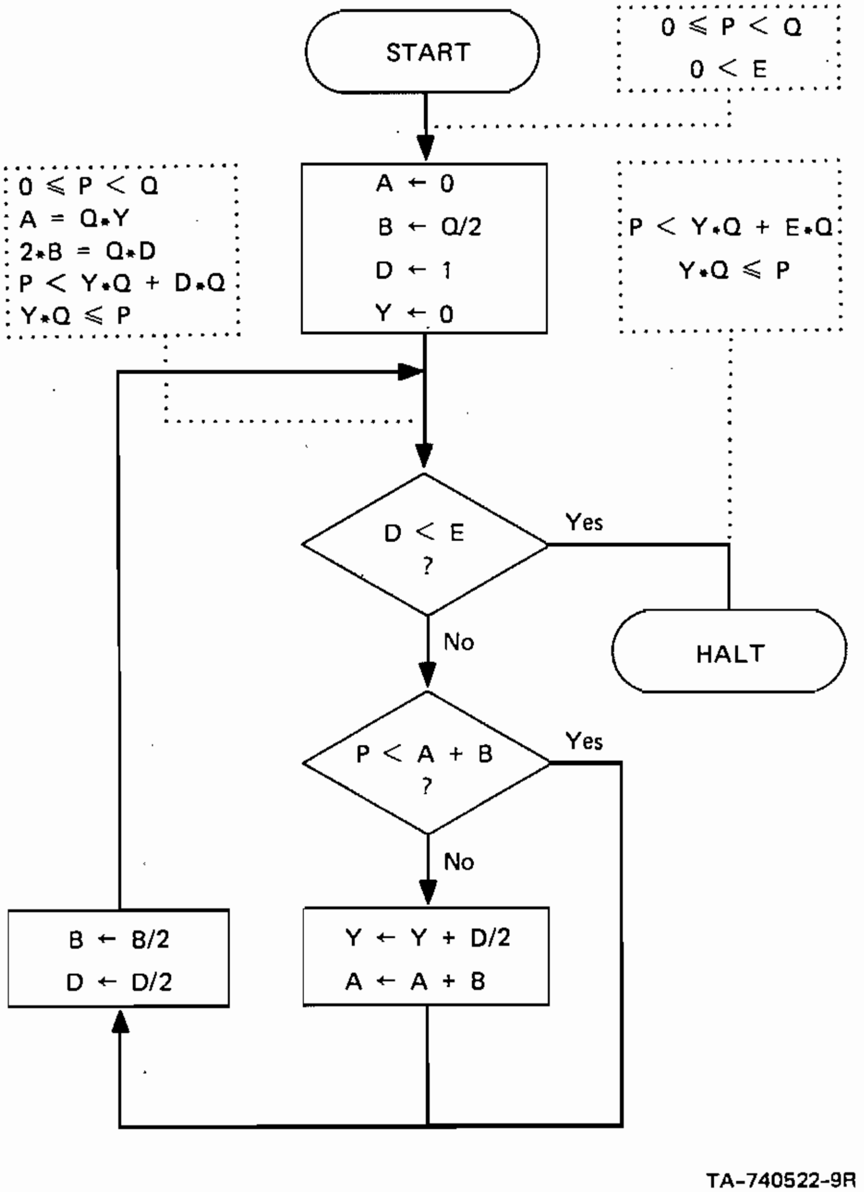


FIGURE 2 THE WENSLEY DIVISION ALGORITHM

$$P < A+B \quad . \tag{21}$$

The goal is to prove from these hypotheses that

$$P < Q*Y + Q*(D/2) \quad . \tag{22}$$

These hypotheses and the goal were constructed in a manner precisely analogous to the generation of the condition for the previous example of computing the maximum of an array.

The proof goes as follows. After an abortive attempt at using the assertion (18), the system tries to show that the conclusion follows from (21). It therefore tries to show that

$$A+B \leq Q*Y + Q*(D/2) \quad . \tag{23}$$

This goal is broken into the following two:

$$A \leq Q*Y \tag{24}$$

$$B \leq Q*(D/2) \tag{25}$$

Of course, this strategy will not always be successful. However, in this case goal (24) follows from (16), whereas (25) reduces to (17).

A complete trace of this proof and listings of the rules required to achieve it are provided in Appendices A and B.

VII A PATTERN MATCHER

As an experiment in the incorporation of new knowledge into the system, we performed the partial verification of a simple pattern matcher and a recursive version of the unification algorithm [Robinson, 1965]. These algorithms were of special interest to us because they involve concepts we have actually used in the implementation of the QA4 program itself. They are thus in some sense realistic, although neither of these programs appears literally in the QA4 code. The subject domain is as follows.

We assume that <u>expressions</u> are LISP S-expressions [McCarthy, 1962]; for example, (F X (G A B)) is an expression. Atomic elements are designated as either <u>constant</u> or <u>variable</u>, and they can be distinguished by the use of the predicates const and var. Here we use A, B, C, F, and G as constants and U, V, W, X, Y, and Z as variables:

- var(X) is true
- const(A) is true
- var(A) is false
- var((X Y)) is false.

A <u>substitution</u> replaces some of the variables of an expression by terms. Substitutions are represented as lists of dotted pairs. ((X • A) (Y • (F G))) is a substitution. Varsubst(s,e) is the result of making substitution s in expression e. If s is

((X • A) (Y • (G B))) ,

and e is

(F X A (Y B)) ,

then varsubst(s,e) is

(F A A ((G B) B)) .

The LISP functions car, cdr, list, and atom can be used to manipulate expressions. The empty substitution is denoted by EMPTY and has no effect on an expression. An operation called compose, the composition of substitutions, defined by Robinson [1965], has the following property:

varsubst(compose(s1, s2), e) = varsubst(s1, varsubst(s2, e)) .

The problem of pattern matching is defined as follows: Given two expressions called the pattern and the argument, try to find a substitution for the variables of the pattern that makes it identical to the argument. We call such a substitution a <u>match</u>. For example, if the pattern is

(X (Y A B) X) ,

and the argument is

(D (C A B) D) ,

then match(pat, arg) is

((X • D) (Y • C)) .

If there is no substitution that makes the pattern identical to the argument, we want the pattern matcher to return the distinguished atom NOMATCH. Thus, if pat is (X Y X) and arg is (A B C), then match(pat, arg) = NOMATCH, since we cannot expect X to be matched against both A and C.

For simplicity, we assume that the argument contains no variables. A LISP-like program to perform the match might be

```
match(pat, arg) = prog ((m1 m2)
   if const(pat) then (if pat = arg then
     return(EMPTY) else return(NOMATCH))
   if var(pat) then return(list (cons(pat, arg)))
   if atom(arg) then return(NOMATCH)
   m1 ← match(car(pat), car(arg))
   if m1 = NOMATCH then return(NOMATCH)
   m2 ← match(varsubst(m1, cdr(pat)), cdr(arg))
   if m2 = NOMATCH then return(NOMATCH)
   return(compose (m2, m1))   .
```

The program does the appropriate thing in the case of atomic patterns or arguments, and it calls itself recursively on the left and right halves of the expressions in the nonatomic case. The program applies the substitution found in matching the left halves of the expressions to the right half of the pattern before it is matched so as to avoid having the same variable matched against different terms.

We have proved several facts about a version of this program, but we focus our attention here on one of them: If the program does not return MOMATCH, then the substitution it finds actually is a match; i.e., that applying the substitution to the pattern makes that pattern identical to the argument. Thus, the output assertion is:

$$\text{match(pat, arg)} \neq \text{NOMATCH} \supset \text{varsubst(match(pat, arg), pat)} = \text{arg} \quad .$$

Since we assume the argument contains no variables, the input assertion is

$$\text{constexp(arg)} \tag{26}$$

We have verified one condition for the longest path of <u>match</u> with respect to these assertions. This path is followed when the pattern and the argument are both nonatomic and when the recursive calls on <u>match</u> successfully return a substitution. In writing our verification condition, we use the same abbreviations the program does, i.e.,

$$m1 = match(car(pat),\ car(arg))$$

and

$$m2 = match(varsubst(m1,\ cdr(pat)),\ cdr(arg)) \quad .$$

In proving a property of a recursively defined program, we follow Manna and Pnueli [1970] and assume that property about the recursive call to the program. Thus, for this program we have the inductive hypotheses

$$constexp(car(arg)) \wedge m2 \neq NOMATCH \supset$$
$$varsubst(m1,\ car(pat)) = car(arg)$$

(the program works for the car of the pattern) and

$$constexp(cdr(arg)) \wedge m2 \neq NOMATCH \supset$$
$$varsubst(m2,\ varsubst(m1,\ cdr(pat))) = cdr(arg) \quad . \qquad (28)$$

(the program works for the instantiated cdr of the pattern). The verification condition generator would split both of these hypotheses into three cases; we will consider only the case in which the antecedents of both implications are ture. Hence, we assume that both the recursive calls to the pattern matcher succeed in finding matches.

By the path we have taken through the program, we know that

$$\neg\, const(pat) \qquad (29)$$

(the pattern is not a constant).

$$\neg\, var(pat) \qquad (30)$$

(the pattern is not a variable).

$$\neg\, atom(arg) \qquad (31)$$

(the argument is not an atom). Since for this path

$$match(pat,\ arg) = compose(m2,\ m1) \quad ,$$

the goal is to prove

varsubst(compose(m2, ml), pat) = arg . (32)

The proof produced by the system proceeds as follows. The goal is split into two subgoals:

varsubst(compose(m2, ml), car(pat)) = car(arg) (33)

and

varsubst(compose(m2, ml), cdr(pat)) = cdr(arg) . (34)

From the property of "compose," the first goal is simplified to

varsubst(m2, varsubst(ml, car(pat))) = car(arg) .

Since

varsubst(ml, car(pat)) = car(arg)

by (27), this simplifies to

varsubst(m2, car(arg)) = car(arg) .

Since arg contains no variables, neither does car(arg). Thus, the goal simplifies to

car(arg) = car(arg) .

The proof of (34) is even simpler:

varsubst(compose(m2, ml), cdr(pat))

simplifies to

varsubst(m2, varsubst(ml, cdr(pat))) .

We know by our hypothesis (28) that

varsubst(m2, varsubst(ml, cdr(pat))) = cdr(arg) .

and this completes the proof.

This proof required not only that we add new rules describing the concepts involved, but also that we extend certain of our older capabilities, particularly our ability to simplify expressions using known equalities. A complete trace of the proof is included in Appendix B.

We worked nearly a week before the system was able to do this proof. However, once the proof was completed, the effort necessary to enable the system to do the proof of the unification algorithm was minimal. The latter proof, though longer than this one, did not require much additional intellectual capacity on the part of the deductive system. We do not show that proof here because it is similar to the pattern matcher proof, but we include the program and the assertion we proved about it.

VIII THE UNIFICATION ALGORITHM

The problem of unification is similar to that of pattern matching except that we allow both arguments to contain variables. We expect the algorithm to find a substitution that makes the two arguments identical when it is applied to both, if such a substitution exists. For example, if x is (F U A) and y is (F B V), then unify(x, y) is ((U•B)(V•A)), where U and V are variables and A, B, and F are constants.

A simple program to unify x and y is

```
unify(x, y) = prog((m1 m2)
  if x = y then return(EMPTY)
  if var(x) then
    return(if occursin(x, y) then NOMATCH
                                 else list (cons(x, y)))
  if var(y) then
    return(if occursin(y, x) then NOMATCH
                                 else list (cons(y, x)))
  if atom(x) then return(NOMATCH)
  if atom(y) then return(NOMATCH)
  m1 ← unify(car(x), car(y))
  if m1 = NOMATCH then return(NOMATCH)
  m2 ← unify(varsubst(m1, cdr(x)),
             varsubst(m1, cdr(y)))
  if m2 = NOMATCH then return(NOMATCH)
  return(compose(m2, m1)))   .
```

The predicate <u>occursin</u>(u,v) tests if u occurs in v. This program is a recursive, list-oriented version of Robinson's iterative, string-oriented program. Again, we have verified only the longest path of the program, not the entire program. Furthermore, we have proved not the strongest possible statement about this program, but only that

$$\text{unify}(x,y) \neq \text{NOMATCH} \supset$$

$$\text{varsubst}(\text{unify}(x,y),\ x) = \text{varsubst}(\text{unify}(x,y),\ y) \quad .$$

IX THE FIND PROGRAM

The program FIND, described by Hoare [1961] is intended to rearrange an array A so that all the elements to the left of a certain index F are less than or equal to A[F], and all those to the right of F are greater than or equal to A[F]. In other words, the relation (STRIP (BAGA A 1 F-1)) $\leq$ A[F] $\leq$ (STRIP (BAGA A F+1, NN)) should hold when the program halts. For instance, if F is NN÷2, then A[F] is the median of the array. The function is useful in computing percentiles and is fairly complex.

Hoare remarks that a sorting program would achieve the same purpose but would usually require much more time; the conditions for FIND are much weaker in that, for example, the elements to the left of F need not be sorted themselves, as long as none of them are greater than A[F].

The ALGOL representation of FIND is as follows:

```
BEGIN
     INTEGER M,N;
     M ← 1;
     N ← NN;
     WHILE M < N DO
          BEGIN INTEGER R,I,J;
               R ← A[F];
               I ← M;
               J ← N;
               WHILE I ≤ J DO
               BEGIN WHILE A[I] < R DO I ← I+1;
                    WHILE R < A[J] DO J ← J-1;
                    IF I ≤ J THEN
                    BEGIN EXCHANGE(A I J)
                         I ← I+1
                         J ← J-1
                    END
               END
               IF F ≤ J THEN N ← J
               ELSE IF I ≤ F THEN M ← I
                    ELSE GO TO L
          END
     L:
END
```

The general strategy of the program FIND is to move "small" elements to the left and "large" elements to the right. These relative size categories are defined as being less than or not less than an arbitrary array element. The algorithm scans the array from left to right looking for a large element; when it finds one, it scans from right to left looking for a small element. When it finds one, it exchanges the large element and the small element it has already found, and the scan from the left continues where it left off until the next large element is found, and so on. When the scan from the left and the scan from the right meet somewhere in the middle, they define a split in the array. We can then show that all the elements to the left of the split are small and all those to the right are large.

The index F can be either to the left or to the right of the split, but suppose it is to the left. Then the elements to the right of the split can remain where they are; they are the largest elements in the array, and the element that will ultimately be in position F is to the left of the split. We then disregard the right portion of the array and repeat the process with the split as the upper bound of the array and with a refined definition of "large" and "small." We will eventually find a new split; suppose this split is to the left of F. We can then leave in place the elements of the array to the left of the split and work only with the elements to the right; we readjust the left bound of the array to occur at the split, and we repeat the process. Thus, the left and right bounds of the array move closer and closer together, but they always have F between them. Finally, they meet at F, and the algorithm halts.

The flow chart in Figure 3 follows Hoare's algorithm closely.

In this program, I is the pointer for the left-to-right scan, J is the pointer for the right-to-left scan, M and N are the lower and upper bounds of the "middle" portion of the array, and R is the value used to discriminate between small and large array elements. Hoare [1971] provided an informal manual proof of the correctness of his program. Igarashi, London, and Luckham [1973] have produced machine proofs. The proof we obtained required a minimal number (three) of intermediate assertions; however, one of the verification conditions produced was quite difficult to prove. This condition corresponds to the statement that the elements to the right of the right boundary dominate the elements to its left after an exchange is performed and a new right boundary is established. We present a sketch of that proof below.

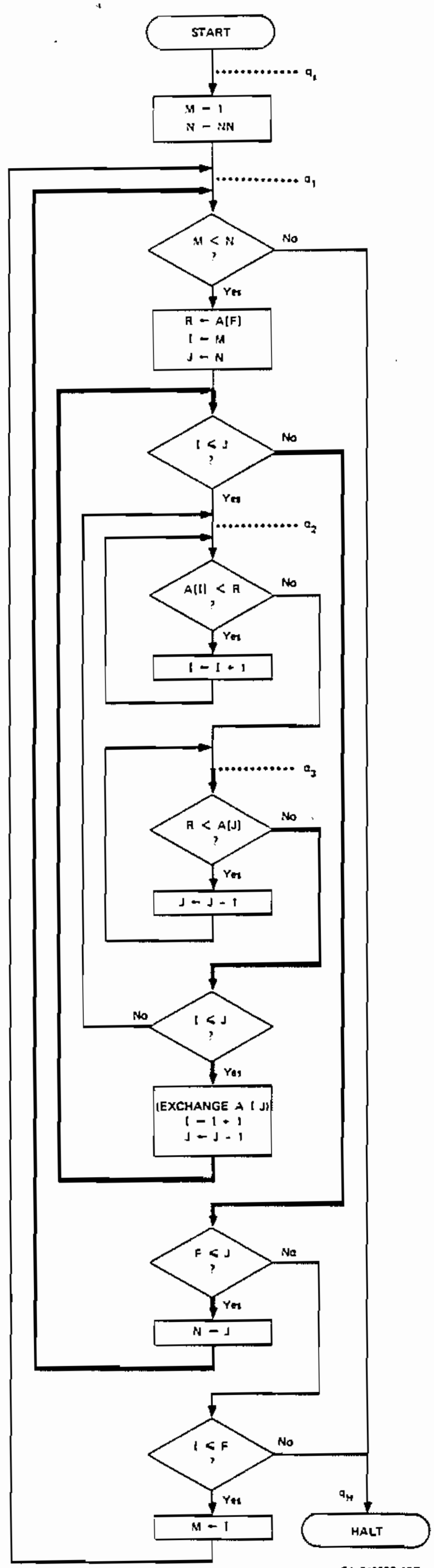


FIGURE 3 THE FIND PROGRAM

A. Assertions for FIND

The input assertion q_s for FIND is (the conjunction of)

$$1 \le F \le NN \quad ,$$
$$A = AP \quad ,$$

The array AP is the initial version of A; we define it in the input assertion so that we can refer to it after we have modified A.

The output assertion q_H is

```
(STRIP (BAGA A 1 F-1)) ≤ A[F] ≤ (STRIP (BAGA A F+1, NN))
      (BAGA A 1 NN) = (BAGA AP 1 NN)    .
```

The second conjunct of q_H states that when the program terminates, the array A is indeed a permutation of the initial array AP.

The intermediate assertion q_1 is

```
1 ≤ M ≤ F ≤ N ≤ NN
(STRIP (BAGA A 1 M-1)) ≤ (STRIP (BAGA A M NN))
(STRIP (BAGA A 1 N)) ≤ (STRIP (BAGA A N+1 NN))
(BAGA A 1 NN) = (BAGA AP 1 NN)     .
```

This assertion is reached whenever a new bound on the middle section of the array is established.

The assertion q_2 is

```
1 ≤ M ≤ F ≤ N ≤ NN
(STRIP (BAGA A 1 M-1)) ≤ (STRIP (BAGA A M NN))
(STRIP (BAGA A 1 N)) ≤ (STRIP (BAGA A N+1 NN))
M ≤ I
J ≤ N
(STRIP (BAGA A 1 I-1)) ≤ R ≤ (STRIP (BAGA A J+1 NN))
(BAGA A 1 NN) = (BAGA AP 1 NN)    .
```

The assertion q_3 is the same as the assertion q_2, with the additional conjunct

$$R \le A[I] \quad .$$

B. The Proof

All but one of the verification conditions for this program were proved fairly easily. The one difficult condition corresponds to the path beginning at q_3 that follows the heavy line and finally ends at q_1. The verification condition generator supplied us with the following hypotheses:

$$1 \le M \le F \le N \le NN \tag{35}$$

$$\text{(STRIP (BAGA A 1 M-1))} \le \text{(STRIP (BAGA A M NN))} \tag{36}$$

$$\text{(STRIP (BAGA A 1 N))} \le \text{(STRIP(BAGA A N+1 NN))} \tag{37}$$

$$M \le I \tag{38}$$

$$J \le N \tag{39}$$

$$\text{(STRIP (BAGA A 1 I-1))} \le R \le \text{(STRIP (BAGA A J+1 NN))} \tag{40}$$

$$R \le A[I] \tag{41}$$

$$\text{(BAGA AP 1 NN)} = \text{(BAGA A 1 NN)} \tag{42}$$

$$\neg (R < A[J]) \tag{43}$$

$$I \le J \tag{44}$$

$$\neg (I+1 \le J-1) \tag{45}$$

$$F \le J-1 \quad . \tag{46}$$

The interesting consequence for this path is

$$\text{(STRIP (BAGA A}' \text{ 1 J-1))} \le \text{(STRIP (BAGA A}' \text{ (J-1)+1 NN))} \tag{47}$$

where

$$A' = \text{(EXCHANGE A I J)} \quad ,$$

the array that results when elements A[I] and A[J] are interchanged in A.

The proof sketched below roughly parallels the proof produced by the inference system. Portions of the trace are shown in Appendix B.

The (J-1)+1 term in the goal (47), is simplified to J, giving the goal

(STRIP (BAGA A′ 1 J-1)) ≤ (STRIP (BAGA A′ J NN)) (48)

The difficulty in the proof arises from the uncertainty about whether J ≤ I. We are reasoning about an array segment, and it is not clear whether that segment is affected by the exchange or not. Hand analysis of hypothesis (44) and (45) reveals that I = J or I = J-1. The value of a term like (BAGA (EXCHANGE A I J) 1 J-1) depends on which possibility is actually the case.

The system "simplifies" the term into

```
(IF J ≤ I THEN (BAGA A 1 J-1)
          ELSE (BAG (STRIP (BAGA A 1 I-1))
                    A[J]
                    (STRIP (BAGA A I+1 J-1))))   .
```

Intuitively, if J ≤ I, then both I and J are outside the bounds of the array segment, whereas if I < J, then the array segment loses A[I] but gains A[J].

Similarly, the term

(BAGA (EXCHANGE A I J) J NN)

is "simplified" into

```
(IF J ≤ I THEN (BAGA A J NN)
          ELSE (BAG (STRIP (BAGA A J J-1))
                    A[I]
                    (STRIP (BAGA A J+1 NN))))   .
```

Note that (BAGA A J J-1) is empty; the ELSE clause is then

(BAG A[I] (STRIP (BAGA A J+1 NN))) .

Our goal can thus be reduced to showing that

```
(IF J ≤ I THEN (STRIP (BAGA A 1 J-1))
          ELSE (STRIP (BAG (STRIP (BAGA A 1 I-1))
                           A[J]
                           (STRIP (BAGA A I+J J-1)))))

≤

(IF J ≤ I THEN (STRIP (BAGA A J NN))
          ELSE (STRIP (BAG A[I]
                           (STRIP (BAGA A J+1 NN)))))    . (49)
```

The system approaches the conditional expression by creating two contexts: In one context, J ≤ I holds, and in the other, I < J is true. In the first context we must prove that

```
(STRIP (BAGA A 1 J-1)) ≤ (STRIP (BAGA A J NN))   .     (50)
```

In the second context, the statement to be proved is

```
(STRIP (BAG (STRIP (BAGA A 1 I-1))
            A[J]
            (STRIP (BAGA A I+1 J-1))))

≤

(STRIP (BAG A[I]
            (STRIP (BAGA A J+1 NN))))    .          (51)
```

Note that in the first context, J = I by (44). In working on (50), (BAGA A J NN) is expanded to (BAG A[J] (STRIP (BAGA A J+1 NN))). Thus, (50) breaks into two subgoals:

```
(STRIP (BAGA A 1 J-1)) ≤ A[J]                       (52)
```

and

```
(STRIP (BAGA A 1 J-1)) ≤ (STRIP (BAGA A J+1 NN))   .     (53)
```

Since I = J, (52) follows from (40) and (41), and (53) follows from (40) alone.

Work on the goal (51) proceeds in the second context, in which I ≤ J.

Since J-1 ≤ I+1 (11), we know (BAGA A I+1 J-1) is empty. The inequality (51) may thus be broken into four inequalities:

$$\text{(STRIP (BAGA A 1 I-1))} \le \text{A[I]} \quad , \tag{54}$$

$$\text{(STRIP (BAGA A 1 I-1))} \le \text{(STRIP (BAGA A J+1 NN))} \quad , \tag{55}$$

$$\text{A[J]} \le \text{A[I]} \quad , \tag{56}$$

and

$$\text{A[J]} \le \text{(STRIP (BAGA A J+1 NN))} \quad . \tag{57}$$

Line (54) follows from hypotheses (40) and (41). Goal (55) follows from (40). Goal (57) follows from (43) and (40). This completes the proof.

This proof is the longest achieved by our deductive system so far.

X SUMMARY OF RESULTS

Complete proofs have been found of the correctness of the following algorithms:

- Finding the largest element of an array
- Finding the quotient of two real numbers
- Hoare's FIND program
- The Euclidean algorithm for finding the greatest common divisor
- The exponentiation program from King's thesis
- Integer quotient and remainder
- Integer multiplication by repeated addition
- The factorial.

Theorems have been proved about the following algorithms:

- The pattern matcher.
- Unification.
- Exchanging two array elements (the theorem is that the bag of the contents of the array is unchanged).
- King's exchange sort.

We believe the system now has the power to do all of King's problem set except the linear inequalities problem, which is not really a proof about an algorithm.

XI FUTURE PLANS

We are currently applying the verifier to more and more complex programs in a variety of subject domains. We are continuously being forced to add new rules and occasionally to generalize old ones; a special purpose rule that worked for one problem may not work for the next.

The deductive system is implemented in the QA4 language. Although QA4 is ideally suited for expressing our rules, it is an experimental system evaluated by an interpreter and is written in LISP; furthermore, it uses space inefficiently. R. Reboh and E. Sacerdoti are in the process of integrating QA4 into BBN-LISP to produce a system known as QLISP [Reboh and Sacerdoti, 1973]. QLISP programs will be LISP programs that can be evaluated by the LISP interpreter or even compiled. Furthermore, QLISP is much more conservative in its use of space. We expect that this system will be considerably faster and more compact than the existing system. Our deductive system is already being translated into QLISP.

QA4 subtly encourages its users to write depth-first search strategies, since it implements the goal mechanism by means of backtracking. The deductive system uses depth-first search, and for the most part, this has been the proper thing to do. There have been times, however, when we have felt the need for something more discriminating. Suppose, for example, we are trying to prove an expression of the form $x = y$. We can do this by trying to simplify x and then proving that the simplified x is equal to y, or we can try to find some assertion $a = b$ and prove $x = a$ and $y = b$. In the current system, we must exhaust one possibility before trying another, whereas we would like to be able to switch back and forth between different approaches, giving more attention to the one that currently seems to be making the best progress. In other words, we hope to use processes rather than backtracking in the implementation of the goal mechanism.

Finally, we hope to apply this work to the generation of counter-examples for "wrong" programs, to the generation of Floyd assertions, and to the automatic construction of programs. It seems inevitable that if we know how to reason about programs, that reasoning should be able to help us in the process of forming or changing a program. Rather than taking a handwritten and hand-debugged program to a verifier for approval, we hope to collaborate with a system that will play an active role in the creation of the algorithm.

Appendix A

LISTING OF THE DEDUCTIVE SYSTEM

Appendix A

THE DEDUCTIVE SYSTEM

The deductive system has the overall structure shown in Figure 4. The names on the chart are either function names or goal classes. Only important substructures are included.

An annotated listing of the programs used for reasoning is presented below. An index of functions and goal classes is included at the end of this appendix. The reader will note how little of the space is devoted to general strategies and how much is devoted to subject-specific knowledge. Some of the programs use QA4 features that are not described in this paper. The reader can rely on the English explication of the programs, or he can refer to the QA4 manual (Rulifson et al. [1973]).

To start a deduction, we say to the system

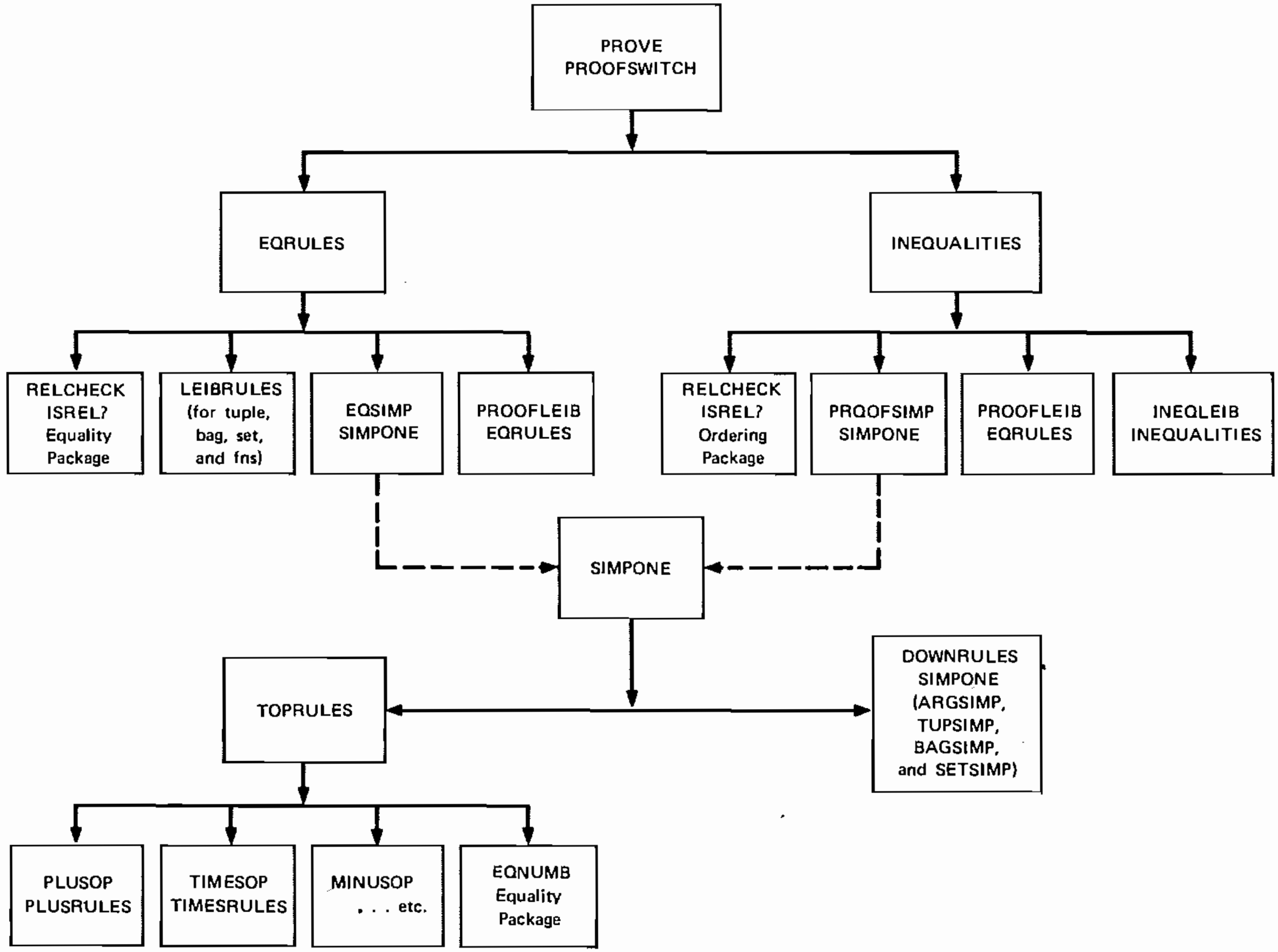
(GOAL $PROVE⟨some statement⟩)

- PROVE is a goal class:*

(TUPLE ANDSPLIT ORSPLIT OPSPLITMANY PROOFSWITCH)

- PROOFSWITCH determines whether the goal is an equality, otherwise it is assumed to be an inequality.

* Bullets are used to indicate the beginning of the description of a new function.



SA-1891-5

FIGURE 4 STRUCTURE OF THE DEDUCTIVE SYSTEM

```
PROOFSWITCH=(LAMBDA (←F ←X)
                    (PROG (DECLARE)
                          (IF (EQUAL $F (QUOTE EQ))
                              THEN
                              (GOAL $EQRULES ($F $X))
                              ELSE
                              (GOAL $INEQUALITIES
                                    ($F $X)))
                          (ASSERT ($F $X))
                          (RETURN ($F $X]+
```

In either case, the appropriate set of rules is applied.

1. Equalities

- The equality class is

```
EQRULES =
(TUPLE ANDSPLIT RELCHECK EQTIMESDIVIDE EQSUBST LEIBT LEIBF
       LEIBB LEIBS EQSIMP PROOFLEIBS)
```

- The rule ANDSPLIT takes a goal that is a conjunction of two or more expressions‡ and tries to prove each conjunct independently.

```
ANDSPLIT=(LAMBDA (AND ←X ←←Y)
                 (ATTEMPT (GOAL $GOALCLASS $X)
                          THEN
                          (ATTEMPT (GOAL $GOALCLASS
                                         (AND $$Y))
                                   ELSE
                                   (FAIL))
                          ELSE
                          (FAIL]
```

If repeated applications of ANDSPLIT are successful eventually, the goal (AND) will be generated. However, (AND) is an assertion in the data base, and so the rule will then succeed.

- ORSPLIT applies to a goal that is the disjunction of two expressions

---

+ The right bracket represents a string of right parentheses long enough to balance the expression.

‡ Variables with double prefixes, "←←" or "$$," respectively match or evaluate to a sequence of terms rather than a single term. In the rule, for example, Y can be bound to a set of terms, including the empty set.

and works on each separately.

```
ORSPLIT=(LAMBDA (OR ←X ←Y)
                (ATTEMPT (GOAL $GOALCLASS $X)
                         ELSE
                         (GOAL $GOALCLASS $Y)
```

The expression x is attempted as a goal first; if this is successful, we are done. Otherwise, ORSPLIT works on y; if it is unsuccessful, then a failure is generated.

- ORSPLITMANY is similar to ORSPLIT, except that it takes as a goal the disjunction of three or more expressions:

```
ORSPLITMANY =(LAMBDA (OR ←X ←Y ←Z ←←W)
                    (ATTEMPT (GOAL $GOALCLASS $X)
                             ELSE
                             (GOAL $GOALCLASS
                                   (OR $Y $Z $$W)
```

The expression x is attempted first; if the proof is successful, the disjunction is true. Otherwise, the disjunction of the remaining expressions is established as a new goal. Continued failure to prove members of a disjunction will eventually cause ORSPLIT to be invoked.

- The rule RELCHECK merely checks the property lists of the expressions to see if they are already known to be equal:

```
RELCHECK=(LAMBDA ←X (ISREL? $X))
```

When RELCHECK is applied, x is bound to an equality statement, which is fed to the ISREL? statement. ISREL? will succeed not only if the equality has been explicitly asserted, but also if the equality follows by the transitive law from other equalities or inequalities. ISREL? is the mechanism for making queries about special relations. It will work with inequality relations, such as LT, GTQ, and NEQ, as well as EQ.

EQTIMESDIVIDE and EQSUBST are rules for reasoning about numbers and substitutions, respectively. They are discussed in the relevant sections.

```
LEIBF=(LAMBDA (EQ (←F ←X)
                  (←F ←Y))
              (PROG (DECLARE)
                    ($ASK (' (EQ $X $Y))
                          PROVE?)
                    (GOAL $EQRULES (EQ $X $Y]

LEIBT=(LAMBDA (EQ (TUPLE ←X ←←Z)
                  (TUPLE ←Y ←←W))
              (PROG (DECLARE)
                    (GOAL $EQRULES (EQ $X $Y))
                    (GOAL $EQRULES (EQ $Z $W]

LEIBS=(LAMBDA (EQ (SET ←X ←←Z)
                  (SET ←Y ←←Z))
              (GOAL $EQRULES (EQ $X $Y]

LEIBB=(LAMBDA (EQ (BAG ←X ←←Z)
                  (BAG ←Y ←←Z))
              (GOAL $EQRULES (EQ $X $Y]
```

The LEIBF rule asks the user if he wants that rule to be applied. The function ASK that performs the interaction is described in the section on utility functions.

- EQSIMP and PROOFLEIB are very time consuming but also very powerful. EQSIMP says that to prove x = y, simplify x and try to prove that the simplified x is equal to y.

```
EQSIMP =[LAMBDA (EQ ←X ←Y)
                (PROG (DECLARE)
                      (SETQ ←X ($SIMPONE $X))
                      (GOAL $EQRULES (EQ $X $Y)))
               BACKTRACK])
```

Since the program uses the BACKTRACK option, and since EQ implicitly takes a set as its argument, EQSIMP can work on y as well as on x. In other words, if it fails to simplify x, it will go ahead and try to simplify y.

- PROOFLEIB tries to make use of information stored in the data base. It is used to prove inequalities as well as equalities.

```
PROOFLEIB =(LAMBDA (←F ←X)
                   (PROG (DECLARE)
                         (EXISTS ($F ←Y))
                         ($ASK (' (EQ $X $Y))
                               PROVE?)
                         (GOAL $EQRULES (EQ $X $Y]
```

It says: to prove u = v, find an assertion of the form a = b and prove u = a and v = b. It relies on user interaction to cut off bad paths. Note that if F is EQ, we can expect X and Y to be sets, so that LEIBS will ultimately be called to prove the equality expression generated by PROOFLEIB.

2. Inequalities

We now turn to the rules for proving inequalities.

```
INEQUALITIES
(TUPLE ANDSPLIT RELCHECK ORSPLIT ORSPLITMANY PROOFSIMP
       INEQIFTHENELSE INEQSTRIPBAG INEQSTRIPSTRIP
       INEQSTRIPTRAN GTQLTQ LTQMANY FSUBTRACT1 FSUBTRACT2
       INEQTIMESDIVIDE EQINEQMONOTONE LTQPLUS PROOFLEIB
       INEQLEIB))
```

RELCHECK has been mentioned above.

- GTQLTQ says that to prove y ≥ x, try to prove x ≤ y:

```
GTQLTQ =(LAMBDA (GTQ ←Y ←X)
                (GOAL $INEQUALITIES (LTQ $X $Y]
```

- LTQMANY takes an inequality goal, such as

$$x_1 \le x_2 \le \ldots \le x_n \quad ,$$

and breaks it into separate goals,

$$x_1 \le x_2 \text{ and } x_2 \le x_3 \text{ and } \ldots x_{n-1} \le x_n \quad .$$

```
LTQMANY =(LAMBDA (LTQ ←←X ←Y ←Z ←←W)
                 (PROG (DECLARE)
                       (GOAL $INEQUALITIES (LTQ $X $Y))
                       (GOAL $INEQUALITIES
                             (LTQ $Y $Z $$W]
```

LTQPLUS, FSUBTRACT1, and FSUBTRACT2 are special rules for reasoning about numbers and are discussed in the relevant section.

- PROOFSIMP proves an expression F(Y) by trying to simplify Y and proving the simplified expression.

```
PROOFSIMP=(LAMBDA (PAND ←X (←F ←Y))
                  (PROG (DECLARE GOALCLASS1)
                        (SETQ ←GOALCLASS1 $GOALCLASS)
                        (ATTEMPT (SETQ ←X ($ARGSIMP $X))
                                 ELSE
                                 (FAIL))
                        (GOAL $GOALCLASS1 $X]
```

It has more general application than just to inequalities, although so far we have used it only for inequalities.

- INEQLEIB is similar to PROOFLEIB, but it works only for inequalities.

```
INEQLEIB=(LAMBDA (←L ←X ←Y)
                 (PROG (DECLARE LOWER UPPER)
                       (EXISTS ($L ←LOWER ←UPPER))
                       ($ASK PROVE (' (LTQ $X $LOWER))
                             AND
                             (' (LTQ $UPPER $Y))
                             ?)
                       (GOAL $INEQUALITIES
                             (AND (LTQ $X $LOWER)
                                  (LTQ $UPPER $Y]
```

L is expected to be LT or LTQ. To prove $x < y$, for example, find an asserted statement LOWER < UPPER and prove $x \leq$ LOWER and UPPER $\leq y$.

• INEQIFTHENELSE is a rule that sets up a case analysis:

```
INEQIFTHENELSE=(LAMBDA (←F ←←W1 (IFTHENELSE ←X ←Y ←Z)
                        ←←W2)
                (PROG (DECLARE VERICON)
                      (ATTEMPT (SETQ ←VERICON
                                     (CONTEXT PUSH
                                              LOCAL))
                               (ASSERT $X WRT $VERICON)
                               THEN
                               (GOAL $INEQUALITIES
                                     ($F $$W1 $Y $$W2)
                                     WRT $VERICON))
                      (ATTEMPT (SETQ ←VERICON
                                     (CONTEXT PUSH
                                              LOCAL))
                               (DENY $X WRT $VERICON)
                               THEN
                               (GOAL $INEQUALITIES
                                     ($F $$W1 $Z $$W2)
                                     WRT $VERICON)
                               ELSE
                               (RETURN (SUCCESS WITH
                                     INEQIFTHENELSE]
```

For example, suppose the goal is (IF x THEN y ELSE z) $\leq$ w. This rule establishes two subcontexts of the local context. In one of these contexts, x is true; in the other, x is false. In the first context, the rule tries to prove y $\leq$ w, whereas in the second, it tries to prove z $\leq$ w. Note that the system that stores equalities and inequalities will cause a failure if an assertion (or a denial) would lead it to contradict what it knows. In that case the goal is considered to be achieved.

• INEQSTRIPBAG is an inequality rule that has a bag as one of its arguments.

```
INEQSTRIPBAG=(LAMBDA (←F ←←W (STRIP (BAG ←X ←←Y))
                      ←←Z)
              (GOAL $INEQUALITIES
                    (AND ($F $$W $X $$Z)
                         ($F $$W (STRIP (BAG $$Y))
                             $$Z]
```

This rule would be invoked when we want to show, for example, $w_1 \leq$ (STRIP (BAG c_1 c_2 ...)) $\leq w_2$. The intention here is to demonstrate that $w_1 \leq c_1 \leq w_2$ and $w_1 \leq c_2 \leq w_2$, and so forth. Ultimately, we might have to demonstrate that $w_1 \leq$ (STRIP (BAG)) $\leq w_2$. The special relations handler (ISREL?) succeeds vacuously with any inequality relation where one of the arguments is (STRIP (BAG)).

3. Deduce

- DEDUCE is a goal class of rules that are guaranteed to terminate quickly. It is used when we want something more inquisitive than EXISTS but less timeconsuming than PROVE, EQRULES, or INEQUALITIES.

```
DEDUCE=
(TUPLE RELCHECK ANDSPLIT ORSPLIT ORSPLITMANY LTPLUS FSUBTRACT1
       FSUBTRACT2 LTQPLUS NOTATOM CONSTCAR CONSTCDR))
```

We have already described RELCHECK.

The other DEDUCE rules are for special applications and are discussed in the appropriate sections.

4. Simplification

- The top-level simplification function is SIMPONE. This function does not try to simplify its argument completely. It will find a partial simplification; repeated applications, if necessary, will completely simplify the expression.

The simplification rules may, of course, be added by the user. We expect that each simplification rule should make the expression simpler in some sense. Otherwise, the program may loop interminably.

```
SIMPONE=(LAMBDA ←GOAL1 (PROG (DECLARE SIMPGOAL)
                             (IF (EQUAL (STYPE $GOAL1)
                                        NUMBER)
                                 THEN
                                 (FAIL))
                             ($ASK $GOAL1 SIMPLIFY?)
                             [SETQ ←SIMPGOAL
                                   (ATTEMPT
                                      (GOAL $TOPRULES $GOAL1)
                                      ELSE
                                      ($TRY $TOPRULES
                                            (GOAL $DOWNRULES
                                                  $GOAL1]
                              (PUT $GOAL1 SIMPLIFIED $SIMPGOAL
                                   WRT ETERNAL)
                              (RETURN $SIMPGOAL]
```

SIMPONE fails if it cannot simplify its argument at all. It treats numbers as being completely simplified. It asks the user for permission to go ahead. It tries a goal class, TOPRULES, on the expression.

- TOPRULES is a set of rules that work on the top level of the expression:

```
TOPRULES
(TUPLE HASSIMP FAILINTODOWNRULES PLUSOP TIMESOP MINUSOP
       FIFTHENELSE BAGAOP SUBSTOP EXPZERO EXPEXP SUBPLUS
       SUBNUM GCDEQ ACCH CONSDIFF DIFDIF DIFFCONS DIFFONE
       MAXPLUS MAXONE BAGSTRIP ACCEX EQNUMB)
```

If any of these rules apply, SIMPONE returns the simplified expression as its value. Otherwise, it tries to simplify some subexpression of the given expression:

```
DOWNRULES=(TUPLE ARGSIMP TUPSIMP BAGSIMP SETSIMP))

ARGSIMP=(LAMBDA (←F ←X)
                (SUBST (' ($F $X))
                       (TUPLE $X ($SIMPONE $X]

TUPSIMP=[LAMBDA (TUPLE ←←X ←Y ←←Z)
                (TUPLE $$X ($SIMPONE $Y)
                       $$Z)
                BACKTRACK])

BAGSIMP=[LAMBDA (BAG ←X ←←Y)
                (BAG ($SIMPONE $X)
                     $$Y)
                BACKTRACK])

SETSIMP=[LAMBDA (SET ←X ←←Y)
                (SET ($SIMPONE $X)
                     $$Y)
                BACKTRACK])
```

The DOWNRULES simplify a complex expression by simplifying the component parts of the expression. If any of the DOWNRULES apply, SIMPONE applies the TOPRULES again to the new expression. SIMPONE calls the functions ASK and TRY that are described in the section on utility functions.

• SIMPONE puts the simplified expression on the property list of the original expression. In this way, if it ever comes across the original expression again, one of the TOPRULES, HASSIMP, will immediately know what simplification was found before.

```
HASSIMP=(LAMBDA ←X (IF (NOT (IN (SETQ ←X (GET $X SIMPLIFIED))
                                (TUPLE DONE NOSUCHPROPERTY)))
                       THEN $X ELSE (FAIL]
```

• If the expression to be simplified is a set, tuple or bag rather than a function application, none of the TOPRULES will apply to it. To avoid the cost of searching for a match among all the TOPRULES, the rule FAILINTODOWNRULES will first test for this condition and cause the entire goal statement to fail should it arise:

```
FAILINTODOWNRULES= (LAMBDA ←X
                           (IF (IN (STYPE $X)
                                   (TUPLE TUPLE SET BAG))
                             THEN
                             (FAIL GOAL)
                             ELSE
                             (FAIL]
```

SIMPONE will then apply the DOWNRULES to the argument to see if any of its subexpressions can be simplified.

- One of the most general TOPRULES is EQNUMB, which replaces any expression by the "smallest" known equal expression:

```
EQNUMB =(LAMBDA ←X (PROG (DECLARE BEST EQSET)
                         (IF (EQUAL (SETQ ←EQSET (GET $X EQ))
                                    NOSUCHPROPERTY)
                             THEN
                             (FAIL))
                         (SETQ ←BEST ($SHORTEST $EQSET))
                         (IF (EQUAL $BEST $X)
                             THEN
                             (FAIL)
                             ELSE
                             (RETURN $BEST]
```

The "smallest" element of a set is computed by the QA4 function SHORTEST, described among the utility functions.

If EQNUMB fails to find a smaller representation for x, it fails.

-
```
FIFTHENELSE= (LAMBDA (←F (IFTHENELSE ←W ←X ←Y))
                     (' (IFTHENELSE $W ($F $X)
                                       ($F $Y]
```

FIFTHENELSE moves conditional expressions outside of function applications. An expression of the form

f(IF w THEN x ELSE y)

translates into

IF w THEN f(x) ELSE f(y) .

The remaining rules in TOPRULES are discussed in the sections dealing with special subject domains.

5. Reasoning About Numbers

a. Equality and Inequality Rules

- EQTIMESDIVIDE is an EQRULE. It means that to prove w = (x/y)*z, prove w*y = x*z:

```
EQTIMESDIVIDE=[LAMBDA (EQ ←W (TIMES (DIVIDES ←X ←Y)
                                    ←←Z))
                      (GOAL $EQRULES (EQ (TIMES $Y $W)
                                         (TIMES $X $$Z)))
                      BACKTRACK])
```

Some inequality rules that know about numbers are presented below.

- LTQPLUS says that to prove i ≤ j+k, prove i ≤ j and 0 ≤ k:

```
LTQPLUS=[LAMBDA (LTQ ←I (PLUS ←J ←K))
                (GOAL $DEDUCE (AND (LTQ $I $J)
                                   (LTQ 0 $K)))
                BACKTRACK])
```

First, the rule attempts to prove that i ≤ j and 0 ≤ k. If either of these proofs is unsuccessful, then the backtracking mechanism will interchange the bindings of the arguments of LTQPLUS. This then leads to an attempt to prove i ≤ k and 0 ≤ j.

- LTPLUS is the analogue of LTQPLUS for LT:

```
LTPLUS=[LAMBDA (LT ←I (PLUS ←J ←K))
               (GOAL $DEDUCE (AND (LTQ $I $J)
                                  (LT 0 $K)))
               BACKTRACK])
```

It means that to prove i < j+k, prove i ≤ j and 0 < k. It can backtrack to reverse the roles of j and k.

FSUBTRACT1 and FSUBTRACT2 allow us to make deductions: for example, to prove x-y ≤ z try to prove x ≤ y+z.

```
FSUBTRACT1=(LAMBDA (←F (SUBTRACT ←X ←Y)
                       ←Z)
                   (GOAL $GOALCLASS ($F $X (PLUS $Y $Z]

FSUBTRACT2=(LAMBDA (←F ←X (SUBTRACT ←Y ←Z))
                   (GOAL $GOALCLASS ($F (PLUS $X $Z)
                                        $Y]
```

- INEQMONOTONE says that to prove $w+x \leq y+z$, prove $w \leq y$ and $x \leq z$ or $w \leq z$ and $x \leq y$.

```
EQINEQMONOTONE (LAMBDA (←L (PLUS ←W ←X)
                           (PLUS ←Y ←Z))
                       [PROG (DECLARE)
                             ($ASK PROVE (' ($L $W $Y))
                                   AND
                                   (' ($L $X $Z))
                                   ?)
                             (GOAL $GOALCLASS
                                   (AND ($L $W $Y)
                                        ($L $X $Z]
                       BACKTRACK])
```

- The rule INEQTIMESDIVIDE is similar to EQTIMESDIVIDE except that it must check that the denominator is nonnegative before multiplying out:

```
INEQTIMESDIVIDE=[LAMBDA (←F ←W (TIMES (DIVIDES ←X ←Y)
                                      ←←Z))
                        [PROG (DECLARE)
                              (GOAL $DEDUCE (LT 0 $Y))
                              (GOAL $INEQUALITIES
                                    ($F (TIMES $Y $W)
                                        (TIMES $X $$Z]
                        BACKTRACK])
```

This rule says that to prove $w < (x/y)*z$, say, in the case that $0 < y$, try to prove $w*y < x*z$.

b. <u>Numerical Demons</u>

- When $x \geq y$ is asserted, assert that $y < x$:

```
(WHEN EXP (GTQ ←X ←Y)
      INDICATOR MODELVALUE THEN (ASSERT (LTQ $Y $X)
                                        WRT $VERICON]
```

These demons make their assertions with respect to the current context, VERICON.

- Whenever x+y ≤ x+z is asserted, we want to conclude that y ≤ z:

```
(WHEN EXP (LTQ (PLUS ←X ←Y)
               (PLUS ←X ←Z))
      INDICATOR MODELVALUE THEN
      (ASSERT (LTQ $Y $Z)
              WRT $VERICON]
```

- Whenever w-x ≤ y is asserted, assert w ≤ x+y, simplifying the right side if possible:

```
(WHEN EXP (LTQ (SUBTRACT ←W ←X)
               ←Y)
      INDICATOR MODELVALUE THEN
      (PROG (DECLARE RTSIDE)
            [SETQ ←RTSIDE
                  ($TRYALL $PLUSRULES
                           (' (PLUS $Y $X]
            (ASSERT (LTQ $W $RTSIDE)
                    WRT $VERICON]
```

- Whenever (w-x)+x ≤ y is asserted, then assert w ≤ y:

```
(WHEN EXP (LTQ (PLUS (SUBTRACT ←W ←X)
                     ←X)
               ←Y)
      INDICATOR MODELVALUE THEN
      (ASSERT (LTQ $W $Y)
              WRT $VERICON]
```

Certain demons are intended exclusively for the integer domain.

- $x < y \supset x+1 \leq y$:

```
(WHEN EXP (LT ←X ←Y)
      INDICATOR MODELVALUE THEN
      (ASSERT (LTQ (PLUS $X 1)
                   $Y)
              WRT $VERICON]
```

- $x > y \supset y+1 \leq x$:

```
(WHEN EXP (GT ←X ←Y)
      INDICATOR MODELVALUE THEN
      (ASSERT (LTQ (PLUS $Y 1)
                   $X)
              WRT $VERICON]
```

Whenever $w-x < y$ is denied, deny $w \leq y+x-1$, simplifying if possible:

```
(WHEN EXP (LT (SUBTRACT ←W ←X)
              ←Y)
      INDICATOR MODELVALUE PUTS FALSE THEN
      (PROG (DECLARE RTSIDE)
            [SETQ
              ←RTSIDE
              ($TRYALL $PLUSRULES
                       (' (PLUS $Y $X
                                (MINUS 1]
            (DENY (LTQ $W $RTSIDE)
                  WRT $VERICON]
```

c. Numerical Simplification

Much of the knowledge the system has about numbers is embedded in the simplifier. For efficiency, these rules have been arranged hierarchically. For example, only one rule, PLUSOP, in TOPRULES deals with sums.

```
PLUSOP=(LAMBDA (PAND ←Y (PLUS ←←X))
               ($TRYALLFAIL $PLUSRULES $Y))
```

However, this one rule coordinates a multitude of other rules. All the rules that operate on plus expressions are in the goal class PLUSRULES.

```
PLUSRULES=(TUPLE PLUSEMPTY PLUSSINGLE PLUSZERO PLUSPLUS
                 PLUSMINUS PLUSDIFFERENCE PLUSCOMBINE
                 PLUSNUMBER))
```

The strategy PLUSOP uses is to apply all the PLUSRULES to its argument until no further simplification is possible. (The function TRYALLFAIL, that expresses this strategy, is described among the utility functions.) If PLUSOP can find no simplification at all, it fails.

Most of the PLUSRULES are quite simple.

- The sum of the empty bag is 0:

```
PLUSEMPTY=[LAMBDA (PLUS)
                  0])
```

- The sum of a bag of one element is that element itself:

```
PLUSSINGLE=[LAMBDA (PLUS ←X)
                   $X])
```

(i.e., +x = x) .

- x+0 = +x :

```
PLUSZERO=(LAMBDA (PLUS ←←X 0)
                 (' (PLUS $$X]
```

Note that this rule implicitly says

$$0+x = +x$$

$$x+0+y = x+y$$

$$x+y+0+z = x+y+z ,$$

and so forth because PLUS takes a bag as its argument.

- $((x_1+x_2+...)+y_1+y_2+...) = (x_1+x_2+...+y_1+y_2+...)$:

```
PLUSPLUS=(LAMBDA (PLUS (PLUS ←←X)
                       ←←Y)
                 (' (PLUS $$X $$Y]
```

- x+(-x)+y = +y :

```
PLUSMINUS=(LAMBDA (PLUS ←X (MINUS ←X)
                        ←←Y)
                  (' (PLUS $$Y]
```

- x+(y-z)+w = x+y+w+(-z) :

```
PLUSDIFFERENCE =(LAMBDA (PLUS ←X (SUBTRACT ←Y ←Z)
                                  ←←W)
                        ($TRY (TUPLE PLUSMINUS)
                              (' (PLUS $Y $X $$W (MINUS $Z]
```

Note that PLUSDIFFERENCE recommends that PLUSMINUS be attempted immediately afterward. This is merely advice; if PLUSMINUS does not apply, nothing is lost. (TRY is described in the section on utility functions.)

- x+x+y = 2*x+y

```
PLUSCOMBINE=(LAMBDA (PLUS ←X ←X ←←Y)
                    ($TRYSUB $TIMESRULES ON
                             (' (TIMES 2 $X))
                             IN
                             (' (PLUS (TIMES 2 $X)
```

Note that PLUSCOMBINE recommends that the 2*x term be simplified if possible.

- If two elements of a plus expression are syntactically numbers, PLUSNUMBER will add them up:

```
                                                   $$Y]
PLUSNUMBER=[LAMBDA (PLUS ←X ←Y ←←Z)
                   (PROG (DECLARE SUM)
                         ($INSIST (EQUAL (STYPE $X)
                                         NUMBER))
                         ($INSIST (EQUAL (STYPE $Y)
                                         NUMBER))
                         (SETQ ←SUM (PLUS $X $Y))
                         (RETURN (PLUS $SUM $$Z)))
                   BACKTRACK])
```

- The rule TIMESOP is strategically similar to PLUSOP:

```
TIMESOP =(LAMBDA (PAND ←Y (TIMES ←←X))
                 ($TRYALLFAIL $TIMESRULES $Y])
```

It will apply all the TIMESRULES to the expression in question. TIMESRULES is

```
TIMESRULES =
(TUPLE TIMESEMPTY TIMESSINGLE TIMESZERO TIMESONE TIMESPLUS
       TIMESTIMES CANCEL SQRULE TIMESEXP TIMESDIVIDEONE))
```

- The product of the empty bag is 1:

```
TIMESEMPTY=[LAMBDA (TIMES)
                   1])
```

- The product of a bag of one element is that element itself:

```
TIMESSINGLE=[LAMBDA (TIMES ←X)
                    $X])
```

- 0*y = 0 :

```
TIMESZERO=[LAMBDA (TIMES 0 ←←Y)
                  0])
```

- 1*x = x :

```
TIMESONE=(LAMBDA (TIMES 1 ←←X)
                 (' (TIMES $$X]
```

Recall that these rules also imply

x*1*y = x*y ,

x*0*y*z = 0 ,

and so forth.

- (x+y)*z = x*z+(+y)*z (distribution law):

```
TIMESPLUS=(LAMBDA
  (TIMES (PLUS ←X ←←Y)
         ←←Z)
  ($TRY $PLUSRULES ($TRYSUB $PLUSRULES ON (' (PLUS $$Y))
                            IN
                            (' (PLUS (TIMES $X $$Z)
                                     (TIMES (PLUS $$Y)
                                            $$Z]
```

(Some simplification is attempted immediately on y and on x*z + y*z. TRYSUB is explained in the section on utility functions.)

- $((x_1*x_2*\ldots)*y_1*y_2\ldots) = (x_1*x_2*\ldots*y_1*y_2*\ldots)$:

```
TIMESTIMES=(LAMBDA (TIMES (TIMES ←←X)
                          ←←Y)
                   (' (TIMES $$X $$Y]
```

- $x*(1/y)*z = (x/y)*z$:

```
TIMESDIVIDEONE=(LAMBDA (TIMES ←X (DIVIDES 1 ←Y)
                              ←←Z)
                       (' (TIMES (DIVIDES $X $Y)
                                 $$Z]
```

- $x*(y/x)*z = y*z$:

```
CANCEL=(LAMBDA (TIMES ←X (DIVIDES ←Y ←X)
                      ←←Z)
               (' (TIMES $Y $$Z]
```

- $x*x*y = x^2*y$:

```
SQRULE=(LAMBDA (TIMES ←X ←X ←←Y)
               ($TRY (TUPLE TIMESSINGLE)
                     (' (TIMES (EXP $X 2)
                               $$Y]
```

- $x*x^n*y = x^{n+1}*y$:

```
TIMESEXP=(LAMBDA (TIMES ←X (EXP ←X ←N)
                        ←←Y)
                 ($TRYSUB $PLUSRULES ON (' (PLUS $N 1))
                          IN
                          (' (TIMES (EXP $X (PLUS $N 1))
                                    $$Y]
```

- To the reader who has gotten this far, MINUSOP will be self-explanatory:

```
MINUSOP=(LAMBDA (MINUS ←X)
                (GOAL (TUPLE MINUSZERO MINUSMINUS MINUSPLUS)
                      (MINUS $X]
```

Note the MINUSOP, unlike PLUSOP and TIMESOP, does not apply <u>all</u> the rules to the expression, but will return the value of the first rule that does not fail.

- $-0 = 0$:

```
MINUSZERO=[LAMBDA (MINUS 0)
                  0]
```

- $-(-x) = x$:

```
MINUSMINUS=[LAMBDA (MINUS (MINUS ←X))
                   $X]
```

- $-(x+y) = (-x)+(-y)$:

```
MINUSPLUS=(LAMBDA (MINUS (PLUS ←X ←←Y))
                  ($TRY $PLUSRULES (PLUS (MINUS $X) .
                                         (MINUS (PLUS $$Y]
```

At present there are only two subtraction rules, and so we do not combine them into one operator:

- $x-y = x+(-y)$:

```
SUBPLUS=(LAMBDA (SUBTRACT ←X ←Y)
                ($TRY $PLUSRULES (' (PLUS $X (MINUS $Y]
```

- If x and y are both numbers and not variables, SUBNUM actually evaluates x-y:

```
SUBNUM=(LAMBDA (SUBTRACT ←X ←Y)
               (PROG (DECLARE)
                     ($INSIST (AND (EQUAL (STYPE $X NUMBER))
                                   (EQUAL (STYPE $Y)
                                          NUMBER)))
                     (RETURN (= (SUBTRACT $X $Y]
```

The "=" sign forces the system to evaluate what it would otherwise merely instantiate. INSIST is another utility function.

Two more rules about exponentiation are given below.

- $x^0 = 1$:

```
: EXPZERO=[LAMBDA (EXP ←X 0)
                  1])
```

- $(x^y)^z = x^{y*z}$:

```
EXPEXP=(LAMBDA (EXP (EXP ←X ←Y)
                    ←Z)
               ($TRYSUB $TIMESRULES ON (' (TIMES $Y $Z))
                        IN
                        (' (EXP $X (TIMES $Y $Z]
```

Note that **EXPEXP** recommends that the **TIMESRULES** be applied to the product y*z; this is heuristic advice that could have been omitted.

- (GCD x x) = x

```
GCDEQ=[LAMBDA (GCD ←X ←Y)
              (PROG (DECLARE)
                    (GOAL $DEDUCE (EQ ←X ←Y))
                    (RETURN $X]
```

The GCD is the greatest common divisor.

6. Reasoning About Arrays

Most of the knowledge about arrays embedded in the system is expressed as simplification rules.

- (ACCESS (CHANGE A I T) I) = T

  I ≠ J ⊃ (ACCESS (CHANGE A I T) J) = (ACCESS A J):

```
ACCH=(LAMBDA (ACCESS (CHANGE ←A ←I ←T)
                     ←J)
             (PROG (DECLARE)
                   (ATTEMPT (GOAL $DEDUCE (EQ $I $J))
                            THEN
                            (RETURN $T))
                   (GOAL $DEDUCE (NEQ $I $J))
                   (RETURN (ACCESS $A $J]
```

$$(\text{MAXA A I J+1}) = A[J+1] \quad .$$

On the other hand, if

$$(\text{MAXA A I J}) > A[J+1] \quad ,$$

then

$$(\text{MAXA A I J+1}) = (\text{MAXA A I J}) \quad ;$$

```
MAXPLUS =(LAMBDA
   (MAXA ←A ←I (PLUS ←J 1))
   (PROG (DECLARE)
         [ATTEMPT [GOAL $DEDUCE (LTQ (MAXA $A $I $J)
                                     (ACCESS $A (PLUS $J 1]
                  THEN
                  (RETURN (ACCESS $A (PLUS $J 1]
         (GOAL $DEDUCE (LT (ACCESS $A (PLUS $J 1))
                           (MAXA $A $I $J)))
         (RETURN (MAXA $A $I $J]
```

- Recall that (BAGA A I J) is the bag [A[i], A[i+1], ... A[j]]. Because of the crucial part this function plays in assertions about sort-like programs, we have many rules for it.

```
BAGARULES =
(TUPLE BAGAPLUS BAGAEMPTY BAGAII ARGSIMP BACH BAGEX BAGEX1
       BAGAMINUS BAGALOWERPLUS BAGEXCOMPLICATED))
```

- These rules are controlled by the rule BAGAOP, one of the TOPRULES:

```
BAGAOP=[LAMBDA (PAND ←Y (BAGA ←←X))
               ($TRYALLFAIL $BAGARULES $Y])
```

Thus, the BAGARULES will be tried whenever we are simplifying an expression of the form (BAGA A I J).

- If $I \leq J+1$, then (BAGA A I J+1) = (BAG A[J+1] (STRIP(BAGA A I J))):

```
BAGAPLUS=(LAMBDA (BAGA ←A ←I (PLUS 1 ←J))
                 (PROG (DECLARE)
                       (GOAL $DEDUCE (LTQ $I (PLUS 1 $J)))
                       (RETURN (BAG (ACCESS $A (PLUS $J 1))
                                    (STRIP (BAGA $A $I $J]
```

ACCH is one of the TOPRULES, as are the rules below, ACCEX, MAXONE, MAX, and MAXPLUS.

- (EXCHANGE A I J) is a higher level function whose output is the array A with the values of A[I] and A[J] exchanged. The value of (ACCESS (EXCHANGE A I J) K) depends on whether or not K equals I or J, i.e., whether the element here accessed was affected by the exchange. If K = I, the value is A[J]. If K = J, the value is A[I]. If K is neither I nor J, the value is the original value of A[K], since that location has not been affected by the exchange. The rule fails if it cannot be determined that K= I or K=J. This information is embodied in the rule ACCEX:

```
ACCEX=(LAMBDA (ACCESS (EXCHANGE ←A ←I ←J)
                      ←K)
              (PROG (DECLARE)
                    (ATTEMPT (GOAL $DEDUCE (EQ $K $I))
                             THEN
                             (RETURN (ACCESS $A $J)))
                    (ATTEMPT (GOAL $DEDUCE (EQ $K $J))
                             THEN
                             (RETURN (ACCESS $A $I)))
                    (GOAL $DEDUCE (AND (NEQ $K $I)
                                       (NEQ $K $J)))
                    (RETURN (ACCESS $A $K]
```

- The maximum of an array, MAXA, is a function of three arguments: the array, the lower bound, and the upper bound.

(MAXA A I J) = (MAX A[I], A[I+1], ..., A[J])

(MAXA A I I) = A[I]:

```
MAXONE=(LAMBDA (MAXA ←A ←I ←J)
               (PROG (DECLARE)
                     (GOAL $DEDUCE (EQ $I $J))
                     (RETURN (ACCESS $I]
```

- If (MAXA A I J) $\le$ A[J+1] ,

then

- If I < J, (BAGA A J I) is the empty bag:

```
BAGAEMPTY=(LAMBDA (BAGA ←A ←J ←I)
                  (PROG (DECLARE)
                        (GOAL $DEDUCE (LT $I $J))
                        (RETURN (BAG]
```

- (BAGA A I I) is [A[I]]:

```
BAGAII=(LAMBDA (BAGA ←A ←I ←J)
               (PROG (DECLARE)
                     (GOAL $DEDUCE (EQ $I $J))
                     (RETURN (BAG (ACCESS $A $I]
```

- If I ≤ J, then (BAGA A I J) = (BAG (STRIP (BAGA A I J-1)) A[J]):

```
BAGAMINUS=(LAMBDA
   (BAGA ←A ←I ←J)
   (PROG (DECLARE)
         ($INSIST (EQUAL (STYPE $J)
                         IDENT))
         (GOAL $DEDUCE (LTQ $I $J))
         (RETURN (BAG (ACCESS $A $J)
                      (STRIP (BAGA $A $I (SUBTRACT $J 1]
```

Since this rule would apply so often, it is restricted by forcing J to be an identifier rather than a complex expression.

- If L ≤ M, then

(BAGA A L M) = (BAG A[L] (STRIP (BAGA A L+1 M))) :

```
BAGALOWERPLUS=
(LAMBDA (BAGA ←ARNAME ←L ←M)
        (PROG (DECLARE F LOWER UPPER W)
              (EXISTS (←F ←←V (STRIP (BAGA $ARNAME ←LOWER
                                                   ←UPPER))
                          ←←W))
              (GOAL $DEDUCE (EQ $LOWER (PLUS 1 $L)))
              (RETURN (BAG (ACCESS $ARNAME $L)
                           (STRIP (BAGA $ARNAME (PLUS 1 $L)
                                        $M]
```

This rule tries to determine if its application is desirable by checking in the model for any relationship involving an array segment with lower bound equal to L+1; if no such relationship exists, it is doubtful that the proposed simplification will lead to a proof.

- If $I \leq J \leq K$, then

$$\text{(BAGA (CHANGE A J T) I K)} =$$
$$\text{(BAG T (STRIP (BAGA A I K)))} \sim \text{(BAG A[J])} \quad .$$

On the other hand, if $J < I$ or $K < J$,

$$\text{(BAGA (CHANGE A J T) I K)} = \text{(BAGA A I K)} \quad .$$

(The notation $\sim$ means the difference between two bags.) In other words, making an assignment to an array element whose index is outside the bounds of a segment does not affect the segment. However, if the index is within bounds of the segment, then the corresponding bag will lose the old value of the array element but gains the new value:

```
BACH =
 (LAMBDA
   (BAGA (CHANGE ←A ←J ←T)
         ←I ←K)
   (PROG
     (DECLARE)
     [ATTEMPT
       (GOAL $DEDUCE (LTQ $I $J $K))
       THEN
       (RETURN
         (=
           ($TRY
             $DIFFRULES
             ($TRYSUB
               (TUPLE ACCH ACCEX)
               ON
               (' (ACCESS $A $J))
               IN
               ($TRYSUB $BAGARULES ON (' (BAGA $A $I $K))
                        IN
                        (' (DIFFERENCE (BAG $T
                                            (STRIP (BAGA $A $I $K)))
                                       (BAG (ACCESS $A $J]
     (GOAL $DEDUCE (OR (LT $J $I)
                       (LT $K $J)))
     (RETURN (BAGA $A $I $K]
```

The rule BACH contains many recommendations about possible future simplifications. These recommendations are included to promote efficiency; the simplifier would eventually try the recommended rules even if the advice were omitted. The advice-giving functions TRY and TRYSUB are described in the section on utility functions.

- As mentioned above, (EXCHANGE A I J) is the array A with the values of A[I] and A[J] interchanged. If I and J are either both inside or both outside an array segment, then the exchange operation has no affect on the bag corresponding to that segment:

```
BAGEX (LAMBDA (BAGA (EXCHANGE ←A ←I ←J)
                    ←L ←M)
              (PROG (DECLARE)
                    (GOAL $DEDUCE (LTQ $I $J))
                    (ATTEMPT
                      (GOAL $DEDUCE
                            (OR (AND (LTQ $L $I)
                                     (LTQ $J $M))
                                (LT $J $L)
                                (LT $M $I)
                                (AND (LT $I $L)
                                     (LT $M $J]
                      THEN
                      (RETURN (BAGA $A $L $M))
                      ELSE
                      (FAIL]
```

For simplicity, BAGEX requires that $I \leq J$.

- If elements A[I] and A[J] are exchanged, and if J is in the array segment and I is not, or if I is in the segment and J is not, then the corresponding bag is indeed affected by the exchange operation. For instance, in the case in which J is in the segment and I is not, if the segment is bounded by L and M, the new bag is

```
(BAG (STRIP (BAGA A L J-1)
     A[I]
     (STRIP (BAGA A J+1 M)))    :
```

```
BAGEX1 =
(LAMBDA
  (BAGA (EXCHANGE ←A ←I ←J)
        ←L ←M)
  (PROG (DECLARE)
        (GOAL $DEDUCE (LTQ $I $J))
        [ATTEMPT (GOAL $DEDUCE (AND (LT $I $L)
                                    (LTQ $L $J)
                                    (LTQ $J $M)))
                 THEN
                 (RETURN (BAG (STRIP (BAGA $A $L (SUBTRACT
                                                  $J 1)))
                              (ACCESS $A $I)
                              (STRIP (BAGA $A (PLUS 1 $J)
                                                $M]
        (ATTEMPT (GOAL $DEDUCE (AND (LT $M $J)
                                    (LTQ $L $I)
                                    (LTQ $I $M)))
                 THEN
                 [RETURN (BAG (STRIP (BAGA $A $L (SUBTRACT
                                                  $I 1)))
                              (ACCESS $A $J)
                              (STRIP (BAGA $A (PLUS 1 $I)
                                                $M]
                 ELSE
                 (FAIL]
```

- BAGEXCOMPLICATED handles the case in which it can be determined that one of the exchanged elements is within or outside the array segment, but the location of the other array element is uncertain. The result is then a conditional expression. For example, if J is known to be outside the segment but I is only known to be greater than or equal to the lower limit L, the result is

```
(IF M < I THEN (BAGA A L M)
          ELSE (BAG (STRIP (BAGA A L I-1))
                    A[J]
                    (STRIP (BAGA A I+1 M))))
```

```
BAGEXCOMPLICATED=
(LAMBDA
  (BAGA (EXCHANGE ←A ←I ←J)
        ←L ←M)
  (PROG (DECLARE)
        (GOAL $DEDUCE (LTQ $I $J))
        [ATTEMPT (GOAL $DEDUCE (AND (LTQ $L $I)
                                    (LTQ $M $J)))
                 THEN
                 (RETURN (IFTHENELSE
                          (LT $M $I)
                          (BAGA $A $L $M)
                          (BAG (STRIP (BAGA $A $L (SUBTRACT $I 1)))
                               (ACCESS $A $J)
                               (STRIP (BAGA $A (PLUS 1 $I)
                                                  $M]
        (ATTEMPT (GOAL $DEDUCE (AND (LTQ $J $M)
                                    (LTQ $L $J)))
                 THEN
                 [RETURN (IFTHENELSE
                          (LTQ $L $I)
                          (BAGA $A $L $M)
                          (BAG (STRIP (BAGA $A $L (SUBTRACT $J 1)))
                               (ACCESS $A $I)
                               (STRIP (BAGA $A (PLUS 1 $J)
                                                  $M]
                 ELSE
                 (FAIL]
```

BAGEXCOMPLICATED comes after BAGEX and BAGEX1 in the goal class BAGARULES because we prefer the definite answer they provide to the conditional expression returned by BAGEXCOMPLICATED.

All the rules in this section have been simplification rules. There also are two inequalities rules that pertain to arrays, INEQSTRIPTRAN and INEQSTRIPSTRIP.

- To prove that every element in an array segment is less than (or less than or equal to) some quantity C, find an array segment that properly contains the given segment such that every element in the larger segment is less than some element D that is, in turn, less than or equal to C:

```
INEQSTRIPTRAN=
(LAMBDA (←F (STRIP (BAGA ←ARNAME ←L ←M))
            ←C)
        (PROG (DECLARE LOWER UPPER D)
              (EXISTS ($F (STRIP (BAGA $ARNAME ←LOWER ←UPPER))
                             ←D))
              (GOAL $DEDUCE (AND (LTQ $LOWER $L)
                                 (LTQ $M $UPPER)
                                 (LTQ $D $C]
```

- To prove some ordering relation < or ≤, between all the elements of two array segments, S_1 and S_2, find relations of the same sense involving S_1' and C, and involving D and S_2'. Then show that S_1' and S_2' contain S_1 and S_2 respectively, and that C is less than or equal to D.

```
INEQSTRIPSTRIP=
(LAMBDA (←F (STRIP (BAGA ←A ←I ←J))
            (STRIP (BAGA ←A ←K ←L)))
        (PROG (DECLARE LOWER1 UPPER1 LOWER2 UPPER2 C D)
              (ATTEMPT (EXISTS ($F (STRIP (BAGA $A ←LOWER1
                                                  ←UPPER1))
                                   ←C))
                       [EXISTS ($F ←D
                                   (STRIP (BAGA $A ←LOWER2
                                                  ←UPPER2]
                       (GOAL $DEDUCE (AND (LTQ $LOWER1 $I)
                                          (LTQ $J $UPPER1)
                                          (LTQ $LOWER2 $K)
                                          (LTQ $L $UPPER2)
                                          (LTQ $C $D)))
                       ELSE
                       (FAIL]
```

7. Reasoning About Bags

We have accumulated a number of rules about bags. Many of these rules have set-theoretic counterparts, which could have been included, but we have needed only bags in our proofs.

We use the QA4 function DIFFERENCE to mean the difference between bags, written informally as "∼" .

- (BAG x y) ∼ (BAG x) = (BAG y)

```
DIFFXX=[LAMBDA (DIFFERENCE (BAG ←X ←←Y)
                           (BAG ←X))
               (BAG $$Y])
```

- $$cons(x, y \sim z) = cons(x,y) \sim z:$$

```
CONSDIFF (LAMBDA =(CONS ←X (DIFFERENCE ←Y ←←Z))
                  (' (DIFFERENCE (CONS $X $Y)
                                  $$Z]
```

- $$(x \sim y) \sim z = x \sim y \sim z \quad :$$

```
DIFDIF=(LAMBDA (DIFFERENCE (DIFFERENCE ←X ←←Y)
                             ←←Z)
                (' (DIFFERENCE $X $$Y $$Z]
```

- $$cons(x,y) \sim (BAG\ x) \sim u = x \sim u \quad :$$

```
DIFFCONS=(LAMBDA (DIFFERENCE (CONS ←X ←Y)
                              (BAG ←X)
                              ←←U)
                  ($TRY (TUPLE DIFFONE)
                        (' (DIFFERENCE $Y $$U]
```

- (DIFFERENCE x) is taken to be x itself:

```
DIFFONE=[LAMBDA (DIFFERENCE ←X)
                $X])
```

- $$(BAG\ (strip\ x)) = x:$$

```
BAGSTRIP=[LAMBDA (BAG (STRIP ←X))
                 $X])
```

8. Reasoning About Substitutions

The rules in this section were added to prove assertions about the pattern matcher and the unification algorithm.

- An atom is either a variable or a constant;

$$\neg var(x) \wedge \neg const(x) \supset \neg Atom(x) \quad :$$

```
NOTATOM=(LAMBDA (NOT (ATOM ←X))
                (PROG (DECLARE)
                      (EXISTS (NOT (VAR $X)))
                      (EXISTS (NOT (CONST $X]
```

- If an expression is made of constants, so is the car and the cdr of the expression:

```
CONSTCAR=(LAMBDA (CONSTEXP (CAR ←X))
                 (EXISTS (CONSTEXP $X]

CONSTCDR=(LAMBDA (CONSTEXP (CDR ←X))
                 (EXISTS (CONSTEXP $X]
```

NOTATOM, CONSTCAR, and CONSTCDR are DEDUCE rules.

- The empty substitution does not change the expression:

```
SUBSTEMPTY=(LAMBDA (VARSUBST EMPTY ←X)
                   $X])
```

- No substitution changes an expression made up entirely of constants:

```
SUBSTCONST=(LAMBDA (VARSUBST ←S ←Y)
                   (PROG (DECLARE)
                         (GOAL $DEDUCE (CONSTEXP $Y))
                         $Y])
```

SUBSTEMPTY and SUBSTCONST are simplification rules.

- To prove

$$varsubst(s, car(x)) = car(y) \quad ,$$

prove

$$varsubst(s,x) = y$$

```
SUBSTCAR=(LAMBDA (EQ (VARSUBST ←S1 (CAR ←X))
                     (CAR ←Y))
                 (GOAL $EQRULES (EQ (VARSUBST $S1 $X)
                                    $Y]
```

- Similarly, to prove

$$varsubst(s, cdr(x)) = cdr(y) \quad ,$$

prove

$$varsubst(s,x) = y$$

```
SUBSTCDR=(LAMBDA (EQ (VARSUBST ←S1 (CDR ←X))
                     (CDR ←Y))
                 (GOAL $EQRULES (EQ (VARSUBST $S1 $X)
                                    $Y]
```

- To prove

$$varsubst(s, x) = y \quad ,$$

where x and y are nonatomic, prove

$$varsubst(s, car(x)) = car(y)$$

and

$$varsubst(s, cdr(x)) = cdr(y) \quad .$$

```
SUBSTCONS=(LAMBDA (EQ (VARSUBST ←S1 ←X)
                      ←Y)
                  (PROG (DECLARE)
                        (GOAL $DEDUCE (NOT (ATOM $X)))
                        (GOAL $DEDUCE (NOT (ATOM $Y)))
                        (GOAL (= ($REMOVE EQSUBST FROM
                                          $EQRULES))
                              (EQ (VARSUBST $S1 (CAR $X))
                                  (CAR $Y)))
                        (GOAL (= ($REMOVE EQSUBST FROM
                                          $EQRULES))
                              (EQ (VARSUBST $S1 (CDR $X))
                                  (CDR $Y]
```

- SUBSTCAR, SUBSTCDR and SUBSTCONS are equality rules. They are clustered together in a goal class:

```
EQSUBSTRULES=(TUPLE SUBSTCAR SUBSTCDR SUBSTCARCDR SUBSTCONS))
```

- EQSUBSTRULES is called from EQSUBST, an EQRULE.

```
EQSUBST=(LAMBDA (PAND ←Y (EQ (VARSUBST ←S ←X)
                             ←Z))
                (GOAL $EQSUBSTRULES $Y))
```

Note that SUBSTCONS removes EQSUBST from the EQRULES. This prevents the system from looping by applying SUBSTCONS followed by SUBSTCAR.

- To prove

$$varsubst(s,u) = varsubst(s,v) \quad ,$$

where u and v are nonatomic, prove

$$varsubst(s,car(u)) = varsubst\ (s, car(v))$$

and

varsubst(s,cdr(u)) = varsubst(s,cdr(v)) .

```
SUBSTCARCDR=(LAMBDA (EQ (VARSUBST ←S ←U)
                        (VARSUBST ←S ←V))
                    (PROG (DECLARE)
                          (GOAL $DEDUCE (NOT (ATOM $U)))
                          (GOAL $DEDUCE (NOT (ATOM $V)))
                          [GOAL $EQRULES
                                (EQ (VARSUBST $S (CAR $U))
                                    (VARSUBST $S (CAR $V]
                          (GOAL $EQRULES
                                (EQ (VARSUBST $S (CDR $U))
                                    (VARSUBST $S (CDR $V]
```

Substitutions are represented as lists of dotted pairs.

- If v is a variable,

varsubst(((v•y)), v) = y

```
SUBSTLIST=[LAMBDA (VARSUBST (LIST (CONS ←V ←Y))
                            ←V)
                  (PROG (DECLARE)
                        (EXISTS (VAR $V))
                        $Y])
```

- The composition operator has the property:

varsubst(compose(s1, s2), x) = varsubst(s1, varsubst(s2, x)) .

```
SUBSTCOMPOSE=(LAMBDA (VARSUBST (COMPOSE ←S1 ←S2)
                               ←X)
                     ($TRY $SUBSTRULES
                           (' (VARSUBST $S1 (VARSUBST $S2 $X]
```

- These simplification rules are members of the goal class

```
SUBSTRULES=(TUPLE SUBSTEMPTY SUBSTLIST SUBSTCOMPOSE SUBSTCONST)
)
```

which is called by SUBSTOP, a member of TOPRULES:

```
SUBSTOP=[LAMBDA (PAND (VARSUBST ←←X)
                      ←Y)
                (GOAL $SUBSTRULES $Y])
```
.

9. Utility Functions

- TRY is like a GOAL statement that will not fail if none of the goal class apply but instead returns its argument.

```
TRY=(LAMBDA (TUPLE ←GOALCLASS ←GOAL1)
            (ATTEMPT (GOAL $GOALCLASS $GOAL1)
                     ELSE $GOAL1))
```

It evaluates (GOAL $GOALCLASS $GOAL1), but, if failure results, it returns GOAL1.

- TRYALL will try a goal class on an expression. If any member of the goal class applies, it will apply the same goal class to the resulting expression, and so on, until no rules applies. TRYALL returns the last expression it has derived, which may be the same as the first expression. TRYALL will not fail:

```
TRYALL=(LAMBDA (TUPLE ←GOALCLASS1 ←GOAL1)
               (PROG (DECLARE)
                     TOP
                     (ATTEMPT (SETQ ←GOAL1 (GOAL $GOALCLASS1
                                                 $GOAL1))
                              THEN
                              (GO TOP))
                     (RETURN $GOAL1]
```

- TRYALLFAIL is like TRYALL, except it will fail if none of the goal class apply to the argument.

```
TRYALLFAIL=(LAMBDA (TUPLE ←GOALCLASS1 ←GOAL1)
                   ($TRYALL $GOALCLASS1 (GOAL $GOALCLASS1
                                              $GOAL1]
```

- TRYSUB applies a goal class to a specially designated subexpression of the given expression:

```
TRYSUB=(LAMBDA (TUPLE ←GOALCLASS ON ←SUB IN ←EXP)
               (SUBST $EXP (TUPLE $SUB ($TRYALL $GOALCLASS
                                                $SUB]
```

- INSIST fails if its argument is FALSE.

```
INSIST=(LAMBDA ←X (IF $X ELSE (FAIL]
```

- REMOVE removes a designated item from a tuple.

```
REMOVE= [LAMBDA (TUPLE ←X FROM ←Y)
                ((QUOTE [LAMBDA (TUPLE ←←U $X ←←V)
                                (TUPLE $$U $$V]]
                 $Y])
```

- ASK queries the user:

```
ASK=(LAMBDA ←X (IF (LISP ASK $X)
                   ELSE
                   (FAIL]
```

It types two expressions, the first a QA4 expression ($Y) and the second an atom($X; e.g., PROVE? or SIMPLIFY?). If the user types YES, TRUE, OK, Y, or T, say, ASK returns TRUE. Otherwise ASK fails. ASK uses a LISP function of the same name.

- SHORTEST computes the "smallest" element of a set, bag, or tuple:

```
SHORTEST =
[LAMBDA
  ←X
  (PROG (DECLARE BEST BESTCOUNT TEMPCOUNT)
        (SETQ ←BESTCOUNT 2000)
        [MAPC $X (QUOTE (LAMBDA
                          ←Y
                          (IF (OR (LT (SETQ ←TEMPCOUNT
                                            (LISP QA4COUNT $Y))
                                      $BESTCOUNT)
                                  (EQUAL (STYPE $Y)
                                         NUMBER))
                              THEN
                              (SETQ ←BEST $Y)
                              (SETQ ←BESTCOUNT $TEMPCOUNT]
        $BEST])
```

The size of an expression is roughly the number of atoms in the expression. It is computed by a LISP function, QA4COUNT. Numbers are assumed to be "smaller" than identifiers.

Table A-1 gives an index of functions and goal classes.

Table A-1

INDEX OF FUNCTIONS AND GOAL CLASSES

*Goal class.

Table A-1 (Concluded)

INDEX OF FUNCTIONS AND GOAL CLASSES

Appendix B

TRACES OF SOLUTIONS

Appendix B

TRACES OF SOLUTIONS

1. The Maximum of an Array (1)

A complete trace of a proof performed by our system is presented below. The verification condition to be proved is derived from the program to compute the maximal element of an array. Although a proof is contained above in the body of the text, following the trace tells us exactly what rules were applied in the proof. Furthermore, we can see exactly what false starts were made by the system and what user interaction was required to keep the program on the right track.

This particular verification condition was derived from the loop path of the program. The hypotheses are

```
(CONTEXT (1 0) 1 0)
(ASSERT (EQ MAX (ACCESS A LOC)) WRT $VERICON)
TRUE
(ASSERT (LTQ (STRIP (BAGA A 0 I)) MAX) WRT $VERICON)
TRUE
(ASSERT (LTQ 0 LOC) WRT $VERICON)
TRUE
(ASSERT (LTQ LOC I) WRT $VERICON)
TRUE
(ASSERT (LTQ I N) WRT $VERICON)
TRUE
(DENY (LT N (PLUS 1 I)) WRT $VERICON)
FLASE
(DENY (LT MAX (ACCESS A (PLUS 1 I))) WRT $VERICON)
FLASE
```

The word "Flase" is a misspelling of "False."

Since the hypotheses for the different verification conditions of a program may contradict each other, each proof is done in a separate context. The name of that context is VERICON. That is the meaning of the phrase "WRT $VERICON," which follows our assertions and goal. (For this proof VERICON is ((1 0) 1 0).) Assertions made with respect to one VERICON will not affect problems solved with respect to another.

```
16 (GOAL $PROVE (LTQ (STRIP (BAGA A 0 (PLUS 1 I))) MAX) WRT $VERICON)
17   LAMBDA PROOFSWITCH (LTQ (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
```

When a traced function is applied to an argument, the trace says

LAMBDA ⟨function name⟩ ⟨argument⟩ .

Some of the utility functions are not traced.

```
18    (GOAL $INEQUALITIES ($F $X))
19      LAMBDA RELCHECK (LTQ (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
20      LAMBDA PROOFSIMP (LTQ (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
21        LAMBDA ARGSIMP (LTQ (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
22          LAMBDA SIMPONE (TUPLE (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
(TUPLE (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
SIMPLIFY?
:Y
```

The system asked us whether we wanted it to simplify

(TUPLE (STRIP (BAGA A 0 (PLUS 1 I))) MAX) .

We said yes.

```
23           (GOAL $TOPRULES $GOAL1)
24             LAMBDA HASSIMP (TUPLE (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
25             (FAIL)
26             LAMBDA EQNUMB (TUPLE (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
27             (FAIL)
28           (GOAL $DOWNRULES $GOAL1)
29             LAMBDA TUPSIMP (TUPLE (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
30               LAMBDA SIMPONE (STRIP (BAGA A 0 (PLUS 1 I)))
(STRIP (BAGA A 0 (PLUS 1 I)))
SIMPLIFY?
:Y
31               (GOAL $TOPRULES $GOAL1)
32                 LAMBDA HASSIMP (STRIP (BAGA A 0 (PLUS 1 I)))
33                 (FAIL)
34                 LAMBDA EQNUMB (STRIP (BAGA A 0 (PLUS 1 I)))
35                 (FAIL)
36               (GOAL $DOWNRULES $GOAL1)
37                 LAMBDA ARGSIMP (STRIP (BAGA A 0 (PLUS 1 I)))
38                   LAMBDA SIMPONE (BAGA A 0 (PLUS 1 I))
(BAGA A 0 (PLUS 1 I))
SIMPLIFY?
:Y
```

We have given the system permission to simplify (BAGA A 0 (PLUS 1 I)).

```
                    (GOAL $TOPRULES $GOAL1)
                      LAMBDA HASSIMP (BAGA A 0 (PLUS 1 I))
                      (FAIL)
                      LAMBDA BAGAOP (BAGA A 0 (PLUS 1 I))
                      (GOAL $GOALCLASS1 $GOAL1)
                        LAMBDA BAGAPLUS (BAGA A 0 (PLUS 1 I))
                         (GOAL $DEDUCE (LTQ $I (PLUS 1 $J)))
```

The system tries to prove that 0 ≤ (PLUS 1 I)).

```
                           LAMBDA RELCHECK (LTQ 0 (PLUS 1 I))
                           LAMBDA LTQPLUS (LTQ 0 (PLUS 1 I))
                           (GOAL $DEDUCE (AND (LTQ $I $J) (LTQ 0 $K)))
                             LAMBDA RELCHECK (AND (LTQ 0 I) (LTQ 0 1))
```

It breaks down the goal to 0 ≤ I and 0 ≤ 1.

```
                             LAMBDA ANDSPLIT (AND (LTQ 0 I) (LTQ 0 1))
                             (GOAL $GOALCLASS $X)
                               LAMBDA RELCHECK (LTQ 0 1)
                               RELCHECK = TRUE
```

When a function returns a value, the trace says

⟨function name⟩ = ⟨value⟩ .

In this case, the system knew that 0 < 1 by performing the corresponding LISP evaluation.

```
                             (GOAL $GOALCLASS $Y)
                               LAMBDA RELCHECK (LTQ 0 I)
                               RELCHECK = TRUE
```

The 0 ≤ I follows from hypothesis 8 and 9.

```
                             ANDSPLIT = TRUE
                           LTQPLUS = TRUE
                         BAGAPLUS = (BAG (ACCESS A (PLUS 1 I)) (STRIP
(BAGA A 0 I)))
```

The system has succeeded in simplifying

(BAGA A 0 (PLUS 1 I))

to

(BAG (ACCESS A (PLUS 1 I)) (STRIP (BAGA A 0 I))) .

```
60                        (GOAL $GOALCLASS1 $GOAL1)
61                        BAGAOP = (BAG (ACCESS A (PLUS 1 I)) (STRIP (BAGA
A 0 I)))
62                     SIMPONE = (BAG (ACCESS A (PLUS 1 I)) (STRIP (BAGA
A 0 I)))
63                  ARGSIMP = (STRIP (BAG (ACCESS A (PLUS 1 I)) (STRIP
(BAGA A 0 I))))
64                (GOAL $GOALCLASS $GOAL1)
65                  LAMBDA HASSIMP (STRIP (BAG (ACCESS A (PLUS 1 I))
(STRIP (BAGA A 0 I))))
66                  (FAIL)
67                  LAMBDA EQNUMB (STRIP (BAG (ACCESS A (PLUS 1 I)) (STRIP
(BAGA A 0 I))))
68                  (FAIL)
69               SIMPONE = (STRIP (BAG (ACCESS A (PLUS 1 I)) (STRIP
(BAGA A 0 I))))
70            TUPSIMP = (TUPLE (STRIP (BAG (ACCESS A (PLUS 1 I)) (STRIP
(BAGA A 0 I)))) MAX)
71          (GOAL $GOALCLASS $GOAL1)
72            LAMBDA HASSIMP (TUPLE (STRIP (BAG (ACCESS A (PLUS 1 I))
(STRIP (BAGA A 0 I)))) MAX)
73            (FAIL)
74            LAMBDA EQNUMB (TUPLE (STRIP (BAG (ACCESS A (PLUS 1 I))
(STRIP (BAGA A 0 I)))) MAX)
75            (FAIL)
76          SIMPONE = (TUPLE (STRIP (BAG (ACCESS A (PLUS 1 I)) (STRIP
(BAGA A 0 I)))) MAX)
77        ARGSIMP = (LTQ (STRIP (BAG (ACCESS A (PLUS 1 I)) (STRIP (BAGA
A 0 I)))) MAX)
```

The problem now is to prove

(STRIP (BAG (ACCESS A (PLUS 1 I)) (STRIP (BAGA A 0 I)))) ≤ MAX :

```
78     (GOAL $GOALCLASS1 $X)
79       LAMBDA RELCHECK (LTQ (STRIP (BAG (ACCESS A (PLUS 1 I)) (STRIP
(BAGA A 0 I)))) MAX)
80       RELCHECK = TRUE
```

But since the system already knows

(ACCESS A (PLUS 1 I)) ≤ MAX from (14),

and

(STRIP (BAGA A 0 I)) ≤ MAX from (4) ,

the proof is complete:

```
81     PROOFSIMP = TRUE
82   (ASSERT ($F $X))
83   PROOFSWITCH = (LTQ (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
84 (LTQ (STRIP (BAGA A 0 (PLUS 1 I))) MAX)
```

The Maximum of an Array (2)

The following is the trace of the proof for another verification condition for the program that computes the maximal element of an array. This verification condition is derived from the halt path of the program.

```
1  (CONTEXT (1 0) 1 0)
2  (ASSERT (EQ MAX (ACCESS A LOC)) WRT $VERICON)
3  TRUE
4  (ASSERT (LTQ (STRIP (BAGA A 0 I)) MAX) WRT $VERICON)
5  TRUE
6  (ASSERT (LTQ 0 LOC) WRT $VERICON)
7  TRUE
8  (ASSERT (LTQ LOC I) WRT $VERICON)
9  TRUE
10 (ASSERT (LTQ I N) WRT $VERICON)
11 TRUE
12 (ASSERT (LT N (PLUS 1 I)) WRT $VERICON)
```

There is a demon that knows that in the integer domain,

$$x < y \supset x+1 \leq y \quad .$$

This demon is responsible for the assertion

```
13 (ASSERT (LTQ (PLUS 1 $X) $Y) WRT $VERICON)
```

The system now knows $N+1 \leq I+1$. This assertion wakes up another demon:

```
14 (ASSERT (LTQ $Y $Z) WRT $VERICON)
15 TRUE
```

The system now knows that $N \leq I$. Since $I \leq N$ has just been asserted (5), the mechanism for storing ordering relations silently tells the system that $I = N$.

The system proceeds with the proof:

```
16 (GOAL $PROVE (LTQ (STRIP (BAGA A 0 N)) (ACCESS A LOC)) WRT $VERICON)
17   LAMBDA PROOFSWITCH (LTQ (STRIP (BAGA A 0 N)) (ACCESS A LOC))
18    (GOAL $INEQUALITIES ($F $X))
19      LAMBDA RELCHECK (LTQ (STRIP (BAGA A 0 N)) (ACCESS A LOC))
20      LAMBDA PROOFSIMP (LTQ (STRIP (BAGA A 0 N)) (ACCESS A LOC))
21        LAMBDA ARGSIMP (LTQ (STRIP (BAGA A 0 N)) (ACCESS A LOC))
22          LAMBDA SIMPONE (TUPLE (STRIP (BAGA A 0 N)) (ACCESS A LOC))
(TUPLE (STRIP (BAGA A 0 N)) (ACCESS A LOC))
SIMPLIFY?
:N
23          (FAIL)
24      (FAIL)
25      LAMBDA PROOFLEIB (LTQ (STRIP (BAGA A 0 N)) (ACCESS A LOC))
26      (EXISTS ($F ←Y))
```

The system searches the data base for an assertion of the form (LTQ ←Y), i.e., the gross form of the goal we are trying to prove. It finds one [assertion (2)] and asks us if it should try to prove that the argument of the assertion it has found is equal to the argument of the goal:

```
(EQ (TUPLE (STRIP (BAGA A 0 N)) (ACCESS A LOC)) (TUPLE (STRIP (BAGA
A 0 I)) MAX))
PROVE?
```

We say yes, and the proof proceeds.

```
:Y
27      (GOAL $EQRULES (EQ $X $Y))
28        LAMBDA RELCHECK (EQ (TUPLE (STRIP (BAGA A 0 N)) (ACCESS A
LOC)) (TUPLE (STRIP (BAGA A 0 I)) MAX))
29        RELCHECK = TRUE
30      PROOFLEIB = TRUE
31    (ASSERT ($F $X))
32    PROOFSWITCH = (LTQ (STRIP (BAGA A 0 N)) (ACCESS A LOC))
33 (LTQ (STRIP (BAGA A 0 N)) (ACCESS A LOC))
```

The two tuples were found to be equal because N = I (from 10 and 14), and MAX = A[LOC]. The proof is complete.

2. <u>The Wensley Division Algorithm</u>

The following is the complete trace of the proof included in the body of the text:

```
1  (CONTEXT (1 0) 1 0)
2  (ASSERT (EQ AA (TIMES QQ YY)) WRT $VERICON)
3  TRUE
4  (ASSERT (EQ (TIMES 2 BB) (TIMES QQ DD)) WRT $VERICON)
5  TRUE
6  (ASSERT (LT PP (PLUS (TIMES QQ YY) (TIMES QQ DD))) WRT $VERICON)
7  TRUE
8  (ASSERT (LTQ (TIMES QQ YY) PP) WRT $VERICON)
9  TRUE
10 (ASSERT (LT PP (PLUS AA BB)) WRT $VERICON)
11 TRUE
12 (DENY (LT (DIVIDES DD 2) EE) WRT $VERICON)
13 FALSE
```

The goal is to prove PP < QQ*YY + QQ*(DD/2):

```
14 (GOAL $PROVE (LT PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES DD 2))))
WRT $VERICON)
15   LAMBDA PROOFSWITCH (LT PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES
DD 2)))))
16    (GOAL $INEQUALITIES ($F $X))
17      LAMBDA RELCHECK (LT PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES
DD 2)))))
18      LAMBDA PROOFSIMP (LT PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES
DD 2)))))
19        LAMBDA ARGSIMP (LT PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES
DD 2)))))
20          LAMBDA SIMPONE (TUPLE PP (PLUS (TIMES QQ YY) (TIMES QQ
(DIVIDES DD 2))))
21            LAMBDA ASK (TUPLE (TUPLE PP (PLUS (TIMES QQ YY) (TIMES
QQ (DIVIDES DD 2)))) SIMPLIFY?)
(TUPLE PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES DD 2))))
SIMPLIFY?
:NO
22            (FAIL)
23      (FAIL)
24      LAMBDA PROOFLEIB (LT PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES
DD 2)))))
25      (EXISTS ($F ←Y))
26        LAMBDA ASK (TUPLE (EQ (TUPLE PP (PLUS (TIMES QQ YY) (TIMES
QQ (DIVIDES DD 2)))) (TUPLE PP (PLUS (TIMES QQ YY) (TIMES QQ DD))))
PROVE?)
(EQ (TUPLE PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES DD 2)))) (TUPLE
PP (PLUS (TIMES QQ YY) (TIMES QQ DD))))
PROVE?
:NO
```

```
27        (FAIL)
28        LAMBDA ASK (TUPLE (EQ (TUPLE PP (PLUS (TIMES QQ YY) (TIMES
QQ (DIVIDES DD 2)))) (TUPLE PP (PLUS AA BB))) PROVE?)
(EQ (TUPLE PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES DD 2)))) (TUPLE
PP (PLUS AA BB)))
PROVE?
:NO
29        (FAIL)
30      LAMBDA INEQLEIB (LT PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES
DD 2))))
31      (EXISTS ($L (TUPLE ←LOWER ←UPPER)))
32        LAMBDA ASK (TUPLE PROVE (LTQ PP PP) AND (LTQ (PLUS (TIMES
QQ YY) (TIMES QQ DD)) (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES DD 2))))
?)
PROVE
(LTQ PP PP)
AND
(LTQ (PLUS (TIMES QQ YY) (TIMES QQ DD)) (PLUS (TIMES QQ YY) (TIMES
QQ (DIVIDES DD 2))))
?
```

After several false starts the system reaches using hypothesis (10), it generates two subgoals: PP ≤ PP and AA + BB ≤ QQ*YY + QQ*(DD/2). We give our approval of this tactic:

```
:NO
33        (FAIL)
34        LAMBDA ASK (TUPLE PROVE (LTQ PP PP) AND (LTQ (PLUS AA BB)
(PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES DD 2)))) ?)
PROVE
(LTQ PP PP)
AND
(LTQ (PLUS AA BB) (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES DD 2))))
?
:YES
35        ASK = TRUE
```

It proves the first subgoal immediately.

```
36      (GOAL $INEQUALITIES (AND (LTQ $X $LOWER) (LTQ $UPPER $Y)))
37        LAMBDA ANDSPLIT (AND (LTQ PP PP) (LTQ (PLUS AA BB) (PLUS
(TIMES QQ YY) (TIMES QQ (DIVIDES DD 2)))))
38        (GOAL $GOALCLASS $X)
39          LAMBDA RELCHECK (LTQ PP PP)
40          RELCHECK = TRUE
41        (GOAL $GOALCLASS (AND $$Y))
42          LAMBDA ANDSPLIT (AND (LTQ (PLUS AA BB) (PLUS (TIMES QQ
YY) (TIMES QQ (DIVIDES DD 2)))))
43          (GOAL $GOALCLASS $X)
44            LAMBDA RELCHECK (LTQ (PLUS AA BB) (PLUS (TIMES QQ YY)
(TIMES QQ (DIVIDES DD 2))))
45            LAMBDA INEQMONOTONE (LTQ (PLUS AA BB) (PLUS (TIMES QQ
YY) (TIMES QQ (DIVIDES DD 2))))
46              LAMBDA ASK (TUPLE ((LTQ AA (TIMES QQ (DIVIDES DD 2)))
(LTQ BB (TIMES QQ YY))) PROVE?)
((LTQ AA (TIMES QQ (DIVIDES DD 2))) (LTQ BB (TIMES QQ YY)))
PROVE?
:NO
```

```
47                (FAIL)
48              LAMBDA INEQMONOTONE (LTQ (PLUS AA BB) (PLUS (TIMES QQ
YY) (TIMES QQ (DIVIDES DD 2))))
49                LAMBDA ASK (TUPLE ((LTQ AA (TIMES QQ YY)) (LTQ BB (TIMES
QQ (DIVIDES DD 2)))) PROVE?)
((LTQ AA (TIMES QQ YY)) (LTQ BB (TIMES QQ (DIVIDES DD 2))))
PROVE?
```

It divides the second subgoal into two subsubgoals: AA $\leq$ QQ*YY and BB $<$ QQ*(DD/2):

```
:YES
50                ASK = TRUE
51              (GOAL $GOALCLASS (AND ($F (TUPLE $W $Y)) ($F (TUPLE $X
$Z))))
52                LAMBDA ANDSPLIT (AND (LTQ AA (TIMES QQ YY)) (LTQ BB
(TIMES QQ (DIVIDES DD 2))))
53                (GOAL $GOALCLASS $X)
54                  LAMBDA RELCHECK (LTQ AA (TIMES QQ YY))
55                  RELCHECK = TRUE
```

The first subsubgoal follows from hypothesis (2).

```
56                (GOAL $GOALCLASS (AND $$Y))
57                  LAMBDA ANDSPLIT (AND (LTQ BB (TIMES QQ (DIVIDES DD
2))))
58                  (GOAL $GOALCLASS $X)
59                    LAMBDA RELCHECK (LTQ BB (TIMES QQ (DIVIDES DD 2)))
60                    LAMBDA INEQTIMESDIVIDE (LTQ BB (TIMES QQ (DIVIDES
DD 2)))
61                    (GOAL $DEDUCE (LT 0 $Y))
62                      LAMBDA RELCHECK (LT 0 2)
63                      RELCHECK = TRUE
64                    (GOAL $INEQUALITIES ($F (TUPLE (TIMES $Y $W) (TIMES
$X $$Z))))
65                      LAMBDA RELCHECK (LTQ (TIMES 2 BB) (TIMES QQ DD))
66                      RELCHECK = TRUE
```

Checking that 2 $>$ 0, the system multiplied out the second subgoal into 2*BB $\leq$ QQ*DD. This follows from assertion (4). The proof is complete:

```
67                    INEQTIMESDIVIDE = TRUE
68                  (GOAL $GOALCLASS (AND $$Y))
69                  ANDSPLIT = (AND)
70                ANDSPLIT = (AND)
71              INEQMONOTONE = (AND)
72            ((GOAL $GOALCLASS (AND $$Y))
73            ANDSPLIT = (AND)
74          ANDSPLIT = (AND)
75        INEQLEIB = (AND)
76    (ASSERT ($F $X))
77    (RETURN ($F $X))
78    PROOFSWITCH = (LT PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES DD
2))))
79 (LT PP (PLUS (TIMES QQ YY) (TIMES QQ (DIVIDES DD 2))))
```

3. The Pattern Matcher

As an abbreviation, let

m1 = match(car(pat), car(arg))

and

m2 = match(varsubst(m1, cdr(pat)), cdr(arg)) .

The hypotheses are that

varsubst(m1, car(pat)) = car(arg) ;

or in unabbreviated form,

```
1  (ASSERT (EQ (VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CAR PAT)) (CAR
ARG)))
2  TRUE
```

and that

varsubst(m2, varsubst(m1, cdr(pat))) = cdr(arg) :

```
3  (ASSERT (EQ (VARSUBST (MATCH (VARSUBST (MATCH (CAR PAT) (CAR ARG))
(CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CDR PAT)))
(CDR ARG)))
4  TRUE
```

The other hypotheses are

```
5  (ASSERT (CONSTEXP ARG))
6  TRUE
7  (ASSERT (NOT (CONST PAT)))
8  TRUE
9  (ASSERT (NOT (ATOM ARG)))
10 TRUE
11 (ASSERT (NOT (VAR PAT)))
12 TRUE
```

The goal is to prove

varsubst(compose(m2, m1), pat) = arg :

```
13 (GOAL #PROVE (EQ (VARSUBST (COMPOSE (MATCH (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG)))
PAT) ARG))
```

The proof begins:

```
14    LAMBDA PROOFSWITCH (EQ (VARSUBST (COMPOSE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG)))
PAT) ARG)
15     (GOAL $EQRULES ($F $X))
16      LAMBDA RELCHECK (EQ (VARSUBST (COMPOSE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG)))
PAT) ARG)
17      LAMBDA EQSUBST (EQ (VARSUBST (COMPOSE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG)))
PAT) ARG)
18      (GOAL $EQSUBSTRULES $Y)
19        LAMBDA SUBSTCONS (EQ (VARSUBST (COMPOSE (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (COR PAT)) (CDR ARG)) (MATCH (CAR PAT)
(CAR ARG))) PAT) ARG)
20         (GOAL $DEDUCE (NOT (ATOM $X)))
21           LAMBDA RELCHECK (NOT (ATOM PAT))
22           LAMBDA NOTATOM (NOT (ATOM PAT))
23           (EXISTS (NOT (VAR $X)))
24           (EXISTS (NOT (CONST $X)))
25           NOTATOM = (NOT (CONST PAT))
26         (GOAL $DEDUCE (NOT (ATOM $Y)))
27         (GOAL (= ($REMOVE (TUPLE EQSUBST FROM $EQRULES))) (EQ (VARSUBST
$S1 (CAR $X)) (CAR $Y)))
```

Reasoning that pat is not an atom since it is neither a variable nor a constant, the system breaks the goal into two subgoals:

varsubst (compose(m2, m1), car(pat)) = car(arg)

and

varsubst (compose(m2, m1), cdr(pat)) = cdr(arg) .

It begins work on the first of these:

```
28           LAMBDA RELCHECK (EQ (CAR ARG) (VARSUBST (COMPOSE (MATCH
(VARSUBST (MATCH (CAR PAT) (CAR ARG)) (COR PAT)) (CDR ARG)) (MATCH
(CAR PAT) (CAR ARG))) (CAR PAT)))
29           LAMBDA EQSIMP (EQ (CAR ARG) (VARSUBST (COMPOSE (MATCH (
VARSUBST (MATCH (CAR PAT) (CAR ARG)) (COR PAT)) (CDR ARG)) (MATCH
(CAR PAT) (CAR ARG))) (CAR PAT)))
30             LAMBDA SIMPONE (VARSUBST (COMPOSE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG)))
(CAR PAT))
(VARSUBST (COMPOSE (MATCH (VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CDR
PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG))) (CAR PAT))
SIMPLIFY?
```

We give the system our permission to simplify the left side of the first subgoal,

varsubst (compose(m2, ml), car(pat)) :

```
31            (GOAL $TOPRULES $GOAL1)
32              LAMBDA HASSIMP (VARSUBST (COMPOSE (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT)
(CAR ARG))) (CAR PAT))
33              (FAIL)
34              LAMBDA SUBSTOP (VARSUBST (COMPOSE (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT)
(CAR ARG))) (CAR PAT))
35              (GOAL $SUBSTRULES $Y)
36                LAMBDA SUBSTCOMPOSE (VARSUBST (COMPOSE (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT)
(CAR ARG))) (CAR PAT))
37                (GOAL $GOALCLASS $GOAL1)
38                  LAMBDA SUBSTCONST (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
39                  (GOAL $DEDUCE (CONSTEXP $Y))
40                    LAMBDA RELCHECK (CONSTEXP (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CAR PAT)))
41                SUBSTCOMPOSE = (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
42              SUBSTOP = (VARSUBST (MATCH (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR ARG))
(CAR PAT)))
43            (RETURN $SIMPGOAL)
44            SIMPONE = (VARSUBST (MATCH (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR ARG))
(CAR PAT)))
```

The system has succeeded in simplifying the left half of the goal into

varsubst(m2, varsubst(ml, car(pat))) .

It now tries to prove this new expression equal to car(arg):

```
45            (GOAL $EQRULES (EQ $X $Y))
46              LAMBDA RELCHECK (EQ (CAR ARG) (VARSUBST (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CAR PAT))))
47              LAMBDA EQSUBST (EQ (CAR ARG) (VARSUBST (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CAR PAT))))
48              (GOAL $EQSUBSTRULES $Y)
49                LAMBDA SUBSTCONS (EQ (CAR ARG) (VARSUBST (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CAR PAT))))
50                (GOAL $DEDUCE (NOT (ATOM $X)))
```

```
51                  LAMBDA RELCHECK (NOT (ATOM (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CAR PAT))))
52                  LAMBDA NOTATOM (NOT (ATOM (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT))))
53                  (EXISTS (NOT (VAR $X)))
54              LAMBDA EQSIMP (EQ (CAR ARG) (VARSUBST (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CAR PAT))))
55                LAMBDA SIMPONE (VARSUBST (MATCH (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR
ARG)) (CAR PAT)))
(VARSUBST (MATCH (VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CDR PAT))
(CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CAR PAT)))
SIMPLIFY?
```

The system asks permission to simplify

varsubst(m1, varsubst(m2, car(pat)))

further. Permission is granted:

```
:Y
56                (GOAL $TOPRULES $GOAL1)
57                  LAMBDA HASSIMP (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
58                  (FAIL)
59                  LAMBDA SUBSTOP (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
60                  (GOAL $SUBSTRULES $Y)
61                    LAMBDA SUBSTCONST (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
62                    (GOAL $DEDUCE (CONSTEXP $Y))
63                      LAMBDA RELCHECK (CONSTEXP (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CAR PAT)))
64                  LAMBDA EQNUMB (VARSUBST (MATCH (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR
ARG)) (CAR PAT)))
65                  (FAIL)
66                (GOAL $DOWNRULES $GOAL1)
67                  LAMBDA ARGSIMP (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
68                    LAMBDA SIMPONE (TUPLE (MATCH (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR
ARG)) (CAR PAT)))
(TUPLE (MATCH (VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR
ARG)) (VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CAR PAT)))
SIMPLIFY?
:Y
69                    (GOAL $TOPRULES $GOAL1)
70                      LAMBDA HASSIMP (TUPLE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
```

```
71                         (FAIL)
72                        LAMBDA EQNUMB (TUPLE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
73                         (FAIL)
74                       (GOAL $DOWNRULES $GOAL1)
75                        LAMBDA TUPSIMP (TUPLE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
76                          LAMBDA SIMPONE (MATCH (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CDR PAT)) (CDR ARG))
(MATCH (VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG))
SIMPLIFY?
:N
77                          (FAIL)
78                        LAMBDA TUPSIMP (TUPLE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT)))
79                          LAMBDA SIMPONE (VARSUBST (MATCH (CAR PAT) (CAR
ARG)) (CAR PAT))
(VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CAR PAT))
SIMPLIFY?
```

The system wants to simplify varsubst(ml, car(pat)), a subexpression of our goal. We give our blessings:

```
:Y
80                          (GOAL $TOPRULES $GOAL1)
81                            LAMBDA HASSIMP (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT))
82                             (FAIL)
83                            LAMBDA SUBSTOP (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT))
84                             (GOAL $SUBSTRULES $Y)
85                              LAMBDA SUBSTCONST (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CAR PAT))
86                               (GOAL $DEDUCE (CONSTEXP $Y))
87                                 LAMBDA RELCHECK (CONSTEXP (CAR PAT))
88                                 LAMBDA CONSTCAR (CONSTEXP (CAR PAT))
89                                  (EXISTS (CONSTEXP $X))
90                            LAMBDA EQNUMB (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CAR PAT))
91                             (RETURN $BEST)
92                            EQNUMB = (CAR ARG)
93                          (RETURN $SIMPGOAL)
94                          SIMPONE = (CAR ARG)
```

The subexpression varsubst(ml, car(pat)) is known to be equal to car(arg) by hypothesis (1). The rule EQNUMB has found this simplification. Work continues on simplifying the entire left-hand side.

```
95                    TUPSIMP = (TUPLE (MATCH (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (CAR ARG))
96                  (GOAL $GOALCLASS $GOAL1)
97                    LAMBDA HASSIMP (TUPLE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (CAR ARG))
98                    (FAIL)
99                    LAMBDA EQNUMB (TUPLE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (CAR ARG))
100                   (FAIL)
101               (RETURN $SIMPGOAL)
102               SIMPONE = (TUPLE (MATCH (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CDR PAT)) (CDR ARG)) (CAR ARG))
103             ARGSIMP = (VARSUBST (MATCH (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (CAR ARG))
```

The expression being simplified is now varsubst(m2, car(arg)):

```
104             (GOAL $GOALCLASS $GOAL1)
105               LAMBDA HASSIMP (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (CAR ARG))
106               (FAIL)
107               LAMBDA SUBSTOP (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (CAR ARG))
108               (GOAL $SUBSTRULES $Y)
109                 LAMBDA SUBSTCONST (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (CAR ARG))
110                 (GOAL $DEDUCE (CONSTEXP $Y))
111                   LAMBDA RELCHECK (CONSTEXP (CAR ARG))
112                   LAMBDA CONSTCAR (CONSTEXP (CAR ARG))
113                   (EXISTS (CONSTEXP $X))
114                   CONSTCAR = (CONSTEXP ARG)
115                 SUBSTCONST = (CAR ARG)
116               SUBSTOP = (CAR ARG)
117             (RETURN $SIMPGOAL)
118             SIMPONE = (CAR ARG)
```

Since arg consists entirely of constants, so does car(arg). Therefore, substitutions have no effect on car(arg), and the left-hand side of our subgoal reduces to car(arg) itself, which is precisely the same as the right-hand side.

```
119           (GOAL $EQRULES (EQ $X $Y))
120             LAMBDA RELCHECK (EQ (CAR ARG))
121             RELCHECK = TRUE
122           EQSIMP = TRUE
123         EQSIMP = TRUE
```

We have yet to prove the second subgoal:

varsubst (compose(m2, ml), cdr(pat)) = cdr(arg) :

```
124        (GOAL (= ($REMOVE (TUPLE EQSUBST FROM $EQRULES))) (EQ (VARSUBST
$S1 (CDR $X)) (CDR $Y)))
125          LAMBDA RELCHECK (EQ (CDR ARG) (VARSUBST (COMPOSE (MATCH
(VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH
(CAR PAT) (CAR ARG))) (CDR PAT)))
126          LAMBDA EQSIMP (EQ (CDR ARG) (VARSUBST (COMPOSE (MATCH (
VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH
(CAR PAT) (CAR ARG))) (CDR PAT)))
127            LAMBDA SIMPONE (VARSUBST (COMPOSE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG)))
(CDR PAT))
(VARSUBST (COMPOSE (MATCH (VARSUBST (MATCH (CAR PAT) (CAR ARG)) (CDR
PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG))) (CDR PAT))
SIMPLIFY?
:Y
128             (GOAL $TOPRULES $GOAL1)
129               LAMBDA HASSIMP (VARSUBST (COMPOSE (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT)
(CAR ARG))) (CDR PAT))
130               (FAIL)
131               LAMBDA SUBSTOP (VARSUBST (COMPOSE (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT)
(CAR ARG))) (CDR PAT))
132               (GOAL $SUBSTRULES $Y)
133                 LAMBDA SUBSTCOMPOSE (VARSUBST (COMPOSE (MATCH (VARSUBST
(MATCH (CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT)
(CAR ARG))) (CDR PAT))
134                 (GOAL $GOALCLASS $GOAL1)
135                   LAMBDA SUBSTCONST (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CDR PAT)))
136                   (GOAL $DEDUCE (CONSTEXP $Y))
137                     LAMBDA RELCHECK (CONSTEXP (VARSUBST (MATCH (CAR
PAT) (CAR ARG)) (CDR PAT)))
138                 SUBSTCOMPOSE = (VARSUBST (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CDR PAT)))
139               SUBSTOP = (VARSUBST (MATCH (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR ARG))
(CDR PAT)))
140             (RETURN $SIMPGOAL)
141             SIMPONE = (VARSUBST (MATCH (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR ARG))
(CDR PAT)))
```

The subgoal has been simplified to

varsubst(m2, varsubst(ml, cdr(pat))) = cdr(arg) .

However, this is precisely our hypothesis (3).

```
142          (GOAL $EQRULES (EQ $X $Y))
143          EQSIMP = (EQ (VARSUBST (MATCH (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR ARG))
(CDR PAT))) (CDR ARG))
```

```
144      SUBSTCONS = (EQ (VARSUBST (MATCH (VARSUBST (MATCH (CAR PAT)
(CAR ARG)) (CDR PAT)) (CDR ARG)) (VARSUBST (MATCH (CAR PAT) (CAR ARG))
(CDR PAT))) (CDR ARG))
145     EQSUBST = (EQ (VARSUBST (MATCH (VARSUBST (MATCH (CAR PAT) (CAR
ARG)) (CDR PAT)) (CDR ARG)) (VARSUDST (MATCH (CAR PAT) (CAR ARG))
(CDR PAT))) (CDR ARG))
146  (ASSERT ($F $X))
147  (RETURN ($F $X))
148  PROOFSWITCH = (EQ (VARSUBST (COMPOSE (MATCH (VARSUBST (MATCH
(CAR PAT) (CAR ARG)) (CDR PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG)))
PAT) ARG)
149
   (EQ (VARSUDGT (COMPOSE (MATCH (VARSUBST (MATCH (CAR PAT) (CAR ARG))
(CDR PAT)) (CDR ARG)) (MATCH (CAR PAT) (CAR ARG))) PAT) ARG)
```

The proof is complete.

4. FIND

Only a selection from the trace for the interesting verification condition of FIND is presented here because of the length of the entire trace. We will focus on the use of the case analysis during the proof.

The antecedent hypotheses for this condition are

$1 \le M \le F \le NN$

$M \le I$

$J \le N$

(STRIP (BAGA A I M-1)) $\le$ (STRIP (BAGA A M NN))

(STRIP (BAGA A 1 N)) $\le$ (STRIP (BAGA A N+1 NN))

(STRIP (BAGA A 1 I-1)) $\le$ R

R $\le$ (STRIP (BAGA A 1+J NN))

$A[J] \le R$

$R \le A[I]$

$I \le J$

$J-1 < I+1$

$F \le J-1$.

The theorem to be proved is

```
(STRIP (BAGA (EXCHANGE A I J) 1 J-1))

  ≤ (STRIP (BAGA (EXCHANGE A I J) (J-1)+1 NN)    .
```

This goal is simplified to

```
(IF J-1 < I THEN (STRIP (BAGA A 1 J-1))
            ELSE (STRIP (BAG (STRIP (BAGA A 1 I-1))
                             A[J]
                             (STRIP (BAGA A I+1 J-1)))))

≤ (IF J ≤ I THEN (BAGA A J NN)
            ELSE (STRIP (BAG A[I]
                             (STRIP (BAGA A J+1 NN))
                             (STRIP (BAGA A J J-1)))))    :
```

```
1   (GOAL $INEQUALITIES (LTQ (IFTHENELSE (LT (SUBTRACT J 1) I) (STRIP
(BAGA A 1 (SUBTRACT J 1))) (STRIP (BAG (STRIP (BAGA A 1 (SUBTRACT
I 1))) (ACCESS A J) (STRIP (BAGA A (PLUS 1 I) (SUBTRACT J 1))))))
(IFTHENELSE (LTQ J I) (STRIP (BAGA A J NN)) (STRIP (BAG (ACCESS A
I) (STRIP (BAGA A (PLUS 1 J) NN)) (STRIP (BAGA A J (SUBTRACT J 1))))))))
2     LAMBDA RELCHECK (LTQ (IFTHENELSE (LT (SUBTRACT J 1) I) (STRIP
(BAGA A 1 (SUBTRACT J 1))) (STRIP (BAG (STRIP (BAGA A 1 (SUBTRACT
I 1))) (ACCESS A J) (STRIP (BAGA A (PLUS 1 I) (SUBTRACT J 1))))))
(IFTHENELSE (LTQ J I) (STRIP (BAGA A J NN)) (STRIP (BAG (ACCESS A
I) (STRIP (BAGA A (PLUS 1 J) NN)) (STRIP (BAGA A J (SUBTRACT J 1)))))))
3     LAMBDA INEQIFTHENELSE (LTQ (IFTHENELSE (LT (SUBTRACT J 1) I)
(STRIP (BAGA A 1 (SUBTRACT J 1))) (STRIP (BAG (STRIP (BAGA A 1 (SUBTRACT
I 1))) (ACCESS A J) (STRIP (BAGA A (PLUS 1 I) (SUBTRACT J 1))))))
(IFTHENELSE (LTQ J I) (STRIP (BAGA A J NN)) (STRIP (BAG (ACCESS A
I) (STRIP (BAGA A (PLUS 1 J) NN)) (STRIP (BAGA A J (SUBTRACT J 1)))))))
4     (ASSERT $X WRT $VERICON)
```

Since the left side of the goal has an IF-THEN-ELSE form, it causes the rule INEQIFTHENELSE to be applied. This rule sets VERICON to be a new lower context and asserts

$$J-1 < I$$

with respect to the new VERICON. This question triggers off a demon:

```
5     (ASSERT (LTQ (PLUS 1 $X) $Y) WRT $VERICON)
```

The new assertion is

$$(J-1)+1 \leq I \quad .$$

The new assertion triggers off another demon, which makes still another assertion with respect to VERICON:

```
6      (ASSERT (LTQ $W $Y) WRT $VERICON)
```

This new assertion is

$$J \leq I \quad .$$

(Later in the proof, another context will be established; J-1 < I will be <u>denied</u> with respect to the new context.)

The THEN clause of the IF-THEN-ELSE expression must now be proved less than or equal to the right side of the goal:

```
7      (GOAL $INEQUALITIES ($F (TUPLE $$W1 $Y $$W2)) WRT $VERICON)
```

This goal is attempted with respect to the new context VERICON. In other words, we are trying to prove

```
        (STRIP (BAGA A 1 J-1))
      ≤ (IF J ≤ I THEN (STRIP (BAGA A J NN))
                  ELSE (STRIP (BAG A[I]
                                    (STRIP (BAGA A J+1 NN))
                                    (STRIP (BAGA A J J-1)))))
```

with respect to the context in which J ≤ I has been asserted:

```
8       LAMBDA RELCHECK (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1))) (
IFTHENELSE (LTQ J I) (STRIP (BAGA A J NN)) (STRIP (BAG (ACCESS A I)
(STRIP (BAGA A (PLUS 1 J) NN)) (STRIP (BAGA A J (SUBTRACT J 1)))))))
9       LAMBDA INEQIFTHENELSE (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1)))
(IFTHENELSE (LTQ J I) (STRIP (BAGA A J NN)) (STRIP (BAG (ACCESS A
I) (STRIP (BAGA A (PLUS 1 J) NN)) (STRIP (BAGA A J (SUBTRACT J 1))))))))
```

Since the right side of the inequality is still in IF-THEN-ELSE form, the rule INEQIFTHENELSE applies again. A new context VERICON, even lower than the last, is established, and the (redundant) statement

$$J \le I$$

is asserted with respect to the new context:

```
10      (ASSERT $X WRT $VERICON)
```

A new goal is established with respect to the new context.

```
11      (GOAL $INEQUALITIES ($F (TUPLE $$W1 $Y $$W2)) WRT $VERICON)
```

The new goal is

(STRIP (BAGA A 1 J-1)) ≤ (STRIP (BAGA A J NN))

```
12          LAMBDA RELCHECK (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1))) (STRIP
(BAGA A J NN)))
13          LAMBDA PROOFSIMP (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1))) (STRIP
(BAGA A J NN)))
14            LAMBDA ARGSIMP (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1))) (STRIP
(BAGA A J NN)))
15               LAMBDA SIMPONE (TUPLE (STRIP (BAGA A 1 (SUBTRACT J 1)))
(STRIP (BAGA A J NN)))
```

The simplifier is invoked. We will omit some steps from the trace here and mention only that the rule BAGALOWERPLUS played an important part in the simplification of the second element of the tuple.

```
90             SIMPONE = (TUPLE (STRIP (BAGA A 1 (SUBTRACT J 1))) (STRIP
(BAG (STRIP (BAGA A (PLUS 1 J) NN)) (ACCESS A J))))
91           ARGSIMP = (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1))) (STRIP
(BAG (STRIP (BAGA A (PLUS 1 J) NN)) (ACCESS A J))))
92         (GOAL $GOALCLASS1 $X)
```

The simplified goal is

(STRIP (BAGA A 1 J-1))

≤ (STRIP (BAG (STRIP (BAGA A J+1 NN))

A[J]))   :

```
93           LAMBDA RELCHECK (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1)))
(STRIP (BAG (STRIP (BAGA A (PLUS 1 J) NN)) (ACCESS A J))))
94           LAMBDA INEQSTRIPBAG (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1)))
(STRIP (BAG (STRIP (BAGA A (PLUS 1 J) NN)) (ACCESS A J))))
```

INEQSTRIPBAG breaks up the goal into two subgoals. The first of these goals is

$$(STRIP\ (BAGA\ A\ 1\ J-1)) \le (STRIP\ (BAGA\ A\ J+1\ NN)) \quad :$$

```
95          (GOAL $INEQUALITIES ($F (TUPLE $$W $X $$Z)))
96            LAMBDA RELCHECK (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1)))
(STRIP (BAGA A (PLUS 1 J) NN)))
```

The rule INEQSTRIPSTRIP is applicable to this goal:

```
103           LAMBDA INEQSTRIPSTRIP (LTQ (STRIP (BAGA A 1 (SUBTRACT
```

Since it is known that

$$(STRIP\ (BAGA\ A\ 1\ I-1)) \le (STRIP\ (BAGA\ A\ J+1\ NN)) \quad ,$$

and, in this context, $J \le 1$, INEQSTRIPSTRIP succeeds:

The other subgoal to be proved is

$$(STRIP\ (BAGA\ A\ 1\ J-1)) \le (STRIP\ (BAG\ A[J])) \quad :$$

```
143         (GOAL $INEQUALITIES ($F (TUPLE $$W (STRIP (BAG $$Y)) $$Z)))
144           LAMBDA RELCHECK (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1)))
(STRIP (BAG (ACCESS A J))))
```

INEQSTRIPBAG applies again, splitting this goal into two subgoals, one of which is trivial.

```
145           LAMBDA INEQSTRIPBAG (LTQ (STRIP (BAGA A 1 (SUBTRACT J
1))) (STRIP (BAG (ACCESS A J))))
146           (GOAL $INEQUALITIES ($F (TUPLE $$W $X $$Z)))
```

The nontrivial goal is

$$(STRIP\ (BAGA\ A\ 1\ J-1)) \le A[J] \quad :$$

```
147             LAMBDA RELCHECK (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1)))
(ACCESS A J))
```

This goal invokes the rule INEQSTRIPTRAN. We will examine the operation of this rule in detail:

```
154             LAMBDA INEQSTRIPTRAN (LTQ (STRIP (BAGA A 1 (SUBTRACT
J 1))) (ACCESS A J))
155             (EXISTS ($F (TUPLE (STRIP (BAGA $ARNAME ←LOWER·←UPPER))
←U)))
```

The rule finds the hypothesis

$$(STRIP\ (BAGA\ A\ 1\ I-1)) \le R \quad .$$

It tests if this relation is appropriate:

```
156             (GOAL $DEDUCE (AND (LTQ $LOWER $L) (LTQ $M $UPPER)
(LTQ $D $C)))
157               LAMBDA RELCHECK (AND (LTQ 1 1) (LTQ (SUBTRACT J 1)
(SUBTRACT I 1)) (LTQ R (ACCESS A J)))
```

The system is testing whether the array segment between 1 and I-1 includes the segment between 1 and J-1, and also whether R ≤ A[J]:

```
158               LAMBDA ANDSPLIT (AND (LTQ 1 1) (LTQ (SUBTRACT J 1)
(SUBTRACT I 1)) (LTQ R (ACCESS A J)))
159               (GOAL $GOALCLASS $X)
160                 LAMBDA RELCHECK (LTQ 1 1)
161                 RELCHECK = TRUE
162               (GOAL $GOALCLASS (AND $$Y))
163                 LAMBDA RELCHECK (AND (LTQ (SUBTRACT J 1) (SUBTRACT
I 1)) (LTQ R (ACCESS A J)))
164                 LAMBDA ANDSPLIT (AND (LTQ (SUBTRACT J 1) (SUBTRACT
I 1)) (LTQ R (ACCESS A J)))
165                 (GOAL $GOALCLASS $X)
166                   LAMBDA RELCHECK (LTQ (SUBTRACT J 1) (SUBTRACT
I 1))
167                   RELCHECK = TRUE
168                 (GOAL $GOALCLASS (AND $$Y))
169                   LAMBDA RELCHECK (AND (LTQ R (ACCESS A J)))
170                   LAMBDA ANDSPLIT (AND (LTQ R (ACCESS A J)))
171                   (GOAL $GOALCLASS $X)
172                     LAMBDA RELCHECK (LTQ R (ACCESS A J))
173                     RELCHECK = TRUE
174                   (GOAL $GOALCLASS (AND $$Y))
175                   ANDSPLIT = (AND)
176                 ANDSPLIT = (AND)
177               ANDSPLIT = (AND)
```

The tests prove to be successful, and INEQSTRIPTRAN returns:

```
178             INEQSTRIPTRAN = (AND)
```

The trivial subgoal is achieved:

```
179           (GOAL $INEQUALITIES ($F (TUPLE $$W (STRIP (BAG $$Y))
$$Z)))
180             LAMBDA RELCHECK (LTQ (STRIP (BAGA A 1 (SUBTRACT J 1)))
(STRIP (BAG)))
181             RELCHECK = TRUE
```

The call to INEQSTRIPBAG from line 145 returns successfully:

```
182           INEQSTRIPBAG = TRUE
```

The call to INEQSTRIPBAG from line 94 also returns:

```
183          INEQSTRIPBAG = TRUE
```

Thus, the goal established in Line 11 has been successfully proved:

```
184       PROOFSIMP = TRUE
```

That goal was established by the rule INEQIFTHENELSE. This rule asserted J ≤ I with respect to a lower context and set up the goal with respect to that context. The rule now attempts to deny J ≤ I with respect to another context and to establish a new goal with respect to the new context.

```
185     (DENY $X WRT $VERICON)
```

However, J ≤ I was also asserted with respect to a higher context in line 6. Therefore, denying J ≤ I contradicts this assertion, causing the denial to fail. Since the situation is contracidtory and could not arise, it is unnecessary to achieve the goal, and the call to INEQIFTHENELSE from line 9 returns successfully:

```
186     (RETURN (SUCCESS (TUPLE WITH INEQIFTHENELSE)))
187     INEQIFTHENELSE = (SUCCESS (TUPLE WITH INEQIFTHENELSE))
```

The goal established in line 7 has been achieved. This goal was set up by an earlier call to INEQIFTHENELSE line 3) with respect to a context in which J-1 < I was asserted line 4). It is now necessary to set up a new goal with respect to a new context; in this new context, J-1 < I is denied:

```
188   (DENY $X WRT $VERICON)
```

This denial activates a demon that denies

J ≤ I :

```
189     LAMBDA TRYALL (TUPLE (TUPLE PLUSEMPTY PLUSSINGLE PLUSZERO
PLUSPLUS PLUSMINUS PLUSDIFFERENCE PLUSCOMBINE PLUSNUMBER) (PLUS 1
I (MINUS 1)))
190     (GOAL $GOALCLASS1 $GOAL1)
191       LAMBDA PLUSMINUS (PLUS 1 I (MINUS 1))
192       PLUSMINUS = (PLUS I)
193     (GOAL $GOALCLASS1 $GOAL1)
194       LAMBDA PLUSSINGLE (PLUS I)
195       PLUSSINGLE = I
```

```
196     (GOAL $GOALCLASS1 $GOAL1)
197     (RETURN $GOAL1)
198     TRYALL = I
199   (DENY (LTQ $W $RTSIDE) WRT $VERICON)
```

The new goal

```
(STRIP (BAG (STRIP (BAGA A 1 I-1)
            A[J]
            (STRIP (BAGA A I+1 J-1))))
≤ (IF J ≤ I THEN (STRIP (BAGA A J NN))
            ELSE (STRIP (BAG A[I]
                             (STRIP (BAGA A J+1 NN))
                             (STRIP (BAGA A J J-1))))
```

is established with respect to the new context:

```
200   (GOAL $INEQUALITIES ($F (TUPLE $$W1 $Z $$W2)) WRT $VERICON)
201      LAMBDA RELCHECK (LTQ (STRIP (BAG (STRIP (BAGA A 1 (SUBTRACT
I 1))) (ACCESS A J) (STRIP (BAGA A (PLUS 1 I) (SUBTRACT J 1)))))) (
IFTHENELSE (LTQ J I) (STRIP (BAGA A J NN)) (STRIP (BAG (ACCESS A I)
(STRIP (BAGA A (PLUS 1 J) NN)) (STRIP (BAGA A J (SUBTRACT J 1)))))))
```

INEQIFTHENELSE is invoked because the right-side of the goal is of the form IF-THEN-ELSE.

```
202      LAMBDA INEQIFTHENELSE (LTQ (STRIP (BAG (STRIP (BAGA A 1 (SUBTRACT
I 1))) (ACCESS A J) (STRIP (BAGA A (PLUS 1 I) (SUBTRACT J 1)))))) (
IFTHENELSE (LTQ J I) (STRIP (BAGA A J NN)) (STRIP (BAG (ACCESS A I)
(STRIP (BAGA A (PLUS 1 J) NN)) (STRIP (BAGA A J (SUBTRACT J 1)))))))
```

Again the rule creates two contexts: In one context J ≤ I is asserted, and in the other J ≤ I is denied. However, since J ≤ I was denied in a higher context (line 199), the assertion of J ≤ I fails; this contradictory case can safely be ignored, and attention focuses on the second context:

```
204     (DENY $X WRT $VERICON)
```

The goal is established using the ELSE clause of the previous goal:

```
   (STRIP (BAG (STRIP (BAGA A 1 I-1))
               A[J]
               (STRIP (BAGA A I+1 J-1))))

 ≤ (STRIP (BAG A[I]
               (STRIP (BAGA A J+1 NN))
               (STRIP (BAGA A J J-1))))
```

```
205     (GOAL $INEQUALITIES ($F (TUPLE $$W1 $Z $$W2)) WRT $VERICON)
206       LAMBDA RELCHECK (LTQ (STRIP (BAG (STRIP (BAGA A 1 (SUBTRACT
I 1))) (ACCESS A J) (STRIP (BAGA A (PLUS 1 I) (SUBTRACT J 1))))) (STRIP
(BAG (ACCESS A I) (STRIP (BAGA A (PLUS 1 J) NN)) (STRIP (BAGA A J
(SUBTRACT J 1))))))
207       LAMBDA INEQSTRIPBAG (LTQ (STRIP (BAG (STRIP (BAGA A 1 (SUBTRACT
I 1))) (ACCESS A J) (STRIP (BAGA A (PLUS 1 I) (SUBTRACT J 1))))) (STRIP
(BAG (ACCESS A I) (STRIP (BAGA A (PLUS 1 J) NN)) (STRIP (BAGA A J
(SUBTRACT J 1))))))
```

The proof from this point will only be summarized, since it is lengthy but uneventful. The goal is divided into nine subgoals by successive applications of INEQSTRIPBAG. Each of these goals turns out to be easily proved, and the proof ends successfully.

```
558     INEQIFTHENELSE = TRUE
559   INEQIFTHENELSE = TRUE
560
    TRUE
```

Appendix C

EXAMPLE OF HOW A VERIFICATION CONDITION IS GENERATED

Appendix C

EXAMPLE OF HOW A VERIFICATION CONDITION IS GENERATED

For those readers unfamiliar with the Floyd-Naur method of producing verification conditions, we give below an example of its application, a complete trace of how the first condition in Section III was produced.

The program under discussion is reproduced again in Figure 5.

The path under consideration begins at point C, travels around the loop through point E, and returns again to C. We will try to prove the second conjunct at C.

This statement is

$$A[0] \leq MAX,\ A[1] \leq MAX,\ \ldots,\ A[I] \leq MAX \quad . \tag{1}$$

We pass this assertion backward around the loop to point E, making the corresponding substitution. The transformed assertion is then

$$A[0] \leq A[I],\ A[1] \leq A[I],\ \ldots,\ A[I] \leq A[I] \quad , \tag{2}$$

Since LOC does not appear explicitly in (2), the assignment LOC $\leftarrow$ I has no effect.

To reach point E, the test

$$MAX < A[I]? \tag{3}$$

must have been true. Passing the assertion back before the test gives the implication

$$MAX < A[I] \supset A[0] \leq A[I],\ A[1] \leq A[I],\ \ldots,\ A[I] \leq A[I]$$

If this implication (4) is true before the test (3), the assertion (2) will be true after the test.

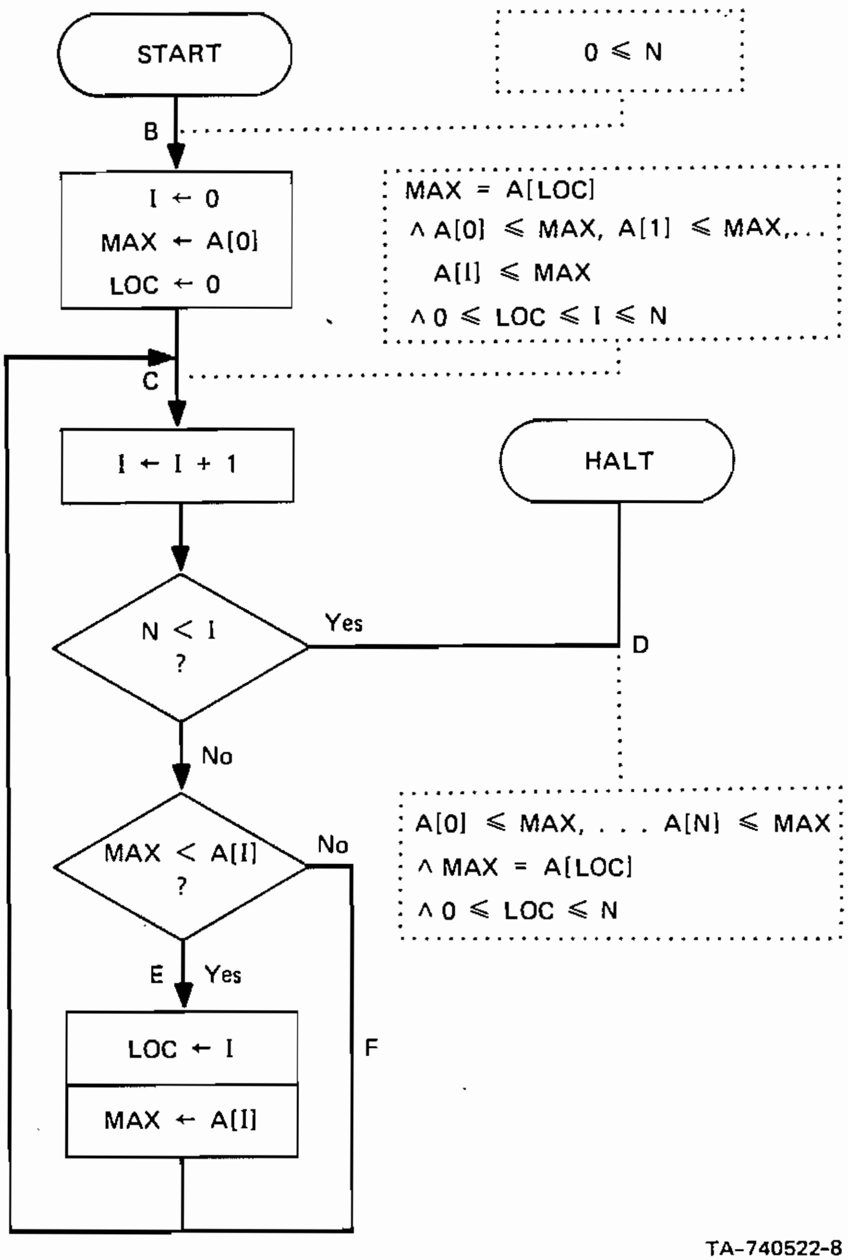


FIGURE 5 FINDING THE MAXIMUM OF AN ARRAY

To travel around the loop at all, the result of test

$$N < I? \tag{5}$$

must have been false. Passing the assertion (4) back over the test (5) gives

$$\neg(N < I) \wedge MAX < A[I]$$
$$\supset A[0] \leq A[I],\ A[1] \leq A[I],\ \ldots,\ A[I] \leq A[I] \quad . \tag{6}$$

Passing (6) back over the assignment statement

$$I \leftarrow I+1 \tag{7}$$

gives

$$\neg(N < I+1) \wedge MAX < A[I+1]$$
$$\supset A[0] \leq A[I+1],\ A[1] \leq A[I+1],\ \ldots,\ A[I+1] \leq A[I+1] \quad . \tag{8}$$

This statement has been generated in such a way that if it is true when control passes through point C, then (1) will be true if control passes around the loop through point E and returns to C. If we consider this path as a straight line program with the assertion at C as both its start assertion and its halt assertion, then proving the correctness of the second conjunct (1) at C reduces to proving

$$\begin{aligned}
&MAX = A[LOC] \wedge \\
&A[0] \leq MAX,\ \ldots,\ A[I] \leq MAX \wedge \\
&0 \leq LOC \leq I \leq N \wedge \\
&\neg(N < I+1) \wedge \\
&MAX < A[I+1] \supset \\
&A[0] \leq A[I+1],\ \ldots,\ A[I+1] \leq A[I+1] \quad .
\end{aligned}$$

Finally, the antecedents of this implication are expressed as separate hypotheses, and the consequent is represented as a goal.

REFERENCES

Boyer, R. S., and J. S. Moore, "Proving Theorems about LISP Functions," Advance Papers of the Conference, Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, California, pp. 486-493 (August 1973).

Deutsch, P., "An Interactive Program Verifier," Ph.D. Thesis, University of California, Berkeley, California (1973).

Elspas, B., M. W. Green, K. N. Levitt, and R. J. Waldinger, "Research in Interactive Program-Proving Techniques," SRI Project 8398, Phase II, Stanford Research Institute, Menlo Park, California (May 1972a).

Elspas, B., K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, Vol. 4, No. 2 (1972b).

Elspas, B., K. N. Levitt, and R. J. Waldinger, "Design of an Interactive System for Verification of Computer Programs," SRI Project 1891, Stanford Research Institute, Menlo Park, California (July 1973).

Floyd, R. W., "Assigning Meanings to Programs," Proc. American Mathematical Society Symp. in Applied Mathematics, Vol. 19, pp. 19-31 (1967).

Hewitt, C., "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts (1971).

Hoare, C.A.R., "Algorithm 65, FIND," CACM, Vol. 4, No. 7, p. 321 (1961).

__________, "An Axiomatic Basis for Computer Programming," CACM, Vol. 12, No. 10, p. 576 (1969).

__________, "Proof of a Program: FIND," CACM, Vol. 14, No. 1, p. 39 (1971).

Igarashi, S., R. London, and D. Luckham, "Automatic Verification of Programs I: A Logical Basis and Implementation" (tentative title), Stanford University, Stanford, California (to appear 1973).

Katz, S. M., and Z. Manna, "A Heuristic Approach to Program Verification," Advance Papers of the Conference, Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, California, pp. 500-512 (August 1973).

King, J. C., "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania (1969).

Knuth, D., The Art of Computer Programming, Vol. 1: Fundamental Algorithms (Addison-Wesley Publishing Company, Reading, Massachusetts, 1968).

________, The Art of Computer Programming, Vol. 2: Seminumerical Algorithms (Addison-Wesley Publishing Company, Reading, Massachusetts, 1969).

Manna, Z., and A. Pnueli, "Formalization of Properties of Functional Programs," JACM, Vol. 17, No. 3, p. 555 (1970).

McCarthy, J., et al., LISP 1.5 Programmer's Manual (MIT Press, Cambridge, Massachusetts, 1962).

McCarthy, J., and J. A. Painter, "Correctness of a Compiler for Arithmetic Expressions," Proc. American Mathematical Society Symp. in Applied Mathematics, Vol. 19, pp. 33-41 (1967).

Naur, P., "Proof of Algorithms by General Snapshots," BIT, Vol. 6, pp. 310-316 (1966).

Reboh, R., and E. Sacerdoti, "Preliminary QLISP Manual," Technical Note 81, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California (August 1973).

Robinson, J. A., "A Machine Oriented Logic Based on the Resolution Principle," JACM, Vol. 12, No. 1, pp. 23-41 (1965).

Rulifson, J. F., J. A. Derksen, and R. J. Waldinger, "QA4: A Procedural Calculus for Intuitive Reasoning," Technical Note 73, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California (1972).

Teitelman, W., et al., "BBN-LISP TENEX Reference Manual," Bolt, Beranek, and Newman, Incorporated, Cambridge, Massachusetts (1971).

von Newmann, J., H. H. Goldstine, "Planning and Coding Problems for an Electronic Computer Instrument, part 2, vol. 1-3," John von Neumann Collected Works, Vol. 5, pp. 80-235 (Pergamon Press, New York, 1963).

Wegbreit, B., "Heuristic Methods for Mechanically Deriving Inductive Assertions," Advance Papers of the Conference, Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, California, pp. 524-536 (August 1973).

Wensley, J. H., "A Class of Non-Analytical Iterative Processes," Computer Journal, Vol. 1, pp. 163-167 (1958).