AN INTERACTIVE PROGRAM VERIFIER

BY L. PETER DEUTSCH

XEROX
PALO ALTO RESEARCH CENTER

AN INTERACTIVE PROGRAM VERIFIER

BY L. PETER DEUTSCH

CSL-73-1 MAY 1973 REPRINTED JULY 1976

Program verification refers to the idea that the intent or effect of a program can be stated in a precise way that is not a simple "rewording" of the program itself, and that one can prove (in the mathematical sense) that a program actually conforms to a given statement of intent. This thesis describes a software system which can verify (prove) some non-trivial programs automatically.

The system described here is organized in a novel manner compared to most other theorem-proving systems. It has a great deal of specific knowledge about integers and arrays of integers, yet it is not "special-purpose", since this knowledge is represented in procedures which are separate from the underlying structure of the system. It also incorporates some knowledge, gained by the author from both experiment and introspection, about how programs are often constructed, and uses this knowledge to guide the proof process. It uses its knowledge, plus contextual information from the program being verified, to simplify the theorems dramatically as they are being constructed, rather than relying on a super-powerful proof procedure. The system also provides for interactive editing of programs and assertions, and for detailed human control of the proof process when the system cannot produce a proof (or counter-example) on its own.

KEY WORDS
program verification, theorem proving, interactive systems, context trees

CR CATEGORIES
4.19, 4.22, 5.21, 5.24

XEROX
PALO ALTO RESEARCH CENTER
3333 COYOTE HILL ROAD / PALO ALTO / CALIFORNIA 94304

TABLE OF CONTENTS

PRELIMINARIES

Introduction

This thesis is about *automatic program verification. Program verification* refers to the idea that one can state the intended effect of a program in a precise way that is not merely another program, and then prove rigorously that the program does (or does not) conform to this specification. *Automatic* refers to the hope that if we can understand enough about the structure of programs and the structure of proofs, we can build systems that perform some or all of this verification task for us.

The computing community has two good reasons for being interested in automatic verification. The first is practical. Most large software systems never work reliably, and a study suggests [Bell] that the initial quality of such systems has a surprisingly large influence over their reliability during their entire lifetime. For such systems, it is clear that automated aids to verification will be required, since hand-verification would be impossibly tedious. In the short run, the "structured programming" approach of Dijkstra [Dijk1], which attempts to codify and enforce good programming practices such as limiting interactions between program modules, is likely to prove more valuable than verification: this approach is already being adopted [Mills1] in an effort to hold down software costs. However, such an approach can only reduce the likelihood of errors, and does not offer the kind of security attainable through verification. Another idea advocated by a few, that of "graceful recovery" from software errors [Bas1], seems like a serious misapplication of an idea which works very well for hardware errors.

Verification is also important to computing because of its place in the historical trend which has constantly brought programming languages closer to the problem domain of the end user. This process has required transferring more and more of what were originally thought of as "programming" tasks to the machine, and this in turn has required increasingly deep understanding of the process of programming. The author believes that his automatic verification system succeeds as well as it does because it represents a successful encoding of the structure of programs. To this extent, it constitutes a contribution to the emerging area of *automatic programming*, a term which refers to a group of ambitious attempts to throw much more of the detailed design of algorithms and data structures onto the machine than was previously thought possible. A recent survey by Balzer [Bal1] gives a good idea of the scope of these attempts.

The author has built and checked out an automatic verification system called PIVOT (Programmer's Interactive Verification and Organizational Tool). There has been one other published automatic verification system, that of James C. King, reported in his Ph.D. thesis (issued in 1970). PIVOT owes an enormous amount to King's system in certain specific areas, notably representation and simplification of arithmetic expressions, although the organization of PIVOT is quite different from that of King's Verifier. There will be numerous references and comparisons to King's work strewn throughout this thesis. However, there have been considerable advances in artificial intelligence in the past three years, and the present work has made use of some of them, particularly the work of the QA4 group at Stanford Research Institute (Derksen, Rulifson, Waldinger, et al.) [Ru11] to which there will also be frequent references.

Perhaps the most interesting result of the author's experiences over the year and a half spent developing PIVOT is the observation that program verification seems to be a task significantly different from that of proving mathematically interesting theorems. The verification task involves verifying the consistency of a network of state descriptions: the verification of each individual arc of this network, which corresponds to a partial path through the program, is as much like executing the program in an abstract algebraic domain as it is like proving theorems about what the program is doing. This observation has been made independently by other researchers [Burl] [Robl]. The author was not committed to this view when he began; it arose progressively from his experience with his system. The sections "Program overview" and "Theorem generator" in the next chapter will describe the structure of the system in more detail in this regard.

The organization of this thesis roughly follows the structure of the actual program. Chapter I is introductory. Chapter II describes the language used to input programs. Chapter III, which covers the organization of the program, and Chapter IV, which describes the mechanisms used to perform ongoing deductions, case analysis, and global simplification contain the principal original material. Chapter V describes the internal representation of expressions and programs. Chapter VI describes the appearance of the system to the user. Chapter VII is an attempt to assess the author's work in the larger context of research in artificial intelligence and general computer science.

Many theses represent the endpoint of a piece of research. This one does not: it represents a stage in a long-term research plan at which the author felt it was appropriate to pause to communicate his work. The instantaneous state of such a research project is always a mixture of implemented procedures, well-understood but unprogrammed solutions to problems, and more or less clear personal understandings of problems which have not yet reached the stage of algorithmic solution. Consequently, there are numerous statements in this thesis of the form "X is understood but not implemented yet". Large programs such as PIVOT also do not remain unchanged from one month to the next; the reader may forgive possible minor inconsistencies arising from changes in PIVOT during the time this thesis was being written.

Notation

Every mathematical or computer-oriented paper prepared with computer assistance has the problem that the available character set is deficient in some crucial area. In the first printing of this thesis, it was necessary to substitute the following symbols:

Intended	**Appeared as**	**Meaning**
$\wedge$	&	Conjunction, logical AND
$\vee$	!	Disjunction, logical OR
$\supset$	=>	Implication
$\leftarrow$	:=	Assignment
A_i	A{i}	Subscripting

In the current printing, however, the desired symbols are available and have been used where appropriate. The universal and existential quantification symbols, $\forall$ and $\exists$, have also replaced the overstrike combinations used in the first printing.

Every paper of this sort also has a quotation problem, i.e., some device must be chosen to distinguish between $\underline{x}$ used to mean the actual symbol "x" and $\underline{x}$ used to stand for any of a class of symbols, expressions, etc. The device adopted here is to use upper case for the former and lower case for the latter. For example, (fn X) is a list whose first element is any function and whose second element is the actual symbol "X".

The syntax equations in this thesis are written in a dialect of the Backus-Naur Form, BNF. The example immediately below illustrates all features of this notation.

```
<sum> ::= <term> ${+ <term> | - <term>}
<term> ::= [-] <primary>
<primary> ::= <id> | <num> | ( <sum> )
```

Dollar sign ($) means "0 or more of the following". Braces {} are used as meta-parentheses for grouping. Square brackets mean "optionally" or "0 or 1 of". Quotation marks are used to delimit literals containing the meta-syntactic characters: "$", "|", "{", "}", "[", and "]". There are four special defined entities: <id> (identifier), <num> (number), <str> (string), and <punc> (punctuation mark). A <str> is any string of characters surrounded by quotation marks. A <punc> is any printing character on the keyboard other than letters, digits, or quotation mark. The other two are defined below:

```
<id> ::= <letter> ${<letter> | <digit>}
<num> ::= <digit> $<digit>
<letter> ::= A | B | C | D | E | F | G | H | I | J
           | K | L | M | N | O | P | Q | R | S | T
           | U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Theoretical background

The present work is based on the Floyd theory of program verification. This theory has been discussed and explained at great length elsewhere, in particular in King's thesis, so the sketch here will be brief. The idea is that we take programs represented as flow charts and attach predicates to their edges. These predicates are in a formal language (typically a first-order predicate calculus in which the function and predicate symbols are interpreted). The free variables of the predicates refer to the values of the program variables, so that each predicate can be thought of as describing a set of machine states; the predicate on a given edge specifies a state set which includes the set of possible states when control reaches that edge. We can represent these predicates as applied to a single variable S, the entire machine state, rather than the individual variables. Now if we have predicates P(S) and Q(S) separated by a flowchart box whose effect can be described as S $\leftarrow$ f(S), then we say that part of the program is "partially correct" if we can prove the theorem

$$(\forall S)(P(S) \supset Q(S,f(S))).$$

Similarly, if the effect of the box is to test r(S), the theorem is

$$(\forall S)(P(S) \wedge r(S) \supset Q(S)).$$

We say the program as a whole is "partially correct" if we can prove all these theorems. (This requires choosing predicates for the starting and stopping points of the program as well as all the internal edges, of course.) A similar method can be used to show that the program actually reaches its stopping point if the initial values of the variables satisfy the starting predicate: if the program is partially correct and also stops for all legal initial values, we say the program is "correct".

While this approach has an appealing simplicity, it suffers from a number of difficulties in practical use. One is that the semantics of the programming language must be fully understood. Recent work by Hoare, attempting to define precisely certain common program constructs such as the ALGOL 60 procedure call [Hoare1], reveal just how difficult this task is. The author feels that this difficulty will eventually be overcome by the development of "cleaner" programming languages, and has not attempted to produce a system which accepts any real language. The input language for PIVOT is a simple statement-oriented language reminiscent of ALGOL W [Wirth1]: it has arrays and records, WHILE loops but no GOTO, compound statements but not block structure and only call-by-value for procedures. The semantics of pointers and structured data follow what Morris [Mor1] calls the "pointer-swinging" approach. The complexities of numerical analysis, a good deal of which is concerned with correctness questions of a different sort, have been avoided by restricting scalar variables to unbounded integers.

A second drawback of the Floyd approach is that for it to be reasonably amenable to machine manipulation, the author of the program must supply enough predicates to cut every loop in the flowchart, although for human proof this can sometimes be avoided if the inner loops are simple enough. It turns out that generation of the proper predicates is a substantial intellectual task. Knuth suggests that a programmer does not fully understand his program if he cannot supply these predicates, and there is some merit to this view, but it presents a substantial barrier to acceptance of verification as a tool for producing correct programs. PIVOT does not attack this problem directly, although the author has done a small amount of work not reported in this thesis on generating some of the simpler predicates automatically, especially those required to prove halting, which are usually much simpler. Some very recent work by Wegbreit [Wegl] shows great promise in this area.

A third difficulty with the Floyd assertion method is that each predicate must specify *everything* that needs to be known about the state to show correctness of paths originating at that predicate; this tends to require, for example, that the input specifications be repeated in every predicate even if the input variables never change. This problem arises from the fact that each theorem must in principle be provable independently of every other. PIVOT attacks this problem through a notational convenience which allows the programmer to specify predicates which must remain true over sections of the program rather than being true at a single control point. (Interestingly, a similar idea was suggested by Peter Naur in one of the very earliest papers on verification [Naurl].) PIVOT also automatically propagates through the program any assertions referring only to variables known to be invariant.

In addition to these practical difficulties with the Floyd method, the verification problem using this method is theoretically only semi-decidable if the predicates are expressed in a first-order calculus, which seems to be the weakest suitable language. Even if the program deals exclusively with integers, recent results show that certain simple problems are unsolvable [Davl] or only solvable in exponentially long time [Rabl]. However, these theoretical limitations have not caused the author any practical problems: the theorems that arise from the programs he has investigated are all quite simple, and he feels that this is true of programs in general, provided that their operation does not depend on subtle mathematical theorems (which relatively few do).

Quite apart from intrinsic difficulties with the method, there are implementation problems in organizing the theorem-proving process even for the situation where one is fairly sure a proof exists. Like King's system, PIVOT implicitly adopts a pragmatic attitude: when they exhaust a fixed set of theorem-proving techniques, they give up. However, since one of PIVOT's techniques is a restricted form of resolution [Robla], the computation may proceed for an unreasonably long time and perhaps even fail to terminate. PIVOT can detect that a theorem has a counter-example in certain cases, and is currently being modified to exhibit it to the user. Interactive direction of the proof procedure by the user, at least to the extent of being able to define and force application of simplification or deduction rules, is not difficult to implement: a number of other research groups are following this route [Luck1] [Mil1] [Guard1] and such a facility has recently been added to PIVOT. Interactive assistance also seems like a promising solution to the inability of most known proof systems (including PIVOT) to generate inductive proofs automatically, although some efforts [Weg1] [Buc1] [Wal1] [Boy1] are being directed towards automatic recognition of some common types of induction.

INPUT LANGUAGE

Program structure

PIVOT uses an algebraic, statement-oriented language for the programs to be analyzed. The author rejected the obvious alternative, LISP, on the grounds that it was not particularly suitable for transformation into flowchart form, that the published programs chosen as benchmarks were written in ALGOL already, and that using a parser which provided the ability to define the syntax of the language would allow experimentation with the idea that the language structure could affect ease of verification.

PIVOT programs are organized in terms of *modules.* A *procedure module,* or simply *procedure,* consists of a *header* giving the procedure name, declarations for the input and output variables and structured local variables, and a *body* containing assertions and executable statements. (Procedure modules are like ALGOL procedures, except that textual nesting is not allowed and all parameters are transmitted by value.) A *declaration module* has a header and declarations only. Declaration modules are similar to macro-instructions: the INSTANCE statement inserts a copy of a declaration module in-line. Declaration modules are not especially useful in PIVOT at present: their existence is motivated by the "structured programming" principle that definitions which are required in many places should only be written in a single place.

Within each module statements are numbered. The numbers are strictly for editing and sequencing purposes. Intra-program references are not by number, as in BASIC for example, but by label. The only semantic significance of statement numbers is that numbers below 100 are reserved for declarations and numbers above are reserved for non-declarations.

The semantics of variables are especially simple. All variables are local to the procedure module in which they appear. Undeclared variables are assumed to be local scalars. This requires that all information accessible to a procedure be passed to it by the caller. Of course, this attitude is not satisfactory for a real programming system; it was adopted in order to defer questions relating to global side-effects until a later stage of the research.

```
<procedure-head> ::= PROCEDURE <id> <formal-parameters>

<declaration-head>::=DECLARATIONS <id> <formal-parameters>

<declaration> ::= DECLARE <declaration-mode> <name-list> |
                  INSTANCE <id> <optional-parameters> |
                  LET <id> <formal-parameters> =
                       <general-expression>

<declaration-mode> ::= ARRAY [ ( <number-of-
                           dimensions> ) ] |
                       RECORD <formal-parameters> |
                       SCALAR |
                       <record-name>

<number-of-dimensions> ::= <num>

<record-name> ::= <id>

<name-list> ::= <id> ${, <id>}

<formal-parameters> ::= [( <name-list> )]

<optional-parameters> ::= [( <parameters> )]

<parameters> ::= [<expression> ${, <expression>}]
```

Fig. II-1

```
<statement> ::= [: <label> :] <statement-body>

<statement-body> ::= ASSERT <q-relation-list> |
                     LEMMA <q-relation-list> |
                     ASSUME <q-relation-list> |
                     IF <rel-expression> THEN
                         {BEGIN | <simple-list>} |
                     BEGIN |
                     LOOP |
                     WHILE <rel-expression> :
                         {BEGIN | <simple-list>} |
                     REPEAT : BEGIN |
                     ELSE {BEGIN |
                          <simple-list> |
                          IF <rel-expression> THEN
                               {BEGIN | <simple-
                                         list>}} |
                     END |
                     DECLARE <q-relation-list>' |
                     CANCEL <label> |
                     "..." |
                     <simple-list>

<simple-list> ::= <assignment> ${; <assignment>}
                 [; <branch>]
                 | <branch>

<assignment> ::= <left-hand-side> {← | :=} <expression>

<branch> ::= EXIT <label> |
             NEXT <label>

<label> ::= <id>
```

Fig. II-2

```
<q-relation-list> ::= <q-relation> ${, <q-relation>}

<q-relation> ::= <general-expression>

<rel-expression> ::= <general-expression>

<expression> ::= <general-expression>

<general-expression> ::= IF <rel-expression>
                            THEN <general-expression>
                            ELSE <general-expression>
                         | <non-q-relation>

<non-q-relation> ::= <disjunction> [IMP <disjunction>]

<disjunction> ::= <conjunction> ${<or-symbol> <conjunction>}

<or-symbol> ::= OR | !

<conjunction> ::= <relation> ${<and-symbol> <relation>}

<and-symbol> ::= AND | &

<relation> ::= NOT <relation> |
               .FA <quantifier-tail> |
               .EX <quantifier-tail> |
               <arith-expression> ${<rel-symbol>
                                    <arith-expression>}

<rel-symbol> ::= = | # | < [=] | > [=]

<quantifier-tail> ::= ( <general-expression> ${,
                         <general-expression>}
                         [: <q-relation-list>] )
                         <relation>

<arith-expression> ::= <term> ${+ <term> | - <term>}

<term> ::= <factor> ${* <factor> | / <factor> | MOD <factor>}

<factor> ::= <negative> ${<exponentiation-symbol> <negative>}

<exponentiation-symbol> ::= * * | ↑

<negative> ::= [-] <primary>

<primary> ::= CREATE <record-type> |
              <left-hand-side> [( <parameters> )] |
              <num> |
              <str> |
              ( <general-expression> )

<record-type> ::= <id>

<left-hand-side> ::= [$] <id> {"[" <expression> ${,
                                 <expression>} "]" |
                                 ${. <field-name>}}

<field-name> ::= <id>
```

Fig. II-3

Declarations

Figure II-1 gives the syntax of declarations. A variable (name) may be declared as one of four types: an array of a given number of dimensions, a scalar (unbounded integer), a pointer to a record of a previously defined type, or a record type. Record types carry a list of field names (formal parameters): the field selection operator takes a record variable on the left and a field name appropriate to that record on the right. Neither field names nor array names are values, i.e., they cannot be computed, so the relationship between record pointers and field names is essentially the same as between array indices and array names. These simple semantics allow expression of many programs that manipulate structured data, but avoid the more difficult questions concerning design of data structuring facilities in programming languages, such as the proper treatment of coercions and unions.

To cover the need for abbreviation, the LET statement provides a macro-definition capability at the expression level. For example, one could define an abbreviation for the maximum of two quantities as

```
LET MAX(X,Y) = IF X>Y THEN X ELSE Y.
```

The substitution is macro-like, but preserves the intended scope of operators by effectively surrounding the actual parameters and the expanded definition with parentheses when required: for example, MAX(X,Y)+1 effectively expands to (IF X>Y THEN X ELSE Y)+1 rather than IF X>Y THEN X ELSE Y+1. Note that since the expression on the right is a general-expression, LET can be used to define predicates or assertions, i.e., it is not restricted to arithmetic contexts.

INSTANCE calls for insertion of a copy of a declaration module with actual parameters substituted for formal. As with LET, the substitution is macro-like. In particular, variables in the declaration module which are not formal parameters of the module are treated like any other variables of the procedure in which the INSTANCE statement appears, since the statements of the declaration module are literally copied in-line to effectively replace the INSTANCE statement. The intended use of declaration modules and INSTANCE is to provide a library of parameterized assertions. For example, suppose one wants to have the idea of an array being sorted readily available. The following declaration module would serve:

```
DECLARATIONS SORTED(A,N)
100 ASSERT .FA(J:1<=J<N)A[J]<=A[J+1]
```

Then to assert that a particular array A1 of size N1 was sorted one could write

```
250 INSTANCE SORTED(A1,N1).
```

As it happens, in this case one could achieve the same effect by the declaration

```
1 LET SORTED(A,N) = .FA(J:1<=J<N)A[J]<=A[J+1]
```

and a subsequent call

```
250 ASSERT SORTED(A1,N1).
```

The distinction between LET definitions, which only generate a single expression and are local to one procedure, and declaration modules, which generate one or more statements and are accessible to all procedures, is somewhat arbitrary; both constructs will probably be superseded by a more general scope structure and a more general definition facility in some future version of PIVOT.

Statements

The syntax of statements other than declarations appears in Figure II-2. Groups of ASSERT statements mark points in the program with two properties: the asserted relations are to be *assumed* for all control paths leading away from the point, and the relations are to be *proved* necessarily true for all control paths leading to the point. DECLARE statements are like ASSERTs except that they are implicitly copied as part of every interior group of ASSERTs. (This provides the notational convenience mentioned in the introductory section on theoretical background.) For cases where it is temporarily necessary to override a DECLARE, the CANCEL statement appearing in an ASSERT group cancels an enclosing DECLARE statement with the specified label. The LEMMA statement allows the user to provide "hints" for PIVOT: a LEMMA must be proved for all incoming paths and is assumed for all outgoing paths, but does not itself delimit paths. The ASSUME statement allows the user to specify truths (axioms) to be accepted on faith.

The semantics of the control statements are simple. For tests, there are IF statements and ELSE statements (the THEN clause is part of the IF). Loops are a little less conventional. The LOOP statement signals the beginning of a loop, but the WHILE statement (or REPEAT, which means "repeat forever") actually begins the body. This arrangement allows insertion of assertions between the LOOP and the WHILE, which are checked both before the first entry into the loop and after each iteration. (Earlier versions of PIVOT did not provide this facility, which made it necessary to write two copies of the assertions if these semantics were desired.) For abnormal exit from a loop, the EXIT statement looks for an enclosing statement with the specified label and branches to the statement following the end of the loop body, if a LOOP statement, or the END, if a BEGIN. The NEXT statement searches for an enclosing IF or ELSE IF and effectively revokes the decision, forcing the succeeding ELSE to be executed. EXIT and NEXT are present in some form in several modern programming languages and seemed conceptually cleaner than GOTO: for one thing, they make the problem of detecting unreachable statements trivial. Doing all control in terms of BEGIN-END pairs also makes possible attractively indented listings of procedures which indicate the logical structure. Early versions of PIVOT used GOTO, which was not much harder to analyze for purposes of path generation but which produced less readable programs.

This leaves assignments and the "..." statement. Assignments to simple variables, array elements, or structure components are allowed. Assignment to a variable declared as a record pointer just changes the pointer: this is like LISP and most other languages that recognize the existence of pointers. "..." is used to indicate a gap in the program. Its only effect is to prevent verification from being attempted on any paths that pass through it. In the future, "..." will prompt PIVOT to analyze the surrounding program for qualities which the missing code must possess, such as requirements that certain variables be changed, or even the requirement that a certain assignment be done. This statement was added to PIVOT very recently, as an implementation of the idea suggested in Appendix C.

Expressions

The syntax of expressions appears in figure II-3. Most of the syntax is quite conventional and transparent. The distinction between the three kinds of expressions is a semantic one: <q-relation>s may contain quantifiers and must be predicates, <rel-expression>s must not contain quantifiers and must be predicates, and <expression>s must not contain quantifiers and must be arithmetic. Thus, for example, quantifiers are only allowed in ASSERT, LEMMA, and DECLARE statements. (There is a school of thought which says that quantifiers should be considered as abbreviations for loops and should therefore always be restricted in ways which make them potentially executable [Weg2]. While the present author has some sympathy for this view, he considers it too restrictive.) Quantifiers have the general form

```
        . {FA | EX} ( <variables> : <range> ) <body>
```

where the range and body are any predicates. As variations, the range may be omitted, or the variables may be omitted and marked with a preceding $ at any occurrence in the range, or a combination of the two:

```
                .FA(1<=$J<N)A[J]=0
```

is equivalent to

```
                .FA(J:1<=J<N)A[J]=0
```

or to

```
                .FA(J)(1<=J AND J<N IMP A[J]=0).
```

The CREATE operator creates a new instance of the given record and returns a pointer to it. The only other point worth noting is the syntax of <left-hand-side>. The initial $ is for quantifiers, as just mentioned. Brackets are used for array indexing and "." for field selection. As remarked above, the semantics disallow computing an array name or a field name, so the syntax does not allow it either.

PROCESSING STRUCTURE

Program overview

As Figure III-1 shows, PIVOT's processing of a program occurs in stages. When the program is input, either by being typed or from file storage, it is parsed strictly statement-by-statement and stored. When further processing is required, each statement is individually converted to a canonical form. All internal computations are in this form. (The specification of the canonical form appears as Chapter V.) The next step is a scan over the program to collect declarations and to match up BEGIN-END pairs. Then another scan is made to determine all control paths through the program: the endpoints of these paths are (groups of) ASSERT statements, and there must be no loops lacking such statements. Figure III-2 shows this process in more detail for a simple example.

In further processing, each path is treated separately. PIVOT works forward along the path, building up data bases which record the assertions at the beginning of the path and the outcomes of IF and WHILE tests along the path (the "clause data bases") and the assignments made to variables (the "value data bases"). Assigned values are represented in terms of the values possessed by variables at the beginning of the path: after the assignments

```
X ← Y+Z
Y ← Z+1
```

the value of X would be recorded as "Y+Z" and of Y as "Z+1". When the ASSERT statements at the end of the path are reached, the assigned values are substituted for variable names and the negations of the result are added to the relation data bases, in anticipation of the theorem proving process (whose aim is to produce a contradiction). King's system works backward along the path, making a substitution into the goal assertion at every assignment; the author's method substitutes previously assigned values into each assignment, but only substitutes into the (generally much larger) goal assertion once, at the end of the path, and hence is more efficient. (The difference between the two methods is analogous to evaluating lambda-expressions using an A-list [McCar3] instead of by substituting values for bound variables.) The author was also led to choose the forward method for PIVOT because he felt more comfortable with it and because it is better adapted to the author's intuitively motivated treatment of arrays.

The final theorem proving stage works on each theorem separately. One path may produce many theorems, because there may be several ASSERTs at the end of it or because several cases may have been created to deal with array assignments (the latter possibility is discussed in detail later in this chapter). Each theorem is stored as a set of clauses, and the theorem prover just iterates over these clauses with a set of transformations until a contradiction appears or no further transformations are applicable.

The major departure from King's system arises in the theorem generating process. Whereas King's system largely ignores interactions between the various assertions and assignments during its theorem generation phase, PIVOT builds up a set of clauses which are known to be "true" as it moves along a path and can use them to simplify expressions encountered later. For example, when PIVOT passes through a test like WHILE I<N on a path that assumes that it succeeds, PIVOT enters the clause I<N into a data base. Then, for example, if an assignment is made to A[N] and a later reference is made to A[I], the information that the latter cannot refer to the same element as the former is instantly available. In fact, this kind of on-the-fly deduction prevents a sizable number of theorems from reaching the theorem prover at all, since they are deduced to be "true" as they are generated. Chapter IV discusses this process (which is the heart of the author's original work) in detail, with particular reference to the mechanism used to make it inexpensive.

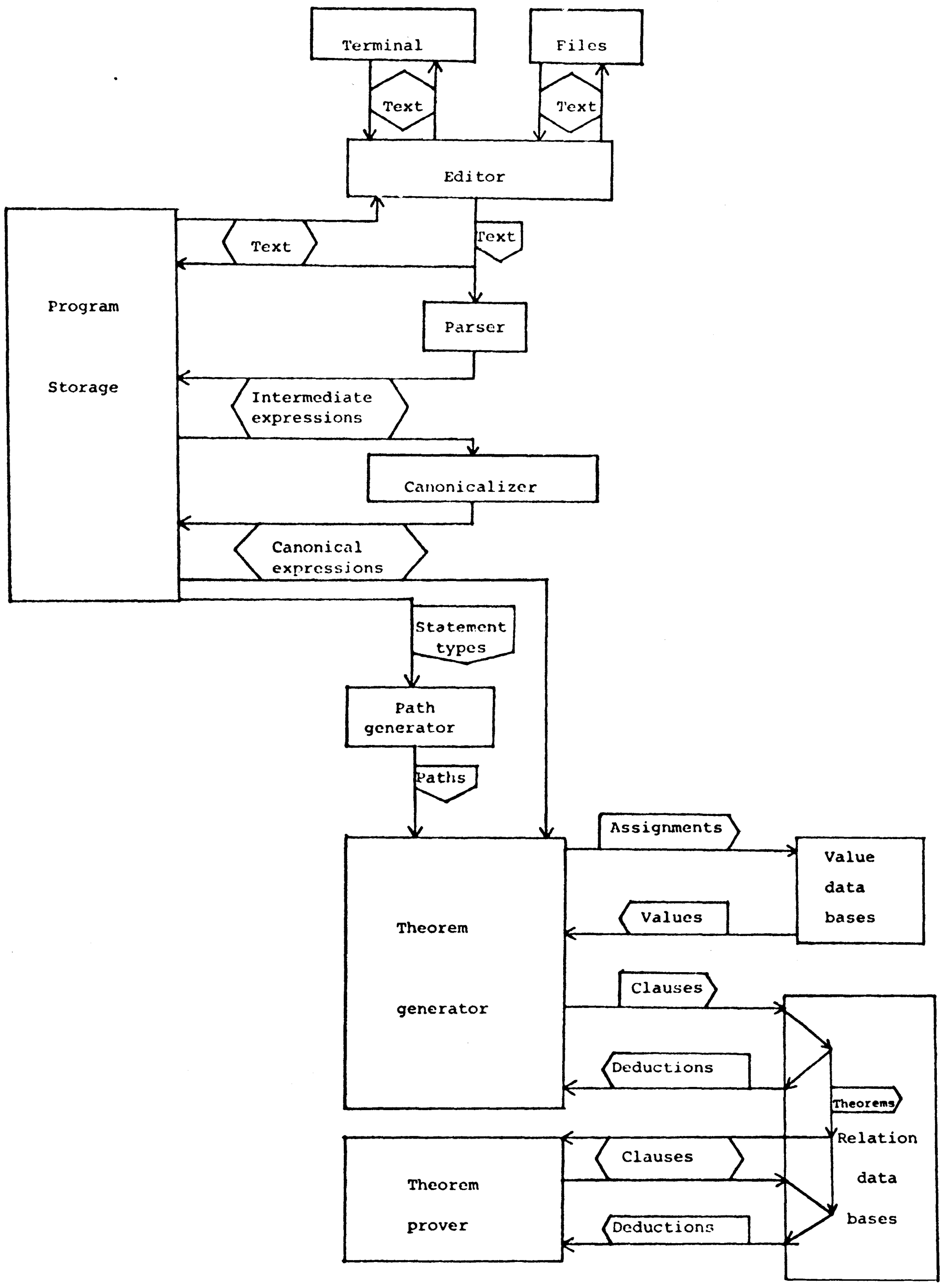
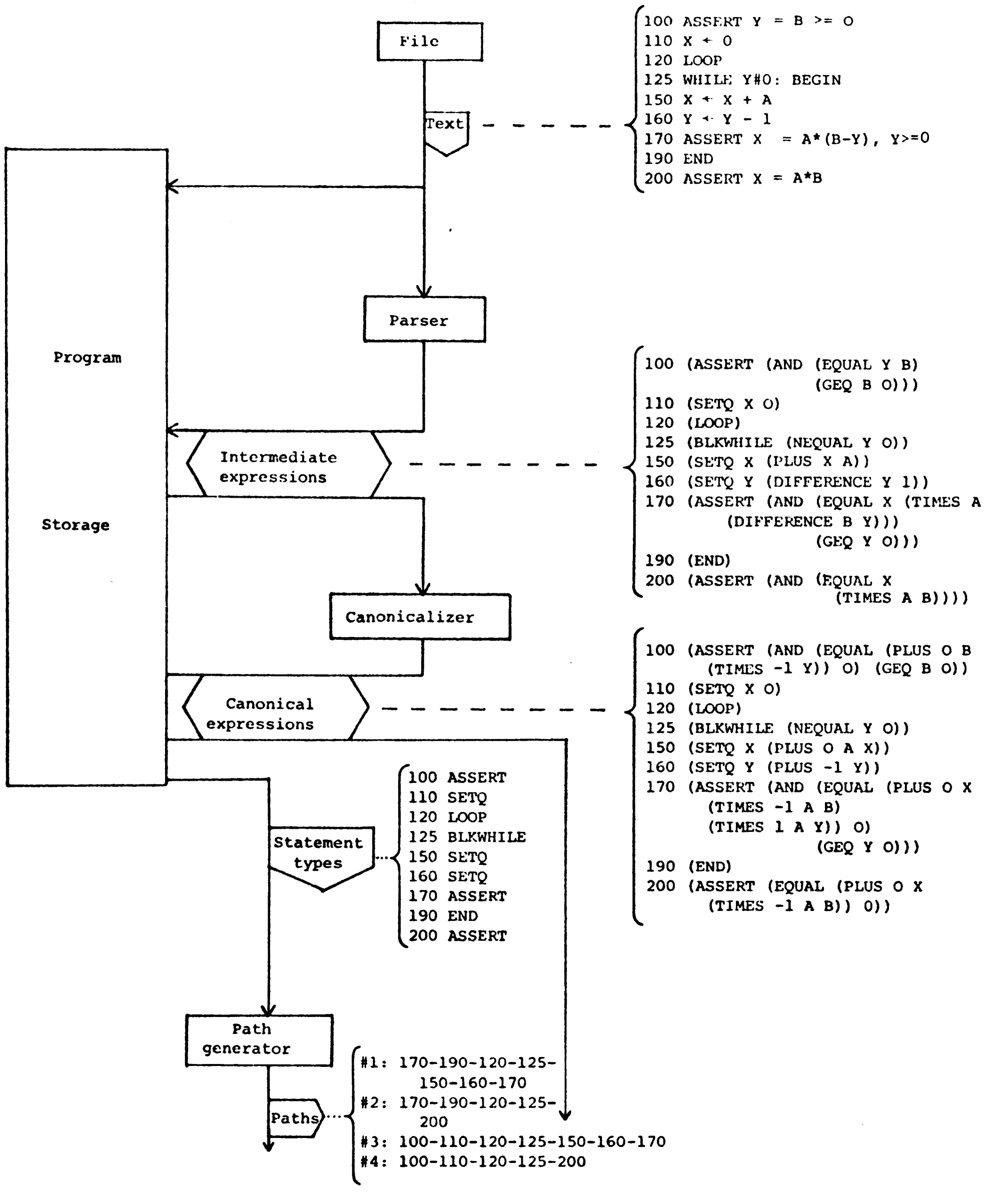

Terminal
Files
Text
Text
Editor
Text
Text
Program
Storage
Parser
Intermediate expressions
Canonicalizer
Canonical expressions
Statement types
Path generator
Paths
Assignments
Value data bases
Values
Theorem generator
Clauses
Deductions
Theorems
Relation data bases
Theorem prover
Clauses
Deductions

Fig. III-1



```
100 ASSERT Y = B >= 0
110 X ← 0
120 LOOP
125 WHILE Y#0: BEGIN
150 X ← X + A
160 Y ← Y - 1
170 ASSERT X  = A*(B-Y), Y>=0
190 END
200 ASSERT X = A*B
```

```
100 (ASSERT (AND (EQUAL Y B)
                 (GEQ B 0)))
110 (SETQ X 0)
120 (LOOP)
125 (BLKWHILE (NEQUAL Y 0))
150 (SETQ X (PLUS X A))
160 (SETQ Y (DIFFERENCE Y 1))
170 (ASSERT (AND (EQUAL X (TIMES A
          (DIFFERENCE B Y)))
                 (GEQ Y 0)))
190 (END)
200 (ASSERT (AND (EQUAL X
                   (TIMES A B))))
```

```
100 (ASSERT (AND (EQUAL (PLUS 0 B
       (TIMES -1 Y)) 0) (GEQ B 0))
110 (SETQ X 0)
120 (LOOP)
125 (BLKWHILE (NEQUAL Y 0))
150 (SETQ X (PLUS 0 A X))
160 (SETQ Y (PLUS -1 Y))
170 (ASSERT (AND (EQUAL (PLUS 0 X
       (TIMES -1 A B)
       (TIMES 1 A Y)) 0)
                 (GEQ Y 0)))
190 (END)
200 (ASSERT (EQUAL (PLUS 0 X
       (TIMES -1 A B)) 0))
```

```
100 ASSERT
110 SETQ
120 LOOP
125 BLKWHILE
150 SETQ
160 SETQ
170 ASSERT
190 END
200 ASSERT
```

```
#1: 170-190-120-125-
      150-160-170
#2: 170-190-120-125-
      200
#3: 100-110-120-125-150-160-170
#4: 100-110-120-125-200
```

Fig. III-2

Contexts

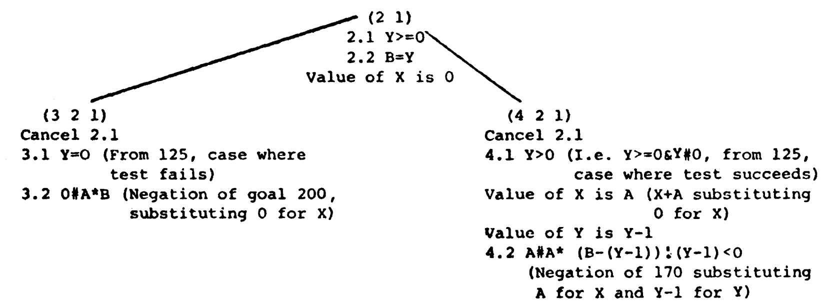
PIVOT incorporates a powerful and general mechanism for incremental incorporation of information, i.e., representing theorems and program states in terms of their differences from previous states. To see the value of such a mechanism, consider the process of proof by contradiction which PIVOT uses, i.e., proving a theorem $c_1 \wedge ... \wedge c_n \supset d$ means deriving a contradiction from $c_1 \wedge ... \wedge c_n \wedge \sim d$. The proof process transforms this set of clauses by adding, modifying, or deleting individual clauses; it is clearly desirable to have a method for doing this without copying the entire theorem each time. If at some point it seems desirable to try a proof by cases, it is valuable to be able to do each subproof independently, yet without making a copy of the theorem and with the ability to reflect transformations that do not depend on assumptions particular to the subproof back into the main proof. An incremental representation mechanism is also useful in theorem generation: when a path reaches an IF statement, one would like a method of splitting it into two paths without having to copy the clauses and assignments to variables which have been made up to that point.

PIVOT provides this mechanism in a form due to the QA4 group: rather than providing many autonomous instances of the data bases for the different paths, theorems, etc., each data base is actually organized as a tree. The trees for all data bases grow in parallel. Each group of parallel nodes is assigned a number called its *context index*: the group is identified internally by a *context* which is a list of all the context indices on the path from the given node to the root (whose index is 1). All processing occurs with respect to a particular context: the state of the data bases at any moment, as seen by any program which uses the data bases, is obtained by logically "overlaying" the successive nodes from the root to the referenced context. The example in figure III-3 may help clarify this idea.

Program being verified:

```
        100 ASSERT Y>=0, B=Y
        110 X ← 0
        120 LOOP
        125 WHILE Y#0: BEGIN
        150 X ← X + A
        160 Y ← Y - 1
        170 ASSERT X=A*(B-Y), Y>=0
        190 END
        200 ASSERT X=A*B
```

Context tree, clauses and assignments for paths beginning 100...125:



```
                           (2 1)
                         2.1 Y>=0
                         2.2 B=Y
                      Value of X is 0

 (3 2 1)                                    (4 2 1)
Cancel 2.1                                 Cancel 2.1
3.1 Y=O (From 125, case where              4.1 Y>0 (I.e. Y>=0&Y#0, from 125,
       test fails)                                case where test succeeds)
3.2 0#A*B (Negation of goal 200,           Value of X is A (X+A substituting
          substituting 0 for X)                             0 for X)
                                           Value of Y is Y-1
                                           4.2 A#A* (B-(Y-1))!(Y-1)<0
                                                (Negation of 170 substituting
                                                 A for X and Y-1 for Y)
```

Clauses "seen" in leaf contexts:

```
  (3 2 1)
2.2 B=Y
3.1 Y=0
3.2 0#A*B

  (4 2 1)
2.2 B=Y
4.1 Y>0
4.2 A#A *(B-(Y-1))!(Y-1)<0
```

Fig. III-3

As the example shows, entries can be removed in a deeper context (e.g., clause 2.1 in contexts 3 and 4) or replaced (e.g., the value of X being reset in context 4) as well as added (e.g., clauses 3.2 and 4.2). The process of determining what information is "seen" in a given context can be represented by imagining each context as a transparent sheet on which information added in that context appears and deletion lines are drawn in places corresponding to information in lower contexts which has been cancelled or superseded: if one stacks up the sheets, with the root context at the bottom, the correct composite state is visible from the top.

The context mechanism has been one of the most versatile internal mechanisms in PIVOT. Its main use in the early versions was for proofs by cases, but it came to be employed for more and more purposes as more attention was paid to minimizing repetition of computation. Because of this, it is worth describing the implementation in some detail. The basic problem is to take a key (a variable or an expression), a data base name (property name), and a context, and logically search backwards toward the root looking for the first context in which there is an entry (or cancellation, i.e., deliberate non-entry) for that key in that data base. For example, in the clause data base the keys are the clauses themselves and the associated value is a flag indicating the presence and origin of the clause. PIVOT first takes the key and looks it up in a hash table (the mechanism required to make this work for expressions is described in Chapter V). The corresponding "value" is a list of entries of the form (context . value), sorted in such a way that if the context C1 is an ancestor of C2, then the entry for C1 appears on the list after that for C2 if both are present. Cancellations are denoted by value=NIL. Now if C is the context for which the value is desired, it is only necessary to search down this list until one encounters an entry where the context component is an ancestor of C (i.e., a "tail" of C in the sense that (2 1) is a tail of (3 2 1)). Since a given expression is likely only to have an entry in a few contexts, the search is likely to be short: a run of PIVOT on a substantial program gave an average of 3.98 entries examined per successful lookup, and 2.44 per unsuccessful lookup.

Theorem generator

PIVOT's theorem generation process works on one path at a time: as it passes through each statement, it calls a routine associated with the statement type (ASSERT, IF, assignment, etc.). The semantics associated with the different statement types are as follows:

Type	**Effect**
Assignment	Evaluate the right-hand side, and any subscripts on the left-hand side. Record the assignment in the appropriate "value data base" (there are separate data bases for simple variables and for arrays).
BEGIN, END, LOOP, EXIT, NEXT, ELSE, REPEAT	None. These statements only affect the flow of control.
IF, WHILE, ELSE IF	Evaluate the condition, and assert (add as a clause) either the condition or its negation, depending on a marker in the path description.
...	Abandon the theorem generating process for this path.
LEMMA	Evaluate the lemma. Create two new contexts underneath the current context. In one, assert the negation of the lemma (this context will be used to prove the lemma). In the other, assert the lemma; this context becomes the current context for the generation process for the remainder of the path.
ASSERT, DECLARE	Evaluate the assertion. If at the beginning of a path, assert the assertion; otherwise (at the end of a path) create a new context underneath the current context and assert the negation of the assertion in this context, which will be used for proving the assertion. (Each goal assertion receives its own context so that the proofs will be independent.)
ASSUME	Evaluate the assumption, and assert it.

Evaluate", in the context of the theorem generator, basically means "replace variables by assigned values, if any". The convention was chosen that variable names in clauses refer to the values of those variables at the beginning of the path, hence values are stored and clauses generated in those terms. For example, if the assignments X ← 2*X and X ← X-Y appear on a path, then the value recorded for X after the second assignment would be 2*X-Y and an assertion to be proved that mentioned X would have the X replaced by 2*X-Y. This approach also works for array elements, in the following more complex manner: whenever an assignment is made to an array element, the (evaluated) subscript and assigned value are placed at the head of a list associated with that array. When evaluating an array reference, A[k], PIVOT builds the internal equivalent of IF k=i_1 THEN v_1 ELSE IF k=i_2 THEN v_2 ... ELSE A[k], where the (i_j,v_j) are the subscript-value pairs associated with A. For example, consider a program which interchanges two elements of an array by

W ← A[I], A[I] ← A[I-1], A[I-1] ← W.

A later reference to A[J] would evaluate to the equivalent of

IF J=I-1 THEN A[I] ELSE IF J=I THEN A[I-1] ELSE A[J],

which is the correct expression in terms of the *original* contents of A. A reference to A[I] would evaluate to A[I-1], since I=I-1 simplifies to "false" and I=I to "true".

In addition to the capabilities discussed above, which are similar to those of King's theorem generator, PIVOT incorporates three improvements which have an enormous effect on its ability to deal with complex programs. These improvements deal respectively with equalities, induction, and case analysis.

If PIVOT encounters an assertion or a test which evaluates to an equality which can be solved for a variable that has not yet been assigned to, PIVOT effectively treats the assertion as an assignment. For example, if ASSERT X=X0 appears among the initial assertions on a path, it is treated like X ← X0. (The clause X=X0 is also put into the data base in case X or X0 is actually assigned to later in the path, or in case references to X have already occurred in earlier assertions.) This occasionally allows simplifications to take place during theorem generation that otherwise would have to wait for the theorem proving phase, in which a scan of the theorem notices the equality and performs the substitution. A similar and much more important action occurs for assertions of the form

.FA(J:p(J))(A[J]=g(J)),

which specify the values of some range of elements of an array. Such assertions are put on the assignment list for the array A and considered just like assignments to specific elements: they produce tests of the form IF p(k) THEN g(k) when forming the conditional expression for a reference to A[k]. Of course, the same result could be obtained by making the correct instance of the assertion during the theorem proving phase (by paramodulation [Rob3]), but this approach produces the same result more directly. The most striking application of this technique occurs in Appendix B, where the proof would have been completely unmanageable without it.

The assertion which cuts a loop is frequently of the form .FA(J:r(J))(p(J)), where r (the range) is some simple relation involving a loop variable (one that changes in a simple way) and p does not involve loop variables directly. In such cases, it is advantageous to divide the proof of the evaluated assertion into two parts: one part deals with that part of the new range of J which overlaps the old, and one part which deals with the new elements. For example, if r(J) is something of the form $M \leq J < N$ and N is replaced by N+1 in the loop, then the overlapping part of the range is $M \leq J < N$ and the new part is J=N (N meaning, as usual, the original value of N). PIVOT actually uses this range-splitting technique whenever the same assertion of the form just mentioned occurs at both the beginning and end of a path: this typically results from a loop, but it may result from a DECLARE. Letting r' be the evaluated form of r ($M \leq J < N+1$ in the example) and p' of p, PIVOT converts the goal

.FA(J:r'(J))(p'(J))

into the two goals

.FA(J:r'(J)$\wedge$r(J)$\wedge$p(J))(p'(J))

and

.FA(J:r'(J)$\wedge$~r(J))(p'(J)).

The reason for including p(J) in the first goal is that it is known to be true (if it is true for all J satisfying r, then it is certainly true for all J satisfying both r and r') and it makes it more likely that the body of the quantifier will simplify to "true". (This is just a short-cut: the theorem prover could deduce p(J) on its own.) Furthermore, p is evaluated to produce p' in a new context in which one of $r' \wedge r$ or $r' \wedge \sim r$ has already been asserted, so that the deduction process described in the next chapter can operate more powerfully in simplifying array references. For example, in the case where a loop variable is being incremented and an assignment is being made to an array element, it is likely that there is enough information in the assertions cutting the loop to determine whether the subscript falls in $r' \wedge r$ or in $r' \wedge \sim r$; this allows elimination of the conditional expression which would otherwise have to be generated for array references in the goal, since one of the two tests will evaluate to "true" via the deduction process described in the next chapter and the other will evaluate to "false". The proofs for King's Example 6, Appendix A, illustrate the value of this strategy.

The range-splitting technique just discussed helps reduce the incidence of conditional expressions generated to cope with assignments to array elements, but in fact PIVOT does not tolerate their intrusion into evaluated expressions at all. When such an expression actually results from evaluating an array reference, PIVOT aborts the evaluation process, creates as many new contexts as there are alternatives in the conditional expression, asserts the respective predicates of the conditional in the new contexts, and starts the evaluation over in all the new contexts "in parallel". For example, in the situation where there is an assignment to A[I] and there is a later reference to A[J] where it cannot be resolved whether J=I, PIVOT creates two subcontexts of the current context, asserting J=I in one and $J \neq I$ in the other, and starts over. This time, the difficulty does not arise, since in one of the two it is known that J=I and in the other, that $J \neq I$. It is true that this approach leads to some proliferation of contexts. The alternative, a proliferation of conditional expressions, is worse since the conditional expression must undergo substantial reprocessing each time a new operator is applied to it, i.e., f(IF p THEN e ELSE e') becoming IF p THEN f(e) ELSE f(e'). Furthermore, in the not unlikely event that there is more than one reference with the same subscript, the context splitting only happens once, and future occurrences of the same subscript are resolved without further difficulty, whereas a new conditional expression must be generated for each reference in the other approach. Finally, case splitting leads to simpler theorems, and the effort required to prove a theorem tends to rise faster than linearly with the complexity of the theorem. Appendix B, the Constable-Gries pair-enumeration program, illustrates the advantages of this approach.

Theorem prover

The PIVOT theorem prover takes a data base containing clauses and tries a series of transformations on it in an attempt to produce a contradiction. There is a fixed list of transformations (procedures) to try: when a procedure succeeds, the prover starts over at the top of the list; if the list is exhausted before a contradiction is attained, the theorem prover gives up. In this respect PIVOT's theorem prover is just like King's. Most of the added theorem-proving power comes from one new procedure which makes instances of quantifiers, and from the deduction mechanism described in the next chapter which is not part of the theorem prover at all.

The theorem prover first collects all equalities which allow elimination of variables. Then it removes the equalities and substitutes the values for the variables in all other clauses. Even though equalities entered by the theorem generator are also used for substitution during theorem generation, there may have been references to the substituted variables (I if I=J is entered) before the equality was reached, and the theorem prover eliminates these. In some sense this procedure is a special case of paramodulation; it is useful because it actually decreases the number of clauses in the theorem.

The next procedure is to take in turn each clause which is a disjunction and truth-valuate each disjunct with respect to the remaining clauses by the techniques described in the next chapter. If any disjunct becomes "true", the clause is dropped; if a disjunct becomes "false", the disjunct is dropped. This procedure essentially extends to disjunctions the automatic truth-valuation procedure for non-disjunctions: if $x \leq y$ is a clause and $x > y$ is asserted, a contradiction is detected, whereas if $x \leq y$! $x \leq z$ is a clause and $x > y$ is asserted, the disjunction is not replaced by $x \leq z$. Its main utility arises in proofs by cases, where there may be important consequences of the newly asserted disjunct (i.e., p being asserted in place of p!q).

If disjunction valuation fails to produce progress, the prover tries a weak form of resolution called *quantifier instantiation*, which adds clauses obtained by substituting expressions for bound variables in universally quantified (.FA) clauses. The basis of this procedure is a pattern matcher which takes a *form* (part of a quantified assertion), an expression to match against the form, and a list of bound variables in the form which may match anything in the expression. The pattern matcher produces all possible matches, each match being a list of assignments of subexpressions to bound variables and a score which tries to measure the "relevance" of the match. The score is computed as follows: the first match of a bound variable against an expression subtracts the number of cells in the expression from the score, on the grounds that simpler matches are more likely to yield useful results; successive matches of the same bound variable do not alter the score; a match of any other variable against the same variable in the expression adds 10 to the score, on the grounds that this is a good measure of relevance. (This scoring algorithm is somewhat adhoc, but seems to produce high scores for good matches, i.e., those that yield instances which contribute to the eventual success of the proof.) The matching procedure itself is straightforward, with one exception: if the form contains a subform $x \geq c$ and the expression being matched is of the form $y \geq d$, the matcher only matches x against y (disregarding the constant term); if the form is $x \geq c$ and the expression is $y \leq d$, the matcher tries x against -y. The rationale is that the matcher is only trying to produce good candidates for instantiation of universally quantified assertions, so the match doesn't have to be perfect. This "semantic matching" capability is unique with PIVOT and will probably be extended in the future.

The quantifier instantiation procedure cycles through the clauses which are universal quantifications, matching the range and the body of each against all other appropriate clauses. To reduce search time, for each operator (e.g., FA, $\geq$, !, etc.) there is a list of all clauses that have that operator as their outermost one: if the outermost operator of a quantifier body is $\geq$, for example, it is only necessary to try matches against clauses whose outermost operator is $\geq$ or $\leq$. After all possible matches have been collected, the matches are sorted by score. Each score is biased by 5 times the depth of the test clause in the context tree, on the grounds that more specialized clauses (e.g., goals as opposed to assertions) are somewhat more promising candidates for a quick contradiction. Furthermore, matches against the range (as opposed to the body) are biased by -200, on the grounds that body matches are more specific in some sense: a range of the form $1 \leq J < N$, for example, will produce a match with any assertion of the form $x \geq c$, leading to many probably irrelevant matches. Starting with the best match, the appropriate instance of the quantifier is asserted. For example, if the clauses

$$.FA(J{:}1 \leq J < N)A[J] \leq A[M]$$

and

$$A[I-1] > A[M]$$

are present, then the clause

$$I-1 < 1 \ ! \ I-1 \geq N$$

would be added since the first clause is actually

$$.FA(J)(J < 1 \ ! \ J \geq N \ ! \ A[J] \leq A[M]).$$

If this instance has a valuation of "true", i.e., is a consequence of existing clauses, then the next best match is tried: instances of this sort frequently have already been asserted as negations of goal assertions. The prover also will not consider any match with a score below -100, which prevents the avalanche of bad instances which often overwhelms such theorem provers.

If there are no sufficiently good matches, the prover looks for a clause of the form $f(a_1 \ldots a_n) \neq f(b_1 \ldots b_n)$ (e.g., $A[I] \neq A[J]$), constructs a subcontext in which the clause is replaced by $a_1 \neq b_1 ! \ldots ! a_n \neq b_n$ (e.g., $I \neq J$), and tries to produce a contradiction in this subcontext, i.e., this is a recursive call of the prover. The rationale is that the inequality on the arguments is a weaker condition, and is worth trying even though the theorem is stronger than the original and may be false (not contradictory) even if the original is true (contradictory). This function removal procedure is of marginal utility and may be removed in the future, although it was the deciding factor in one or two proofs. Function removal is also attempted on clauses of the form

$$.FA(\ldots)f(\ldots) \neq f(\ldots),$$

where the removal is done inside the body of the quantifier.

The next procedure to be tried is proof by cases. The prover selects a clause which is a disjunction, $c_1 ! \ldots ! c_n$, and creates two new contexts: in one of them it asserts c_1, in the other $\sim c_1 \wedge (c_2 ! \ldots ! c_n)$. It then tries to produce a proof in both contexts, independently. Since this procedure can lead to a proliferation of subproofs, it is only tried on clauses whose "size" (number of list cells occupied) is less than 40, on the heuristic suspicion that simpler clauses impose stronger conditions on the theorem and are more likely to produce a contradiction quickly.

If there are no suitable disjunctions, PIVOT tries quantifier instantiation again with a threshold of -500. If this fails, it tries proof by cases with no limit on the clause size. This two-pass scheme seems to work very well in terms of preventing proofs from getting sidetracked by premature proliferation of instances or cases.

If all these procedures fail, the prover tries a desperation measure: it searches for any clause containing certain operators (division, MOD) and replaces the operators by their definition. Division, for example, is defined by $x/y=$"q such that $x=q*y+r$ where $0 \leq r < y$" for positive y and analogously for negative y. The rationale for this procedure is that PIVOT's direct knowledge of these operators is somewhat limited, and there may be some value in converting them to operators which PIVOT knows more about such as addition and multiplication.

There is actually a flaw in the prover design on which the reader will probably have remarked already: whenever progress is made, the prover starts over at the first procedure. This means, for example, that quantifier instantiation will be tried many, many times on the same clauses, since there is no mechanism for preventing it. It turns out that this is the only case which really causes trouble, since the other procedures are quite cheap to try when they are not applicable. This problem has been circumvented (but not solved) by storing, rather than recomputing, the results of past matches. While the control mechanisms in more recent systems such as Planner [Hew3], Conniver [Sus3], and QA4 allow more selective invocation of proof procedures, it is not clear that they allow easy construction of the organization used by PIVOT, which has the advantage that the procedure most likely to produce valuable results (if applicable at all) is always tried first. It is interesting to observe that the present configuration of the PIVOT prover is a result of early hand-simulations and also of experience gained from interactive testing and tracing. In this area, at least, the ability to stop, step, and monitor the dynamic progress of proofs interactively has proved extremely valuable.

Those accustomed to the orderliness of resolution theorem provers will probably feel that PIVOT's theorem prover is a hodge-podge. In fact, the theorem prover plays a minor role in the verification process, and is not meant to be particularly powerful. Almost all of PIVOT's power comes from the constant and economical use of the truth-valuation process of chapter IV. (This was not the author's original intention: the early version of PIVOT relied heavily on the theorem prover.) The prover's relative lack of generality may impede PIVOT's application to arbitrary semantic domains, but the very fact that prover power is not required for the specific domain supports the author's contention that there are other powerful ways of integrating domain-dependent knowledge into a verification system.

DEDUCTION

Data base overview

PIVOT's functioning revolves around two separate groups of data bases. One group records assignments of values to scalar variables, array elements, and functions, and is only used during the theorem generation phase. The role of this group was discussed indirectly in the earlier section on theorem generation. The other group records clauses (predicates asserted or determined to be true in a particular context) and is used during both theorem generation and proof. Both groups are organized according to the context structure described in the previous chapter.

Just as LISP property lists follows the general paradigm

```
            (GETP object property) = value,
```

where "object" is a symbolic atom, "property" is a property name (another atom), and "value" is any LISP datum, so PIVOT data bases follow the paradigm

```
            (GETX object property context) = value,
```

where "object" is a variable or an expression, "property" is the data base name (an atom), "context" is a context name, and "value" is some LISP datum whose nature varies from one data base to another but which usually also involves expressions. (For example, in the data base that records assignments to array elements, the "value" is essentially a list of subscript/assigned value pairs; the subscripts and assigned values are expressions.)

For consistency, variable names in all expressions in all data bases refer uniformly to the values of the program variables at the beginning of the relevant path. This implies that whenever the theorem generator is about to process an expression, it must subject it to an evaluation process which replaces all variables with assigned values by those values, expressed in terms of the values which the variables had originally.

Value data bases

PIVOT uses the "value" group of data bases to record assignments of values to names: these data bases are the source of information required by the evaluation (value substitution) process of the theorem generator. There are three data bases, corresponding to the three kinds of named objects that can appear in PIVOT programs:

Name	Mnemonic	Function
CVAL	Current VALue	Associates values with scalar variables.
ELLIST	ELement LIST	Associates values with array elements.
SFDEF	Special Function DEFinition	Associates definitions with functions.

(Program modules, which are another kind of named object, are kept track of using LISP property lists in a much more mundane manner.)

As PIVOT moves forward along a path during theorem generation, there are two events that can lead to recording an assignment of a value to a scalar variable. One, of course, is an assignment statement. The other is an assertion of equality, either through an ASSERT statement or an IF test, e.g., ASSERT I=I0 or the "true" path through IF I=I0 THEN ... An assignment statement leads to a new value being entered in CVAL under the variable name being assigned to. Since a variable can only have one CVAL at a time on a given path, an equality only leads to a CVAL entry if none exists for the variable already; also, an equality does not produce a CVAL entry if the assigned value would be too complicated (not a linear combination of variables), on the grounds that there is a good chance that subsequent events may lead to a simplification before the substitution is made by the theorem prover. (Equalities are recorded as clauses, and will therefore be seen by the theorem prover, even if no CVAL entry is made: the CVAL entry just makes the substitutions earlier, in the hope that simplification will result.)

CVAL provides a nice example of the usefulness of the context tree in theorem generation. Consider the program fragment in figure IV-1. The assignment statement 200 is "seen" in all three subsidiary contexts, and superseded independently in two while remaining unaltered in the third.

The situation for array elements is more complicated. Besides assignments and asserted equalities, PIVOT takes note of quantified equalities, i.e., assertions of the form .FA(j:r(j))A[j]=f(j). (This process was described as part of the theorem generation phase.) If one views assignment to an array element as assignment of a new value to the entire array, one can imagine assigning values to many elements at once as well as making assertions about many elements at once. PIVOT does not actually provide a notation for such assignment to a partial array, although it could: the only widely used language with this ability is APL [Iv4] [Ger4].

There are four kinds of defined functions in PIVOT. The first kind, procedure modules, receive their definitions as described in Chapter II. They are always treated as closed subroutines: a call to a procedure causes PIVOT to retrieve the initial and final assertions from the procedure definition and construct an appropriate theorem and assertion for the caller. The second kind is LET definitions. These have their definitions recorded in SFDEF during a preliminary scan over the procedure being analyzed, and calls to them are always replaced by the substituted definition during theorem generation. The third kind is built-in functions, currently MIN, MAX, ABS, and SIGN. The definitions of these are entered in SFDEF under context 1 (the root of the context tree) when PIVOT first initializes itself and are never altered; calls are again always replaced by the substituted definition. The fourth kind is the internal functions used for division and modulus (QUOTIENT and MOD): these are also defined permanently, but calls are only expanded as a last resort during the theorem proving phase. All definitions in SFDEF have the same form, namely

(<dummy-variable-list> <defining-expression> <always-expand-flag>).

If function definitions (LAMBDA expressions) were added to PIVOT at some later date, then the values in SFDEF would be precisely such expressions together with an always-expand flag.

There is an interesting parallel between arrays and functions which PIVOT may exploit at some future time. Arrays are generally assigned values element-by-element, or asserted to have certain values over a range of indices as a function of the index. Functions are generally assigned "closed forms" which given their values at all points, but some kinds of functions (such as absolute value) are more conveniently defined in disjoint ranges, and component selectors for records, which can be thought of as functions and in some languages are even written with function-call syntax, are usually assigned values one point at a time (i.e., one usually assigns a value to a component of only a single record at once). At the moment, PIVOT treats arrays and selectors identically, allowing pointwise or piecewise specification of values, and functions differently, requiring total specification. In the future it may be a desirable simplification to treat all three structures the same.

Program:

```
         .
         .
         .
    200 X ← Z
    210 IF X>0 THEN X ← X-1
    220 ELSE IF X<0 THEN X ← X+1
    230 LEMMA Z=0 ! X**2<Z**2
         .
         .
         .
```

A possible context tree, with clauses and CVAL entries:

```
                              (6 2 1)
                            CVAL(X)=Z


  (7 6 2 1)                   (8 6 2 1)                    (9 6 2 1)
7.1 Z>0                 8.1 Z<0 (i.e Z≯0& Z<0)       9.1 Z=0 (i.e Z≯0& Z≮0)
CVAL(X)=Z-1                 CVAL(X)=Z+1
```

The evaluation of the expression in statement 230, in each resulting context:

```
  (7 6 2 1)                   (8 6 2 1)                    (9 6 2 1)
Z=0 ! 2*Z-1>0             Z=0 ! 2*Z+1<0                Z=0
(from Z=0 ! (Z-1)**2<Z**2 (from Z=0 ! (Z+1)**2<Z**2)   (from Z=0 ! Z**2<Z**2)
```

Fig. IV-1

Clause data bases

PIVOT uses the "clause" group of data bases to record clauses, i.e. expressions which are deemed to be "true". The theorem generator places assertions and the outcomes of IF and WHILE tests in these data bases, and the negations of goal ASSERTs and LEMMAs; the theorem prover transforms the clauses in an effort to produce a contradiction. There are four clause data bases, each of which stores information about clauses in a different form:

Name/Mnemonic	**Key/Value**	**Function**
THCLAUSEP/ THeorem CLAUSE	Clause/ origin flag	Stores the clauses themselves
C$1/ Coordinate 1	Operator/ list of clauses	Collects clauses with the same outermost operator
RC$2/ Relation Coordinate 2	Arithmetic expression/ list of clauses	Collects arithmetic relations with the same non-constant part
ULBL/ Upper-Lower Bound List	Variable or arithmetic expression/ coefficient triples	Stores $\leq$ and $\geq$ relations and their consequences

Unlike the value data bases, where each data base stores a different kind of information, the clause data bases all store subsets of the same information, in different forms for different kinds of processing.

The idea behind the clause data bases is that it is important to be able to make certain kinds of trivial deductions very cheaply, where "deduction" means "assignment of a truth-value to a logical expression". The most important application of this ability is in eliminating cases of subscript conflict, i.e., being able to deduce that $J \neq I$ in the situation where there has been an assignment to A[I] and there is a reference to A[J], but it also has a very significant effect in preventing trivial theorems from ever reaching the somewhat ponderous machinery of the theorem prover.

The simplest case in which an expression has an immediately determinable truth-value is when the expression, or its negation, is a clause: if the former, the truth-value is "true", if the latter, "false". Consequently, there is a data base (THCLAUSEP) in which the key-expressions are the clauses themselves and the values are basically irrelevant since they only serve to indicate that the clause is present. Thus if THCLAUSEP(e)$\neq$NIL, e's truth-value is "true"; if THCLAUSEP(~e)$\neq$NIL, e's truth-value is "false". (PIVOT actually does use the THCLAUSEP value, to distinguish between clauses put in as a result of ASSERT statements, clauses put in as negations of goals, and other clauses, since this information is useful when printing theorems for the user.)

PIVOT extends the idea that an expression has a trivially deducible truth-value to a number of other situations. For example, if $X \geq 0$ is a clause, then $X<0$ and $X=-2$ are both false. More generally, if e is any expression with no constant term, r and r' are relations ($=$, $\neq$, $>$, $<$, $\geq$, $\leq$), and c and c' are constants, there is a table which gives the truth-value of e r' c' given that e r c is a clause (figure IV-2). This observation (due to King) makes it advantageous to group together all clauses with the same e. RC$2(e) is just the list of all such clauses, and is consulted whenever it is desired to determine a truth-value for an expression e r' c'.

r'	e=c'	e≠c'	e≥c'	e≤c'
e=c	c' = c: T c' ≠ c: F	c' = c: F c' ≠ c: T	c'≤c: T c'>c: F	c'≥c: T c'<c: F
e≠c	c' = c: F	c' = c: T		
e≥c	c'<c: F	c'<c: T	c'≤c: T	c'<c: F
e≤c	c'>c: F	c'>c: T	c'>c: F	c'≥c: T

This table indicates under what conditions e r' c' has a deducible truth-value, given that e r c is "true". (For integers, e>c is e≥c+1 and e<c is e≤c-1).

Fig. IV-2

A more complex situation arises when a deduction must be made from more than one relation, e.g., the deduction that $X<Z$ is false when $X\geq Y$ and $Y\geq Z$ are known to be true. King's system used a procedure called the "linear solver" to determine satisfiability of a set of simultaneous linear inequalities. While the "linear solver" could make the type of deduction just illustrated, it would be slow and painful. PIVOT uses a more restricted but more efficient technique, which is original with the present author and requires a certain amount of algebraic justification.

Consider a chain of relations, assumed to be true, of the form $x_1\geq x_2+c_1$, ..., $x_{n-1}\geq x_n+c_{n-1}$, $x_n\geq c_n$, where the x's are variables and the c's constants. Suppose we want to determine whether this assumption fixes the truth-value of a single relation of the form SUM[i=1,n]a_i*x_i r c, where the a's and c are constants and r is any arithmetic relation. (The SUM notation is a concession to the lack of a summation symbol. Its presence in the paradigm form does not imply its presence in the PIVOT program language, e.g., $x_1-x_3<0$ is an example of the form.) The information contained in the chain of known relations can equally well be expressed as $x_i=x_{i+1}+c_i+d_i$, where the d's are unknown non-negative quantities. With this representation, we have a direct representation of the x's in terms of the c's and d's:

x_i=SUM[j=i,n]c_j+d_j.

By substituting this expression into the relation under investigation, we can rewrite this relation as:

SUM[i=1,n]SUM[j=i,n]a_i*(c_j+d_j) r c.

We now break the left side of the relation into constant and non-constant parts, and interchange the order of summation in the latter (these operations are legal since the sums are finite):

(SUM[i=1,n]SUM[j=i,n]a_i*c_j)+

(SUM[j=1,n]SUM[i=1,j]a_i*d_j) r c.

Now let us compute two sets of cumulative coefficients:

b_j=SUM[i=1,j]a_i for j=1, ..., n;

e_i=SUM[j=i,n]c_j for i=1, ..., n,

and rewrite the relation again:

(SUM[i=1,n]a_i*e_i)+(SUM[j=1,n]b_j*d_j) r c.

By moving the first term on the left to the other side, we finally reduce the test relation to the form

s r k, where

s=SUM[j=1,n]b_j*d_j, and

k=c-SUM[i=1,n]a_i*e_i.

Note that k is a constant computed from the asserted relation chain and the test relation, and s is a linear sum of non-negative unknown quantities whose coefficients are likewise computable. The truth-value of s r k is the same as that of the test relation, in the sense that if it has a determined, identical truth-value for every assignment of non-negative values to the d's, the test relation necessarily has that truth value. The situations in which s r k has a unique truth-value are easily discovered to be those in the following table:

r	s r k true if	s r k false if
=	All b's=0 and k=0	All b's$\geq$0 and k<0, or all b's$\leq$0 and k>0
$\geq$	All b's$\geq$0 and k$\leq$0	All b's$\leq$0 and k>0
$\leq$	All b's$\leq$0 and k$\geq$0	All b's$\geq$0 and k<0

The entries for the remaining relations are filled in by observing that s>k is true precisely when s$\leq$k is false, and so on.

An example may clarify the testing procedure which the foregoing analysis justifies. Suppose we know that U$\geq$V, V>W (or equivalently V$\geq$W+1), and W$\geq$0, and we wish to determine a truth-value for U-V+W$\geq$0 by the method just described. This situation fits into the paradigm as follows:

x_1	U	a_1	1	e_1	1	k	0
x_2	V	a_2	-1	e_2	1		
x_3	W	a_3	1	e_3	0		
c_1	0	c	0	b_1	1		
c_2	1			b_2	0		
c_3	0			b_3	1		

All b's are $\geq$0, and k$\leq$0, so the table says the test relation is necessarily true.

PIVOT actually implements the algorithm of the preceding paragraphs quite directly. The asserted relation chains are maintained in the ULBL data base: ULBL(x) is a list of elements of the form (y m e . l), where y is another variable in the chain, m is the difference between y's subscript and x's, e is the sum of all the intervening c's, and l is a list of all the clauses that contributed to this entry (required for trace output and to make deletion of clauses remove ULBL entries properly). By convention, the last relation in the chain is represented by a fictional variable "0". Thus the ULBL entries for the three-variable chain N>J, J$\geq$I, I>0 might appear as follows:

```
x            ULBL(x)

N            ((J 1 1 N>J)
              (I 2 1 N>J J≥I)
              (0 3 2 N>J J≥I I>0))

J            ((N -1 -1 N>J)
              (I 1 0 J≥I)
              (0 2 1 J≥I I>0))

I            ((N -2 -1 N>J J≥I)
              (J -1 0 J≥I)
              (0 1 1 I>0))

0            ((N -3 -2 N>J J≥I I>0)
              (J -2 -1 J≥I I>0)
              (I -1 -1 I>0))
```

(The fact that the space and updating time for ULBL are proportional to the square of the number of entries in the chain does not appear to be a practical problem: the longest chain observed in the test cases had 5 links, and most had only 2 or 3.) When a new relation of the form $x \geq y+c$ is asserted, all the ULBL's in the chain must be updated. Testing a relation is straightforward: the terms must be sorted by subscript first, then the method applied directly. There are two slight complications. One is that if the last relation in the chain (of the form $x \geq c$) is missing, it is still possible to make deductions if the coefficient of c_n in the expression for k is zero, i.e., if the sum of all the a's is zero. The other is that it is necessary to verify that the variables in the test relation actually fall in a single chain, since this information is not represented explicitly in ULBL and would be very messy to update if it were: PIVOT handles this by brute force, namely checking, for each x_i that appears in the test relation, that each x_j has an entry on ULBL(x_i) for all $j>i$. This procedure is costly in principle, but the vast majority of test relations only have two variables and thus require only one check. Finally, the ULBL computation does not come into play at all for relations e r c where RC$2(e) can provide a truth-value.

The foregoing discussion should make it clear that PIVOT's clause storage mechanisms are designed to make querying cheap at the expense of updating time. In fact, the truth-value testing process is so cheap that it is actually applied to every logical expression as soon as it is constructed, and still only accounts for 15% of the CPU time. This approach has two advantages over delaying all deduction until theorem generation is completed. First, it performs semantically conditioned simplification as early as possible, which helps hold down the size of intermediate expressions and the number of cases which must be considered. Second, it leads to automatic analyses which bear a much stronger resemblance to human proofs than those of King or Slagle and Norton [Sla4]. This idea of continuous consultation of a data base which is in some sense "active" rather than "passive" pervades a good deal of current work in artificial intelligence; its application to PIVOT has grown out of experience and experimentation, and has more than justified the effort spent.

REPRESENTATION AND SIMPLIFICATION

Expression overview

Like King's system, PIVOT represents expressions internally in a simplified canonical form. Each operator has an associated routine which is responsible for producing the canonical expression resulting from applying that operator to operands, e.g., there is a routine MKPLUS which takes a list of expressions and produces their canonical sum. This organization reduces the size of intermediate expressions (an important factor since a good deal of the processing involves scans over entire expressions) and, since most simplifications are inherently local anyway, provides an inexpensive unifying framework within which new operators and their associated simplification rules can easily be added. Restricting the variety of forms for the same expression can also cut processing time: for example, keeping the terms of a sum sorted reduces the time to add two existing sums (of m and n terms) and combine like terms from O(m*n) to O(m+n). It is worth noting that these operator routines, particularly those for addition, multiplication, and logical connectives, were a major speed bottleneck in the early versions of PIVOT and had to be rewritten: in the process, they were tuned for the relatively simple expressions that form the great majority of those actually encountered in proofs.

PIVOT, unlike King's system, relies heavily on the ability to store and retrieve properties of expressions, i.e., to use expressions as keys in tables. For example, computing the negation of a logical expression is a frequent and moderately time-consuming operation: it would be convenient to compute the negation only once, store it in a table, and then look it up when it was needed thereafter. A problem arises from the fact that hash tables [Mor5], which afford the most rapid lookup procedure, require a fast mapping of the key into a small range of integers. Doing this for an expression, in the absence of any other strategem, requires examining the entire expression, if not to compute the hash code, at least to compare the expression to the key in the table. However, if means can be found to ensure that each expression actually only appears once in storage, then the *location* of the expression can be used as the key and lookup can be extremely fast. It appears that the extra time required to ensure this property using PIVOT's technique (which is described in the next paragraph) is far outweighed by the reduction in space and the increased speed with which properties of expressions can be determined compared to more conventional methods. As Chapters III and IV make clear, the entire organization of PIVOT assumes that looking up expressions in tables is cheap. (The QA4 group uses a different technique called a "discrimination net" to achieve similar ends; their application requires more general kinds of matching than strict identity when looking for a matching key.)

The PIVOT technique for assuring expression uniqueness is similar to techniques used in compilers to detect common subexpressions. PIVOT expressions are LISP list structures, and all such structures are built up from two-component nodes (e.g., the head node of the expression (PLUS 3 X) has as components the symbol PLUS and a pointer to the sublist (3 X)), so the problem reduces to ensuring that no two nodes exist with the identical components. PIVOT implements this with a function (HCONS c1 c2) which constructs a new node with the given components if none exists, or returns the existing node if there is one. If c2 is NIL (the list terminator), HCONS looks up c1 in a hash table, using the location as the key. (The locations of literal symbols like X are guaranteed to be unique by LISP.) The value found in the hash table is a node with the key as first component and NIL as second. If c2 is not NIL, it is looked up in a different hash table, where the value is a list of nodes with c2 as second component: this list is searched linearly for a node with the proper first component (c1). Thus, after constructing the list (PLUS 3 X), the NIL table would have the following entry:

X: (X)

and the non-NIL table would have these entries:

(X): ((3 X))

(3 X): ((PLUS 3 X))

All routines in PIVOT which construct expressions use HCONS, either directly, or to copy the expression before returning it, so all expressions are guaranteed unique representation. This hashing scheme was partly a result of the presence in BBN-LISP of a hashing function that only took a single pointer as the hash key, and the absence of any facilities for building a two-key hashing function: the latter would probably have been preferable to the scheme just described, although not greatly since the average length of the lists is only about 3 entries.

Arithmetic expressions

The syntax of internal arithmetic expressions appears in the figure below. Conditional and choice expressions will be discussed in the next section. Note that the internal syntax is quite a bit simpler than the external: for example, there is no subtraction operator. Note also that some simplification, such as combining constant terms, is implicit in the syntax. These syntax equations, and the others in this chapter, refer to LISP lists rather than strings of characters; in particular, parentheses refer to list boundaries (as is customary in the external representation of LISP data).

```
<arithmetic-expression> ::= (PLUS <num> <term>
      $<term>) | <num> | <term>

<term> ::= (TIMES <num> <primary> $<primary>) |
      <primary>

<primary> ::= <id> | <conditional> | <choice> |
      (<function> $<arithmetic-expression>)

<function> ::= EL | QUOTIENT | REMAINDER | IEXPT |
      <user-defined-function>
```

Simplifications for addition include combining constant terms, combining other like terms (e.g., adding X and (TIMES 2 X) to produce (TIMES 3 X)), and eliminating terms that sum to zero. After simplification, to produce the canonical form the non-constant terms are sorted using an ordering which initially disregards constant factors, and places variables first, then other primaries. (This is the same as King's ordering: its advantages over an arbitrary ordering based on, for example, the locations of their terms, are that it places variables and negatives of variables first, which is useful in determining whether an equality can be solved for a variable, and that it makes collecting like terms easy. It is not clear that these advantages outweigh the expense of sorting according to such a complex ordering.) If the final sum is just a constant or a variable, the canonical form is the constant or variable by itself rather than a PLUS expression. King's canonical form essentially keeps all expressions in the form (PLUS <num> $(TIMES <num> $<primary>)); the PIVOT form seems preferable since it conserves space in expressions not involving multiplication, but it admittedly complicates the operator routines for addition and multiplication. The decision to always have a constant term in a sum was a compromise with King's form.

Simplifications for multiplication are essentially the same as those for addition: like factors are combined to produce exponentiation (IEXPT). Products of sums are multiplied out, e.g., multiplying (PLUS -1 X) by Y produces (PLUS 0 (TIMES -1 Y) (TIMES 1 X Y)). Although this occasionally produces larger expressions, it simplifies the internal syntax drastically.

EL is the subscripting operator and has no speical simplification rules. Division recognizes x/x=>1, x/1=>x, and 0/x=>0; if the denominator is a constant or a simple variable, and the denominator divides each term of the numerator, simplification also occurs. Exponentiation of a sum or a product to a constant power results in explicit multiplying-out; the only other transformations for the IEXPT operator are 0**x=0, 1**x=1, and x**1=x.

PIVOT currently has no provision for user-defined simplifications, but extending it to deal effectively with user-defined functions and domains will require such provision. The main problem is not how to express such rules but how to incorporate them into the program in an efficient manner. It is the author's opinion that this will involve user-defined canonical forms as well as some kind of efficient pattern detection mechanism; this question will receive further attention in the last chapter of this thesis.

Logical expressions

The syntax of internal logical expressions appears below. Conditional and choice expressions are actually arithmetic expressions, but their processing and semantics are more closely related to those for logical expressions.

```
<conditional> ::= (LCOND ${<logical-expression>
      <arithmetic-expression>})

<logical-expression> ::= <quantifier> |
      <disjunction> | <conjunction>

<disjunction> ::= (OR <conjunction> <conjunction>
      $<conjunction>) | <logical-primary>

<conjunction> ::= (AND <disjunction> <disjunction>
      $<disjunction>) | <logical-primary>

<logical-primary> ::= <quantifier> | (<relationb-
      symbol> <arithmetic-expression> <num>)

<relation-symbol> ::= EQUAL | NEQUAL | LEQ | GEQ
```

LCOND stands for "linear conditional" and represents an expression which may have different values depending on the truth of some condition or conditions. Unlike the familiar IF-THEN-ELSE of programming languages, however, the logical-expressions of a PIVOT conditional are always mutually exclusive, so that in principle their order is unimportant and they could be evaluated in parallel. (In fact, for canonicalization, the clauses are sorted on their logical-expressions.) (This format is called "linear" somewhat in the sense of "linearly independent".) Thus the ALGOL conditional expression IF p1 THEN e1 ELSE IF p2 THEN e2 ELSE e3 would become (LCOND p1 e1 p2∧~p1 e2 ~p2∧~p1 e3). In the early versions of PIVOT, the conditional expression did have sequential (ALGOL-like) semantics, but it turned out that this was ill-suited to processing by the LISP mapping functions and also that the linear (independent) conditions like p2∧~p1 were usually what was wanted.

Three kinds of simplification apply to conditionals. One is de-nesting: (LCOND p1 e1 p2 (LCOND p3 e3 p4 e4)) becomes (LCOND p1 e1 p2∧p3 e3 p2∧p4 e4). Another is removal: if any condition is NIL (false), that clause is removed, and if a condition is T (true: this implies that all the others are NIL by exclusiveness), its associated expression becomes the value of the entire conditional. Another is merging of clauses with identical consequents: (LCOND p1 e1 p2 e2 p3 e1) becomes (LCOND p1$\vee$p3 e1 p2 e2). A fourth transformation related to conditionals is carried out by a pre-processor that precedes every operator routine: if op is any arithmetic operator (PLUS, TIMES, EL, QUOTIENT, REMAINDER, IEXPT, or a user-defined function), then (op ... (LCOND p1 e1 ... pn en) ...) becomes (LCOND p1 (op ... e1 ...) ... pn (op ... en ...)), i.e., the operator is pushed inside the conditional. (This procedure can lead to a proliferation of cases, for example if both terms of a sum involve conditionals, but PIVOT is sufficiently successful at avoiding conditionals through case analysis that this difficulty does not seem to arise in practice.) In this way, the conditional eventually becomes an argument of a predicate (in the current PIVOT, always the first argument of arithmetic relation), where it undergoes a similar transformation and vanishes: (relation-symbol (LCOND p1 e1 ... pn en) .num) becomes (OR (AND p1 (relation-symbol e1 .num)) ... (AND pn (relation-symbol en .num))). King's system has all of these transformations except clause merging, but it is not clear from his thesis whether his conditionals are sequential or parallel.

The alert reader will have remarked that there is no NOT operator in PIVOT. It happens that the negation of a canonical expression is always another canonical expression without needing a NOT operator: AND and OR become each other, likewise EQUAL and NEQUAL, LEQ and GEQ. (The last arises from the fact that for integers, $\sim(x\leq y) \equiv x>y \equiv x\geq y+1$.) In fact, the PIVOT internal expression syntax does allow for NOT, and there is a routine associated with it, in expectation of future additions such as user-defined predicates.

Simplifications for AND include the obvious ones: (AND)=T, (AND x)=x, (AND x ... ~x ...)=NIL, (AND x ... x)=(AND x ...), (AND (AND ...) ...)=(AND ... ...). A less obvious set, discovered by King, results from the representation of arithmetic relations in the form (<relation-symbol> <arithmetic-expression> <num>). Any conjunction of relations of this form with the same <arithmetic-expression> can be simplified as follows:

(1) If any of the relations is an equality, then substitute the numeric value obtained from that relation into all the others: if they are all true, then the simplified conjunction is the equality alone, otherwise NIL (false).

(2) If more than one of the relations is a LEQ, drop all such relations except the one with the smallest numeric part (e.g., $x \leq 3 \& x \leq 5$ simplifies to $x \leq 3$).

(3) Similarly, if more than one relation is a GEQ, drop all such relations except the one with the greatest numeric part.

(4) If the upper and low bounds have crossed (e.g., $x \leq 2 \& x \geq 4$), the result is NIL. If they are equal, replace them by an equality and do case 1.

(5) Otherwise, the remaining relations are all inequalities. If any fall outside the range delimited by the upper and lower bounds, delete them. If one falls on an endpoint, delete it and "tighten" the bound, e.g., $x \geq 4 \& x \neq 4$ becomes $x \geq 5$. (This transformation relies on properties of the integers and is not applicable to real arithmetic in general.) This may produce a crossover or equality as in case 4. PIVOT actually carries out this process whenever it forms a conjunction that involves two or more arithmetic relations with the same first argument (non-constant part).

The remaining transformation for AND arises when one of the conjuncts is a disjunction. In this case, each disjunct is compared against every simple conjunct to see whether it is either incompatible with that conjunct, in which case the disjunct is eliminated (e.g., $x \leq 3 \& (x \geq y ! x \geq 5)$ becomes $x \leq 3 \& x \geq y$), or implies that conjunct in which case the entire disjunction is eliminated (e.g., $x \leq 2 \& (x \geq y ! x \leq 3)$ becomes $x \leq 2$). The simplifications for OR are exactly analogous to those for AND.

Simplifications for arithmetic relations are those of King (dividing out constant factors, and recognizing that certain equalities like 4x+2y=3 have no integer solutions), and two more. One is conversion of $x/k \geq n$ to $x \geq k*n$, where k and n are integers, k positive. The other is conversion of x*y=0 to x=0!y=0, where at least one of x and y is not a sum (PIVOT's factoring abilities are rather weak).

This long list of simplifications may seem haphazard, but it was extended from King's list by hand-proof of several complex examples, and every simplification in PIVOT receives sufficient use to warrant retaining it in view of the fact that the same result often cannot be obtained without it. The list is certainly not "complete" in the formal sense of the term; the author believes that this is not a very important criterion for judging the merits of such systems. This question is discussed at length in Chapter VII.

Quantifiers

```
<choice> ::= (CHOICE <bound-variable-list>
      <logical-expression> <arithmetic-expression>)

<quantifier> ::=
      (FA <bound-variable-list> <domain> <disjunction>) |
      (EX <bound-variable-list> <domain> <conjunction>) |
      (FU <bound-variable-list> <logical-expression>
            <logical-expression>)

<bound-variable-list> ::= (BV <id> $<id>)

<domain> ::= <conjunction> | T
```

Quantifier simplifications will be discussed in terms of FA (universal quantifier) although EX (existential quantifier) is treated exactly the same way. PIVOT tries to divide the expression in the range of a quantifier into a "domain" and a "body" for purposes of simplification and as an eventual guide to forming instances of universally quantified assertions. (This representation is motivated by the frequent occurrence in program proofs of quantifiers which describe properties of intervals or subarrays.) If the body of a FA is $c_1!c_2!...!c_n$, then c_i is a "domain predicate" if it is of the form $v \neq e$, $v \geq e$, or $v \leq e$ where v is a bound variable and e is any expression not involving v (but possibly involving other bound variables). The negated disjunction of all domain predicates becomes the domain and the disjunction of the remaining c_j becomes the body, using the equivalence of the three expressions:

$$(\forall x{:}p(x))(q(x)),$$
$$(\forall x)(p(x) \supset q(x)),$$
$$(\forall x)(\sim p(x) \lor q(x)).$$

If any domain predicate is an equality (inequality before negating), e is substituted for v throughout the domain and the body and v is deleted from the list of bound variables. If the body reduces to a logical constant b (T or NIL), then the entire quantifier expression is a constant whose value depends on the type of quantifier and whether there are any values of the bound variables that satisfy the domain expression, according to the following table:

	sat.	**unsat.**
FA	b	T
EX	b	NIL

A simpler version of King's "linear solver" suffices to determine whether the domain is satisfiable: since all relations are of the form $v \geq e$ or $v \leq e$, PIVOT eliminates one variable at a time by replacing

$$v \geq e_1 \& ... \& v \geq e_m \& v \leq e'_1 \& ... \& v \leq e'_n$$

by

$$e'_1 \geq e_1 \& ... \& e'_1 \geq e_m \& e'_2 \geq e_1 \& ... \& e'_n \geq e_m$$

and leaving relations not involving v unchanged. If a logical contradiction arises, the domain is unsatisfiable; if PIVOT is able to eliminate all the variables without producing a contradiction (which may not happen if some of the e's are more complicated than linear combinations of the variables), then the domain is satisfiable; otherwise, the resulting set of conjuncts is partitioned into domain and body and PIVOT starts over. The idea of separating the domain from the body arose after experience with early versions of PIVOT indicated that this provided a good guide to what instantiations to try during the theorem-proving process (namely, those which made the *body* identical to the negation of a clause), and that there would be a significant efficiency gain from not having to negate the domain when negating a quantified expression, i.e., taking advantage of the fact that $\sim(\forall x{:}p(x))(q(x))$ is $(\exists x{:}p(x))(\sim q(x)))$.

Unlike resolution-based systems, which move quantifiers to the outermost level, PIVOT moves quantifiers as far in as possible. This involves moving a FA inside an AND, moving a disjunct of an OR outside a quantifier if none of the bound variables occur in it, and one more complex transformation which was discovered in the course of trying out examples: converting $(\exists x)(c \wedge (x=e \,!\, d))$ to $c[e{:}x] \,!(\exists x)(c \wedge d)$, where $c[e{:}x]$ represents the result of substituting e for x in c. One obvious initial transformation, namely converting a body to conjunctive form for FA or disjunctive form for EX, is only carried out if such a transformation does not cause a size explosion (specifically, if it no more than doubles the length of the expression).

There is a computationally expensive problem associated with converting an expression with bound variables to canonical form. One would like the canonical form to be independent of the names of the bound variables, so that e.g., (FA (BV X) (OR (P V) (P X))) and (FA (BV U) (OR (P U) (P V))) would have the same canonical representation. As this example shows, this is an extremely difficult task; in general, it involves finding a canonical form for an arbitrary labeled tree. PIVOT does not attempt to deal with this problem: the two expressions exhibited above are actually both in canonical form as far as PIVOT is concerned.

The CHOICE function is an attempt to clear up a fuzzy area in King's definition of division. One would like to define x/y according to the Fundamental Theorem of

Arithmetic, i.e., "the q such that $x=q*y+r$ where $0 \leq r<|y|$". Leaving aside some complications introduced by the possibility of a negative denominator, King's definition was a conditional: "if $q*y \leq x<(q+1)*y$ then q", where q was a newly-generated variable name. (For positive y, there is always a q that satisfies the condition.) Making q a new variable implicitly recognizes that q is local to this subexpression; this function is more properly provided by a bound variable. In fact, there are situations where this omission of an implicit quantifier produces errors. Consider the generic expression

[1] .FA(x)(r(x) IMP p(x/2))

where / denotes integer division. King's system converts this to

[2] .FA(x)(r(x) & (x=2*z ! x=2*z+1) IMP p(z)).

Now in the specific case where r(a) is $a \geq 0$ and p(a) is $a<0$, [1] becomes

[3] .FA(x)(x≥0 IMP x/2<0)

which is obviously false. However, [2] becomes

[4] .FA(x)(x≥0 & (x=2*z ! x=2*z+1) IMP z<0)

which is false if the free variable z is ≥ 0 and true if $z<0$; since free variables in clauses are implicitly existentially quantified in both systems, [3] and [4] are not equivalent. The CHOICE operator solves this problem by explicitly indicating the presence of a bound variable: (CHOICE (BV x) (p x) (f x)) means "f applied to the x such that (p x)". Use of CHOICE is restricted to situations where there is precisely one value of x which satisfies p. The quotient x/y, for example, could be defined as (CHOICE (BV Q) Q*y≤x<(Q+1)*y Q). In principle, CHOICE expressions require a transformation similar to the one for conditionals when used as an argument of a function: (f (CHOICE (BV x) exp x)) becomes (CHOICE (BV y) y=f(x)&exp y). It is to avoid this growth of expressions and repeated generation of new variables that CHOICE takes a third argument, to represent the computation to be done once the value has been found: the example would actually be represented as (CHOICE (BV x) exp (f x)). The only simplification applicable to CHOICE arises when the <logical-expression> contains a conjunct which is an equality that can be solved for one of the bound variables. In this case, the variable is dropped and its derived value substituted for all free occurrences of the variable in the two expressions. If all variables are eliminated by this process, the CHOICE itself disappears and is replaced by the <arithmetic-expression>.

When a CHOICE expression becomes the argument of a predicate, (r (CHOICE (BV x) (p x) (f x))), there are two plausible transformations to use to eliminate the CHOICE. One possible transformation produces

[1] $(\forall x)(p(x) \supset r(f(x)))$:

this requires r to hold for f of <u>all</u> x satisfying p, and is vacuously true if no such x exists. The other possibility is

[2] $(\exists x)(p(x) \wedge r(f(x)))$,

which only requires r to hold for <u>some</u> such x, and is false if no such x exists. Since the use of CHOICE is restricted to situations where x is guaranteed to exist and be unique, the two forms are actually equivalent! To see this, consider the unique-existence condition

[3] [3a] $\wedge$ [3b], where

[3a] $(\exists w)(p(w))$ {existence}

[3b] $(\forall y,z)(p(y) \wedge p(z) \supset y=z)$ {uniqueness}.

We want to show [1] & [3] => [2] and [2] & [3] $\Rightarrow$ [1].

The first derivation:

[1] $\wedge$ [3a] $\wedge$ [3b] $\Rightarrow$

[1] $\wedge$ [3a] $\Rightarrow$

[1] $\wedge$ $p(\underline{w})$ {Skolem constant} $\Rightarrow$

$p(\underline{w}) \wedge r(f(\underline{w}))$ {Using $\underline{w}$ for x in [1]} $\Rightarrow$

$(\exists w)(p(w) \wedge r(f(w)))$ {By example} $\Rightarrow$

[2] {Alphabetic variant}.

The second derivation:

$[2] \wedge [3a] \wedge [3b] \Rightarrow$

$[2] \wedge [3b] \Rightarrow$

$p(\underline{x}) \wedge r(f(\underline{x})) \wedge [3b]$ {Skolem constant} $\Rightarrow$

$p(\underline{x}) \wedge r(f(\underline{x})) \wedge (\forall z)(p(z) \supset z=\underline{x})$ {Using $\underline{x}$ for y in [3b]} $\Rightarrow$

$(\forall z)(p(z) \supset r(f(z)))$ {Paramodulation} $\Rightarrow$

[1] {Alphabetic variant}.

However, neither [1] nor [2] is an ideal translation. Consider what happens when a clause which originally contained a CHOICE is asserted as part of a theorem. If the quantifier resulting from the CHOICE is EX, it can be removed, and its variables replaced by Skolem constants; this is preferable to the FA case since the theorem prover is better equipped to deal with unquantified clauses. Whether EX or FA results at the clause level depends both on the original quantifier used to replace the CHOICE and on the number of times the subexpression is negated: neither EX nor FA aş the original quantifier can guarantee EX at the clause level. Therefore, since we have just proved that we can use either EX or FA and produce logically equivalent expressions, we invent a new "hybrid" quantifier FU which has the desirable properties of both EX and FA: ~(FU (BV x) (r x) (p x)) is (FU (BV x) (r x) ~(p x)), and FU can be eliminated at the clause level like EX. This hybrid quantifier is unique to PIVOT and is the result of many frustrating attempts to understand the semantics of CHOICE.

Source program

Each statement in the source program is translated into an internal representation in two steps. The parsing program produces an expression which uses the same operators as the canonical form, plus a few more (e.g., UGREATERP for >), but is not actually in this form. This is then put into canonical form during a preliminary sweep over the program before condition generation or verification is attempted. Both forms are saved with each statement, since the canonicalization of some expressions may depend on non-local semantic information (e.g., A(X) may be a function reference or a selection of a component from a record). For each type of source statement, there is an internal syntax, as follows:

Source	**Internal**
ASSERT $c_1, ..., c_n$	(ASSERT (AND c_1 ... c_n))
LEMMA $c_1, ..., c_n$	(LEMMA (AND c_1 ... c_n))
ASSUME $c_1, ..., c_n$	(ASSUME (AND c_1 ... c_n))
IF c THEN BEGIN	(BLKIF c)
IF c THEN body	(IF c body)
BEGIN	(BEGIN)
LOOP	(LOOP)
WHILE c: BEGIN	(BLKWHILE c)
REPEAT: BEGIN	(BLKREPEAT)
WHILE c: body	(WHILE c body)
ELSE BEGIN	(BLKELSE)
ELSE IF c THEN BEGIN	(BLKELSEIF c)
ELSE IF c THEN body	(ELSEIF c body)
ELSE body	(ELSE body)
END	(END)
DECLARE $c_1, ..., c_n$	(DECLARE (AND c_1 ... c_n))
CANCEL label	(CANCEL label)
body	(PROGN body)
...	(EMPTY)

"Body" is one or more of the following (separated by ; in the source program):

EXIT label	(EXIT label)
NEXT label	(NEXT label)
left-hand-side← expression	(SETQ left-hand-side expression)

In addition to the source text, the two expressions, and the statement number, there is one other important property permanently saved with each source statement: the set of declarations required for that statement to be legal. The only such checks currently required are that the first argument of a subscripting operation is an array and that the arguments of a field selection written with "." are a pointer and an appropriate selector name respectively.

HUMAN INTERFACE

Editing language

Since PIVOT is an interactive system for construction and verification of programs, its input language actually has three distinct components. The language in which programs are written was discussed in Chapter II. The language used for routine editing, filing, etc. of programs is discussed in this section. The language used to control the verification process is described in the next section. The syntax of the editing language appears in the following table.

```
<module-command> ::=
     PROCEDURE <module-name> <formal-parameters> |
     DECLARATIONS <module-name> <formal-parameters> |
     IN <module-name> |
     WHERE |
     MODULES |
     SAVE <file-name> $<module-name> |
     GET <file-name>

<module-name> ::= <id>

<file-name> ::= <id> | <str>

<edit-command> ::=
     DELETE {MODULE <module-name> | <text-range>} |
     RENUMBER [<text-range>] [FROM <num> [BY <num>]] |
     {COPY | MOVE} <text-range> TO <num> [BY <num>] |
     CHANGE <token> <token> <text-range> |
     LABEL <text-addr> [<id>]

<text-range> ::= <text-addr> [- <text-addr>]

<text-addr> ::= [: <module-name> :]
          {<statement-number> | <search> | <last-statement>}
     <statement-number> ::= <num>
     <search> ::= <str> [<offset>]
          <offset> ::= <num>
     <last-statement> ::= "$"

<token> ::= <id> | <num> | <str> | <punc>
```

As mentioned in Chapter II, the program under PIVOT's scrutiny at any moment is a collection of modules, where each module may be a procedure or a declaration module.

Creation of a new module is accomplished by simply typing the module header, i.e., the PROCEDURE or DECLARATIONS command. This also makes the module "current": one module is current at any given time, and the current module is used as the default in a number of editing and control commands such as LIST and PROVE. The IN command sets the current module; WHERE prints the name of the current module. SAVE saves a list of modules, or all modules if no list is given, on a file; GET reads a file prepared by SAVE and prints the module headers as they are encountered.

The editing language is similar to that of BASIC and not particularly original. However, it is important that an interactive system like PIVOT have good editing facilities that are oriented specifically to the manipulation of programs. Thus, for example, there is a special LABEL command to allow easy insertion or deletion of labels without having to retype the affected statement. Also, there are facilities which, though somewhat independent of both the particular command language and the functions it carries out, make an enormous difference in the "helpfulness" of the system as perceived by users. Two such facilities, namely the ability to *undo* commands already executed and the ability to *misspell* command names and still have them recognized, originated in BBN-LISP [Tei6] and are automatically available in PIVOT. UNDO (described in the BBN-LISP manual [Tei6a]) is limited to editing commands at present; automatic spelling correction is currently applicable to command names, and will be extended to statement keywords like ASSERT in the future.

This crude command set is intended as the nucleus of a much more sophisticated system for filing, locating, cataloging, cross-referencing, and administering a library of program modules in various stages of being developed or verified, since the author feels that a "Programmer's Interactive Verification and Organizational Tool" will only be useful if it provides such facilities. Their development has not been a major component of the author's research up to this point, however.

Executive language

The present executive (top-level) control language for PIVOT is rather simple. Its syntax follows:

```
<control-command> ::=
      <list-command> |
      STOP |
      PATHS [<proc-name>] |
      TRACE [<trace-code>] |
      PROVE [<context-index> | <proc-or-path>] |
      IPROVE <context-index>

<list-command> ::=
      LIST [<module-name> | <text-range>] |
      TH [<context-index> [. <clause-index>] |
            <proc-or-path>] |
      EFFORT [<proc-name> | <context-index>] |
      JUSTIFY [<proc-name> | <context-index>]

<proc-name> ::= <module-name>

<path> ::= # <num>

<context-index> ::= <num>

<clause-index> ::= <num>

<proc-or-path> ::= <proc-name> [path] | <path>

<trace-code> ::= V | T | B | P | S | C | NIL
```

Since PIVOT is embedded in BBN-LISP, it was deemed convenient to have a PIVOT command that sends control back to the LISP executive, namely STOP. Any input to PIVOT which does not begin with a PIVOT command word or a number (indicating a statement to be inserted in the current block) is also passed to LISP. The absence of this latter arrangement in the earliest versions of PIVOT caused great inconvenience, since it required constant transitions back and forth between PIVOT (trying new parts of the program) and LISP (fixing problems), and such transitions entailed losing some state information in the form of variable bindings.

As described in Chapter II and illustrated in figure VI-1, PIVOT's processing of a program is accomplished in a pipeline-like manner. To illustrate this process and the control commands, we will use King's Example 1:

```
100   ASSERT Y=B>=0
110   X ← 0
120   LOOP
125      WHILE Y#0: BEGIN
150      X ← X + A
160      Y ← Y - 1
170      ASSERT X=A*(B-Y) AND Y>=0
190      END
200   ASSERT X=A*B
```

The PATHS command moves a given block, the current block if none is specified, through the Canonicalizer and the Path Generator, and prints a list of paths through the program. The output from our example is:

```
#PATHS
759 conses, 12.404 seconds
PATH #1:  170-190-120-125(Y#0)-150...170
PATH #2:  170-190-120-125(Y=0)-200
PATH #3:  100...125(Y#0)-150...170
PATH #4:  100...125(Y=0)-200
```

The long computation time is mostly canonicalization: the path tracing process is very fast. Note that where a path goes through a decision statement (WHILE or IF), the outcome of the decision is printed as part of the path description.

PIVOT offers several levels of trace printout for the Theorem Generator and the Theorem Prover. These are identified by the trace-codes given to the TRACE command. (TRACE alone prints the current trace-code.) NIL turns off all tracing. The other codes select various options regarding printing deductions, clauses, values, and entries into data bases. (The trace-code C, which only prints a record of context generation, was used for the sample output below.) Tracing uses the BBN-LISP "advice" facility, so tracing costs nothing when not being used.

The SETUP command moves the procedure through the Theorem Generator. If a path is specified, only that path is set up. (This is mainly useful for debugging, since PIVOT records which paths have been set up and does not repeat the theorem generating process if no statements on a path have been changed.)

```
     #SETUP
     0 conses, .384 seconds
     Paths ready.
```
(Since no edits have occurred, PIVOT does not recompute the canonical forms or the paths, but simply retrieves them.)
```
     PATH #1: 170-190-120-125(Y#0)-150...170
     193 conses, 5.928 seconds
     NIL
```
(NIL indicates that the goal assertion at the end of this path, namely the evaluated form of statement 170, was deduced to be true by the Theorem Generator and therefore does not require use of the Theorem Prover. Normally the output would be a list of contexts which would have theorems set up in them for proof.)
```
     PATH #2: 170-190-120-125(Y=0)-200
     63 conses, 1.898 seconds
     NIL
```
(Path #2 is not significantly simpler than path #1, but the work done in traversing statements 170, 190, and 120 does not have to be repeated. This accounts for the shorter time.)
```
     PATH #3: 100...125(Y#0)-150...170
     185 conses, 5.354 seconds
     NIL
     PATH #4: 100...125(Y[0)-200
     52 conses, 1.916 seconds
     NIL
     Theorems ready.
```

In this example, all the goals were deduced to be true during theorem generation. While this situation is uncommon, it is common even in complex programs for one-third or more of the goals not to require use of the theorem prover.

The PROVE command allows selective invocation of the Theorem Prover on a procedure, a path, or a context. The reader is referred to Appendix D for an example of the output of this command. IPROVE (Interactive PROVE) is similar to PROVE; but allows detailed user control of the proof process, as described in the next section. The TH command lists

either a single clause in a given context, given its number, or: all the clauses in a given context; all the contexts generated for proofs at the end of a given path; or all the contexts generated for a given procedure.
In the example:

```
#TH

PATH #1: 170-190-120-125(Y#0)-150...170
No proofs
```

and similarly for paths 2, 3, and 4. The reader is again referred to Appendix D for more interesting output.

One of the interesting minor features of PIVOT, not mentioned in earlier chapters, is a certain amount of "introspection": with each context, PIVOT permanently retains the reason why that context was created and the amount of effort (time and CONSes) expended on the proof for that context if applicable. It is intended that PIVOT eventually use the knowledge, for example, that a given context was created for a user-supplied lemma, to prove the lemma before proving theorems which depend on the lemma, and ask the user for help if the proof of the lemma fails. At the moment, however, this information is retained only for the edification of the user. The JUSTIFY command prints the reasons for context generation, and the EFFORT command prints the effort expended on proofs. (The latter is not applicable to the small example: see Appendix D again.)

```
#JUSTIFY
  (188/1) for procedure K1
    (192/188) for paths beginning 100...120
      (194/192) assuming false in 125
      (193/192) assuming true in 125
    (189/188) for paths beginning 170-190-120
      (191/189) assuming false in 125
      (190/189) assuming true in 125
```

Note that, as mentioned before, PIVOT creates contexts for common initial segments of paths to avoid doing work twice.

This command language represents the minimum that the author needed to provide for himself for effective development of the rest of PIVOT. More elaborate facilities for reviewing the course of a proof will be required for a useful system. Also, some mechanism must be provided for the user to try out actual numerical values over sections of a program or in theorems which could not be proved, in an attempt to find a counterexample.

Proof control language

In addition to the executive language, PIVOT incorporates a proof control language which allows the user to monitor and direct individual proofs at a detailed level. The syntax of this language follows.

```
<proof-command> ::=
     <list-command> |
     CASES <clause-name> |
     EXPAND <clause-name> |
     ADD <clause-expression> |
     DELETE <clause-name> |
     INST <clause-name> <substitutions> |
     SET <substitutions> |
     MATCH <clause-name> <clause-name> |
     EVAL <clause-expression> |
     OK | STOP | NEXT | QUIT | TEST

<clause-name> ::= <context-index> . <clause-index>

<clause-expression> ::= <q-relation>

<substitutions> ::= <sub> ${, <sub>}

<sub> ::= <id> = <expression>
```

A proof may be forced into the interactive, user-controlled mode in two ways: by initiating it with IPROVE rather than PROVE, or by typing an interrupt character (control-H) at any time during the proof. (The latter forces the proof into interactive mode the next time the prover completes an application of any of the transformation procedures listed in Chapter III.) The OK command causes PIVOT to revert to automatic mode. Other commands which affect the flow of control are NEXT, which causes PIVOT to do one step (make one transformation) and return control to the user; STOP, which abandons the current subproof (of a proof by cases or function removal) and leaves the user in control at the next higher level; QUIT, which causes the entire proof to fail; and TEST, which creates a subproof (subcontext) so the user can experiment without fear.

There are two command subsets which allow the user to affect the set of clauses which constitute the theorem. CASES initiates a proof by cases of a designated (disjunction) clause. EXPAND expands special functions (division and MOD) in a given clause. MATCH invokes quantifier matching, and asks the user which instantiations he wants asserted if there is more than one. These commands essentially force PIVOT to employ specific theorem-proving techniques. The user can also intervene more directly, by using ADD to add a new clause, DELETE to remove an existing one, INST to add an instance of a quantified predicate, or SET to substitute values for variables throughout the theorem. He can also use EVAL to find the truth-value of a predicate without having to add it as a clause.

While this proof control language is primitive and relatively new to PIVOT, it has already proved useful in locating difficulties in particular proofs (not to mention bugs). It can also be used to help find counter-examples: to examine the consequences of setting X to 0, for example, one can use the command sequence:

```
$TEST
New (79/50) for testing
$SET X=0
```

If tracing is turned on, the user can see the transformations produced by the substitution of 0 for X.

Incremental aspects

To be useful, an interactive system must respond quickly to typed input. PIVOT currently requires about 1.5 CPU seconds to process a statement to be stored. This figure will probably drop to less than .5 seconds when PIVOT is compiled -- currently most of PIVOT is run in interpretive mode for ease of modification.

A more serious question is the speed and organization of PIVOT's computations. The 1.5 second figure includes parsing and storage, but not transformation to canonical form or any further processing. The times for the other stages were measured on King's Example 6, a modest program to move the greatest element of an array to the end by successive interchanges:

```
100   ASSERT N>0
110   I ← 2
200   LOOP
210       WHILE I<=N: BEGIN
300       IF A[I-1]>A[I] THEN BEGIN
400           X ← A[I]; A[I] ← A[I-1]; A[I-1] ← X
410           END
420       ASSERT .FA(1<=$K<I)A[I]>[A[K]
430       ASSERT I<=N
440       I := I+1
450       END
500   ASSERT .FA(1<=$L<N)A[N]>[A[L]
```

The paths through this program, as determined by PIVOT:

```
PATH #1: 420...450-200-210(I<=N)-300(A[I]<A[I-1])-400...430
PATH #2: 420...450-200-210(I<=N)-300(A[I]>=A[I-1])-420-430
PATH #3: 420...450-200-210(I>N)-500
PATH #4: 100...210(I<=N)-300(A[I]<A[I-1])-400...430
PATH #5: 100...210(I<=N)-300(A[I]>=A[I-1])-420-430
PATH #6: 100...210(I>N)-500
```

The timing results:

```
Parsing: 6.2 seconds           (Input from a file is
                                     faster than from the
                                     terminal.)
Canonicalization: 17.4 seconds
Path generation: 4.0 seconds
Theorem generation: 94.3 seconds, of which
           Evaluation: 23.5 seconds
           Entering clauses: 23.1 seconds
           Deduction: 15.0 seconds
Theorem proving: 11.4 seconds
```

Even though these times may diminish by a factor of 5 when PIVOT is compiled, they remain too high for repetition of the entire generation and proof process to be acceptable whenever a change is made in the program. For this reason, PIVOT is actually organized in a manner which minimizes repeated work. Each statement carries a flag to indicate whether it has been canonicalized; the flag is set when the canonicalization is done and reset when the statement is changed or first created. The paths produced by the Path Generator are expressed internally in terms of "path segments": the endpoints of a segment are groups of ASSERT statements, test statements (IFs or WHILEs), or the beginning or the end of the program. In the example, path #1 consists of three segments: 420-430-440-450-200, 210(I<=N), and 300(A[I]<A[I-1])-400-410-420-430. Path #2 also has three segments of which the first two are identical to the first two segments of path #1; path #3 has two segments, of which the first is the same as the first segment of path #1, and so on. Since PIVOT follows paths forward through the program, if a change is made in statement 400 only the last segment of paths #1 and #2 (and #4, as it happens) must be retraced by the theorem generator. Of course, the theorem prover must consider the resulting theorems de novo. This incremental organization did not save any time in debugging the examples in appendices B and D, which are rather complex loops with a single cluster of assertions, but it did save considerable time in the FIND program (Appendix E) from which statement 299 was originally omitted.

The ability to avoid superfluous processing is especially critical in the mixed-initiative situation envisioned in Floyd's IFIP paper (Appendix C), where the computer is expected to make comments on programs that are only partly specified. PIVOT allows for this type of interaction in a modest way with a special "..." statement, which is used to indicate that part of the program is missing and that theorem generation and proof should not be attempted for paths passing through that point. The analysis in Appendix C hopefully will persuade the reader that PIVOT is capable of carrying out the functions envisioned by Floyd; the proofs attached to the Appendix demonstrate that PIVOT possesses the necessary "intellectual" capabilities.

A different need for interaction arises when PIVOT fails to complete a proof. There are at least five reasons why this might occur:

(1) There is an error in the program being verified;

(2) There is an error in the assertions;

(3) PIVOT could have completed the proof, but the user grew impatient and interrupted it;

(4) PIVOT lacks the necessary theorem-proving capabilities;

(5) There is a bug in PIVOT itself.

PIVOT should provide the user with at least the following capabilities for exploring the cause of failure: change the program, add assertions to the particular theorem and retry the proof, retry the proof with additional trace output, tell PIVOT that he knows the theorem is true and PIVOT should take his word for it, try some test cases with real values, add specialized theorem-proving capabilities to PIVOT's repertoire, etc. All of these facilities are planned for PIVOT, but only the first three are currently available. Adding theorem-proving capabilities will only be possible in the limited sense of specifying some properties (e.g., commutative-associative) and simplification rules for new operators, and transformations on clauses involving new predicates; the latter will require some knowledge of the internal workings of PIVOT's theorem prover. The author believes that much of the trouble that arises with general-purpose theorem provers is that they have too many options and not enough knowledge, i.e., too many possible transformations and not enough algorithms for deciding when to apply them, and that interactive direction and algorithmic extension will prove sufficient for solving most real problems.

CONCLUSIONS

Accomplishments and failures

The author feels he has significantly expanded the range of programs which are amenable to automatic verification. The examples in the appendices are more complex than those in King's thesis, or any other published work; the Constable-Gries program in Appendix B is actually a program of some subtlety. PIVOT has also successfully verified the other examples in King's thesis, with the exception of Example 10 which was devised specifically to exercise a routine for handling systems of linear inequalities which King's system incorporates and PIVOT does not. Preliminary analysis of an even more complex program, Floyd's TREESORT3 [Floyd7] [Lon7], indicates that PIVOT will be able to handle it after minor improvements.

More important than PIVOT's successes is the manner in which they are achieved. PIVOT's proofs have a considerably more intuitive "feel" than King's, at least by this author's subjective standards. This was one of the major goals of the research. PIVOT is less inclined to suffer from combinatorial explosion on complex programs. King's Example 9 was presented in King's thesis missing a few critical assertions: King's program generated a few theorems of mind-boggling complexity and was unable to make any headway; PIVOT generated more theorems than with the correct assertions, but not many more, and reported failure to prove those (and only those) which were false. Considerably more care was taken in PIVOT to make its output comprehensible to the user: Appendix B provides an example. This extra care is the first step in making a smooth, helpful interactive system.

There are two respects in which PIVOT has failed to attain the goals set for it by the author. One is stability. Even though PIVOT is restricted to a fairly limited domain, each new test case has required adding simplification rules or extending the logic of PIVOT in some way. King apparently avoided this difficulty by ceasing development of his system when it reached a state capable of verifying a specific set of examples. However, the basic organization and methodology of PIVOT have not changed significantly in the past year, and the author hopes that the incremental incorporation of new techniques will gradually converge.

The other area of partial failure is generality. The simplification rules and data representation are tightly bound into the program. The former could be made accessible to the user, but only at a great cost in speed. While the author believes that much of the power of PIVOT comes from this specific orientation to a single problem domain, this specialization compromises potential extension to other domains. One direction for future development of PIVOT which the author hopes to pursue is the addition of specialized routines for other domains, such as the list-structure domain being studied by Burstall [Bur7], and the investigation of techniques for harmonious cooperation between the specialized packages.

Soundness and completeness

PIVOT's operations are obviously *sound* (in the mathematical sense) in that every transformation of an expression or a theorem preserves the value of the expression or the truth-value of the theorem. (The one exception is the function-splitting rule, which produces a stronger theorem; if the proof of this theorem fails, no harm is done, and other techniques are tried.) A more interesting question is to what extent they are *complete*, i.e., for what variety of inputs PIVOT is guaranteed to produce a definitive answer.

One difficulty arises at the level of expression representation. PIVOT uses a canonical form for arithmetic expressions, but it is not a normal form, i.e., two expressions which always have the same value may have different internal representations, such as (x+y)/2+(x-y)/2 and x. Theoretical research [Mos7] shows that a normal form does exist for the expressions that PIVOT uses, which are built up from addition, multiplication, division, and exponentiation, if the variables are allowed to be real numbers; the results on Hilbert's 10th problem mentioned below seem to imply that no normal form exists in the integer case. If a normal form does not exist, this means that even determining that two expressions are equivalent (proving $e_1-e_2=0$) may be undecidable. (If a normal form exists, then any two equivalent expressions would have the same normal form: King's thesis discusses this problem in more detail.) PIVOT's canonical form for arithmetic expressions is actually a normal form as long as division is not involved, but there are some expressions involving division (such as the example above) which PIVOT cannot reduce. The author conjectures that the peculiar semantics of integer division make it intrinsically difficult to find a normal form for expressions involving it.

There is an obvious normal form for logical expressions with no free variables: they are either "true" or "false"! This is no help, since of course the problem of determining whether a given such expression is true is only semi-decidable. Even in the more restricted domain of systems of linear inequalities involving only addition and multiplication of integers, the problem is very difficult: it is only in the last few years that Hilbert's 10th problem, to determine whether there is a solution to a set of *equalities* of this form, has been solved [Dav7]. PIVOT is relatively weak in its ability to solve linear systems, weaker than King's "linear solver".

The author believes that these partially theoretical limitations do not have much effect on PIVOT's ability to verify real programs. The complexity of programs such as those in the appendices arises from their control structure and their treatment of structured data, and not from mathematical difficulty. The author's approach to program verification is a pragmatic one: this makes it very difficult to characterize the set of provable programs.

Comments and prospects

The author feels that the progress achieved by PIVOT is the result of two differences in methodology from systems based strictly on theorem-proving power. One is the use of techniques borrowed from interpreters: the building up of name-value associations and the continuous application of simplifications. The other is the careful design of data structures specifically for holding knowledge about relationships in the specific value domain, the integers. The PIVOT experience tends to support the author's underlying philosophy: that methods which attempt to build coherent, structured images of the program's data universe will accomplish more than past methods which rely primarily on manipulating expressions.

Almost any project which claims to have produced an advance in artificial intelligence must deal with Dreyfus' assertion [Drey7] that such advances are to be compared with those of a man who climbs successively taller trees in the belief that he will eventually be able to climb to the moon. The author feels that neither Dreyfus' argument nor its denial have sufficient confirming evidence to be considered established, and that such arguments at the present time are founded on personal philosophical bias. The author's bias is to believe that his efforts are to be compared to the construction of aircraft, which also cannot reach the moon, but for less fundamental reasons!

Despite successes such as PIVOT's, current proof, verification, and planning systems are all severely limited in that they cannot deal effectively with the interactions between the various structures in an arbitrary, user-defined semantic universe. PIVOT, for example, has a great deal of knowledge about the universe of integer arithmetic "wired in", and performs badly when applied outside this domain. This knowledge appears both in the simplification rules for arithmetic expressions (e.g., X+-X=0), which could be converted to tabular form, and in the canonical representation form and the strategies used for organizing the data bases, which could not be tabularized. This is not happenstance: there is evidence that much of the power of human mathematical manipulation comes from the way we organize our knowledge as well as its specific content, but it is disturbing that the chief strength of PIVOT may also be its major weakness.

One area of current research, "automatic programming", is attempting to solve the generality problem by automating more of the process of choosing those data structures and processing algorithms appropriate to a particular problem domain. The author believes that this research will not yield useful results in the near future, and that it may well run into the same combinatorial problems as research into improving the power of theorem provers. Another approach, more modest and also being explored under the "automatic programming" rubric, is to try to gain the same kind of systematic, organized experience with other semantic domains as has been gained with the integers. A recent paper [Boy7] indicates a small step in this direction for LISP list structures. Examples of programs whose analysis might lead to further steps are London's LISP compilers [Lon7a], the Courtois-et-al synchronization problem [Cour7], or Knuth's storage allocators [Knuth7]. Some relevant work has already begun to appear in print [Bur7] [Dijk7] [Lev7] and the author hopes to work in this direction.

While these two assaults on the generality problem are trying to improve our ability to build program analyzers, others believe that the key to success is to change the nature of the programs being analyzed. Morris, for example [Mor7], shows that certain semantics for pointers lead to simpler proofs about programs than others. Scott [Scott7] is investigating a characterization of programs as abstract objects, more like the usual mathematical idea of partial functions, in an attempt to bring algebraic ideas to bear on the analysis of program behavior. This approach, as well as higher-level languages like APL, SETL [Schwartz7], and QA4, indicates a desire to reduce the level of detail in programs and make them easier to analyze by bringing the language concepts more in line with the higher-level concepts which occupy the programmer's mind as he writes. The author believes that much of the difficulty encountered by program analyzers, including verifiers, is that they must deduce the macro-intentions of the programmer from the micro-structure of the program, and that improving the expressive power of programming languages may help a great deal.

One interesting aspect of Scott's attempt to reduce the idea of a program to its mathematical essence is that it essentially removes the idea of "control" from programs and converts them to static objects. An early experiment [Cod7] in producing a totally control-free programming language, by defining processes in terms of sets and mappings rather than files and loops, was a practical failure but yielded some tantalizing results such as a description of a simple assembler in 19 equations [Katz7]. The "covering functions" approach of Wegbreit and Poupon [Weg7], which characterizes the state of list-processing programs by the set of nodes visited up to that point rather than by a sequential description of the computation, also has some of the same flavor. The author believes that this attitude of removing the notion of control is very promising; some preliminary experiments with list-following programs by the author suggest that viewing such programs in terms of the history of their accesses to data structures as a static record, rather than trying to analyze the dynamic progress of the computation, may greatly simplify their analysis.

BIBLIOGRAPHY

[Bal1]
Balzer, R. M.
Automatic Programming
Institute Technical Memorandum
University of Southern California
Information Sciences Institute
Sept. 1972
(rough draft)

[Bas1]
Baskin, H. B., Borgerson, B. R., Roberts, R.
PRIME - A modular architecture for terminal-oriented systems
Proceedings of the 1972 SJCC
AFIPS Press, N. J.
pp. 431 - 437

[Bel1]
Belady, L. A., and Lehman, M. M.
Programming System Dynamics, or
The Meta-Dynamics of Systems in Maintenance and Growth
report RC 3546
IBM T. J. Watson Research Center
Yorktown Heights, N. J.
Sept. 1971

[Boy1]
Boyer, R. S., and J. Strother Moore
Proving Theorems About LISP Programs
memo # 60
Department of Computational Logic
School of Artificial Intelligence
University of Edinburgh
May 1973

[Boy7]
see [Boy1]

[Buc1]
Buchanan, Jack
Stanford University
work on program synthesis
(private communication)

[Burl]
Burstall, R. M., and Topor, R.
· Mechanizing Program Correctness Proofs by Symbolic Interpretation
(note on work in progress -- privately circulated)
Nov. 1972

[Bur7]
Burstall, Rod M.
Some Techniques for Proving Correctness of Programs which Alter Data Structures
Machine Intelligence 7
Edinburgh University Press
Nov. 1972

[Cod7]
An Information Algebra: Phase I Report
Language Structure Group of the CODASYL Development Committee
CACM April 1962

[Con8]
Constable, Robert L., and David Gries
On classes of program schemata
SIAM Journal on Computing vol. 1 no. 1 (Mar. 1972)
pp. 66 - 118

[Cour7]
Courtois, P. J., F. Heymans, and D. L. Parnas
Concurrent Control with "READERS" and "WRITERS"
CACM vol. 14 no. 10 (Oct. 1971)
pp. 667 - 668

[Davl]
Hilbert's 10th Problem is Unsolvable
American Mathematical Monthly
vol. 80 no. 3
March 1973
pp. 233 - 269

[Dav7]
see [Davl]

[Dijkl]
Dijkstra, E. W.
Structured Programming
Software Engineering Techniques
J. N. Buxton and B. Randall (eds.)
Oct. 1969
pp. 84 - 88

[Dijk7]
Dijkstra, E. W.
A Constructive Approach to the Problem of Program Correctness
BIT 8
1968
pp. 174 - 186

[Drey7]
Dreyfus, Hubert L.
Alchemy and Artificial Intelligence
The Rand Corporation
publication P-3244
Dec. 1965

[Floyd]
Floyd, Robert W.
Toward interactive design of correct programs
Proceedings of IFIP Congress 71
North Holland Publishing Co.
Amsterdam, Netherlands
Aug. 1971

[Floyd7]
Floyd, Robert W.
Algorithm 245, TREESORT3
CACM 7 (Dec. 1964)
p. 701

[Ger4]
Gerhart, Susan
Verification of APL programs
Ph.D. thesis
Carnegie-Mellon University
Pittsburgh, Pa.
Nov. 1972

[Good]
Good, D. I.
Toward a Man-Machine System for Proving Program Correctness
Ph.D. thesis
Department of Computer Science
University of Wisconsin
Madison, Wisconsin
1970

[Guard1]
Guard, J. R., F. C. Oglesby, J. H. Bennett, L. G. Settle
Semi-Automated Mathematics
JACM vol. 16 no. 1 (Jan. 1969)
pp. 49 - 62

[Hew2]
Hewitt, Carl
PLANNER: A Language for Proving Theorems in Robots
Proceedings of the IJCAI
May 1969

[Hoare]
Hoare, C. A. R.
Algorithm 65: FIND
CACM vol. 4 no. 7 (July 1961)
p. 321

[Hoare1]
Hoare, C. A. R.
Procedures and Parameters: an axiomatic approach
Symposium on Semantics of Algorithmic Languages
E. Engeler (ed.)
Springer-Verlag
1971
pp. 102 - 116

[Iv4]
Iverson, King's Example E.
A Programming Language
John Wiley and Sons, New York
1962

[Katz7]
Katz, Jesse, and McGee, William C.
An Experiment in Non-Procedural Programming
1963 FJCC
Spartan Books, Inc.
Baltimore, Md.
pp. 1 - 13

[King]
King, J. C.
A program verifier
Ph.D. thesis
Carnegie-Mellon University
Pittsburgh
Sept. 1969

[Knuth7]
Knuth, Donald C.
The Art of Computer Programming
Vol. I - Fundamental Algorithms
Addison Wesley, Reading, Mass.
1968

[Lev7]
Levitt, Karl N.
The Application of Program-Proving Techniques to the Verification of Synchronization Processes
Stanford Research Institute Report
Menlo Park, Calif.
May 1972

[Lon7]
London, Ralph L.
Proof of Algorithms: A new kind of certification
(Certification of Algorithm 245: TREESORT3)
CACM vol. 13, no. 6, June 1970
pp. 371 - 373

[Lon7a]
London, Ralph L.
Correctness of two compilers for a LISP subset
Stanford Artificial Intelligence Project memo AIM-157
Stanford University
Oct. 1971

[Luck1]
Luckham, D.
Stanford University
work on user-aided proofs
(private communication)

[Mccar2]
McCarthy, John, et al.
LISP 1.5 Programmer's Manual
M.I.T. Press
Cambridge, Mass.
1962

[Mil1]
Milner, R.
Stanford University
work on user-aided proofs
(private communication)

[Mills1]
Mills, H. D.
Structured Programming in Large Systems
Debugging Techniques in Large Systems
R. Rustin (ed.)
Prentice Hall Inc.
Englewood Cliffs, N. J.
pp. 41 - 55

[Mor1]
Morris, James H.
Verification-Oriented Language Design
Technical Report #7
Computer Science Department
University of California, Berkeley
Dec. 1972

[Mor5]
Morris, Robert
Scatter Storage Techniques
Communications of the ACM
vol. 11 no. 1
Jan. 1968
pp. 38 - 44

[Mor7]
see [Mor1]

[Mos7]
Moses, Joel
Symbolic Integration
Ph. D. thesis
Massachusetts Institute of Technology
Sept. 1967
pp. 160 - 171

[Naur1]
Naur, Peter
Proof of algorithms by general snapshots
BIT 6, 4 (1966)
pp. 310 - 316

[Rab1]
Fischer, Michael, and Rabin, Michael
Absolutely unsolvable problems in algebra and arithmetic
Proceedings of Symposium on Complexity of Real Computational
Processes
New York, April, 1973
to appear in AMS-SIAM series of symposia in Applied Mathematics

[Robl]
Robinson, J. A.
(private communication)

[Robla]
Robinson, J. A.
A machine-oriented logic based on the resolution principle
JACM vol. 12, pp. 23 - 41
Jan. 1965

[Rob2]
Robinson, G., and Wos, L.
Paramodulation and Theorem Proving in First Order Theories With Equality
Machine Intelligence 4
B. Meltzer and D. Michie (eds.)
American Elsevier Publishing Co.
New York
1969
pp. 133 - 150

[Rull]
Rulifson, J. F., Derksen, J. A., and Waldinger, R. J.
QA4: A Procedure Calculus for Intuitive Reasoning
Stanford Research Institute Technical Report
Nov. 1972

[Schwartz7]
Schwartz, J. T.
Abstract Algorithms and a Set-Theoretic Language for Their Expression
Computer Science Department
Courant Institute
New York University
1970 - 71

[Scott7]
Scott, Dana
Outline of a mathematical theory of computation
Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems
1970

[Sla4]
Slagle, James R., and Norton, Lewis M.
Experiments with an Automatic Theorem Prover Having Partial Ordering Rules
Heuristics Laboratory
Division of Computer Research and Technology
National Institutes of Health
Department of Health, Education and Welfare
Bethesda, Maryland 20014

[Sus2]
McDermott, V., and Sussman, G. J.
The CONNIVER Reference Manual
M.I.T. Artificial Intelligence Laboratory
Memo no. 259
Cambridge, Mass.
May 1972

[Tei6]
Teitelman, Warren
Automated programmering - The programmer's assistant
Proceedings of the 1972 FJCC
pp. 917 - 921

[Tei6a]
W. Teitelman, D. G. Bobrow, A. K. Hartley, D. L. Murphy
BBN-LISP TENEX Reference Manual
Bolt Beranek and Newman Inc.
Cambridge, Mass.
August 1972

[Wal1]
Waldinger, Richard J.
Constructing Programs Automatically Using Theorem Proving
Ph.D. thesis
Computer Science Department
Carnegie-Mellon University
Pittsburgh
May 1969

[Weg1]
Wegbreit, Ben
Heuristic Methods for Mechanically Deriving Inductive Assertions
Bolt, Beranek and Newman Inc.
Cambridge, Mass.
Feb. 1973

[Weg2]
Wegbreit, Ben
(private communication)

[Weg7]
Wegbreit, B., and Poupon, J.
Covering Functions
Center for Research in Computing Technology
Harvard University
Cambridge, Mass.
Sept. 1972

[Wirth1]
Wirth, Niklaus, and Hoare, C. A. R.
A Contribution to the Development of ALGOL
CACM vol. 9 no. 6 (June 1966)
pp. 413 - 431

APPENDIX A

Example with Key to Trace Output

This Appendix contains a complete proof of King's Example 6, which moves the largest element of an array to the end by successive interchanges. Its purpose is to give the reader some feel for PIVOT's mental processes and to supply some background for the proofs in Appendices B and C. Familiarity with the material in Chapters III and IV will be helpful in following these proofs. The following comments refer to the line numbers attached to the trace output which begins on page A-4.

(26) A new context is created for the processing of an initial segment of path #1. This saves work, since path #3 (line 174) has the same initial segment.

(27) Lines like this announce which statement is currently under the scrutiny of the theorem generator.

(28) This indicates that the context whose first two indices are 3 and 2 is current. The full index list is only printed when the context is created.

(29)-(31) These lines result from the process described at the bottom of page III-16. Lines like (30)-(31) record the addition of clauses: the first number is the context index, the second is a serial number within the context.

(38) This notes an assignment to a scalar variable.

(45) The value assigned to I on line 38 is retrieved in the course of evaluating the WHILE condition I<=N.

(46) This is the result of the evaluation.

(47) A new subcontext is needed to distinguish the path where the test succeeds (#1) from the one where the test fails (#3); see lines (174)-(179).

(48)-(49) The new clause implies an old one, which is dropped.

(53) This shows the result of evaluating an array reference. No assignments have been made to A yet.

(73) This records an assignment to an array element.

(77)-(78) Since the only entry on A's ELLIST is for A[I+1], and I≠I+1, no case analysis is required to evaluate A[I-1] and return A[I].

(89) This is an application of the range-splitting technique discussed in Chapter III. Line 100 is the other case.

(95)-(97) These are deductions from RC$2, as described in Chapter IV.

(98)-(99) The goal evaluated to T, so the proof for this context is actually complete even though the theorem prover was never invoked.

(102) This is an implicit assignment resulting from (101).

(117) No special assumptions like (89)-(92) were made, but the goal evaluated to T anyway.

(120)-(123) The theorem prover found that the proofs were already complete.

(192) This is a deduction from ULBL, as described in Chapter IV.

(207)-(217) The theorem prover was required, but the proof was trivial. The clauses involved were 12.1 (line 182), 13.3 (line 193), and 13.4 (line 198).

(396)-(427) This displays the context tree with the justifications for each context. Six theorems were generated, of which only one required use of the theorem prover.

```
 (1)  #GET K6
 (2)  K6.PIVOT;3
 (3)  K6
 (4)  New (2/1) for procedure K6
 (5)  815 conses, 8.139 seconds
 (6)  T

 (7)  #LIST
 (8)  PROCEDURE K6(A,N)
 (9)   100   ASSERT N>0
(10)   110   I ← 2
(11)   200   LOOP
(12)   210      WHILE I<=N: BEGIN
(13)   300      IF A[I-1]>A[I] THEN BEGIN
(14)   400         X ← A[I]; A[I] ← A[I-1]; A[I-1] ← X
(15)   410         END
(16)   420      ASSERT .FA(1<=$K<I)A[I]>=A[K]
(17)   430      ASSERT I<=N
(18)   440      I ← I+1
(19)   450      END
(20)   500   ASSERT .FA(1<=$L<N)A[N]>=A[L]

(21)  #TRACE T

(22)  #PROVE
(23)  894 conses, 18.801 seconds
(24)  Paths ready.

(25)  Path #1: 420...450-200-210(I<=N)-300(A[I]<A[I-1])...430
(26)  New (3/2/1) for paths beginning 420...450-200
(27)   420   ASSERT .FA(1<=$K<I)A[I]>=A[K]
(28)  In (3/2):
(29)  New (4/3/2/1) to evaluate .FA(K:K>0 & I>K)A[I]>=A[K]
(30)     4.1    K>0
(31)     4.2    I>K
(32)     3.1    .FA(K:K>0 & I>K)A[I]>=A[K]
(33)   430   ASSERT I<=N
(34)  In (3/2):
(35)     3.2    I<=N
(36)   440   I ← I+1
(37)  In (3/2):
(38)  Set CVAL(I): I+1
(39)   450   END
(40)  In (3/2):
(41)   200   LOOP
(42)  In (3/2):
(43)   210   WHILE I<=N: BEGIN
(44)  In (3/2):
(45)  CVAL(I) = I+1
(46)  Eval: I<=N => I<N
(47)  New (5/3/2/1) assuming TRUE in 210
(48)  Cancel 3.2: I<=N
```

```
(49)     5.1    I<N
(50)   300    IF A[I-1]>A[I] THEN BEGIN
(51)  In (5/3):
(52)  CVAL(I) = I+1
(53)  Eval: A[I] => A[I+1]
(54)  CVAL(I) = I+1
(55)  Eval: A[I-1] => A[I]
(56)  Eval: A[I]<A[I-1] => A[I]>A[I+1]
(57)  New (6/5/3/2/1) assuming TRUE in 300
(58)     6.1    A[I]>A[I+1]
(59)   400    X ← A[I]; ...
(60)  In (6/5):
(61)  CVAL(I) = I+1
(62)  Eval: A[I] => A[I+1]
(63)  Eval: A[I] => A[I+1]
(64)  Set CVAL(X): A[I+1]
(65)   400             ... A[I] ← A[I-1]; ...
(66)  In (6/5):
(67)  CVAL(I) = I+1
(68)  Eval: A[I] => A[I+1]
(69)  Eval: A[I] => A[I+1]
(70)  CVAL(I) = I+1
(71)  Eval: A[I-1] => A[I]
(72)  Eval: A[I-1] => A[I]
(73)  Add ELLIST((ARRAY A)): (A[I+1] A[I])
(74)   400                              ... A[I-1] ← X
(75)  In (6/5):
(76)  CVAL(I) = I+1
(77)  Eval: A[I-1] => A[I]
(78)  Eval: A[I-1] => A[I]
(79)  CVAL(X) = A[I+1]
(80)  Eval: X => A[I+1]
(81)  Add ELLIST((ARRAY A)): (A[I] A[I+1])
(82)   410    END
(83)  In (6/5):

(84)   420    ASSERT .FA(1<=$K<I)A[I]>=A[K]
(85)  In (6/5):
(86)  Goal: .FA(K:K>0 & I>K)A[I]>=A[K]
(87)  CVAL(I) = I+1
(88)  Eval: K>0 & I>K => K>0 & I>=K
(89)  New (7/6/5/3/2/1) for .FA goal 420, case K>0 & I>K [in old
      range]
(90)     7.1    K>0
(91)     7.2    I>K
(92)     7.3    A[I]>=A[K]
(93)  In (7/6):
(94)  CVAL(I) = I+1
(95)  From 7.2: I=K => NIL
(96)  From 7.2: I=K-1 => NIL
(97)  From 7.3: A[I]>=A[K] => T
(98)  Eval: A[I]>=A[K] => T
(99)  Success in (7/6)
```

```
(100)  New (8/6/5/3/2/1) for .FA goal 420, case I=K & K>0 [not in
       old range]
(101)     8.1    I=K
(102)  Set CVAL(K): I
(103)     8.2    K>0
(104)  In (8/6):
(105)  CVAL(I) = I+1
(106)  CVAL(K) = I
(107)  Eval: A[K] => A[I+1]
(108)  From 6.1: A[I]>=A[I+1] => T
(109)  Eval: A[I]>=A[K] => T
(110)  Success in (8/6)

(111)   430   ASSERT I<=N
(112)  In (6/5):
(113)  Goal: I<=N
(114)  CVAL(I) = I+1
(115)  From 5.1: I<N => T
(116)  Eval: I<=N => T
(117)  No proof needed, Eval(goal) = T
(118)  881 conses, 40.993 seconds
(119)  ((7 6 5 3 2 1) (8 6 5 3 2 1))

(120)  Proof for (7/6) for .FA goal 420, case K>0 & I>K [in old
       range]
(121)  Proved

(122)  Proof for (8/6) for .FA goal 420, case I=K & K>0 [not in old
       range]
(123)  Proved

(124)  Path #2: 420...450-200-210(I<=N)-300(A[I]>=A[I-1])-420-430
(125)   300   IF A[I-1]>A[I] THEN BEGIN
(126)  In (5/3):
(127)  CVAL(I) = I+1
(128)  Eval: A[I] => A[I+1]
(129)  CVAL(I) = I+1
(130)  Eval: A[I-1] => A[I]
(131)  Eval: A[I]<A[I-1] => A[I]>A[I+1]
(132)  New (9/5/3/2/1) assuming FALSE in 300
(133)     9.1    A[I]<=A[I+1]
```

```
(134)   420   ASSERT .FA(1<=$K<!)A[I]>=A[K]
(135)  In (9/5):
(136)  Goal: .FA(K:K>0 & I>K)A[I]>=A[K]
(137)  CVAL(I) = I+1
(138)  Eval: K>0 & I>K => K>0 & I>=K
(139)  New (10/9/5/3/2/1) for .FA goal 420, case K>0 & I>K [in old
       range]
(140)    10.1    K>0
(141)    10.2    I>K
(142)    10.3    A[I]>=A[K]
(143)  In (10/9):
(144)  CVAL(I) = I+1
(145)  Eval: A[I] => A[I+1]
(146)  From 10.3 & 9.1: A[K]<=A[I+1] => T
(147)  Eval: A[I]>=A[K] => T
(148)  Success in (10/9)
(149)  New (11/9/5/3/2/1) for .FA goal 420, case I=K & K>0 [not in
       old range]
(150)    11.1    I=K
(151)  Set CVAL(K): I
(152)    11.2    K>0
(153)  In (11/9):
(154)  CVAL(I) = I+1
(155)  Eval: A[I] => A[I+1]
(156)  CVAL(K) = I
(157)  Eval: A[K] => A[I]
(158)  From 9.1: A[I]<=A[I+1] => T
(159)  Eval: A[I]>=A[K] => T
(160)  Success in (11/9)

(161)   430   ASSERT I<=N
(162)  In (9/5):
(163)  Goal: I<=N
(164)  CVAL(I) = I+1
(165)  From 5.1: I<N => T
(166)  Eval: I<=N => T
(167)  No proof needed, Eval(goal) = T
(168)  476 conses, 21.236 seconds
(169)  ((10 9 5 3 2 1) (11 9 5 3 2 1))

(170)  Proof for (10/9) for .FA goal 420, case K>0 & I>K [in old
       range]
(171)  Proved
```

```
(172)  Proof for (11/9) for .FA goal 420, case I=K & K>0 [not in
       old range]
(173)  Proved

(174)  Path #3: 420...450-200-210(I>N)-500
(175)   210   WHILE I<=N: BEGIN
(176)  In (3/2):
(177)  CVAL(I) = I+1
(178)  Eval: I<=N => I<N
(179)  New (12/3/2/1) assuming FALSE in 210
(180)  From 3.2: I>=N => I=N
(181)  Cancel 3.2: I<=N
(182)    12.1    I=N
(183)  Set CVAL(N): I

(184)   500   ASSERT .FA(1<=$L<N)A[N]>=A[L]
(185)  In (12/3):
(186)  Goal: .FA(L:L>0 & L<N)A[L]<=A[N]
(187)  CVAL(N) = I
(188)  Eval: L>0 & L<N => L>0 & I>L
(189)  New (13/12/3/2/1) for .FA goal 500, case L>0 & I>L & L<N [in
       old range]
(190)    13.1    L>0
(191)    13.2    I>L
(192)  From 13.2 & 12.1: L<N => T
(193)    13.3    A[L]<=A[N]
(194)  In (13/12):
(195)  CVAL(N) = I
(196)  Eval: A[N] => A[I]
(197)  Eval: A[L]<=A[N] => A[I]>=A[L]
(198)    13.4    A[I]<A[L]
(199)  New (14/12/3/2/1) for .FA goal 500, case L>0 & I>L & L>=N
(200)   [not in old range]
(201)    14.1    L>0
(202)    14.2    I>L
(203)  From 14.2 & 12.1: L>=N => NIL
(204)  Success in (14/12)
(205)  467 conses, 20.323 seconds
(206)  ((13 12 3 2 1))

(207)  Proof for (13/12) for .FA goal 500, case L>0 & I>L & L<N [in
       old range]:
(208)  New (15/13/12/3/2/1) for proof
(209)  Proving (15/13)...
(210)  Equalities:
(211)    (from 12.1)  N => I
(212)  Cancel 12.1: I=N
(213)  Cancel 13.3: A[L]<=A[N]
(214)  From 13.4: A[I]>=A[L] => NIL
(215)  Success in (15/13)
(216)  Proved: 86 conses, 2.281 seconds
(217)  Success in (13/12)
```

```
(218)  Path #4: 100...200-210(I<=N)-300(A[I]<A[I-1])...430
(219)  New (16/2/1) for paths beginning 100...200
(220)   100   ASSERT N>0
(221)  In (16/2):
(222)    16.1     N>0
(223)   110    I ← 2
(224)  In (16/2):
(225)  Set CVAL(I): 2
(226)   200   LOOP
(227)  In (16/2):
(228)   210   WHILE I<=N: BEGIN
(229)  In (16/2):
(230)  CVAL(I) = 2
(231)  Eval: I<=N => N>=2
(232)  New (17/16/2/1) assuming TRUE in 210
(233)  Cancel 16.1: N>0
(234)    17.1     N>=2
(235)   300    IF A[I-1]>A[I] THEN BEGIN
(236)  In (17/16):
(237)  CVAL(I) = 2
(238)  Eval: A[I] => A[2]
(239)  CVAL(I) = 2
(240)  Eval: A[I-1] => A[1]
(241)  Eval: A[I]<A[I-1] => A[1]>A[2]
(242)  New (18/17/16/2/1) assuming TRUE in 300
(243)    18.1     A[1]>A[2]
(244)   400    X ← A[I]; ...
(245)  In (18/17):
(246)  CVAL(I) = 2
(247)  Eval: A[I] => A[2]
(248)  Eval: A[I] => A[2]
(249)  Set CVAL(X): A[2]
(250)   400           ... A[I] ← A[I-1]; ...
(251)  In (18/17):
(252)  CVAL(I) = 2
(253)  Eval: A[I] => A[2]
(254)  Eval: A[I] => A[2]
(255)  CVAL(I) = 2
(256)  Eval: A[I-1] => A[1]
(257)  Eval: A[I-1] => A[1]
(258)  Add ELLIST((ARRAY A)): (A[2] A[1])
(259)   400                           ... A[I-1] ← X
(260)  In (18/17):
(261)  CVAL(I) = 2
(262)  Eval: A[I-1] => A[1]
(263)  Eval: A[I-1] => A[1]
(264)  CVAL(X) = A[2]
(265)  Eval: X => A[2]
(266)  Add ELLIST((ARRAY A)): (A[1] A[2])
(267)   410   END
(268)  In (18/17):
```

```
(269)   420   ASSERT .FA(1<=$K<I)A[I]>=A[K]
(270)  In (18/17):
(271)  Goal: .FA(K:K>0 & I>K)A[I]>=A[K]
(272)  CVAL(I) = 2
(273)  Eval: K>0 & I>K => K=1
(274)  New (19/18/17/16/2/1) for .FA goal 420, case K=1 & I>K [in
       old range]
(275)    19.1    K=1
(276)  Set CVAL(K): 1
(277)    19.2    I>K
(278)    19.3    A[I]>=A[K]
(279)  In (19/18):
(280)  CVAL(I) = 2
(281)  Eval: A[I] => A[1]
(282)  CVAL(K) = 1
(283)  Eval: A[K] => A[2]
(284)  From 18.1: A[1]>=A[2] => T
(285)  Eval: A[I]>=A[K] => T
(286)  Success in (19/18)
(287)  New (20/18/17/16/2/1) for .FA goal 420, case K=1 & I<=K
(288)   [not in old range]
(289)    20.1    K=1
(290)  Set CVAL(K): 1
(291)    20.2    I<=K
(292)  In (20/18):
(293)  CVAL(I) = 2
(294)  Eval: A[I] => A[1]
(295)  CVAL(K) = 1
(296)  Eval: A[K] => A[2]
(297)  From 18.1: A[1]>=A[2] => T
(298)  Eval: A[I]>=A[K] => T
(299)  Success in (20/18)

(300)   430   ASSERT I<=N
(301)  In (18/17):
(302)  Goal: I<=N
(303)  CVAL(I) = 2
(304)  From 17.1: N>=2 => T
(305)  Eval: I<=N => T
(306)  No proof needed, Eval(goal) = T
(307)  534 conses, 28.367 seconds
(308)  ((19 18 17 16 2 1) (20 18 17 16 2 1))
```

```
(309)  Proof for (19/18) for .FA goal 420, case K=1 & I>K [in old
       range]
(310)  Proved

(311)  Proof for (20/18) for .FA goal 420, case K=1 & I<=K [not in
       old range]
(312)  Proved

(313)  Path #5: 100...200-210(I<=N)-300(A[I]>=A[I-1])-420-430
(314)   300   IF A[I-1]>A[I] THEN BEGIN
(315)  In (17/16):
(316)  CVAL(I) = 2
(317)  Eval: A[I] => A[2]
(318)  CVAL(I) = 2
(319)  Eval: A[I-1] => A[1]
(320)  Eval: A[I]<A[I-1] => A[1]>A[2]
(321)  New (21/17/16/2/1) assuming FALSE in 300
(322)    21.1    A[1]<=A[2]

(323)   420   ASSERT .FA(1<=$K<I)A[I]>=A[K]
(324)  In (21/17):
(325)  Goal: .FA(K:K>0 & I>K)A[I]>=A[K]
(326)  CVAL(I) = 2
(327)  Eval: K>0 & I>K => K=1
(328)  New (22/21/17/16/2/1) for .FA goal 420, case K=1 & I>K [in
       old range]
(329)    22.1    K=1
(330)  Set CVAL(K): 1
(331)    22.2    I>K
(332)    22.3    A[I]>=A[K]
(333)  In (22/21):
(334)  CVAL(I) = 2
(335)  Eval: A[I] => A[2]
(336)  CVAL(K) = 1
(337)  Eval: A[K] => A[1]
(338)  From 21.1: A[1]<=A[2] => T
(339)  Eval: A[I]>=A[K] => T
(340)  Success in (22/21)
(341)  New (23/21/17/16/2/1) for .FA goal 420, case K=1 & I<=K
(342)   [not in old range]
(343)    23.1    K=1
(344)  Set CVAL(K): 1
(345)    23.2    I<=K
(346)  In (23/21):
(347)  CVAL(I) = 2
(348)  Eval: A[I] => A[2]
(349)  CVAL(K) = 1
(350)  Eval: A[K] => A[1]
(351)  From 21.1: A[1]<=A[2] => T
(352)  Eval: A[I]>=A[K] => T
(353)  Success in (23/21)
```

```
(354)   430   ASSERT I<=N
(355)  In (21/17):
(356)  Goal: I<=N
(357)  CVAL(I) = 2
(358)  From 17.1: N>=2 => T
(359)  Eval: I<=N => T
(360)  No proof needed, Eval(goal) = T
(361)  323 conses, 19.579 seconds
(362)  ((22 21 17 16 2 1) (23 21 17 16 2 1))

(363)  Proof for (22/21) for .FA goal 420, case K=1 & I>K [in old
       range]
(364)  Proved

(365)  Proof for (23/21) for .FA goal 420, case K=1 & I<=K [not in
       old range]
(366)  Proved

(367)  Path #6: 100...200-210(I>N)-500
(368)   210   WHILE I<=N: BEGIN
(369)  In (16/2):
(370)  CVAL(I) = 2
(371)  Eval: I<=N => N>=2
(372)  New (24/16/2/1) assuming FALSE in 210
(373)  Cancel 16.1: N>0
(374)    24.1    N=1

(375)   500   ASSERT .FA(1<=$L<N)A[N]>=A[L]
(376)  In (24/16):
(377)  Goal: .FA(L:L>0 & L<N)A[L]<=A[N]
(378)  New (25/24/16/2/1) for goal 500
(379)    25.1    L>0
(380)    25.2    L<N
(381)  In (25/24):
(382)    25.3    A[L]>A[N]
(383)  149 conses, 8.112 seconds
(384)  ((25 24 16 2 1))

(385)  Proof for (25/24) for goal 500:
(386)  New (26/25/24/16/2/1) for proof
(387)  Proving (26/25)...
(388)  Equalities:
(389)    (from 24.1)  N => 1
(390)  Cancel 24.1: N=1
(391)  Cancel 25.2: L<N
(392)  From 25.1: L<=0 => NIL
(393)  Success in (26/25)
(394)  Proved: 56 conses, 1.805 seconds
(395)  Success in (25/24)
```

```
(396)  #JUSTIFY
(397)    (2/1) for procedure K6
(398)      (16/2) for paths beginning 100...200
(399)        (24/16) assuming FALSE in 210
(400)          (25/24) for goal 500
(401)                  Proved
(402)            (26/25) for proof
(403)                    Proved: 56 conses, 1.805 seconds
(404)        (17/16) assuming TRUE in 210
(405)          (21/17) assuming FALSE in 300
(406)            (23/21) for  FA goal 420, case K=1 & I<=K [not in
       old range]
(407)                    Proved
(408)            (22/21) for  FA goal 420, case K=1 & I>K [in old
       range]
(409)                    Proved
(410)          (18/17) assuming TRUE in 300
(411)            (20/18) for .FA goal 420, case K=1 & I<=K [not in
       old range]
(412)                    Proved
(413)            (19/18) for  FA goal 420, case K=1 & I>K [in old
       range]
(414)                    Proved
(415)      (3/2) for paths beginning 420...450-200
(416)        (12/3) assuming FALSE in 210
(417)          (13/12) for .FA goal 500, case L>0 & I>L & L<N [in
       old range]
(418)                  Proved
(419)            (15/13) for proof
(420)                    Proved: 86 conses, 2.281 seconds
(421)        (5/3) assuming TRUE in 210
(422)          (9/5) assuming FALSE in 300
(423)            (11/9) for .FA goal 420, case I=K & K>0 [not in
       old range]
(424)            (10/9) for .FA goal 420, case K>0 & I>K [in old
       range]
(425)          (6/5) assuming TRUE in 300
(426)            (8/6) for .FA goal 420, case I=K & K>0 [not in old
       range]
(427)            (7/6) for .FA goal 420, case K>0 & I>K [in old
       range]
```

```
#MODULES
PROCEDURE CG

#LIST
PROCEDURE CG
   1   DECLARE ARRAY A, AS, AD
   2   LET PAIR(X,Y)=(X+Y+1)*(X+Y)/2+Y+1
 100   K ← 0
 110   I ← 0
 120   A[0] ← 0
 130   AS[0] ← 0
 140   AD[0] ← 0
 150   LOOP
 160      ASSERT .FA(J:0<=J<=K)(A[J]=J)
 170      ASSERT .FA(J:1<=J<=K)(AD[J]=J-1)
 180      ASSERT K=PAIR(I,AS[0]-I)-1
 190      ASSERT 0<=I<=AS[0]<=K
 200      ASSERT .FA(J:0<=J<=I)(AS[J]+J=AS[0])
 210      ASSERT .FA(J:I<J<=AS[0])(AS[J]+J=AS[0]+1)
 215      ASSERT .FA(J:AS[0]<J<=K)(AS[J]=0)
 220      REPEAT: BEGIN
 230      K ← K+1
 250      A[K] ← PAIR(A[I],A[AS[I]])
 260      AS[K] ← 0
 270      AD[K] ← K-1
 280      AS[I] ← AS[I]+1
 290      AD[0] ← AS[0]
 300      I ← AD[I]
 310      END

#PATHS
1 conses, 1.482 seconds
Path #1: 160...310-150...215
Path #2: 1...215

#PROVE
1 conses, 1.301 seconds
Paths ready.

Path #1: 160...310-150...215
New (3/2/1) for paths beginning 160...310-150...215
 160   ASSERT .FA(J:0<=J<=K)(A[J]=J)
In (3/2):
New (4/3/2/1) to evaluate .FA(J:J>=0 & J<=K)J=A[J]
   4.1    J>=0
   4.2    J<=K
   3.1    .FA(J:J>=0 & J<=K)J=A[J]
Add ELLIST((ARRAY A)): ($$ $$>=0 & $$<=K $$)
 170   ASSERT .FA(J:1<=J<=K)(AD[J]=J-1)
In (3/2):
New (5/3/2/1) to evaluate .FA(J:J>0 & J<=K)J=AD[J]+1
   5.1    J>0
   5.2     J<=K
   3.2     .FA(J:J>0 & J<=K)J=AD[J]+1
Add ELLIST((ARRAY AD)): ($$ $$>0 & $$<=K $$-1)
 180   ASSERT K=PAIR(I,AS[0]-I)-1
In (3/2):
Eval: K=PAIR(I,AS[0]-I)-1
    => 2*I+2*K=3*AS[0]+AS[0]↑2-1 ! 2*I+2*K=3*AS[0]+AS[0]↑2
   3.3     2*I+2*K=3*AS[0]+AS[0]↑2-1 ! 2*I+2*K=3*AS[0]+AS[0]↑2
 190   ASSERT 0<=I<=AS[0]<=K
In (3/2):
   3.4     I>=0
   3.5     K>=AS[0]
   3.6     I<=AS[0]
 200   ASSERT .FA(J:0<=J<=I)(AS[J]+J=AS[0])
In (3/2):
New (6/3/2/1) to evaluate .FA(J:J>=0 & I>=J)J+AS[J]=AS[0]
   6.1     J>=0
   6.2     I>=J
   3.7     .FA(J:J>=0 & I>=J)J+AS[J]=AS[0]
Add ELLIST((ARRAY AS)): ($$ $$>=0 & I>=$$ AS[0]-$$)
 210   ASSERT .FA(J:I<J<=AS[0])(AS[J]+J=AS[0]+1)
In (3/2):
New (7/3/2/1) to evaluate .FA(J:I<J & J<=AS[0])J+AS[J]=AS[0]+
   7.1     I<J
   7.2     J<=AS[0]
   3.8     .FA(J:I<J & J<=AS[0])J+AS[J]=AS[0]+1
Add ELLIST((ARRAY AS)): ($$ I<$$ & $$<=AS[0] AS[0]+1-$$)
 215   ASSERT .FA(J:AS[0]<J<=K)(AS[J]=0)
In (3/2):
New (8/3/2/1) to evaluate .FA(J:J>AS[0] & J<=K)AS[J]=0
   8.1     J>AS[0]
   8.2     J<=K
   3.9     .FA(J:J>AS[0] & J<=K)AS[J]=0
Add ELLIST((ARRAY AS)): ($$ $$>AS[0] & $$<=K 0)
 220   REPEAT: BEGIN
In (3/2):
 230   K ← K+1
In (3/2):
Set CVAL(K): K+1
 250   A[K] ← PAIR(A[I],A[AS[I]])
In (3/2):
CVAL(K) = K+1
Eval: A[K] => A[K+1]
Eval: A[K] => A[K+1]
From 3.6: I>AS[0] => NIL
From 3.6 & 3.5: I<=K => T
From 3.6: I<=AS[0] => T
From 3.4: I>=0 => T
Eval: AS[I] => AS[0]-I
From 3.6: I<=AS[0] => T
From 3.4 & 3.5 & 3.6: I+K>=AS[0] => T
Eval: A[AS[I]] => AS[0]-I
From 3.4: I>=0 => T
From 3.6 & 3.5: I<=K => T
Eval: A[I] => I
From 3.4: I>=0 => T
From 3.6 & 3.5: I<=K => T
Eval: A[I] => I
From 3.6: I>AS[0] => NIL
From 3.6 & 3.5: I<=K => T
From 3.6: I<=AS[0] => T
From 3.4: I>=0 => T
Eval: AS[I] => AS[0]-I
From 3.6: I<=AS[0] => T
From 3.4 & 3.5 & 3.6: I+K>=AS[0] => T
Eval: A[AS[I]] => AS[0]-I
From 3.6: I>AS[0] => NIL
From 3.6 & 3.5: I<=K => T
From 3.6: I<=AS[0] => T
From 3.4: I>=0 => T
Eval: AS[I] => AS[0]-I
From 3.6: I<=AS[0] => T
From 3.4 & 3.5 & 3.6: I+K>=AS[0] => T
Eval: A[AS[I]] => AS[0]-I
From 3.4: I>=0 => T
From 3.6 & 3.5: I<=K => T
Eval: A[I] => I
From 3.6: I>AS[0] => NIL
From 3.6 & 3.5: I<=K => T
From 3.6: I<=AS[0] => T
From 3.4: I>=0 => T
Eval: AS[I] => AS[0]-I
From 3.6: I<=AS[0] => T
From 3.4 & 3.5 & 3.6: I+K>=AS[0] => T
Eval: A[AS[I]] => AS[0]-I
Eval: PAIR(A[I],A[AS[I]])
    => AS[0]+(AS[0]+AS[0]↑2)/2+1-I
Add ELLIST((ARRAY A)): (A[K+1] AS[0]+(AS[0]+AS[0]↑2)/2+1-I)
 260   AS[K] ← 0
In (3/2):
CVAL(K) = K+1
Eval: AS[K] => AS[K+1]
Eval: AS[K] => AS[K+1]
Add ELLIST((ARRAY AS)): (AS[K+1] 0)
 270   AD[K] ← K-1
In (3/2):
CVAL(K) = K+1
Eval: AD[K] => AD[K+1]
Eval: AD[K] => AD[K+1]
CVAL(K) = K+1
Eval: K-1 => K
Add ELLIST((ARRAY AD)): (AD[K+1] K)
 280   AS[I] ← AS[I]+1
In (3/2):
From 3.6 & 3.5: I=K+1 => NIL
From 3.6 & 3.5: I=K+1 => NIL
From 3.6: I>AS[0] => NIL
From 3.6 & 3.5: I<=K => T
From 3.6: I<=AS[0] => T
From 3.4: I>=0 => T
Eval: AS[I] => AS[0]-I
Eval: AS[I]+1 => AS[0]+1-I
Add ELLIST((ARRAY AS)): (AS[I] AS[0]+1-I)
 290   AD[0] ← AS[0]
In (3/2):
From 3.4 & 3.6 & 3.5: K=-1 => NIL
From 3.4 & 3.6 & 3.5: K=-1 => NIL
From 3.6 & 3.4: AS[0]<0 => NIL
From 3.4 & 3.6 & 3.5: K>=0 => T
From 3.4: I<0 => NIL
From 3.6 & 3.4: AS[0]>=0 => T
From 3.4: I>=0 => T
New (9/3/2/1) for case I=0, so that AS[0] => AS[0]+1-I
Cancel 3.4: I>=0
   9.1    I=0
Set ELLIST((ARRAY AS)): ((AS[0] AS[0]+1) (AS[K+1] 0) ($$ $$>AS[0] &
$$<=
K 0) ($$ I<$$ & $$<=AS[0] AS[0]+1-$$) ($$ $$>=0 & I>=$$ AS[0]-$$))
Set CVAL(I): 0
New (10/3/2/1) for case I≠0, so that AS[0] => AS[0]
Cancel 3.4: I>=0
  10.1    I>0
In (9/3):
From 9.1 & 3.6 & 3.5: K=-1 => NIL
Eval: AS[0] => AS[0]+1
Eval: AS[0] => AS[0]+1
Add ELLIST((ARRAY AD)): (AD[0] AS[0]+1)
In (10/3):
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
Add ELLIST((ARRAY AD)): (AD[0] AS[0])
 300   I ← AD[I]
In (9/3):
CVAL(I) = 0
Eval: AD[I] => AS[0]+1
Eval: AD[I] => AS[0]+1
Set CVAL(I): AS[0]+1
In (10/3):
From 10.1: I=0 => NIL
From 3.6 & 3.5: I=K+1 => NIL
From 10.1: I>0 => T
From 3.6 & 3.5: I<=K => T
Eval: AD[I] => I-1
```

```
Eval: AD[I] => I-1
Set CVAL(I): I-1
 310   END
In (9/3):
In (10/3):
 150   LOOP
In (9/3):
In (10/3):

 160   ASSERT .FA(J:0<=J<=K)(A[J]=J)
In (9/3):
Goal: .FA(J:J>=0 & J<=K)J=A[J]
CVAL(K) = K+1
Eval: J>=0 & J<=K => J>=0 & J<=K+1
New (11/9/3/2/1) for .FA goal 160, case J>=0 & J<=K [in old range]
  11.1    J>=0
  11.2    J<=K
  11.3    J=A[J]
In (11/9):
From 11.2: J=K+1 => NIL
From 11.1: J>=0 => T
From 11.2: J<=K => T
Eval: A[J] => J
Eval: J=A[J] => T
Success in (11/9)
New (12/9/3/2/1) for .FA goal 160, case J=K+1 & J>=0 [not in old
range]
  12.1    J=K+1
Set CVAL(J): K+1
From 12.1 & 9.1 & 3.6 & 3.5: J>=0 => T
In (12/9):
CVAL(J) = K+1
CVAL(J) = K+1
Eval: A[J] => AS[0]+(AS[0]+AS[0]↑2)/2+1-I
Eval: J=A[J] => T
Success in (12/9)
In (10/3):
Goal: .FA(J:J>=0 & J<=K)J=A[J]
CVAL(K) = K+1
Eval: J>=0 & J<=K => J>=0 & J<=K+1
New (13/10/3/2/1) for .FA goal 160, case J>=0 & J<=K [in old range]
  13.1    J>=0
  13.2    J<=K
  13.3    J=A[J]
In (13/10):
From 13.2: J=K+1 => NIL
From 13.1: J>=0 => T
From 13.2: J<=K => T
Eval: A[J] => J
Eval: J=A[J] => T
Success in (13/10)
New (14/10/3/2/1) for .FA goal 160, case J=K+1 & J>=0 [not in old
range]
```

```
  14.1     J=K+1
Set CVAL(J): K+1
From 14.1 & 10.1 & 3.6 & 3.5: J>=0 => T
In (14/10):
CVAL(J) = K+1
CVAL(J) = K+1
Eval: A[J] => AS[0]+(AS[0]+AS[0]↑2)/2+1-I
Eval: J=A[J] => T
Success in (14/10)

 170   ASSERT .FA(J:1<=J<=K)(AD[J]=J-1)
In (9/3):
Goal: .FA(J:J>0 & J<=K)J=AD[J]+1
CVAL(K) = K+1
Eval: J>0 & J<=K => J>0 & J<=K+1
New (15/9/3/2/1) for .FA goal 170, case J>0 & J<=K [in old range]
  15.1     J>0
  15.2     J<=K
  15.3     J=AD[J]+1
In (15/9):
From 15.1: J=0 => NIL
From 15.2: J=K+1 => NIL
From 15.1: J>0 => T
From 15.2: J<=K => T
Eval: AD[J] => J-1
Eval: J=AD[J]+1 => T
Success in (15/9)
New (16/9/3/2/1) for .FA goal 170, case J=K+1 & J>0 [not in old
range]
  16.1     J=K+1
Set CVAL(J): K+1
From 16.1 & 9.1 & 3.6 & 3.5: J>0 => T
In (16/9):
CVAL(J) = K+1
CVAL(J) = K+1
From 9.1 & 3.6 & 3.5: K=-1 => NIL
Eval: AD[J] => K
Eval: J=AD[J]+1 => T
Success in (16/9)
In (10/3):
Goal: .FA(J:J>0 & J<=K)J=AD[J]+1
CVAL(K) = K+1
Eval: J>0 & J<=K => J>0 & J<=K+1
New (17/10/3/2/1) for .FA goal 170, case J>0 & J<=K [in old range]
  17.1     J>0
  17.2     J<=K
  17.3     J=AD[J]+1
In (17/10):
From 17.1: J=0 => NIL
From 17.2: J=K+1 => NIL
From 17.1: J>0 => T
From 17.2: J<=K => T
Eval: AD[J] => J-1
```

```
Eval: J=AD[J]+1 => T
Success in (17/10)
New (18/10/3/2/1) for .FA goal 170, case J=K+1 & J>0 [not in old
range]
  18.1    J=K+1
Set CVAL(J): K+1
From 18.1 & 10.1 & 3.6 & 3.5: J>0 => T
In (18/10):
CVAL(J) = K+1
CVAL(J) = K+1
From 10.1 & 3.6 & 3.5: K=-1 => NIL
Eval: AD[J] => K
Eval: J=AD[J]+1 => T
Success in (18/10)

 180    ASSERT K=PAIR(I,AS[0]-I)-1
In (9/3):
Goal: K=PAIR(I,AS[0]-I)-1
CVAL(K) = K+1
CVAL(I) = AS[0]+1
Eval: AS[0] => AS[0]+1
Eval: AS[0] => AS[0]+1
Eval: AS[0] => AS[0]+1
Eval: K=PAIR(I,AS[0]-I)-1
    => 2*K=3*AS[0]+AS[0]↑2-1 ! 2*K=3*AS[0]+AS[0]↑2
New (19/9/3/2/1) for goal 180
  19.1    2*K#3*AS[0]+AS[0]↑2-1
  19.2    2*K#3*AS[0]+AS[0]↑2
In (10/3):
Goal: K=PAIR(I,AS[0]-I)-1
CVAL(K) = K+1
CVAL(I) = I-1
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
```

```
Eval: K=PAIR(I,AS[0]-I)-1 => T
No proof needed, Eval(goal) = T

 190    ASSERT 0<=I<=AS[0]<=K
In (9/3):
Goal: I>=0 & K>=AS[0] & I<=AS[0]
CVAL(I) = AS[0]+1
CVAL(K) = K+1
Eval: AS[0] => AS[0]+1
CVAL(I) = AS[0]+1
Eval: AS[0] => AS[0]+1
From 9.1 & 3.6: AS[0]>=-1 => T
From 3.5: K>=AS[0] => T
Eval: I>=0 & K>=AS[0] & I<=AS[0] =>
No proof needed, Eval(goal) = T
In (10/3):
Goal: I>=0 & K>=AS[0] & I<=AS[0]
CVAL(I) = I-1
CVAL(K) = K+1
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
CVAL(I) = I-1
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
From 10.1: I>0 => T
From 3.5: K>=AS[0]-1 => T
From 3.6: I<=AS[0]+1 => T
Eval: I>=0 & K>=AS[0] & I<=AS[0] => T
No proof needed, Eval(goal) = T

 200    ASSERT .FA(J:0<=J<=I)(AS[J]+J=AS[0])
In (9/3):
Goal: .FA(J:J>=0 & I>=J)J+AS[J]=AS[0]
CVAL(I) = AS[0]+1
Eval: J>=0 & I>=J => J>=0 & J<=AS[0]+1
New (20/9/3/2/1) for .FA goal 200, case J>=0 & I>=J & J<=AS[0]+1
 [in old range]
  20.1    J>=0
From 20.1 & 9.1: I>=J => I=J
  20.2    I=J
Set CVAL(J): I
From 20.2 & 3.6: J<=AS[0]+1 => T
  20.3    J+AS[J]=AS[0]
```

```
In (20/9):
CVAL(J) = I
Eval: AS[0] => AS[0]+1
CVAL(J) = I
From 9.1: I=0 => T
Eval: AS[J] => AS[0]+1
From 9.1: I=0 => T
Eval: J+AS[J]=AS[0] => T
Success in (20/9)
New (21/9/3/2/1) for .FA goal 200, case J>=0 & I<J & J<=AS[0]+1
 [not in old range]
  21.1    J>=0
  21.2    I<J
  21.3    J<=AS[0]+1
In (21/9):
Eval: AS[0] => AS[0]+1
From 21.2 & 9.1: J=0 => NIL
From 21.3: J>AS[0] => J=AS[0]+1
From 21.2: I<J => T
From 21.1: J>=0 => T
From 21.2: I>=J => NIL
From 21.3 & 3.5: J>K => J=K+1
From 21.3: J>AS[0] => J=AS[0]+1
From 21.3 & 3.5: J>K => J=K+1
New (22/21/9/3/2/1) for case J<=AS[0] & J#K+1, so that AS[J] =>
AS[0]+1-
J
Cancel 21.3: J<=AS[0]+1
  22.1    J<=AS[0]
From 22.1 & 3.5: J#K+1 => T
New (23/21/9/3/2/1) for case J=AS[0]+1 & J<=K, so that AS[J] => 0
Cancel 21.3: J<=AS[0]+1
  23.1    J=AS[0]+1
Set CVAL(J): AS[0]+1
From 23.1 & 3.5: J<=K => T
New (24/21/9/3/2/1) for case J=K+1, so that AS[J] => 0
  24.1    J=K+1
Set CVAL(J): K+1
In (22/21):
Eval: AS[0] => AS[0]+1
From 21.2 & 9.1: J=0 => NIL
From 22.1 & 3.5: J=K+1 => NIL
From 22.1: J>AS[0] => NIL
From 22.1 & 3.5: J<=K => T
From 21.2: I<J => T
From 22.1: J<=AS[0] => T
Eval: AS[J] => AS[0]+1-J
Eval: J+AS[J]=AS[0] => T
Success in (22/21)
In (23/21):
CVAL(J) = AS[0]+1
Eval: AS[0] => AS[0]+1
CVAL(J) = AS[0]+1
```

```
From 23.1 & 21.2 & 9.1: AS[0]=-1 => NIL
From 3.6: I<=AS[0] => T
From 23.1 & 21.2 & 9.1: AS[0]>=-1 => T
From 3.6: I>AS[0] => NIL
From 3.5: K<=AS[0] => K=AS[0]
From 3.5: K>=AS[0] => T
Eval: AS[J] => 0
Eval: J+AS[J]=AS[0] => T
Success in (23/21)
In (24/21):
CVAL(J) = K+1
Eval: AS[0] => AS[0]+1
CVAL(J) = K+1
From 24.1 & 21.2 & 9.1: K=-1 => NIL
Eval: AS[J] => 0
Eval: J+AS[J]=AS[0] => K=AS[0]
Cancel 3.5: K>=AS[0]
  24.2    K>AS[0]
In (10/3):
Goal: .FA(J:J>=0 & I>=J)J+AS[J]=AS[0]
CVAL(I) = I-1
Eval: J>=0 & I>=J => J>=0 & I>J
New (25/10/3/2/1) for .FA goal 200, case J>=0 & I>J [in old range]
  25.1    J>=0
  25.2    I>J
  25.3    J+AS[J]=AS[0]
In (25/10):
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
From 25.2: I=J => NIL
From 25.2 & 3.6 & 3.5: J=K+1 => NIL
From 25.2 & 3.6: J>AS[0] => NIL
From 25.2 & 3.6 & 3.5: J<=K => T
From 25.2: I<J => NIL
From 25.2 & 3.6: J<=AS[0] => T
From 25.1: J>=0 => T
From 25.2: I>=J => T
Eval: AS[J] => AS[0]-J
Eval: J+AS[J]=AS[0] => T
Success in (25/10)
```

```
 210    ASSERT .FA(J:I<J<=AS[0])(AS[J]+J=AS[0]+1)
In (9/3):
Goal: .FA(J:I<J & J<=AS[0])J+AS[J]=AS[0]+1
CVAL(I) = AS[0]+1
Eval: AS[0] => AS[0]+1
Eval: I<J & J<=AS[0] => NIL
No proof needed, Eval(goal) = T
In (10/3):
Goal: .FA(J:I<J & J<=AS[0])J+AS[J]=AS[0]+1
CVAL(I) = I-1
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
Eval: I<J & J<=AS[0]
    => I<=J & J<=AS[0]
New (26/10/3/2/1) for .FA goal 210, case I<J & J<=AS[0] [in old
range]
  26.1    I<J
  26.2    J<=AS[0]
  26.3    J+AS[J]=AS[0]+1
In (26/10):
From 10.1: I=0 => NIL
From 26.2 & 3.5 & 26.1 & 10.1: K=-1 => NIL
From 26.2 & 26.1 & 10.1: AS[0]<0 => NIL
From 26.2 & 3.5 & 26.1 & 10.1: K>=0 => T
From 10.1: I<0 => NIL
From 26.2 & 26.1 & 10.1: AS[0]>=0 => T
From 10.1: I>=0 => T
From 26.1: I=J => NIL
From 26.2 & 3.5: J=K+1 => NIL
From 26.2: J>AS[0] => NIL
From 26.2 & 3.5: J<=K => T
From 26.1: I<J => T
From 26.2: J<=AS[0] => T
Eval: AS[J] => AS[0]+1-J
Eval: J+AS[J]=AS[0]+1 => T
Success in (26/10)
New (27/10/3/2/1) for .FA goal 210, case I=J & J<=AS[0]
 [not in old range]
  27.1    I=J
Set CVAL(J): I
From 27.1 & 3.6: J<=AS[0] => T
In (27/10):
CVAL(J) = I
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
CVAL(J) = I
Eval: AS[J] => AS[0]+1-I
Eval: J+AS[J]=AS[0]+1 => T
Success in (27/10)

 215   ASSERT .FA(J:AS[0]<J<=K)(AS[J]=0)
In (9/3):
Goal: .FA(J:J>AS[0] & J<=K)AS[J]=0
Eval: AS[0] => AS[0]+1
CVAL(K) = K+1
Eval: J>AS[0] & J<=K
    => J>=AS[0]+2 & J<=K+1
New (28/9/3/2/1) for .FA goal 215, case J>=AS[0]+2 & J<=K [in old
range]
  28.1    J>=AS[0]+2
  28.2    J<=K
  28.3    AS[J]=0
In (28/9):
From 28.1 & 9.1 & 3.6: J=0 => NIL
From 28.2: J=K+1 => NIL
From 28.1: J>AS[0] => T
From 28.2: J<=K => T
Eval: AS[J] => 0
Eval: AS[J]=0 => T
Success in (28/9)
New (29/9/3/2/1) for .FA goal 215, case J=K+1 & J>=AS[0]+2
 [not in old range]
  29.1    J=K+1
Set CVAL(J): K+1
  29.2    J>=AS[0]+2
In (29/9):
CVAL(J) = K+1
From 29.2 & 29.1 & 9.1 & 3.6: K=-1 => NIL
Eval: AS[J] => 0
Eval: AS[J]=0 => T
Success in (29/9)
In (10/3):
Goal: .FA(J:J>AS[0] & J<=K)AS[J]=0
From 10.1: I=0 => NIL
From 10.1 & 3.6 & 3.5: K=-1 => NIL
From 10.1 & 3.6: AS[0]<0 => NIL
From 10.1 & 3.6 & 3.5: K>=0 => T
From 10.1: I<0 => NIL
From 10.1 & 3.6: AS[0]>=0 => T
From 10.1: I>=0 => T
CVAL(K) = K+1
Eval: J>AS[0] & J<=K
    => J>AS[0] & J<=K+1
New (30/10/3/2/1) for .FA goal 215, case J>AS[0] & J<=K [in old
range]
  30.1    J>AS[0]
  30.2    J<=K
  30.3    AS[J]=0
In (30/10):
From 30.1 & 3.6: I=J => NIL
From 30.2: J=K+1 => NIL
From 30.1: J>AS[0] => T
From 30.2: J<=K => T
Eval: AS[J] => 0
Eval: AS[J]=0 => T
Success in (30/10)
New (31/10/3/2/1) for .FA goal 215, case J=K+1 & J>AS[0]
 [not in old range]
  31.1    J=K+1
Set CVAL(J): K+1
From 31.1 & 3.5: J>AS[0] => T
In (31/10):
CVAL(J) = K+1
From 3.6 & 3.5: I=K+1 => NIL
Eval: AS[J] => 0
Eval: AS[J]=0 => T
Success in (31/10)
10286 conses, 477.177 seconds
((11 9 3 2 1) (12 9 3 2 1) (13 10 3 2 1) (14 10 3 2 1) (15 9 3 2 1)
(16 9 3 2 1) (17 10 3 2 1) (18 10 3 2 1) (19 9 3 2 1) (20 9 3 2 1)
(22 21 9 3 2 1) (23 21 9 3 2 1) (24 21 9 3 2 1) (25 10 3 2 1) (26
10 3 2 1) (27 10 3 2 1) (28 9 3 2 1) (29 9 3 2 1) (30 10 3 2 1) (31
10 3 2 1))

Proof for (11/9) for .FA goal 160, case J>=0 & J<=K [in old range]
Proved

Proof for (12/9) for .FA goal 160, case J=K+1 & J>=0 [not in old
range]
Proved

Proof for (13/10) for .FA goal 160, case J>=0 & J<=K [in old range]
Proved

Proof for (14/10) for .FA goal 160, case J=K+1 & J>=0 [not in old
range]
Proved
```

```
Proof for (15/9) for .FA goal 170, case J>0 & J<=K [in old range]
Proved

Proof for (16/9) for .FA goal 170, case J=K+1 & J>0 [not in old
range]
Proved

Proof for (17/10) for .FA goal 170, case J>0 & J<=K [in old range]
Proved

Proof for (18/10) for .FA goal 170, case J=K+1 & J>0 [not in old
range]
Proved

Proof for (19/9) for goal 180:
New (32/19/9/3/2/1) for proof
Proving (32/19)...
   3.1 A  .FA(J:J>=0 & J<=K)J=A[J]
   3.2 A  .FA(J:J>0 & J<=K)J=AD[J]+1
   3.3 A  2*I+2*K=3*AS[0]+AS[0]↑2-1 ! 2*I+2*K=3*AS[0]+AS[0]↑2
   3.5 A  K>=AS[0]
   3.6 A  I<=AS[0]
   3.7 A  .FA(J:J>=0 & I>=J)J+AS[J]=AS[0]
   3.8 A  .FA(J:I<J & J<=AS[0])J+AS[J]=AS[0]+1
   3.9 A  .FA(J:J>AS[0] & J<=K)AS[J]=0
   9.1 A  I=0
  19.1 G  2*K#3*AS[0]+AS[0]↑2-1
  19.2 G  2*K#3*AS[0]+AS[0]↑2
Equalities:
  (from 9.1)   I => 0
Cancel 9.1: I=0
Cancel 3.6: I<=AS[0]
  32.1     AS[0]>=0
Cancel 3.3: 2*I+2*K=3*AS[0]+AS[0]↑2-1 ! 2*I+2*K=3*AS[0]+AS[0]↑2
From 19.1: 2*K=3*AS[0]+AS[0]↑2-1 => NIL
From 19.2: 2*K=3*AS[0]+AS[0]↑2 => NIL
Success in (32/19)
Proved: 163 conses, 6.594 seconds
Success in (19/9)

Proof for (20/9) for .FA goal 200, case J>=0 & I>=J & J<=AS[0]+1
 [in old range]
Proved
```

```
Proof for (22/21) for case J<=AS[0] & J≠K+1, so that AS[J] => AS[0]+
1-J
Proved

Proof for (23/21) for case J=AS[0]+1 & J<=K, so that AS[J] => 0
Proved

Proof for (24/21) for case J=K+1, so that AS[J] => 0:
New (33/24/21/9/3/2/1) for proof
Proving (33/24)...
   3.1 A  .FA(J:J>=0 & J<=K)J=A[J]
   3.2 A  .FA(J:J>0 & J<=K)J=AD[J]+1
   3.3 A  2*I+2*K=3*AS[0]+AS[0]↑2-1 ! 2*I+2*K=3*AS[0]+AS[0]↑2
   3.6 A  I<=AS[0]
   3.7 A  .FA(J:J>=0 & I>=J)J+AS[J]=AS[0]
   3.8 A  .FA(J:I<J & J<=AS[0])J+AS[J]=AS[0]+1
   3.9 A  .FA(J:J>AS[0] & J<=K)AS[J]=0
   9.1 A  I=0
  21.1 A  J>=0
  21.2 A  I<J
  21.3 A  J<=AS[0]+1
  24.1 A  J=K+1
  24.2 G  K>AS[0]
Equalities:
  (from 9.1)   I => 0
  (from 24.1)  K => J-1
Cancel 24.1: J=K+1
Cancel 9.1: I=0
Cancel 3.6: I<=AS[0]
  33.1    AS[0]>=0
Cancel 21.2: I<J
Cancel 21.1: J>=0
  33.2    J>0
Cancel 24.2: K>AS[0]
From 21.3: J>=AS[0]+2 => NIL
Success in (33/24)
Proved: 314 conses, 11.722 seconds
Success in (24/21)

Proof for (25/10) for .FA goal 200, case J>=0 & I>J [in old range]
Proved
```

```
Proof for (26/10) for .FA goal 210, case I<J & J<=AS[0] [in old
range]
Proved

Proof for (27/10) for .FA goal 210, case I=J & J<=AS[0]
 [not in old range]
Proved

Proof for (28/9) for .FA goal 215, case J>=AS[0]+2 & J<=K [in old
range]
Proved

Proof for (29/9) for .FA goal 215, case J=K+1 & J>=AS[0]+2
 [not in old range]
Proved

Proof for (30/10) for .FA goal 215, case J>AS[0] & J<=K [in old
range]
Proved

Proof for (31/10) for .FA goal 215, case J=K+1 & J>AS[0]
 [not in old range]
Proved

Path #2: 1...215
New (34/2/1) for paths beginning 1...215
   1   DECLARE ARRAY A, AS, AD
In (34/2):
   2   LET PAIR(X,Y)=(X+Y+1)*(X+Y)/2+Y+1
In (34/2):
 100   K ← 0
In (34/2):
Set CVAL(K): 0
 110   I ← 0
In (34/2):
Set CVAL(I): 0
 120   A[0] ← 0
In (34/2):
Add ELLIST((ARRAY A)): (A[0] 0)
 130   AS[0] ← 0
In (34/2):
Add ELLIST((ARRAY AS)): (AS[0] 0)
 140   AD[0] ← 0
In (34/2):
Add ELLIST((ARRAY AD)): (AD[0] 0)
 150   LOOP
In (34/2):
 160   ASSERT .FA(J:0<=J<=K)(A[J]=J)
In (34/2):
Goal: .FA(J:J>=0 & J<=K)J=A[J]
CVAL(K) = 0
Eval: J>=0 & J<=K => J=0
New (35/34/2/1) for .FA goal 160, case J=0 & J<=K [in old range]
  35.1    J=0
Set CVAL(J): 0
  35.2    J<=K
  35.3    J=A[J]
In (35/34):
CVAL(J) = 0
CVAL(J) = 0
Eval: A[J] => 0
Eval: J=A[J] => T
Success in (35/34)
New (36/34/2/1) for .FA goal 160, case J=0 & J>K [not in old range]
  36.1    J=0
Set CVAL(J): 0
  36.2    J>K
In (36/34):
CVAL(J) = 0
CVAL(J) = 0
Eval: A[J] => 0
Eval: J=A[J] => T
Success in (36/34)

 170   ASSERT .FA(J:1<=J<=K)(AD[J]=J-1)
In (34/2):
Goal: .FA(J:J>0 & J<=K)J=AD[J]+1
CVAL(K) = 0
Eval: J>0 & J<=K => NIL
No proof needed, Eval(goal) = T

 180   ASSERT K=PAIR(I,AS[0]-I)-1
In (34/2):
Goal: K=PAIR(I,AS[0]-I)-1
CVAL(K) = 0
CVAL(I) = 0
Eval: AS[0] => 0
Eval: AS[0] => 0
Eval: AS[0] => 0
Eval: K=PAIR(I,AS[0]-I)-1 => T
No proof needed, Eval(goal) = T
```

```
 190   ASSERT 0<=I<=AS[0]<=K
In (34/2):
Goal: I>=0 & K>=AS[0] & I<=AS[0]
CVAL(I) = 0
CVAL(K) = 0
Eval: AS[0] => 0
CVAL(I) = 0
Eval: AS[0] => 0
Eval: I>=0 & K>=AS[0] & I<=AS[0] => T
No proof needed, Eval(goal) = T

 200   ASSERT .FA(J:0<=J<=I)(AS[J]+J=AS[0])
In (34/2):
Goal: .FA(J:J>=0 & I>=J)J+AS[J]=AS[0]
CVAL(I) = 0
Eval: J>=0 & I>=J => J=0
New (37/34/2/1) for .FA goal 200, case J=0 & I>=J [in old range]
  37.1    J=0
Set CVAL(J): 0
  37.2    I>=J
  37.3    J+AS[J]=AS[0]
In (37/34):
CVAL(J) = 0
Eval: AS[0] => 0
CVAL(J) = 0
Eval: AS[J] => 0
Eval: J+AS[J]=AS[0] => T
Success in (37/34)
New (38/34/2/1) for .FA goal 200, case J=0 & I<J [not in old range]
  38.1    J=0
Set CVAL(J): 0
  38.2    I<J
In (38/34):
CVAL(J) = 0
Eval: AS[0] => 0
CVAL(J) = 0
Eval: AS[J] => 0
Eval: J+AS[J]=AS[0] => T
Success in (38/34)

 210   ASSERT .FA(J:I<J<=AS[0])(AS[J]+J=AS[0]+1)
In (34/2):
Goal: .FA(J:I<J & J<=AS[0])J+AS[J]=AS[0]+1
CVAL(I) = 0
Eval: AS[0] => 0
Eval: I<J & J<=AS[0] => NIL
No proof needed, Eval(goal) = T
```

```
 215   ASSERT .FA(J:AS[0]<J<=K)(AS[J]=0)
In (34/2):
Goal: .FA(J:J>AS[0] & J<=K)AS[J]=0
Eval: AS[0] => 0
CVAL(K) = 0
Eval: J>AS[0] & J<=K => NIL
No proof needed, Eval(goal) = T
792 conses, 53.961 seconds
((35 34 2 1) (36 34 2 1) (37 34 2 1) (38 34 2 1))

Proof for (35/34) for .FA goal 160, case J=0 & J<=K [in old range]
Proved

Proof for (36/34) for .FA goal 160, case J=0 & J>K [not in old
range]
Proved

Proof for (37/34) for .FA goal 200, case J=0 & I>=J [in old range]
Proved

Proof for (38/34) for .FA goal 200, case J=0 & I<J [not in old
range]
Proved
```

APPENDIX C

Analysis of the Floyd Protocol

In this appendix we present the dialogue envisioned by Floyd in his paper "Toward Interactive Design of Correct Programs" [Floyd]. In this dialogue, the computer takes the initiative in leading the man through the design and coding of a small program. Material copied from the paper will appear indented; each utterance will be numbered for reference. In the interests of brevity, a few of Floyd's comments [in square brackets] have been dropped and the text has been slightly abbreviated. Material not indented is the present author's analysis.

We assume that after each of the man's inputs, the computer performs some analysis or else possibly falls back on a list of standard questions that it knows are appropriate to given situations. It is worth noting that since PIVOT has actually carried out all the proofs required by this scenario, the task of implementing a system like the one envisioned by Floyd can now legitimately be classified as system programming rather than artificial intelligence (a task which the author hopes to accomplish in the near future).

We find the programmer preparing to write a procedure which will locate a symbol in a sorted table of known size. He calls the symbol X, the table A_1, A_2, ... A_N. The man speaks first. I enclose my own comments on the interaction in square brackets.

(1) Man: I want to design a program I shall call 'Locate X in table A_1 ... A_N', or P1 for short. [The name is simply for the programmer's convenience: the system makes no attempt to interpret or understand it.] In P1, X is a symbol, N and L are integers, and A is an array of symbols. The values of X, N, and A may not change in P1. We assume the following true at the start of P1 (its <u>antecedent</u> condition):

(1.1) For all integers i, j, $1<i<j<N \supset A_i<A_j$.

I will abbreviate this by 'A monotone'.

(1.2) There is an integer k, $1<k<N$, for which $A_k=X$.

I will abbreviate this by 'X IN A{1:N}'. After execution, P1 must guarantee (its <u>consequent</u> condition) that

(1.3) $1\leq L\leq N$
(1.4) $A_L=X$.

(2) Computer: The consequent of P1, (1.3) and (1.4), does not logically follow from the antecedent, (1.1) and (1.2) (as I can demonstrate by counterexample), so P1 must apparently change L. [Recall that A, N, and X are protected from change.]

The man has finished specifying the input-output relations for P1, and the computer picks on the most obvious need, namely that L is an output variable (since it is mentioned in the consequent but not the antecedent) which must be set.

(3) Man: Let P1 be an iteration. [He has the idea of keeping pointers B and C, bounding the region of A where the symbol X will be found.] Let P1 be

```
ITERATE UNTIL B=C: P2.
```

(4) Computer: What conditions do you want to assume before and after P2?

I.e., as the inductive assertion. The computer asks this question because the man has specified an interation and the inductive assertion has not been specified yet, although the man has the option of saying that he does not want to specify one (e.g., if an inductive assertion for an inner loop will cut the outer one too).

(5) Man:

(2.1) B and C are integer variables of P1.
(2.2) $1 < B < C < N$.
(2.3) There is an integer k, $B < k < C$, for which $A_k = X$, abbreviated X IN A{B:C}.

(6) Computer: Why does P2 terminate?

Another standard question. PIVOT does not currently concern itself with termination, since this is generally easier to show than correctness of results.

(7) Man: C-B is a non-negative integer, decreasing at each cycle of the iteration.

This is the standard rationale for programs dealing with integers.

(8) Computer: The antecedent of P2 does not follow from the antecedent of P1 and the iteration-nonterminated condition $B \neq C$. Do you want to initialize the iteration later? [This suggestion is usually good programming practice.]

In general programming practice, one wants the inductive assertion to hold even before the terminating condition has been tested, so the clause "and the ... condition" really should not appear. As in (2), the computer notices that B and C have not been set in the path from the initial condition to the entry to P2, and suggests that they be set.

(9) Man: Yes.

(10) Computer: The consequent of P2, with the iteration termination condition, does not seem to imply the consequent of P1. Do you want to finalize the iteration later?

This is also good practice. Again, the computer notices that L is not set on the path from the head of P2 to the consequent of P1.

(11) Man: Yes.

(12) Computer: Do you want to design P2 now?

Design of the other two control paths has been deferred, so this is the only task left at the moment.

(13) Man: Yes.

(14) Computer: In P2, A, X, and N are invariant; B, C, and N are integers. Antecedent condition is:

(3.1) $1<B<C<N$
(3.2) X IN A{B:C}
(3.3) A monotone

[The computer does not mention that X IN A{1:N}, since that logically follows from 3.1 and 3.2.]

Consequent condition is:

(3.4) $1<B'<C'<N$
(3.5) X IN A{B':C'}
(3.6) $C'-B'<C-B$

You must change C or B. [The computer does not mention 'A monotone' as consequent condition, since it is part of the antecedent condition and A is invariant.]

(15) Man: Define P2 as

```
D := (B+C)/2; [Integer division]
IF X≤A_D THEN C := D
         ELSE B := D
```

(16) Computer: Well, suppose B=1, C=2, N=2, X=0, A_1=-1. P2 becomes

```
D := (1+2)/2 (=1);
IF 0≤-1 (=FALSE) THEN ... ELSE B := 1
```

```
resulting in B'=1, C'=2.  This violates the condition needed for
termination, that C'-B'<C-B.  [See Appendix for sketch of how the
computer found the counter example.]
```

Now the computer has a fully specified path, and attempts verification. When the verification of the indicated path fails, the computer must be able to recognize that a counterexample is wanted, but finding it is not difficult: the process followed by PIVOT appears at the end of this appendix.

```
(17)    Man: Sorry.  Let's try modifying the third line of P2: call it P3.

(18)    Computer: Antecedent is

            (4.1)   1<B<C<N [Note that B=C is not possible.]
            (4.2)   X IN A{B:C}
            (4.3)   A monotone
            (4.4)   D=(B+C)/2
            (4.5)   X>A_D

        Consequent is

            (4.6)   1<B'<C'<N
            (4.7)   X IN A{B':C'}
            (4.8)   C'-B'<C-B
```

4.1 is 3.1 modified by going through the end test B=C which fails. (PIVOT will actually replace B<C by B<C in THCLAUSEP.) 4.2 and 4.3 are 3.2 and 3.3: the variables in them have not changed. 4.4 is the CVAL entry produced by the assignment to D. 4.5 is the result of the IF $X<A_D$ failing. 4.6 through 4.8 are just 3.4 through 3.6, since P3 is at the end of the loop.

```
(19)    Man: What must P3 do if B=1, C=2, N=2, X=0, A_1=-1?
```

I.e., "What must P3 do in the case that failed?"

```
(20)    Computer: Consequent is then B'=2, C'=2.
```

This comes from substituting the supplied values into 4.1 through 4.8, wrapping 4.6 through 4.8 in .EX(B',C') (...) in a manner similar to the procedure used to find the counterexample, and doing the proof. The proof appears at the end of this appendix.

(21) Man: Check whether $B<D+1\leq C$ follows from the antecedent of P3.

(22) Computer: True.

I.e., $B<(B+C)/2+1<C$ given that $1<B<C$, a fairly trivial piece of algebra.

(23) Man: Define P3 as B := D+1.

(24) Computer: That works. Would you like to design the initialization of P2?

The verification goes through. The body of P2 is now complete, so the computer retrieves the first deferred task.

(25) Man: Yes.

(26) Computer: Antecedent is

```
(5.1)   A monotone
(5.2)   X IN A{1:N}
```

Consequent is

```
(5.3)   1≤B≤C≤N
(5.4)   X IN A{B:C}
```

5.1 and 5.2 are the antecedent of P1; 5.3 and 5.4 are 2.2 and 2.3, the antecedent of P2.

(27) Man: Initialization of P2 is

```
B := 1; C := N.
```

(28) Computer: That works. Would you like to design the finalization of P2?

This is the same situation as (24).

(29) Man: Yes.

(30) Computer: Antecedent is

(6.1) $1 \leq B=C \leq N$
(6.2) $A_B=X$

Consequent is

(6.3) $1 \leq L' \leq N$
(6.4) $A_{L'}=X$.

May I try?

6.1 is 3.1 with B=C taken into account. (PIVOT already does this.) 6.2 is 3.2 with B=C taken into account. (PIVOT does this too.) 6.3 and 6.4 are the consequent of P1. The computer can produce (32) by a simple constructive proof of .EX(L')(6.3 and 6.4) (PIVOT's proof appears at the end of this appendix); presumably it has already done this and is only asking "May I try?" to avoid damaging the man's ego.

(31) Man: Yes.

(32) Computer: L := B.

(33) Man: That works. May I see the complete program?

(34) Computer: BEGIN

```
INVARIANTS A, N, X; INTEGER B, C, D;
(ASSERT N≥1, A MONOTONE, X IN A{1:N});
B := 1; C := N;
ITERATE (ASSERT 1≤B≤C≤N, X IN A{B:C})
  UNTIL B=C:
      BEGIN
      D := (B+C) DIV 2;
      IF X≤A_D THEN C := D
                ELSE B := D+1;
      END
L := B;
(ASSERT 1≤L≤N, A_L=X)
END
```

The computer has inserted a small amount of syntactic embellishment.

```
#GET FLOYD
FLOYD.PIVOT;3
FLOYD
New (2/1) for procedure FLOYD
1282 conses, 11.812 seconds
T

#LIST
PROCEDURE FLOYD
 100    ASSERT N>=1
 110    ASSERT .FA(I0,J0:1<=I0<=J0<=N)A[I0]<=A[J0]
 120    ASSERT .EX(I1:1<=I1<=N)X=A[I1]
 170    B ← 1
 180    C ← N
 200    LOOP
 205       ASSERT .FA(I2,J2:1<=I2<=J2<=N)A[I2]<=A[J2]
 210       ASSERT 1<=B<=C<=N
 215       ASSERT .EX(I3:B<=I3<=C)X=A[I3]
 219       WHILE B#C: BEGIN
 220       ASSUME B0=B, C0=C
 230       D ← (B+C)/2
 250       IF X > A[D] THEN B ← D
 260       ELSE C ← D
 290       LEMMA C-B<C0-B0
 300       END
 305    L ← B
 310    ASSERT 1<=L<=N AND X=A[L]

#SETUP
1662 conses, 22.082 seconds
Paths ready.

Path #1: 205...215-219(B#C)...230-250(X>A[D])-290-300-200...215
New (3/2/1) for paths beginning 205...215
New (4/3/2/1) to evaluate .FA(I2,J2:I2>0 & I2<=J2 & J2<=N)A[I2]<=
A[J2]
New (5/3/2/1) to evaluate .EX(I3:C>=I3 & B<=I3)X=A[I3]
New (6/3/2/1) assuming TRUE in 219
New (7/6/3/2/1) assuming TRUE in 250
New (8/7/6/3/2/1) for goal 290
New (9/7/6/3/2/1) to bypass lemma 290
New (10/9/7/6/3/2/1) to evaluate .EX(I3:C>=I3 & B<=I3)X=A[I3]
New (11/9/7/6/3/2/1) for goal 215
1546 conses, 33.67 seconds
((8 7 6 3 2 1) (11 9 7 6 3 2 1))

Path #2: 205...215-219(B#C)...230-250(X<=A[D])...300-200...215
New (12/6/3/2/1) assuming FALSE in 250
New (13/12/6/3/2/1) to evaluate .EX(I3:C>=I3 & B<=I3)X=A[I3]
New (14/12/6/3/2/1) for goal 215
482 conses, 10.776 seconds
((14 12 6 3 2 1))
```

```
Path #3: 205...215-219(B=C)-305-310
New (15/3/2/1) assuming FALSE in 219
New (16/15/3/2/1) for goal 310
171 conses, 4.369 seconds
((16 15 3 2 1))

Path #4: 100...215
New (17/2/1) for paths beginning 100...215
New (18/17/2/1) to evaluate .FA(I0,J0:I0>0 & I0<=J0 & J0<=N)A[I0]<=
A[J0]
New (19/17/2/1) to evaluate .EX(I1:I1>0 & I1<=N)X=A[I1]
New (20/17/2/1) for goal 205
New (21/17/2/1) to evaluate .EX(I3:C>=I3 & B<=I3)X=A[I3]
New (22/17/2/1) for goal 215
744 conses, 19.146 seconds
((20 17 2 1) (22 17 2 1))

#TH

Path #1: 205...215-219(B#C)...230-250(X>A[D])-290-300-200...215

(8/7) for goal 290
   3.1 A  .FA(I2,J2:I2>0 & I2<=J2 & J2<=N)A[I2]<=A[J2]
   3.2 A  B>0
   3.4 A  C<=N
   3.5 A  C>=I3
   3.6 A  B<=I3
   3.7 A  X=A[I3]
   6.2 A  B=B0
   6.3 A  C=C0
   7.1 A  X>A[(B+C)/2]
   8.1 G  B=C-1

(11/9) for goal 215
   3.1 A  .FA(I2,J2:I2>0 & I2<=J2 & J2<=N)A[I2]<=A[J2]
   3.2 A  B>0
   3.4 A  C<=N
   3.5 A  C>=I3
   3.6 A  B<=I3
   3.7 A  X=A[I3]
   6.2 A  B=B0
   6.3 A  C=C0
   7.1 A  X>A[(B+C)/2]
   9.1 A  B<=C-2
  11.1 G  .FA(I3$1:C>=I3$1 & B+C<=2*I3$1+1)X#A[I3$1]

Path #2: 205...215-219(B#C)...230-250(X<=A[D])...300-200...215

(14/12) for goal 215
   3.1 A  .FA(I2,J2:I2>0 & I2<=J2 & J2<=N)A[I2]<=A[J2]
   3.2 A  B>0
   3.4 A  C<=N
   3.5 A  C>=I3
   3.6 A  B<=I3
   3.7 A  X=A[I3]
   6.1 A  B<C
   6.2 A  B=B0
   6.3 A  C=C0
  12.1 A  X<=A[(B+C)/2]
  14.1 G  .FA(I3$2:B+C>=2*I3$2 & B<=I3$2)X#A[I3$2]

Path #3: 205...215-219(B=C)-305-310

(16/15) for goal 310
   3.1 A  .FA(I2,J2:I2>0 & I2<=J2 & J2<=N)A[I2]<=A[J2]
   3.2 A  B>0
   3.4 A  C<=N
   3.5 A  C>=I3
   3.6 A  B<=I3
   3.7 A  X=A[I3]
  15.1 A  B=C
  16.1 G  X#A[B]

Path #4: 100...215

(20/17) for goal 205
  17.1 A  N>0
  17.2 A  .FA(I0,J0:I0>0 & I0<=J0 & J0<=N)A[I0]<=A[J0]
  17.3 A  I1>0
  17.4 A  I1<=N
  17.5 A  X=A[I1]
  20.1 A  I2>0
  20.2 A  I2<=J2
  20.3 A  J2<=N
  20.4 G  A[I2]>A[J2]

(22/17) for goal 215
  17.1 A  N>0
  17.2 A  .FA(I0,J0:I0>0 & I0<=J0 & J0<=N)A[I0]<=A[J0]
  17.3 A  I1>0
  17.4 A  I1<=N
  17.5 A  X=A[I1]
  22.1 G  .FA(I3:I3>0 & I3<=N)X#A[I3]

#PROVE
1 conses, .934 seconds
Paths ready.

Path #1: 205...215-219(B#C)...230-250(X>A[D])-290-300-200...215
1 conses, .057 seconds
((8 7 6 3 2 1) (11 9 7 6 3 2 1))

Proof for (8/7) for goal 290:
New (23/8/7/6/3/2/1) for proof
Proving (23/8)...
```

```
New (24/23/8/7/6/3/2/1) to prove one disjunct of 23.5
New (25/23/8/7/6/3/2/1) to prove remaining disjuncts of 23.5
Proving (24/23)...
Success in (24/23)
Proving (25/23)...
Failure in (25/23)
Failure in (23/8)
No proof: 2477 conses, 48.415 seconds

Proof for (11/9) for goal 215:
New (26/11/9/7/6/3/2/1) for proof
Proving (26/11)...
Success in (26/11)
Proved: 784 conses, 11.267 seconds

Path #2: 205...215-219(B#C)...230-250(X<=A[D])...300-200...215
1 conses, .055 seconds
((14 12 6 3 2 1))

Proof for (14/12) for goal 215:
New (27/14/12/6/3/2/1) for proof
Proving (27/14)...
New (28/27/14/12/6/3/2/1) to show 27.2 by showing B+C<2*I3
Proving (28/27)...
New (29/28/27/14/12/6/3/2/1) to prove one disjunct of 28.6
New (30/28/27/14/12/6/3/2/1) to prove remaining disjuncts of 28
Proving (29/28)...
Success in (29/28)
Proving (30/28)...
Success in (30/28)
Success in (28/27)
Success in (27/14)
Proved: 2029 conses, 39.271 seconds

Path #3: 205...215-219(B=C)-305-310
1 conses, .054 seconds
((16 15 3 2 1))

Proof for (16/15) for goal 310:
New (31/16/15/3/2/1) for proof
Proving (31/16)...
Success in (31/16)
Proved: 267 conses, 5.006 seconds

Path #4: 100...215
1 conses, .043 seconds
((20 17 2 1) (22 17 2 1))

Proof for (20/17) for goal 205:
New (32/20/17/2/1) for proof
Proving (32/20)...
Success in (32/20)
Proved: 335 conses, 5.694 seconds
```

```
Proof for (22/17) for goal 215:
New (33/22/17/2/1) for proof
Proving (33/22)...
Success in (33/22)
Proved: 272 conses, 5.434 seconds

#TH 25
   3.1 A  .FA(I2,J2:I2>0 & I2<=J2 & J2<=N)A[I2]<=A[J2]
   3.2 A  B>0
  23.3 A  B<N
  25.4 A  A[B]<A[B+1]
  25.5 G  .FA(I2:I2>0 & B>=I2)A[B]>=A[I2]
  25.6 G  .FA(J2:B<=J2 & J2<=N)A[B]<=A[J2]
  25.7 G  A[B]<=A[N]
   -.-    T

#COUNTER 25

Proof for (25/23) to prove remaining disjuncts of 23.5:
New (36/25/23/8/7/6/3/2/1) for proof
Proving (36/25)...
New (37/36/25/23/8/7/6/3/2/1) for counterexample
Proving (37/36)...
Equalities:
  (from 3.2)   B => 1
Cancel 3.2: B>0
Cancel 23.3: B<N
  37.1     N>=2
Cancel 25.7: A[B]<=A[N]
  37.2     A[1]<=A[N]
Cancel 25.4: A[B]<A[B+1]
  37.3     A[1]<A[2]
Cancel 25.6: .FA(J2:B<=J2 & J2<=N)A[B]<=A[J2]
  37.4     .FA(J2:J2>0 & J2<=N)A[1]<=A[J2]
Cancel 25.5: .FA(I2:I2>0 & B>=I2)A[B]>=A[I2]
[I2=>1: A[1]>=A[I2] => T]
Matching 37.3 against 3.1 (score=70):
  J2 => 1
  I2 => 2
Matching 37.2 against 3.1 (score=70):
  J2 => 1
  I2 => N
From 37.2: A[1]>=A[N] => A[1]=A[N]
From 37.1: N#1 => T
Matching 37.1 against 37.4 (score=-150):
  J2 => N
From 37.1: N<=0 => NIL
From 37.2: A[1]<=A[N] => T
Matching 37.1 against 3.1 (score=-150):
  I2 => N
[J2=>N: J2=N & N>0 => N>0]
[J2=>N: A[J2]>=A[N] => T]
```

```
Equalities:
  (from 37.1)  N => 2
Cancel 37.1: N>=2
Cancel 37.2: A[1]<=A[N]
From 37.3: A[1]<=A[2] => T
Cancel 37.4: .FA(J2:J2>0 & J2<=N)A[1]<=A[J2]
[J2=>1: A[1]<=A[J2] => T]
[J2=>2: A[1]<=A[J2] => A[1]<=A[2]]
From 37.3: A[1]<=A[2] => T
Cancel 3.1: .FA(I2,J2:I2>0 & I2<=J2 & J2<=N)A[I2]<=A[J2]
  37.5    .FA(I2,J2:I2>0 & J2<=2 & I2<=J2)A[I2]<=A[J2]
Matching 37.3 against 37.5 (score=70):
  J2 => 1
  I2 => 2
Cancel 37.5: .FA(I2,J2:I2>0 & J2<=2 & I2<=J2)A[I2]<=A[J2]
From 37.3: A[1]<=A[2] => T
Equalities:
  (from 37.3)  A[2] => 0
Cancel 37.3: A[1]<A[2]
  37.6    A[1]<0
Equalities:
  (from 37.6)  A[1] => -1
Cancel 37.6: A[1]<0
Counterexample in (37/36)
Counterexample in (36/25)
Counterexample: 1354 conses, 57.767 seconds

#VALUES 37
C0=2, B0=1, X=A[2], C=2, I3=2, B=1, N=2, A[2]=0, A[1]=-1
```

```
#GET FINDERS
FINDERS.PIVOT;2
FINDVAL
New (2/1) for procedure FINDVAL
FINDL
New (3/1) for procedure FINDL
1110 conses, 14.105 seconds
T

#IN FINDVAL

#LIST
PROCEDURE FINDVAL
 110    ASSERT 1<=B<C<=N
 120    ASSERT .EX(B<=$K<=C)X=A[K]
 130    ASSERT .FA(I,J:1<=I<=J<=N)A[I]<=A[J]
 140    D ← (B+C)/2
 210    ASSUME X>A[D]
 215    ASSUME B=1, C=2, N=2, X=0, A[1]=-1
 220    LEMMA .EX(B1,C1)(1<=B1<=C1<=N & .EX(K1:B1<=K1<=C1)X=A[K1] &
  ..
             C1-B1<C-B)

#PROVE
1298 conses, 23.944 seconds
Paths ready.

Path #1: 110...220
New (4/2/1) for paths beginning 110...220
 110    ASSERT 1<=B<C<=N
In (4/2):
   4.1    B>0
   4.2    B<C
   4.3    C<=N
 120    ASSERT .EX(B<=$K<=C)X=A[K]
In (4/2):
New (5/4/2/1) to evaluate .EX(K:C>=K & B<=K)X=A[K]
   5.1    C>=K
   5.2    B<=K
   4.4    C>=K
   4.5    B<=K
   4.6    X=A[K]
 130    ASSERT .FA(I,J:1<=I<=J<=N)A[I]<=A[J]
In (4/2):
New (6/4/2/1) to evaluate .FA(I,J:I>0 & I<=J & J<=N)A[I]<=A[J]
   6.1    I>0
   6.2    I<=J
   6.3    J<=N
   4.7    .FA(I,J:I>0 & I<=J & J<=N)A[I]<=A[J]
 140    D ← (B+C)/2
In (4/2):
Set CVAL(D): (B+C)/2
```

```
 210   ASSUME X>A[D]
In (4/2):
CVAL(D) = (B+C)/2
Eval: A[D] => A[(B+C)/2]
Eval: X>A[D] => X>A[(B+C)/2]
   4.8    X>A[(B+C)/2]
 215   ASSUME B=1, C=2, N=2, X=0, A[1]=-1
In (4/2):
Cancel 4.1: B>0
   4.9    B=1
Set CVAL(D): (C+1)/2
Set CVAL(B): 1
   4.10   C=2
Set CVAL(D): 1
Set CVAL(C): 2
   4.11   N=2
Set CVAL(N): 2
   4.12   X=0
Set CVAL(X): 0
   4.13   A[1]=-1
 220   LEMMA .EX(B1,C1)(1<=B1<=C1<=N & .EX(K1:B1<=K1<=C1)X=A[K1] &
  ..
             C1-B1<C-B)
In (4/2):
Goal: .EX(B1,C1,K1:B1>0 & C1>=K1 & B+C1<B1+C & B1<=C1 & B1<=K1 & C1<
=N)X
=A[K1]
New (7/4/2/1) to evaluate .EX(B1,C1,K1:B1>0 & C1>=K1 & B+C1<B1+C &
B1<=
C1 & B1<=K1 & C1<=N)X=A[K1]
CVAL(B) = 1
CVAL(C) = 2
CVAL(N) = 2
   7.1    B1=C1
Set CVAL(C1): B1
   7.2    B1>0
   7.3    C1>=K1
   7.4    C1<=2
From 7.3 & 7.1: B1<=K1 => B1=K1
   7.5    B1=K1
Set CVAL(K1): B1
CVAL(X) = 0
[B1=>C1: B1=C1 & B1>0 & C1>=K1 & C1<=2 & B1<=K1
    => C1=K1 & (C1=1 ! C1=2)]
[B1=>C1: A[K1]=0 => A[K1]=0]
[C1=>1: C1=1 & C1=K1 => K1=1]
[C1=>1: A[K1]=0 => A[K1]=0]
[K1=>1: A[K1]=0 => A[1]=0]
[C1=>2: C1=2 & C1=K1 => K1=2]
[C1=>2: A[K1]=0 => A[K1]=0]
[K1=>2: A[K1]=0 => A[2]=0]
From 4.13: A[1]=0 => NIL
Eval: .EX(B1,C1,K1:B1>0 & C1>=K1 & B+C1<B1+C & B1<=C1 & B1<=K1 & C1<
=N)X
=A[K1] => A[2]=0
New (8/4/2/1) for goal 220
   8.1    A[2]#0
New (9/4/2/1) to evaluate .EX(B1,C1,K1:B1>0 & C1>=K1 & B+C1<B1+C &
B1<=
C1 & B1<=K1 & C1<=N)X=A[K1]
CVAL(B) = 1
CVAL(C) = 2
CVAL(N) = 2
   9.1    B1=C1
Set CVAL(C1): B1
   9.2    B1>0
   9.3    C1>=K1
   9.4    C1<=2
From 9.3 & 9.1: B1<=K1 => B1=K1
   9.5    B1=K1
Set CVAL(K1): B1
CVAL(X) = 0
[B1=>C1: B1=C1 & B1>0 & C1>=K1 & C1<=2 & B1<=K1
    => C1=K1 & (C1=1 ! C1=2)]
[B1=>C1: A[K1]=0 => A[K1]=0]
[C1=>1: C1=1 & C1=K1 => K1=1]
[C1=>1: A[K1]=0 => A[K1]=0]
[K1=>1: A[K1]=0 => A[1]=0]
[C1=>2: C1=2 & C1=K1 => K1=2]
[C1=>2: A[K1]=0 => A[K1]=0]
[K1=>2: A[K1]=0 => A[2]=0]
From 4.13: A[1]=0 => NIL
Eval: .EX(B1,C1,K1:B1>0 & C1>=K1 & B+C1<B1+C & B1<=C1 & B1<=K1 & C1<
=N)X
=A[K1] => A[2]=0
New (10/4/2/1) to bypass lemma 220
  10.1    A[2]=0
6032 conses, 151.672 seconds
((8 4 2 1))

Proof for (8/4) for goal 220:
New (11/8/4/2/1) for proof
Proving (11/8)...
   4.2 A  B<C
   4.3 A  C<=N
   4.4 A  C>=K
   4.5 A  B<=K
   4.6 A  X=A[K]
   4.7 A  .FA(I,J:I>0 & I<=J & J<=N)A[I]<=A[J]
   4.8 A  X>A[(B+C)/2]
   4.9 A  B=1
   4.10 A C=2
   4.11 A N=2
   4.12 A X=0
   4.13 A A[1]=-1
   8.1 G  A[2]#0
```

```
Equalities:
  (from 4.12)  X => 0
  (from 4.11)  N => 2
  (from 4.10)  C => 2
  (from 4.9)   B => 1
Cancel 4.9: B=1
Cancel 4.10: C=2
Cancel 4.11: N=2
Cancel 4.12: X=0
Cancel 4.5: B<=K
  11.1    K>0
Cancel 4.6: X=A[K]
  11.2    A[K]=0
Cancel 4.4: C>=K
Cancel 11.1: K>0
  11.3    K=1 ! K=2
Cancel 4.3: C<=N
Cancel 4.2: B<C
Cancel 4.8: X>A[(B+C)/2]
From 4.13: A[1]<0 => T
Cancel 4.7: .FA(I,J:I>0 & I<=J & J<=N)A[I]<=A[J]
  11.4    .FA(I,J:I>0 & J<=2 & I<=J)A[I]<=A[J]
Try proof by cases on 11.3: K=1 ! K=2
New (12/11/8/4/2/1) to prove one disjunct of 11.3
New (13/11/8/4/2/1) to prove remaining disjuncts of 11.3
Cancel 11.3: K=1 ! K=2
  12.1    K=2
Proving (12/11)...
Equalities:
  (from 12.1)  K => 2
Cancel 12.1: K=2
Cancel 11.2: A[K]=0
From 8.1: A[2]=0 => NIL
Success in (12/11)
Cancel 11.3: K=1 ! K=2
  13.1    K#2
Cancel 13.1: K#2
  13.2    K=1
Proving (13/11)...
Equalities:
  (from 13.2)  K => 1
Cancel 13.2: K=1
Cancel 11.2: A[K]=0
From 4.13: A[1]=0 => NIL
Success in (13/11)
Success in (11/8)
Proved: 1050 conses, 35.074 seconds
Success in (8/4)
```

```
#IN FINDL

#LIST
PROCEDURE FINDL
 100   ASSERT 1<=B=C<=N
 110   ASSERT A[B]=X
 200   LEMMA .EX(L)(1<=L<=N & A[L]=X)

#PROVE
332 conses, 8.441 seconds
Paths ready.

Path #1: 100...200
New (14/3/1) for paths beginning 100...200
 100   ASSERT 1<=B=C<=N
In (14/3):
  14.1    B=C
Set CVAL(C): B
  14.2    B>0
  14.3    C<=N
 110   ASSERT A[B]=X
In (14/3):
  14.4    X=A[B]
Set CVAL(X): A[B]
 200   LEMMA .EX(L)(1<=L<=N & A[L]=X)
In (14/3):
Goal: .EX(L:L>0 & L<=N)X=A[L]
New (15/14/3/1) to evaluate .EX(L:L>0 & L<=N)X=A[L]
  15.1    L>0
  15.2    L<=N
CVAL(X) = A[B]
Eval: .EX(L:L>0 & L<=N)X=A[L]
    => .EX(L:L>0 & L<=N)A[B]=A[L]
New (16/14/3/1) for goal 200
  16.1    .FA(L:L>0 & L<=N)A[B]#A[L]
New (17/14/3/1) to evaluate .EX(L:L>0 & L<=N)X=A[L]
  17.1    L>0
  17.2    L<=N
CVAL(X) = A[B]
Eval: .EX(L:L>0 & L<=N)X=A[L]
    => .EX(L:L>0 & L<=N)A[B]=A[L]
New (18/14/3/1) to bypass lemma 200
  18.1    L>0
  18.2    L<=N
  18.3    A[B]=A[L]
473 conses, 25.522 seconds
((16 14 3 1))
```

```
Proof for (16/14) for goal 200:
New (19/16/14/3/1) for proof
Proving (19/16)...
  14.1 A  B=C
  14.2 A  B>0
  14.3 A  C<=N
  14.4 A  X=A[B]
  16.1 G  .FA(L:L>0 & L<=N)A[B]≠A[L]
Equalities:
  (from 14.1)  C => B
  (from 14.4)  X => A[B]
Cancel 14.4: X=A[B]
Cancel 14.1: B=C
Cancel 14.3: C<=N
  19.1    B<=N
[L=>B: B=L & L>0 & L<=N => B>0 & B<=N]
[L=>B: NIL => NIL]
From 19.1: B>N => NIL
From 14.2: B<=0 => NIL
Success in (19/16)
Proved: 409 conses, 13.747 seconds
Success in (16/14)
```

PROGRAMS VERIFIED

This Appendix is a complete list of programs successfully verified by PIVOT. Proofs of King's Example 6, the Constable-Gries program, and Floyd's search program appear in Appendices A, B, and C respectively.

King's Example 1

```
PROCEDURE K1
 100   ASSERT Y=B>=0
 110   X ← 0
 120   LOOP
 125      WHILE Y#0: BEGIN
 150      X ← X + A
 160      Y ← Y - 1
 170      ASSERT X=A*(B-Y) AND Y>=0
 190      END
 200   ASSERT X=A*B
```

King's Example 2

```
PROCEDURE K2(A,B)
 100   ASSERT A>=0, B>=0
 110   Q ← 0
 120   R ← A
 200   LOOP
 210      ASSERT A=Q*B+R, R>=0
 220      WHILE R>=B: BEGIN
 300      Q ← Q+1
 310      R ← R-B
 320      END
 400   ASSERT A=Q*B+R, 0<=R<B
```

King's Example 3

```
PROCEDURE K3(A,B)
 100   ASSERT X=A, Y=B>=0
 110   Z ← 1
 200   LOOP
 210      ASSERT Y>=0, Z*(X**Y)=A**B
 220      WHILE Y#0: BEGIN
 300      IF Y MOD 2=1 THEN Z ← Z*X
 310      Y ← Y/2
 320      X ← X*X
 330      END
 400   ASSERT Z=A**B
```

King's Example 4

```
PROCEDURE K4(A)
 100   ASSERT A>=2
 110   I ← 2
 200   :L: BEGIN
 300      LOOP
 310         ASSERT .FA(2<=$K<I)A MOD K#0, I<=A
 320         WHILE I<A: BEGIN
 400         IF A MOD I#0 THEN I ← I+1
 410         ELSE BEGIN
 420            J ← 1
 430            EXIT L
 440            END
 450         END
 460      J ← 0
 500      END
 510   ASSERT J=0 IMP .FA(2<=$K<A)A MOD K#0
 520   ASSERT J=1 IMP A MOD I=0
```

King's Example 5

```
PROCEDURE K5(A,N)
 110   I ← 1
 200   LOOP
 210      ASSERT .FA(1<=$J<I)A[J]=0
 220      WHILE I<=N: BEGIN
 300      A[I] ← 0
 310      I ← I+1
 320      END
 400   ASSERT .FA(1<=$J<=N)A[J]=0
```

King's Example 6
(see Appendix A for proof)

```
PROCEDURE K6(A,N)
 100   ASSERT N>0
 110   I ← 2
 200   LOOP
 210      WHILE I<=N: BEGIN
 300      IF A[I-1]>A[I] THEN BEGIN
 400         X ← A[I]; A[I] ← A[I-1]; A[I-1] ← X
 410         END
 420      ASSERT .FA(1<=$K<I)A[I]>=A[K]
 430      ASSERT I<=N
 440      I ← I+1
 450      END
 500   ASSERT .FA(1<=$L<N)A[N]>=A[L]
```

King's Example 7

```
PROCEDURE K7
 200   :PASS: LOOP
 205      REPEAT: BEGIN
 210      J ← 0
 220      I ← 2
 300      LOOP
 350         ASSERT .FA(L:2<=L<I)(J#0 OR A[L-1]<=A[L])
 360         WHILE I<=N: BEGIN
 400         IF A[I-1] > A[I] THEN BEGIN
 410            X ← A[I-1]; A[I-1] ← A[I]; A[I] ← X
 420            J ← 1
 430            END
 450         I ← I + 1
 499         END
 550      IF J=0 THEN EXIT PASS
 599      END
 600   ASSERT .FA(M:2<=M<=N)A[M-1]<=A[M]
```

King's Example 8

```
PROCEDURE K8(A,B)
 100   ASSERT A=DA
 110   Y ← 0
 200   LOOP
 210      WHILE A#0: BEGIN
 220      ASSERT Y=(DA-A)*B
 300      IF A>0 THEN BEGIN
 310         XB ← B
 320         LOOP
 330            WHILE XB#0: BEGIN
 340            IF XB>0 THEN Y ← Y+1; XB ← XB-1
 350            ELSE Y ← Y-1; XB ← XB+1
 370            ASSERT Y=(DA-A)*B+B-XB
 380            END
 390         A ← A-1
 399         END
 400      ELSE BEGIN
 410         XB ← B
 420         LOOP
 430            WHILE XB#0: BEGIN
 440            IF XB>0 THEN Y ← Y-1; XB ← XB-1
 450            ELSE Y ← Y+1; XB ← XB+1
 470            ASSERT Y=(DA-A)*B-B+XB
 480            END
 490         A ← A+1
 499         END
 500      END
 510   ASSERT Y=DA*B
```

King's Example 9

King's system was unable to verify this program. Several people subsequently remarked that his assertions were not strong enough, but King himself believes that the correct assertions are too complex for his system to handle.

```
PROCEDURE K9(A,N)
      DECLARE ARRAY A
 110  I ← 1
 120  LOOP
 130     WHILE I < N: BEGIN
 131     J ← I+1
 132     X ← A[I]
 133     K ← I
 135     LOOP
 140       ASSERT I<=K<J<=N+1
 145       ASSERT I<N
 150       ASSERT X=A[K]
 160         ASSERTI>1IMP(.FA(M:I<=M<=N)(A[I-1]<=A[M]))
 170       ASSERT .FA(L:2<=L<I)(A[L-1]<=A[L])
 180       ASSERT .FA(M:I<=M<J)(X<=A[M])
 190       WHILE J <= N: BEGIN
 200       IF X > A[J] THEN BEGIN
 210           X ← A[J]
 220           K ← J
 230           END
 240         J ← J+1
 250         END
 260     A[K] ← A[I]
 270     A[I] ← X
 280     I ← I+1
 290     END
 300  ASSERT .FA(L:2<=L<=N)(A[L-1]<=A[L])
```

Constable-Gries program for enumerating formulas built from one constant symbol and one binary function symbol [Con8]. This program was adapted for PIVOT by replacing the original statement 250, A[k] ← f(A[i],A[AS[i]]), by the one below, which essentially makes f the familiar function that enumerates the pairs of integers in "diagonal" order. This was only done because PIVOT cannot express the idea of enumerating all formulas without repetition.

```
PROCEDURE  CG
       DECLARE ARRAY A, AS, AD
       LET PAIR(X,Y)=(X+Y+1)*(X+Y)/2+Y+1
 100   K ← 0
 110   I ← 0
 120   A[0] ← 0
 130   AS[0] ← 0
 140   AD[0] ← 0
 150   LOOP
          ASSERT .FA(J:0<=J<=K)(A[J]=J)
 160      ASSERT .FA(J:0<=J<=K)(A[J]=J)
 170      ASSERT .FA(J:1<=J<=K)(AD[J]=J-1)
 180      ASSERT K=PAIR(I,AS[0]-I)-1
 190      ASSERT 0<=I<=AS[0]<=K
 200      ASSERT .FA(J:0<=J<=I)(AS[J]+J=AS[0])
 210      ASSERT .FA(J:I<J<=AS[0])(AS[J]+J=AS[0]+1)
 215      ASSERT .FA(J:AS[0]<J<=K)(AS[J]=0)
 220      REPEAT: BEGIN
 230      K ← K+1
 250      A[K] ← PAIR(A[I],A[AS[I]])
 260      AS[K] ← 0
 270      AD[K] ← K-1
 280      AS[I] ← AS[I]+1
 290      AD[0] ← AS[0]
 300      I ← AD[I]
 310      END
```

Hoare's FIND program

```
PROCEDURE FIND
 198   ASSERT 1<=F<=NN
 200   M ← 1; N ← NN
 210   :REDUCE: LOOP
 211      DECLARE 1<=M<=F<=N<=NN
 212      DECLARE .FA(P1,Q1:1<=P1<M<=Q1<=NN)A[P1]<=A[Q1]
 213      DECLARE .FA(P2,Q2:1<=P2<=N<Q2<=NN)A[P2]<=A[Q2]
 220      WHILE M<N: BEGIN
 230      R ← A[F]
 240      I ← M; J ← N
 250      LOOP
 255         DECLARE I>=M, J<=N
 256         DECLARE .FA(P0:1<=P0<I)A[P0]<=R
 257         DECLARE .FA(Q0:J<Q0<=NN)R<=A[Q0]
 260         WHILE I<=J: BEGIN
 270         LOOP
 271            ASSERT 0=0
 280            WHILE A[I]<R: I ← I+1
 290         LOOP
 299            ASSERT A[I]>=R
 300            WHILE R<A[J]: J ← J-1
 301         LEMMA A[J]<=R<=A[I]
 310         IF I<=J THEN BEGIN
 320            W ← A[I]; A[I] ← A[J]; A[J] ← W
 321            LEMMA A[I]<=R<=A[J]
 330            I ← I+1; J ← J-1
 340            END
 350         END
 360      IF F<=J THEN N ← J
 370      ELSE IF I<=F THEN M ← I
 380      ELSE EXIT REDUCE
 430      END
 431   ASSERT .FA(P:1<=P<=F)A[P]<=A[F]
 432   ASSERT .FA(Q:F<=Q<=NN)A[F]<=A[Q]
```

Example from Floyd's IFIP paper
(see Appendix C for proofs)

```
PROCEDURE FLOYD
 100   ASSERT N>=1
 110   ASSERT .FA(I0,J0:1<=I0<=J0<=N)A[I0]<=A[J0]
 120   ASSERT .EX(I1:1<=I1<=N)X=A[I1]
 170   B ← 1
 180   C ← N
 200   LOOP
 205      ASSERT .FA(I2,J2:1<=I2<=J2<=N)A[I2]<=A[J2]
 210      ASSERT 1<=B<=C<=N
 215      ASSERT .EX(I3:B<=I3<=C)X=A[I3]
 219      WHILE B#C: BEGIN
 220      ASSUME B0=B, C0=C
 230      D ← (B+C)/2
 250      IF X > A[D] THEN B ← D
 260      ELSE C ← D
 290      LEMMA C-B<C0-B0
 300      END
 305   L ← B
 310   ASSERT 1<=L<=N AND X=A[L]
```