VERIFICATION OF APL PROGRAMS

by

Susan Lucille Gerhart

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213
November, 1972

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

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.
ACKNOWLEDGEMENTS

I would like to express my appreciation to my thesis advisor, Donald L. Loveland, for his patience and helpful suggestions and also to the other members of the thesis committee, Nico Habermann, Allen Newell, and Roger-Pederson.

This thesis has traded heavily on the novelty of APL and in the sense that it expresses a unique approach to program verification, perhaps it exemplifies the famous aphorism, due to A.J. Perlis, that "One man's constant is another man's variable." Finally, I would like to express my thanks and best wishes to the "brownie plate" group.
## TABLE OF CONTENTS

### ABSTRACT
vi

### INTRODUCTION
1

#### Chapter 1. INTRODUCTION TO APL
6
   History of APL
6
   Function definition
7
   Expressions
8
   Operators
9
   Assignment
21
   Example

#### Chapter 2. VERIFICATION METHODOLOGY
22
   Definition of correctness
22
   Review of previous work
23
   Description of the inductive assertion method
28
   Formal basis for verification of APL programs
32
   Example of informal verification-vector left justification
47
   Pragmatic questions about verification
52
   Summary
56

#### Chapter 3. FORMAL DEFINITION OF APL OPERATORS
58
   Introduction
58
   Notation and conventions
60
   Values
61
   Primitive operators
62
   Nonprimitive operators
67
   Constraints
74
   Definition of assignment
78
   Derived properties
78
   Discussion of the definition
79
   Summary
82

#### Chapter 4. VERIFICATION OF CONSTRAINTS
83
   Introduction
83
   Errors in APL programs
84
   Goals of the method
86
   Outline of the method
88
   Basic algorithms
95
      SLC
      Examples
109
      SLA
115
      BA and BC
116
   Coding methods and performance
121
   Summary and conclusions
124
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Scalar operators</td>
<td>16</td>
</tr>
<tr>
<td>1.2</td>
<td>Syntax and examples of APL operators</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Scalar operators</td>
<td>71</td>
</tr>
<tr>
<td>3.2</td>
<td>Summary of definition of APL operators</td>
<td>72</td>
</tr>
<tr>
<td>3.2</td>
<td>Constraints on APL operators</td>
<td>76</td>
</tr>
<tr>
<td>4.1</td>
<td>Properties of APL operators</td>
<td>96</td>
</tr>
<tr>
<td>4.2</td>
<td>Shape constraints and attributes</td>
<td>97</td>
</tr>
<tr>
<td>4.3</td>
<td>Bounds constraints and attributes</td>
<td>98</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Scalar operator bounds</td>
<td>99</td>
</tr>
<tr>
<td>4.4</td>
<td>Mode constraints and attributes</td>
<td>100</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Scalar operator modes</td>
<td>101</td>
</tr>
<tr>
<td>5.1</td>
<td>APL deductive steps</td>
<td>132</td>
</tr>
<tr>
<td>5.2</td>
<td>List of derived properties</td>
<td>139</td>
</tr>
<tr>
<td>E.1</td>
<td>TTY Operator codes</td>
<td>212</td>
</tr>
</tbody>
</table>
ABSTRACT

VERIFICATION OF APL PROGRAMS

APL differs from other programming languages in having operators designed to facilitate uniform array operations. Those APL programs which submerge their control (looping and branching) into these operators require relatively few inductive assertions when verified by Floyd's method. The thesis studies the nature of verification for this class of programs.

A formal definition of the APL language which emphasizes those properties of the operators necessary for verification is given. A few operators are chosen as primitive and defined by ad hoc but precise descriptions and the remaining operators are defined in a standard form in terms of the primitive operators. This definition is used as the basis for an informal deductive system in which it is possible to manually carry out the proofs of assertions about APL programs.

APL permits variables to take on any mode (integer, real, etc.) or shape (scalar, vector, etc.) but the operators have constraints on their operands which must be satisfied for the operation to be properly executable. The formal basis for verifying programs with partial operators is given and an implemented system for verifying these constraints is described. The system generates initial and

vi
intermediate predicates which, if satisfied, guarantee that no errors in performing operations will occur. It is also possible to verify assigned assertions related to mode and shape in this same manner. The overall effect of this limited approach to verification is that the system user is relieved of a large part of the burden of making assertions for and verifying the properties of his program which have to do with consistency of shapes and modes, and to some extent, subscripts.

Several programs are verified, partially through the mechanized system and partially informally in the deductive system. These programs illustrate the difficulty of finding assertions, the usefulness of derived properties of the operators, and the inherent mathematical character of some verification proofs, for which current mechanical theorem proving techniques appear inadequate. APL, with the addition of a bounded quantification operator, serves as a suitable assertion language.

The theme of the thesis is the study of the verification task within the context of a specific programming language, APL, which has features which make it well-suited for program verification. This has made it possible to perform the informal verification of much more difficult problems than previously verified, while requiring and permitting the development of mechanical verification techniques for the shape, mode, and subscript properties of APL programs.
INTRODUCTION

Establishing that a program is "correct" is a two-step process: first, specifying the intended purpose of the program and, second, showing that the program achieves that purpose. Every programmer is familiar with the usual requirements and difficulties of debugging and testing a program to establish the second step. In recent years, much attention has been placed on the development of techniques for proving the correctness of programs since testing, in general, is inconclusive. The first step falls into the domain of design methodology and will be considered in this thesis only to the extent of providing a language for stating specifications.

The most widely used such correctness technique is the inductive assertion method suggested by Floyd[FL1]. It presupposes that the program's purpose can be stated precisely in terms of a restriction on the possible inputs of the program and a condition which is to be satisfied after the execution of the program for input satisfying the restriction. The method requires that, in addition to the input and output conditions, every loop of the program have at least one (inductive) predicate associated with it that characterizes the computation of the loop. A program is "correct with respect to the input-output relation" if it can be shown that during any computation of the program for data satisfying the input condition when any predicate associated with the program is true and the statements leading to the next predicate are executed, the latter predicate is also true. In this way, the truth of the predicates is maintained for a computation of any number of steps, and consequently, if the output
INTRODUCTION

predicate is reached, it is also true. The basic nature of the method involves showing the consistency of the program with the statements made about the program.

Though theoretically complete, given all the inductive predicates, in practice difficulties frequently arise. It is not always easy to create the inductive predicates, even knowing the program well. Nor is it easy to prove the consistency of assertions with program statements, primarily because a vast amount of detail is usually involved. Since larger programs may have several loops, many of which accomplish some relatively simple action, it becomes quite tedious to create the inductive assertions and then to carry out the proofs. It would be helpful to factor out some of the more trivial loops so that attention may be focused on the assertions essential to the problem for which the program represents a solution.

APL has identified a number of common loop formats and incorporated them directly into the language as operators. Examples are element by element application of a scalar operator to two variables, accumulative application of a scalar operator to a vector, search for the first element of a vector satisfying a certain property, rearrangement of elements of an array, construction of a sequence of integers, and selection of elements from an array. The operators are said to be "structured"; they are applied to data according to the array structure of that data.

The overall effect of the use of these operators is that many expressions have a clear synthetic character, where intermediate
values are created with certain well-defined properties. This effect is important for verification, since the structure imposed by such expressions will be directly expressible in terms of the meaning of the operators of the expressions without the need for many inductive assertions. For example, if two nonempty vectors $A$ and $B$ of the same length are added element by element, written in APL as $A+B$, then the result, $R$, can be described by the statement "every element, $R[i]$, of the result has the value $A[i]+B[i]$ ". There is no need for an inductive assertion and the full meaning of the operator (and its associated loop) can be expressed directly by the above formal definition of the operation.

The use of such powerful operators entails some complications, since each operator has restrictions, referred to here as CONSTRAINTS, on the operands for which it can be executed. Because APL does not require declaration of attributes such as mode, rank, or size, the presence or absence of violations of these restrictions in a program is completely determined by the attributes of the values of the variables at the time the program is executed.

THIS THESIS STUDIES PROGRAM VERIFICATION IN THE LIGHT OF THESE OBSERVATIONS.

Several important general questions arose in the course of the study and the thesis will address the questions in the specific context of the APL language.

1. What does it mean to say that a program is correct, especially when there are conditions under which the program operations may be undefined?
INTRODUCTION

2. What are the requirements on the description of a language such that programs in the language can be verified in some rigorous fashion?

3. What forms do assertions take and, consequently, what is a suitable language for making assertions?

4. What techniques, informal and mechanical, should or must be available to verify the programs? To what extent can verification be mechanized?

Chapter 1 presents a short history and summary of APL. It is intended as a review of or introduction to the language for the reader not familiar with APL, though all readers should become acquainted with the terminology and operator classification.

Chapter 2 discusses the methodology of program verification and extends the formalism in current use to cover the particular features of APL, especially the partial operators. APL will also be used as the basis for an assertion language.

The formal description of the APL operators is presented in Chapter 3. Certain operators are regarded as primitive and the remaining operators are given in a standard form consisting of restrictions on the operands, an expression for the shape of the result of the operation, and an expression for a single element of the result.

The problem of verifying that the operations of a program can be executed properly is considered in Chapter 4. Since the attributes of mode, shape, and rank are undeclared in APL, it is fair to make assertions about these attributes. A collection of algorithms are presented which accomplish part of the verification task by reducing assertions and constraints while determining the consistency of
constraints. An interesting side effect of the underlying procedure is that assertions about attributes may be generated or strengthened.

Finally, proving general properties of APL programs is discussed in Chapter 5. An informal (nonmechanical) deductive system is presented and used in verifying several examples. The next steps toward a mechanical prover are discussed.

The appendices include examples and details not supplied in the text. Appendix E contains a summary of all notation.

The conclusion suggests a broader view of verification including various forms of semantic checking of programs in lieu of debugging.

This thesis represents a shift in emphasis from current and previous work on program verification in that a single, commonly used language, APL, serves as the base for a broad study of program verification technicalities. The language feature of most interest is APL's set of operators into which it is possible to embed many of the loops that would otherwise occur in programs. The study of APL operators has led to the development of formalism and mechanical procedures for dealing with partial operators as well as a description of the semantics of the operators which leads to some interesting proof techniques.
CHAPTER 1
AN INTRODUCTION TO APL

HISTORY OF APL

In the early 1960's, K. Iverson's book, "A Programming Language"[IV1], introduced a programming notation which has since been applied to a wide variety of programming problems and has spurred the development of one of the first successful interactive language systems.

Such common mathematical concepts as sets, positional number systems, the sigma function, inner products, random numbers, and transposition of arrays have all been incorporated as operators of the language. Iverson's book illustrates the use of the notation for representation of trees and graphs and for description of such algorithms as searching and sorting, microprogramming, and translation of expressions to prefix form. The notation has also been used for describing the IBM 360[FIS].

The language with its associated system, APL\360, has gathered a great number of followers in the past few years. A newsletter published by ACM SIGPLAN reports user group activities, continued development of the language and system, and algorithms and suggestions submitted by users.

Note: This chapter may be skimmed by persons familiar with APL. It is intended as an informal introduction to APL that should be supplemented by the reference manuals if questions arise.
The first system was IBM's APL\3G0, and variations of the system and language have been implemented on many different computers. APL\360 is more than just a programming language since it includes facilities for libraries, text editing, communication with other users, and interactive program execution.

See the user's manuals, Falkoff and Iverson[FI2] and Pakin[PA], for more details of the language and system.

FUNCTIONS

Programs are constructed by defining functions, where each function consists of a function header including the function's name, parameters, local variables, and return value, followed by a function body which is a series of numbered statements.

EXAMPLE:

```
PASCAL N
[1] J←1 1
[2] OUTPUT: J
[4] J←(J,0)+(0,J)
[5] →OUTPUT
```

`PASCAL` is the function's name and `N` is its only parameter. `J` is a local variable. The `∇` marks the beginning and end of the function definition. Functions may take 0, 1, or 2 parameters, with prefix form for 1 parameter and infix for 2 parameters.

Control flows from one statement to the next higher numbered one except where redirected by a branch. The syntax of a branch is `→E` where `E` is any APL expression which meets certain restrictions that
will be discussed later. The value of the expression designates the next statement number to be executed if the function has such a numbered statement, otherwise the function is exited. In example 1, line 3 is a conditional exit, statement 5 is a branch to line 2, and \textit{OUTPUT} is the label of statement 2.

\textbf{EXPRESSIONS-VALUES}

The primitive values in APL are numbers and characters. Numbers are written in decimal notation or using the familiar "E" notation, e.g.

\begin{align*}
1. & \quad 0.005 \quad 3.14E+1
\end{align*}

Negative numbers are written using a special character ( - ) distinguished from the minus symbol, e.g.

\begin{align*}
-1 & \quad -4E-10
\end{align*}

Characters are enclosed in single quotes, e.g.

\begin{align*}
'\text{A}' & \quad '1'
\end{align*}

All values may be considered to be arrays in the sense of having a "shape", which gives the rank (number of dimensions) and size (number of components along each dimension). The following table summarizes the shapes for the terminology commonly used:

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\textit{ARRAY TYPE} & \textit{SHAPE} & \textit{SIZE} \\
\hline
\textit{SCALAR} & \quad \textit{RANK} & \quad \textit{EMPTY VECTOR} \\
\textit{VECTOR} & \quad 0 & \quad \textit{L} \\
\textit{MATRIX} & \quad 1 & \quad \textit{L}, \textit{L1}, \textit{L2} \\
\textit{N-DIMENSIONAL ARRAY} & \quad 2 & \quad \textit{L1,}, \textit{L2}, \ldots, \textit{LN} \\
\hline
\end{tabular}
\end{center}
Shape constitutes one property of a value, the other being the actual elements of the array. Two APL operators provide for "selecting" these two aspects of values: \( \text{SHAPE}(p) \) applied to a value returns its shape while \( \text{RAVEL}(,) \) returns its elements in row major order, i.e., with last subscript varying fastest. For example, where \( M \) is the matrix
\[
\begin{pmatrix}
0 & 1 & 2 \\
3 & 4 & 5 \\
\end{pmatrix}
\]
\( pM \) is 2,3 and \( ,M \) is 0,1,2,3,4,5. Ravel is also used as notation for a one-element vector, e.g. \( ,5 \). The elements are referred to in subscripts relative to the origin, a system parameter which can be set to either 0 or 1.

Constant numerical vectors are written by separating numerical scalars by blanks or commas, e.g.
\[
1 2 3 \quad 1,2,3
\]
Constant character vectors are strings of characters enclosed in quotes, e.g.
\[
'\text{ABC}' \quad 'A'
\]

EXPRESSIONS-OPERATORS

APL has a large set of built-in operators; some are applied to scalars but others are particularly designed to operate on arrays, unlike in many other programming languages. A monadic operator is written in prefix form while dyadic operators are written in infix. Many operator symbols are employed for both monadic and dyadic
operators and it is useful to distinguish among some terms:

An "operator" has a fixed form (number of arguments) and meaning. There will be names for all operators.

An "operator symbol" denotes either the monadic or dyadic operator with which it is associated, depending on syntactic context.

An "operation" is the execution of an operator according to its meaning and with evaluated operands.

The evaluation rule for APL expressions is quite simple: Every operation takes as its right operand, the entire expression to its right. There is no hierarchy among operators, although, of course, parentheses may be used to delimit expressions. Therefore, operations are executed in the order of their appearance from right to left except where reordered by parentheses.

Just as there were two aspects to values (shape and elements), there are two aspects to operators. The meaning of an operator is given both by the shape of the result of the operation and by the values of the individual elements of the result. But there is one more consideration for operators: for an operation to be performed, its operands must usually conform to some constraints. The restrictions are mentioned in passing here but will be given completely in Chapter 3.

Some coherence can be given to the operators by grouping them according to their effect—construction, selection, rearrangement, scalar arithmetic and logical, compositions of scalar operators, searching, number system representation, and random number generation. These classes are discussed separately and examples for all operators
appear in table 1.2.

1) Construction operators

We have seen that values may be scalars, vectors, matrices, etc. so there must be operators to construct such values.

CATENATION(,) is the usual operation of concatenating scalars or vectors together into new vectors.

INDEX GENERATOR(\) creates a vector which is a sequence of integers, starting at the origin (a system parameter) and of length N, where N is the argument. For example, \(13\) is 0,1,2 in 0 origin and 1,2,3 in 1 origin.

The two aspects of values (shape and elements) are combined through the RESHAPE(\) operator. The left operand is a vector which is to be the shape of the constructed result while the elements of the result are taken in row-major order cyclically from the right operand.

EXPANSION(\) * is also a construction operation in the sense that a new object is created with 0's (or blanks) inserted corresponding to the 0's in the left operand, which is a boolean vector. The remaining elements are those of the right operand.

2) Selection

In most programming languages, the common form of selection is the subscripting(indexing) operation. APL follows the form for the

*Many operators have an additional operand which designates a coordinate, in which case the operation is applied along that coordinate. See the examples(Table 1.2).
simple case where a single element is selected by giving a list of scalar subscripts which designate that element. But indexing in APL is much more general in that a subscript may be elided, in which case every element along that dimension is selected, and in permitting arbitrary shaped values as subscripts. In the latter case, the elements which are selected are designated by the "Cartesian product" of the elements of all subscripts. The shape of the result is the concatenation of the shapes of all subscripts.

The $\text{T\!A\!K\!E}^{(\uparrow)}$ and $\text{DROP}^{(\downarrow)}$ operators permit the selection of initial and final segments of the right operand of a size designated by the left operand.

$\text{COMPRESS}^{\theta} \text{ION}$ is a form of selection governed by a boolean vector as left operand. The elements corresponding to 1's in the vector are those selected.

3) Rearrangement

$\text{REVERSE}^{\text{MONADIC }} \phi$ creates an object in which the elements are in reverse order along the designated coordinate.

$\text{ROTATE}^{\text{DYADIC }} \phi$ uses the left operand to govern the left or right rotation of elements of the right operand along the designated coordinate.

$\text{TRANSPOSE}^{(\pi)}$ has several forms. As a monadic operator, the last two coordinates are reversed; for a matrix, this gives the mathematical transpose. In general, the left operand is used as a permutation on the subscripts. When the left operand has duplicate
.elements it is possible to select and rearrange simultaneously, for example to obtain diagonals.

4) Scalar

As can be seen from Table 1.1, the scalar operators cover a variety of common mathematical notions. As the name implies, scalar operators are applied to scalar values, but the operators are extended to non-scalar operands by applying the operator on an element-by-element basis. For the dyadic case, this means that the operator is applied between corresponding pairs of elements of the operand. In the case where an operand has a single element, the operation is applied between that element and every element of the other operand. This implies that there is some correspondence between shapes of operands and that the shape of the result is the shape of the "largest" operand.

5) Composition of scalar operations

The scalar operators are composed in three different senses: reduction, inner product and outer product.

REDUCTION(D/) is the accumulative application of a dyadic scalar operator D to a vector. It is a generalization of the mathematical sigma function to the other dyadic operators while following the right to left evaluation rule. If D is a dyadic scalar operator and V a vector then D/V is
i) if $V$ is empty, then the identity for $D$, if any, and

ii) if $V$ has one element, then that element, otherwise

iii) if $V$ is $V_0, V_1, \ldots, V_N$ then $V_0 D D1 V_1, \ldots, V_N$.

The application of a dyadic operator to all pairs of elements in row-major order is the $\text{OUTER PRODUCT}(\cdot, D)$.

The $\text{INNER PRODUCT}(D1, D2)$ is a generalization of the standard matrix and vector products of linear algebra to the full range of dyadic scalar operators. $D2$ is applied element-by-element to a row of the left operand and a column of the right operand, followed by reduction with $D1$ on that result.

An operator related to reduction is $\text{SCAN}(D\backslash)$ where the reduction is applied over successive initial sequences of the operand.

$\text{BASE VALUE}(\cdot)$ is related to the inner product in that the left operand supplies a base for a number system from which a weight vector is formed. The result is the inner product of the weight vector and the right operand.

6) Searching

The $\text{RANKING}(\text{DYADIC} \cdot)$ operator locates the first occurrence of the right operand in the left, returning the index of that occurrence if one is found. If there is no occurrence, then the value returned is $(\text{length of the left operand})\cdot\text{origin}$.

$\text{MEMBERSHIP}(\cdot)$ returns a 1 or 0 indicating whether or not the left operand was located in the right operand.
GRADE-UP(\(\delta\)) and GRADE-DOWN(\(\Psi\)) produce a vector which gives the indices of the elements of the operand in ascending and descending order, respectively.

7) Number System Representation

\textit{REPRESENTATION}(\(r\)) produces the representation of the right operand in the number system for which the left operand is the base.

8) Random number generation

\textit{ROLL}(\text{MONADIC} \ ?) returns a number randomly chosen from the range \textit{ORIGIN} to \(N+\text{ORIGIN}-1\), where \(N\) is the parameter.

\textit{DEAL}(\text{DYADIC} \ ?) returns a vector for which the length is given by the left operand and the elements are chosen without repetition from the range supplied by the right operand.
<table>
<thead>
<tr>
<th>Monadic form fA</th>
<th>Dyadic form f fB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PLUS</strong></td>
<td><strong>PLUS</strong></td>
</tr>
<tr>
<td><strong>NEGATIVE</strong></td>
<td><strong>MINUS</strong></td>
</tr>
<tr>
<td><strong>SIGNUM</strong></td>
<td><strong>TIMES</strong></td>
</tr>
<tr>
<td><strong>RECIPIROCAL</strong></td>
<td><strong>DIVIDE</strong></td>
</tr>
<tr>
<td><strong>CEILING</strong></td>
<td><strong>MAXIMUM</strong></td>
</tr>
<tr>
<td><strong>FLOOR</strong></td>
<td><strong>MINIMUM</strong></td>
</tr>
<tr>
<td><strong>EXPONENTIAL</strong></td>
<td><strong>POWER</strong></td>
</tr>
<tr>
<td><strong>NATURAL</strong></td>
<td><strong>LOGARITHM</strong></td>
</tr>
<tr>
<td><strong>LOGARITHM</strong></td>
<td><strong>LOGARITHM</strong></td>
</tr>
<tr>
<td><strong>MAGNITUDE</strong></td>
<td><strong>RESIDUE</strong></td>
</tr>
<tr>
<td><strong>FACTORIAL</strong></td>
<td><strong>BINOMIAL COEFFICIENT</strong></td>
</tr>
<tr>
<td><strong>ROLL</strong></td>
<td><strong>DEAL</strong></td>
</tr>
<tr>
<td><strong>PI TIMES</strong></td>
<td><strong>CIRCULAR(TRIGONOMETRIC)</strong></td>
</tr>
<tr>
<td><strong>NOT</strong></td>
<td><strong>AND</strong></td>
</tr>
<tr>
<td></td>
<td><strong>OR</strong></td>
</tr>
<tr>
<td></td>
<td><strong>NAND</strong></td>
</tr>
<tr>
<td></td>
<td><strong>NOR</strong></td>
</tr>
<tr>
<td></td>
<td><strong>LESS</strong></td>
</tr>
<tr>
<td></td>
<td><strong>NOT GREATER</strong></td>
</tr>
<tr>
<td></td>
<td><strong>EQUAL</strong></td>
</tr>
<tr>
<td></td>
<td><strong>NOT LESS</strong></td>
</tr>
<tr>
<td></td>
<td><strong>GREATER</strong></td>
</tr>
<tr>
<td></td>
<td><strong>NOT EQUAL</strong></td>
</tr>
</tbody>
</table>
TABLE 1.2  
NON-SCALAR OPERATORS

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>SYNTAX</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHAPE</strong></td>
<td>\p/A</td>
<td>\p/3=\p/3, 5=\p/10, (3,4)/12=\p/3,4</td>
</tr>
<tr>
<td><strong>RAVEL</strong></td>
<td>\A</td>
<td>,\p/3=0,1,2, ,\p/5=5, ,\p/3,4,\p/12=12</td>
</tr>
<tr>
<td><strong>CATENATE</strong></td>
<td>\V 1,2</td>
<td>,\p/3=0,1,2, ,\p/10=1, ,\p/3,\p/12=0,1,2</td>
</tr>
<tr>
<td><strong>INDEX</strong></td>
<td>\i/S</td>
<td>,\p/3=0,1,2, ,\p/11=0</td>
</tr>
<tr>
<td><strong>GENERATOR</strong></td>
<td>I</td>
<td>,\p/3=0,1,2</td>
</tr>
<tr>
<td><strong>RESHAPE</strong></td>
<td>\V \p/A</td>
<td>(3,4)/12=0 1 2 3, 4 5 6 7 8 9 10 11</td>
</tr>
<tr>
<td><strong>EXPANSION</strong></td>
<td>\V [S]A</td>
<td>0 1 0 0 0 0 1 0 2, 1 0 1 0 1 0 (3,4)/12=0 1 2 3, 0 0 0 0 4 5 6 7, 8 9 10 11</td>
</tr>
</tbody>
</table>

**note:** 0 origin is assumed throughout

- **Operand symbols:**
  - **S** Scalar or one-element shape
  - **V** Vector or scalar or one-element shape
  - **A** Any shape
  - **D** Dyadic operator

Identity of left and right sides is denoted by \leftrightarrow.

The following values are used throughout the examples:

\,\p/3=0,1,2, \,\p/5=5, \,\p/3,4,\p/12=12

\,\p/3=0,1,2, \,\p/10=1, \,\p/3,\p/12=0,1,2
### INDEXING

\[ V[A] \]
\[ A[A0;...] \]

| \((3,0)\) | \((0,2,1,2)\) | \(2,10,6\) |
| \((3,4)\) | \((2,2)\) | \(3;\) |

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
0 & 1 & 2 & 3 \\
\end{array}
\]

\[ \rho((3,4)\backslash12)((2,2)\backslash3);\leftrightarrow2,2,4 \]

### TAKE

\[ V \uparrow A \]

\[
\begin{array}{c}
2\uparrow1;3\leftrightarrow0,1 \\
-2\uparrow1;3\leftrightarrow1,2 \\
(2,2)\uparrow(3,4)\backslash12\leftrightarrow2,3 \\
6 & 7 \\
\end{array}
\]

### DROP

\[ V \downarrow A \]

\[
\begin{array}{c}
2\downarrow1;3\leftrightarrow2 \\
-2\downarrow1;3\leftrightarrow0 \\
(2,2)\downarrow(3,4)\backslash12\leftrightarrow8,9 \\
\end{array}
\]

### COMPRESSION

\[ V\backslash[S]A \]

\[
\begin{array}{c}
101\backslash1;3\leftrightarrow0,2 \\
101\backslash[0];(3,4)\backslash12\leftrightarrow \\
0 & 1 & 2 & 3 \\
8 & 9 & 10 & 11 \\
\end{array}
\]

### REVERSE

\[ \phi[S]A \]

\[
\begin{array}{c}
\phi13\leftrightarrow2,1,0 \\
\phi(3,4)\backslash12\leftrightarrow \\
3 & 2 & 1 & 0 \\
7 & 6 & 5 & 4 \\
11 & 10 & 9 & 8 \\
\end{array}
\]

### ROTATE

\[ V\phi[S]A \]

\[
\begin{array}{c}
1\phi(3,4)\backslash12\leftrightarrow \\
1 & 2 & 3 & 0 \\
5 & 6 & 7 & 4 \\
9 & 10 & 11 & 8 \\
(13)\phi(3,4)\backslash12\leftrightarrow \\
0 & 1 & 2 & 3 \\
5 & 6 & 7 & 4 \\
10 & 11 & 8 & 9 \\
\end{array}
\]

### TRANSPOSE

\[ \psi[A] \]

\[
\begin{array}{c}
\psi(3,4)\backslash12\leftrightarrow \\
0 & 4 & 8 \\
1 & 5 & 9 \\
2 & 6 & 10 \\
3 & 7 & 11 \\
\end{array}
\]

\[ V\psi[A] \]

\[
\begin{array}{c}
0 & 0\psi(3,4)\backslash12\leftrightarrow0,5,10 \\
\end{array}
\]

### REDUCTION

\[ D\backslash[S]A \]

\[
\begin{array}{c}
+/13\leftrightarrow0,1+2\leftrightarrow3 \\
-\backslash13\leftrightarrow-1\rightarrow2\leftrightarrow1 \\
+/0\leftrightarrow0 \\
+/\psi(3,4)\backslash12\leftrightarrow6,22,38 \\
\end{array}
\]
# Introduction to APL Operator Tables

## Outer Product

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 .D .A2</td>
<td>(i3) .+ .i3 ↔ 0 1 2 1 2 3 2 3 4</td>
<td></td>
</tr>
</tbody>
</table>

## Inner Product

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 D1.D2 .A2</td>
<td>(i3) .× .i3 ↔ / (i3) .i3 ↔ +/0 1 4 ↔ 5</td>
<td></td>
</tr>
</tbody>
</table>

## Scan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>D \S .A</td>
<td>+ \i3 ↔ 0 1 3 - \i3 ↔ 0 - 1 1</td>
<td></td>
</tr>
</tbody>
</table>

## Base

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>V \A</td>
<td>10 10 10 \i3 ↔ 100 10 1 .× 0 1 2 ↔ 12</td>
<td></td>
</tr>
<tr>
<td>(i3) .i(3,4) .p .i12 ↔ 2 2 1 .× (3,4) .p .i12 ↔ 16 21 26 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Ranking

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>V \v2</td>
<td>(i3) \i1 ↔ 1 (i3) \i12 ↔ 3 (i3) \i4 2 ↔ 3,2</td>
<td></td>
</tr>
</tbody>
</table>

## Membership

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 .A2</td>
<td>1 \i3 ↔ 1 4 \i3 ↔ 0 1 4 \i3 ↔ 1 0</td>
<td></td>
</tr>
</tbody>
</table>

## Grade Up

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>§</td>
<td>$5 1 2 1 3 ↔ 1 3 2 4 0</td>
<td></td>
</tr>
</tbody>
</table>

## Grade Down

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>¶</td>
<td>¶5 1 2 1 3 ↔ 0 4 2 1 3</td>
<td></td>
</tr>
</tbody>
</table>

## Representation

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>V \A</td>
<td>10 10 10 \i12 ↔ 0 1 2 14 - 2 7 \i91 ↔ - 105 10 10 10 \i12 8 ↔ 0 0 1 0 2 8</td>
<td></td>
</tr>
</tbody>
</table>

## Roll

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>?V</td>
<td>?3 COULD BE 0 OR 1 OR 2</td>
<td></td>
</tr>
</tbody>
</table>

## Deal

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ?S</td>
<td>2 ?3 COULD BE ANY OF 0 1 0 2 1 2 2 0 2 1</td>
<td></td>
</tr>
</tbody>
</table>
ASSIGNMENT

Assignments may occur within expressions, in which case the value of the expression is the expression to the right of the assignment operator (\(\gets\)). For example, after \(A \gets (B+1)+2\), \(A\) is 3 and \(B\) is 1.

Indexing may also occur on the left of an assignment and follows rules similar to the usual indexing operator. Assignment is made to all elements designated by the "Cartesian product" of the subscripts. However the shapes of the value to be assigned must match the concatenation of the shapes of the subscripts. The shape of the variable assigned to is unchanged.

EXAMPLES (using 0 origin):

\(A \gets (3,4)\) 2+1

\[
\begin{array}{c}
A \gets (3,4)\ 2+1
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11
\end{array}
\]

After

\(A[;1] \gets 13\)

\[
\begin{array}{cccc}
A[;1] \gets 13
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 2 & 3 \\
4 & 1 & 6 & 7 \\
8 & 2 & 10 & 11
\end{array}
\]

\(A[1;(2,2)\ 4] \gets (2,2)\ 4\)

\[
\begin{array}{cccc}
A[1;(2,2)\ 4] \gets (2,2)\ 4
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
8 & 9 & 10 & 11
\end{array}
\]

\(A[0,1;2,3] \gets (2,2)\ 4\)

\[
\begin{array}{cccc}
A[0,1;2,3] \gets (2,2)\ 4
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
4 & 5 & 2 & 3 \\
8 & 9 & 10 & 11
\end{array}
\]

APL variables have no restrictions on the kinds of values which may be assigned, unlike most languages where variables are rigidly controlled by declarations of type and shape. This property will be referred to as "undeclared attributes."
EXAMPLE: Left Justification of a Vector

Assume that $M$ is a nonempty vector. The following program will left justify $M$, i.e. rotate $M$ such that the leftmost nonzero element, if any, is the first element of the result, $N$.

$$N<-(M≠0)\,1\,\phi\,M$$

Let $M<=>0\,0\,9\,0\,4$. The operation $M≠0$ creates a boolean vector in which 1's mark the nonzero positions of $M$.

$$M≠0<=>0\,0\,1\,0\,1$$

$(M≠0)\,1$ locates the 1 in $M≠0$ with lowest index, i.e. marking the first nonzero element of $M$.

$$(M≠0)\,1<=>2$$

The position of that nonzero element is the desired amount of rotation to the left to place that nonzero element first in the result.

$$2\phi\,M<=>9\,0\,4\,0\,0$$

What if $M<=>0\,0\,0$? Then

$$(M≠0)<=>0\,0\,0\quad (M≠0)\,1<=>3\quad 3\phi\,M<=>M$$
CHAPTER 2
VERIFICATION METHODOLOGY

This chapter considers various aspects of the basis for verifying APL programs. First, the definition of correctness that is used throughout the thesis is given. The review discusses several of the papers most relevant to issues arising from APL program verification. The requirements for verifying programs in a particular language using the inductive assertion method are enumerated. Conventions and formalism are then adopted for these requirements with emphasis on the form of verification conditions and how these establish the correctness of a program. The issues involved are illustrated in an example of informal verification of two versions of a simple APL program. Finally, several pragmatic questions about verification are raised and discussed.

DEFINITION: PROGRAM CORRECTNESS

A program will be considered CORRECT with respect to some restriction on range of input values (called an INPUT PREDICATE) and some prescribed condition of output variables (called an OUTPUT PREDICATE) if any computation of the program on data satisfying the input predicate terminates successfully by making a normal exit with program variables which satisfy the output predicate. A program may be incorrect if for some satisfactory input data it loops infinitely, halts with an error while executing any operation, or produces a "wrong" result. Program verification is the proof, formal or
informal, that the program is indeed correct according to this definition.

REVIEW OF THE LITERATURE

The literature on program verification is extensive, ranging from theoretical properties of program schemata to implemented program verifiers [LO2]. An excellent survey and assessment is Elpas, et al.[EL2] and the accompanying report on the state of the art by London[LO5]. The review here discusses some of the papers relevant to verification of APL programs.

The Inductive Assertion Method

Floyd[FL1, FL2] formalised some ideas that had appeared originally in Gorn[GN], Von Neumann[GVN], and Turing[TU] where the suggestions are made that program correctness can be shown by attaching assertions to certain key points in the program. In Floyd's description, the program (in flow chart form) is augmented by assertions inserted before and after every program statement. The proofs that each assertion follows from the immediately preceding assertion and the semantics of the statement in between constitute a proof of partial correctness of the entire program in the sense that any path of control to the exit guarantees that the final predicate is true since all intermediate assertions were true. However, usually this method requires a separate proof of termination.

Actually only certain key locations of a program need have associated assertions, namely the entrance and exit and at least one
within each loop. The person making the assertions has the choice of location and density of assertions. Each assertion usually gives a description of the state of certain variables at the key locations, or "snap shots" in Naur's[NA] terminology. Assertions on loops are called INDUCTIVE ASSERTIONS since they play the role of the induction statement of an induction proof and commonly describe some invariant property of the loop. The formal statement that "from a given assertion and the execution of statements up to the next assertion, the latter assertion is true" is called a VERIFICATION CONDITION.

EXAMPLE: (in flow chart schema form)

```
                   P1
                   ↓
                X=F(X)
                   ↓
               P2
                   ↓
            Q(X)  T  EXIT
                   ↓
                F
                   ↓
           X=I(X)
```

*P1* states the input relation and *P3* the output relation. The inductive assertion is *P2*. There are three verification conditions:

vc1: "from *P1* and the execution of *X=F(Y)*, prove *P2* ".
vc2: " from *P2* and the truth of *Q(X)*, prove *P3* ".
vc3: " if *P2* is true and *Q(X)* is false and *X=I(X)* is executed, then *P2* will be true".

The basic procedure for forming a verification condition involves substitution into the assertions of expressions for variables, from assignment statements (see [K11],[GO],[EL3]). The class of APL programs that will be of most interest are those where the operators replace loops with a resulting reduction in frequency of assertions and verification conditions. However, the remaining verification
conditions may entail more complicated proof techniques.

A large number of programs have been verified using the inductive assertion method (London[L05]). Experience has shown the need for mechanical assistance both in the bookkeeping task of constructing verification conditions and in the more difficult task of proving the verification conditions. The task of constructing verification conditions looks easy since it only involves bringing together assertions and a series of program statements into a single formula, but in reality it involves extraction of the semantics of the program's statements, which can be nontrivial depending on the language. Statement types such as assignments and simple transfer are easily handled but block structure and procedure calls introduce complexities. (A point which Floyd emphasizes is that the rules for constructing verification conditions are, in effect, a definition of the programming language).

The second task of proving verification conditions has been attacked through a specialized theorem prover (King[KI1,KF]), and schemes for proof checking, especially interactively, have been proposed ([GO,EL3]). Chapter 5 of this thesis will describe and illustrate an informal deductive system for proving APL verification conditions.

Other Methods

Several variations on this basic formalism have been developed. It is sometimes confusing to compare these formalisms without knowing all the basic assumptions. These assumptions may cover the basic
operations (partial or total, and if partial, whether considered), the
way in which the basic operations are defined (in verification
conditions or in the deductive system), whether function calls are
permitted in expressions (which might result in nontermination),
whether expressions are decomposed, the intended verification style
(informal, partially formal, completely mechanized), the domains of
variables (integer, declared, or restricted by input predicate), and
the statement forms.

Hoare\[HO1\] presents axioms and rules of inference for proving
assertions about programs given directly as text (as opposed to
Floyd's flow charts). Hoare argues that the axiomatic approach allows
a language to be described in a succinct form leaving certain aspects
such as precision of elementary arithmetic operations undefined. The
definition of APL which will appear in Chapter 3 and the deductive
system based on the definition will illustrate this axiomatic
approach. Also in Hoare [HO2,HO3] axioms and rules of inference are
given for programs with function calls and jumps and labels.

London\[LO1,LO3\] advocates the development of informal techniques
such as case analysis, flow of control, program sectioning, and
decision logic, as well as the inductive assertion method. Many
examples show that a satisfactory proof can be given rigorously yet
less formally than through the inductive assertion method.

Manna[MA] and Cooper[C02] have shown how the proof of total
correctness (satisfaction of input-output relation and termination)
can be formulated in terms of first and second order predicate
calculus where the existence of any set of satisfactory assertions is to be proved.

Another property of programs which is of interest is equivalence. Consider the situation where a program has been proven correct according to some accepted method and another program exists for the same problem in the same or a different language. If the second can be proved equivalent (in an appropriate sense), then its correctness is also established. Manna, et al. [MNV] survey a variety of induction techniques for proving such properties of programs as correctness, termination, and equivalence, based on the fixed-point approach of Scott [SC]. These methods include structural induction (Burstable), recursion induction (McCarthy), truncation induction (Morris), and parallel induction. An example will be given in Chapter 5 where a form of parallel induction seems better than inductive assertions for a particular APL program.

Allen [AL1] presents a formulation of correctness methodology following Manna [MA] in which the possibility of partial operators is considered by developing correctness and convergence predicates for all points of a program. Examples are given to illustrate that the inductive assertions or predicates (ala Floyd) can be created in the process of showing the existence of the correctness and convergence predicates.

State of the Art of Program Verification

London [LO5] enumerates several representative examples of published proofs of programs, including simple non-numerical
VERIFICATION METHODOLOGY
REVIEW OF THE LITERATURE

algorithms, numerical calculations, and operating system components. The size and range of examples leave little doubt that programs can be proved correct, but the task of proving even a small program is still formidable.

Several systems have been developed which provide some assistance in verification by automatically analyzing the program to produce verification conditions, and, in some cases, proving or checking the proofs of verification conditions. The most successful and best described of these systems is King's [K11] which is restricted to integer arithmetic problems for which a partial decision procedure and powerful simplification rules provide a proof (or refutation). There are currently several verification systems under development, but little information is available (see London [L05]).

DESCRIPTION OF THE INDUCTIVE ASSERTION METHOD

It will be possible to verify a program $P$ in a given programming language $L$ once the following have been specified for $L$:

1) A standard format for programs, e.g., flow charts (represented either graphically or linearized) or text.

2) A way of designating certain places of the program in the given format and associating predicates with those places.

3) A language with well-defined syntax and semantics for expressing the predicates.

4) A definition of the programming language.

5) Rules for forming verification conditions using the definitions of the programming and assertion languages.

6) A deductive system for proving that verification conditions are true.
Note that 3)-6) require that there be some way of relating assertion and programming languages such that a suitable proof system can be developed.

Then to verify a specific program P according to the definition of correctness requires these steps:

a) Choose a subset of the places of P which includes the entrance, all exits, and at least one place in every loop. Associate a predicate with each place. The predicates associated with the entrance and exit are the input and output predicates, respectively.

b) Construct and then prove or disprove, in the deductive system, the verification conditions. If all verification conditions have been proved, it is then possible to state that for any computation of the program where the input values satisfy the input predicate and the exit is reached, the output predicate holds. If the verification conditions cannot all be proved, then there are three possibilities: the input-output relation does not hold, the inductive assertions are inconsistent with the program, or the deductive system is not complete.

c) Prove that all computations for the input values which satisfy the input predicates do terminate at an exit, i.e. that there are no infinite loops and that all operations are executable.

Which steps of this process can be mechanized?

a) Using standard graph algorithms, it is possible to find all
candidate sets of places, but the final choice depends on the requirement for associating predicates with places. Only the programmer can specify the desired input-output relation. Since it will be necessary to prove verification conditions incorporating the inductive assertions and the inductive assertions must support the input-output relation, this step almost surely must be completed by the programmer, although it might be assisted by mechanical techniques such as those to be presented in Chapter 4.

b) The construction of verification conditions is usually a very intricate process since it involves both the finding of all sequential pairs of assertions and the extraction of the semantics of the program according to the language definition. This step can be completely mechanized assuming that the language definition can be adequately formalized and represented ([GO], [K11], [EL3]). The success of automating the proof of verification conditions depends on the choice of deductive system and problem domain. King[K11] presented a system for proving verification conditions of a subset of ALGOL restricted to integer arithmetic operations. The system was successful on a number of examples but suffered from combinatorial explosion in dealing with arrays. The more general requirement is for a theorem-prover with capabilities on the order of predicate calculus. Another alternative is the proof-checking style where the user gives a series of key steps of the proof and the checker performs the necessary symbol manipulation and rules of inference
to justify or disprove each step of the proof. Chapter 5 will discuss some steps toward automating APL verification.

c) There exist several techniques for proving termination. This may not be too difficult since termination may be independent of what the program computes. Showing that every program operation can be performed is very tedious and can be mechanized to some extent: Assume that all predicates must not only be assertions about what the program computes but also that the sequence of statements up to the next assertion can be executed. Then this executability step can be distributed among the various verification conditions by including the stipulation that a verification condition is true only if the statements can be executed properly when the first assertion is satisfied. The proof of individual verification conditions then is extended in the same way as the proof of the input-output relation.
The inductive assertion method will be applicable to APL once the six requirements stated in the previous section have been met:

1) The format for APL functions is a series of numbered lines preceded by a header. For now, we will make the restriction that only single functions will be considered since function calls present other problems (side effects), which will be discussed later. Thus the header can be discarded and all parameters, local variables, and global variables will be treated alike.

2) Predicates can be associated with line numbers using the same syntax as APL comments, a lamp symbol (揶) followed by a string of characters. In the verification system, however, the character string following the lamp will be interpreted as an assertion. The last line of the function will be reserved for the output predicate so that branches to 0 (function exits) will be interpreted as branches to that last line followed by function exit when the program is being considered for verification. If the last line is not an assertion, TRUE will be assumed as the output predicate. Since it is possible to branch to a line with a comment, assertions are introduced directly into the flow of control through the use of the comment facility.

Example 1:

[1] \( \alpha (\sqrt{M \neq 0}) \wedge (1 \neq \rho P M) \)
[2] \( N \in \{(M \neq 0) \cap 1\} \neq M \)
[3] \( \alpha N(0) \neq 0 \)
Example 2:

\[ I \leftarrow (v/M \neq 0) \land (I = \text{pp}M) \]
\[ I \leftarrow 0 \]
\[ I \leftarrow I + 1 \]
\[ I \leftarrow I + 1 \]
\[ I \leftarrow I + 1 \]
\[ I \leftarrow I + 1 \]
\[ I \leftarrow I + 1 \]
\[ I \leftarrow I + 1 \]
\[ I \leftarrow I + 1 \]

3) The assertion language will basically be APL (extended by a few operators to be discussed in Chapter 5).

4) The language definition will be given in two parts: statement forms in this section and the operators in Chapter 3. We shall make several simplifications in statement forms throughout.

1. Functions and function calls will generally be ignored.

2. The statement forms will be restricted to
   i) Unconditional branch \( \rightarrow L \)
   ii) Conditional branch \( \rightarrow I, IF EI \)
   iii) Assignment expression \( V \leftarrow EI \)
   iv) Assertion (or comment) \( A E \)

where

\( L \) is a statement number
\( EI \) is an APL expression with at most one operator
\( V \) is a variable
\( E \) is a conjunction of APL expressions

At first glance these restrictions appear to be quite severe. However, any program with labels can easily be translated to one with the appropriate statement numbers substituted for labels. Also, any expression can be translated into a series of assignment statements where the expressions have at most one operator. All that is required is

1) A simple recursive algorithm which operates on the parse tree of an expression to produce an assignment statement for each subexpression node in the order dictated by the precedence rule.
2) A way of generating names for the variables assigned to such that there is no conflict with regular program variables.

For example, \( N \leftarrow ((M \neq 0) \land l) \phi M \) has parse tree

```
          N
         / \  /  \\
        /   /  \\
       M   l   1
         \   \  /  \\
          \ / \  \\
         M \ # \ 0
```

and can be translated into (where the reserved variable names all start with letter \( R \) followed by \( \$ \) and a number)

\[
R$.0 \leftarrow M \neq 0 \\
R$.1 \leftarrow R$.0 \land l \\
N \leftarrow R$.1 \phi M
\]

This translation process will be called EXPRESSION DECOMPOSITION.

The reason for making this restriction at this point is that the verification of APL programs is dependent on the semantics of the operators and it will be desirable to formulate the verification conditions in terms of the semantics of individual operators rather than expressions. APL also permits assignments embedded within expressions, requiring decomposition in order to form verification conditions. It should be stressed that programs without these two restrictions can be entered into the APL verification system (soon to be presented) with the translation as part of a preprocessing stage and all further verification taking place with respect to the translated version of the program.

One further restriction is not so easily dealt with. The APL
branch has the form $\rightarrow E X$ where $E X$ is any expression which evaluates to a vector or a single-element value. This makes it very difficult to analyze program control flow to form verification conditions. We will require that in the conditional branch $E$ will evaluate to 1 (true) or 0 (false). $I E$ can be regarded as a dyadic function which returns $L$ if $E$ evaluates to 1, otherwise 0, causing either a transfer to $L$ or continuation to the next line, respectively. It would frequently be possible to automatically translate the branch expressions that commonly occur in APL programs into this form although in some cases it would be impossible without some information about the range of possible values of $E$. This restriction is adopted both to make programs more readable (the conditional branch is much more transparent in this standard form) and to avoid the difficulty just discussed.

Function calls will be discussed later.

5) It remains to show how to construct verification conditions for APL. This will be presented in several steps:

5a) The general forms of verification conditions
5b) The notation for control segments, the basic units for forming verification conditions
5c) The formation of the simplest kind of verification conditions (to illustrate the issues)
5d) The general rules for forming verification conditions for arbitrary control segments
5e) The proof that the rules accomplish what is intended for verification

5a) The general forms of verification conditions

The general form of a verification condition for a single statement $T$ joining two assertions $P$ and $Q$ will be
\[(P \supset C) \land (P \land S) \supset Q\]

where

\(C\) is an APL expression for the constraints, the restrictions on
operands for operators to be executable, of the statement \(T\).
\(S\) is a conjunction of identities derived from the statement \(T\).
\(\supset\) has a slightly different meaning from standard logic because we
are dealing with a three-valued logic. \(\neg \supset \neg B \leftrightarrow \text{IF } A \text{ IS TRUE THEN } B \text{ ELSE TRUE}\). See appendix E for a more complete explanation. \(\leftrightarrow\) denotes
that the expression on the left is notation for the expression on the
right, even though the defining expression may not yet be completely
formalized.

It is claimed that this formula expresses the intent that "For
input values that satisfy \(P\), the statement \(T\) can be executed and then
\(Q\) will be true for the result of that execution."

The definition of program correctness given at the start of the
Chapter calls for the proof of both the input-output relation and of
the executability of the program. It has been shown elsewhere (e.g.
Floyd[FL1]) how the inductive assertion method aids in proving the
input-output relation. The same basic principle works in proving
executability of the program: the inductive assertions must provide
that all statements up to the next assertion can be executed. Then if
all verification conditions are proved, it will have been shown that
all statements along any control path can be executed.

5b) Control segments

In general, verification conditions are constructed for a program
by breaking the control paths of the program into segments bounded by
assertions and with no inner assertions. In example 2 the segments
are 1-2-3, 3-4-8-9, 3-4-5-8-9, and 3-4-5-6-7-3.
Notation: A LINE of an APL program, L, will be written as a pair (Int,St) where Int is the integer line number and St is the statement on that line. N(L) will produce Int and S(L) will produce St.

Definition: A CONTROL SEGMENT is a sequence of lines L1,...,Ln* such that
   ia) S(L1) and S(Ln) are assertions
   ib) For no i, 1<i<n, is S(Li) an assertion
   iia) if S(Li) is an assignment statement then N(L(i+1)) is N(Li)+1
   iib) if S(Li) is an unconditional branch \rightarrow L then N(L(i+1)) is L.
   iic) if S(Li) is a conditional branch \rightarrow I IF K then N(L(i+1)) is N(Li)+1 or N(L(i+1)) is L.

The statement S(L1) will be called the PREASSERTION and the statement S(Ln) will be called the POSTASSERTION.

The control segments can be obtained by standard graph traversal algorithms. The ultimate goal is to prove each verification condition as a theorem in the deductive system.

We will first consider in 5c) the formation of verification conditions for control segments with a single statement simply to clarify the issues. The complete rules for forming the verification conditions for any control segment will then be given in 5d).

5c) Formation of simple verification conditions

Consider a setting for the simplest possible verification condition: a control segment with a preassertion line (N1,S1), postassertion line (N3,S3), and statement (N2,S2). The forms for verification conditions will be given in terms of P,M,Q, which are expressions to be developed from S1,S2,S3, respectively. C will represent the constraints for execution of S2.

*Subscripts are linearized in the text. L is written L1 and L is written L(i+1).
form of S2  \[ \rightarrow \text{L} \]
\[ \rightarrow \text{I, IF E} \]
\[ V \leftarrow E \]

form for verification condition
\[ P \supseteq Q \]
\[ (P \supseteq C) \land ((P \land M) \supseteq Q) \]

\[ (P \supseteq C) \land ((P \land M) \supseteq S(V; V'; Q)) \]

where \( S(V; V'; Q) \) means "substitute \( V' \) uniformly for \( V \) in \( Q \)." The conjuncts \( P \supseteq C \) insure that the statement \( S2 \) can be executed if \( P \) is true. Now, what is \( M \)? For the conditional branch \( M \) must contain the semantics of the branch condition \( E \) and also the direction of the branch. We will write \( M \) in this case as \( B \cdot E \) where \( B \) is 1 if \( N3 \) is \( L \) and \( B \) is 0 if \( N3 \) is \( N2 + 1 \). The assignment statement introduces the complication of change of variable name since \( Q \) refers to the new value of \( V \) while \( E \) and \( P \) refer to the old value. So \( M \) will have the form \( V' \leftarrow E \) establishing the identity of values of \( V' \) and \( E \), where \( V' \) will be a new name generated from \( V \). \( \leftarrow \) will denote the identity relation of the deductive system.

We will assume the existence of an underlying mechanism for generating unique names of variables. For example, for every program variable there might be an alteration counter which is initialized at entry to 0 and incremented by 1 at every assignment to the variable. The unique name can be formed from this counter and the variable name. (For a more formal development of a model for this procedure, see Good[GO]).

\( P \) and \( Q \) are to be developed from the preassertion and postassertion, respectively. The assertion language being APL, there is the possibility that the operators of the assertion are not total. To assert that a particular APL expression \( E \) is true is to require
that \( E \) evaluate to 1 (true). In the case of both the preassertion and the postassertion the operator constraints are implicitly assumed to hold. Therefore \( P \) and \( Q \) will have the form \( \land(EI \land CI) \) where \( \land EI \) is the form of the assertion and where \( CI \) includes the constraints on the operators of \( EI \). The constraints will be formally presented in Chapter 3. An example will show the kinds of difficulties that arise.

**EXAMPLE:**

\[
\begin{align*}
&\{1\} A1 \in A \\
&\{2\} I \leftarrow A1 \\
&\{3\} A1 \in pA
\end{align*}
\]

The verification condition is

\[
(((1 \in A) \land \text{CONSTRANTS ON } 1 \in A) \Rightarrow \text{CONSTRANTS ON } A1) \land \nonumber \\
(((1 \in A) \land \text{CONSTRANTS ON } 1 \in A) \land (l' \leftarrow A1)) \Rightarrow \nonumber \\
(\text{CONSTRANTS ON } I' \in pA) \land (l' \in pA)
\]

\text{CONSTRANTS ON } 1 \in A \Rightarrow \text{TRUE} \text{ (there are none)}

\text{CONSTRANTS ON } A1 \Rightarrow \nonumber (l = ppA) \lor (l = pA)

\text{CONSTRANTS ON } I \in pA \Rightarrow \nonumber \nonumber

The verification condition cannot be proved as it stands since it is not the case that \( (1 \in A) \Rightarrow (l = ppA) \lor (l = pA) \). With the additional conjunct \( l = ppA \) in the preassertion, the resulting verification condition would be provable. The definition of correctness requires that the input predicate be satisfied only for input for which the program and assertion operators are meaningful. Thus using the expressions \( A1 \) and \( \in pA \) implicitly restricts the range of input values for the program. It will be shown in Chapter 4 that it is not obligatory to make assertions about the executability of the operators. Instead the constraints which are known from the definition can be reduced mechanically and the input condition can be strengthened, if necessary.

5d) General rules for construction of verification conditions

The fundamental form of the verification conditions considered the case of a single statement control segment with simple

\( \land EI \) is used here to denote the finite conjunction \( EI \land \ldots \land EN \), where the subscript is suppressed.
expressions. As mentioned earlier, assertions will be written as a conjunction and expressions in assignment statements will contain at most one operator. Consider the control segment L1,...,Ln. The complete form of a verification condition for the control segment will be expressed in terms of intermediate formulae:

\[ C_j \text{ represents a condition for executing the operations through line } j \text{ of the control segment} \]

\[ A_j \text{ represents the accumulated assertions and statements through line } j. \]

Each successive statement of a control segment will be processed producing a sequence of these formulae, \( C_j \) expressing the condition for executability and \( A_j \) the effect of the execution of the statements.

**STEP 1**: L1 (preassertion form \( \wedge E_l \))

\[ C_1 \leftrightarrow \text{TRUE} \]

\[ A_1 \leftrightarrow \wedge (C_1 \wedge E_l) \text{ where } C_1 \text{ is a conjunction of the constraints of } E_l. \]

**STEPS** for program statements Lj where j=2,...,j=n-1:

(a) Unconditional branch \( \rightarrow L \) (Lj plays a role only in forming the control segment)

\[ C_j \leftrightarrow C(j-1) \]

\[ A_j \leftrightarrow A(j-1) \]

(b) Conditional branch \( \rightarrow L \text{ IF } B \) (Lj can affect both the executability and consistency of the segment)

\[ C_j \leftrightarrow C(j-1) \wedge (A(j-1) \supset CONSTRANTS ON E) \]

\[ A_j \leftrightarrow A(j-1) \wedge (B=E) \]

where B is 1 if N(Lj)= L, B is 0 if N(Lj)=N(L(j-1))+1

(c) Assignment \( V \leftarrow E \)

\[ C_j \leftrightarrow C(j-1) \wedge (A(j-1) \supset CONSTRANTS ON E) \]

\[ A_j \leftrightarrow A(j-1) \wedge (V \leftarrow E) \]

where \( V' \) is a unique name generated from V and from here until the next assignment to V, \( V' \) will be the current name for the variable V.

**STEP n**: (postassertion \( \wedge E_l \))

\[ C_n \leftrightarrow C(n-1) \wedge (A(n-1) \supset (\wedge CONSTRANTS ON E)) \]

\[ A_n \leftrightarrow A(n-1) \supset \wedge E_l \]

The final form for the verification condition is \( C_n \wedge A_n. \)
EXAMPLE: In example 2 in the control segment with lines 3-4-5-8-9, the verification condition would be

\[ C1 \iff \text{TRUE} \]

\[ A1 \iff ((\text{CONSTRAINTS ON } \vee M \neq 0) \land (\neg \exists M \neq 0)) \land \\
((\text{CONSTRAINTS ON } 1_{pp} M) \land (1_{pp} M)) \land \\
((\text{CONSTRAINTS ON } \neg 0 = 1 \land M) \land (\neg 0 = 1 \land M)) \land \\
((\text{CONSTRAINTS ON } 1 \langle 1 + p M \rangle) \land (1 \langle 1 + p M \rangle)) \]

\[ C2 \iff \neg C1 \land (A1 \iff \text{CONSTRAINTS ON } D \geq p M) \]

\[ A2 \iff \neg A1 \land (\neg (D \geq p M)) \]

\[ C3 \iff \neg C2 \land (A2 \iff \text{CONSTRAINTS ON } M[I] \neq 0) \]

\[ A3 \iff \neg A2 \land (\neg M[I] \neq 0) \]

\[ C4 \iff \neg C3 \land (A3 \iff \text{CONSTRAINTS ON } I \neq M) \]

\[ A4 \iff \neg A3 \land (\neg I \neq M) \]

\[ C5 \iff \neg C4 \land (A4 \iff \text{CONSTRAINTS ON } N[0] \neq 0) \]

\[ A5 \iff \neg A4 \land N[0] \neq 0 \]

The verification condition is \( C5 \land A5 \).

It is assumed that the deductive system will have axioms for the operators which occur in expressions, though there is no reason why those operators could not be translated or defined in terms of different operators. Indeed, the formal definition that will be presented in Chapter 3 will define some APL operators in terms of primitive operators and such a translation could take place.

5e) Meaning of verification conditions

It is now necessary to show that any formula VC constructed from the above rules expresses what is intended for verification. A control segment was formed by simply grouping together a sequence of assertions and statements which MIGHT be executed for some particular values. Verification conditions are formed for both branches of a conditional branch but one branch may be blocked by a previous branch, constraint, or assertion (this is especially likely in verification by cases).
Definition: A control segment L₁,...,Ln is INCONSISTENT for some input values that satisfy the preassertion if there is branch statement S(Lk): \( \rightarrow I \) IF \( E \) where the sequence of statements up to and including the branch is executable but \( E \) evaluates to 1 and \( N(L(k+1)) \) is \( N(Lk)+1 \) or \( E \) evaluates to 0 and \( N(L(k+1)) \) is \( L \).

Definition: A control segment is VERIFIED if for input values that satisfy the preassertion either
i) the control segment is inconsistent or
ii) the sequence of statements is executable and the values after execution satisfy the postassertion

Let VC be the verification condition formed from the control segment L₁,...,Ln according to the rules just presented. VC is to be proved eventually as a theorem in the deductive system and we want to establish the soundness of the system in the following sense:

1) if VC is true then the control segment is verified.

2) if the control segment is verified then VC is true.

Lemma: For given input values that satisfy the preassertion the sequence of lines L₂,...,Lk is executable and consistent (abbreviated e.c.) iff \( CK \land AK \).

Proof: Assume that the preassertion evaluates to 1. Then \( / \) is true since all constraints on operators are satisfied and every conjunct evaluates to 1.

Assume the lemma holds for the sequence L₂,...,Lk. Consider the statement \( S(L(k+1)) \):

a) Unconditional branch \( \rightarrow I \)
   (the statement can affect neither executability nor consistency)
   \( C(K+1) \iff CK \)
   \( A(K+1) \iff AK \)

   \( L₁,..,L(k+1) \) e.c. iff \( L₁,...,Lk \) e.c. iff \( CK \land AK \) iff \( C(K+1) \land A(K+1) \)

b) Conditional branch \( \rightarrow I \) IF \( E \)
   (the branch can affect both the executability and the consistency of the sequence)
   \( C(K+1) \iff CK \land (AK \Rightarrow CONSTRAINTS \ ON \ E) \)
   \( A(K+1) \iff AK \land (B \iff E) \)

   \( L₁,..,L(k+1) \) e.c.
   iff \( L₁,...,Lk \) e.c. \( \wedge (S(L(k+1)) \) executable \( ) \wedge (B \iff E) \)
   iff \( CK \land AK \wedge (\text{constraints on } E \text{ satisfied}) \land (B \iff E) \)
if \( C\!K \land (AK \supset \text{constraints on } E \text{ satisfied}) \land (AK \land (B \leftrightarrow E)) \)
if \( C(K+1) \land A(K+1) \)

c) Assignment \( V \leftrightarrow E \)
(a)ssignment can affect only the executability of the sequence)
\( C(K+1) \leftrightarrow CK \land (AK \supset \text{CONSTRAINTS \ ON \ E}) \)
\( A(K+1) \leftrightarrow AK \land (V' \leftrightarrow E) \)
\( L_1, \ldots, L(k+1) \) e.c. iff
\( L_1, \ldots, L_k \) e.c. \( \land \) (contraints on \( E \) satisfied) \( \land (V' \leftrightarrow E) \)
if \( CK \land AK \land (AK \supset \text{constraints on } E) \land (V' \leftrightarrow E) \)
if \( C(K+1) \land A(K+1) \)

1) if \( VC \) is true then the control segment is verified

If \( VC \) is true then \( AI \) is true and the preassertion is satisfied. The form of \( VC \)
is
\( CN \land AN \leftrightarrow CN \land (A(N-1) \supset EI) \)

Case 1: \( A(N-1) \) false
Since \( A(N-1) \) is a conjunction there must be a \( T \) such that \( AT \) is false but \( A(T-1) \) is true. \( AT \) can become false only if \( B \leftrightarrow E \) for some conditional branch \( \rightarrow L \) \( IF \ E \). By the lemma, \( L_2, \ldots, L(T-1) \) is executable and consistent, therefore the control segment is inconsistent but verified.

Case 2: \( A(N-1) \) true
By the lemma \( L_2, \ldots, L(n-1) \) is executable and consistent.
\( A(N-1) \supset \langle \text{CONSTRAINTS \ OF \ EI} \rangle \) since \( CN \) is true. \( A(N-1) \supset EI \).
Therefore the postassertion is satisfied and the control segment is verified.

II) If the control segment is verified, then \( VC \) is true.

Assume the preassertion is true. Then \( AI \) is true. Either
Case 1: The control segment is inconsistent.
Then there is some statement \( S(Lk) \) which is a conditional branch \( \rightarrow L \) \( IF \ E \) and the expression outcome is inconsistent with the \( B \) in \( AK \). Then \( AK \) is false and for all \( k < n \) \( AJ \) is false and \( CJ \) is therefore true. But \( CN \) and \( AN \) are both true therefore \( VC \) is true.

Case 2: The control segment is executable and consistent.
Then \( A(N-1) \land C(N-1) \) by the lemma and since the postassertion is satisfied \( \land \langle \text{CONSTRAINTS \ ON \ EI} \rangle \) hold and therefore \( CN \) is true. Since the control segment is verified \( \land EI \) holds and therefore \( AN \) is true. Therefore \( VC \) is true.
Discussion of the Formalism

There are several major differences in the formalism given here from that of Floyd[FL1], King[K11,K12], and Good[GO].

1) The expressions are decomposed for two reasons: first, embedded assignments enforce the requirement for change of variables within expressions and second, it is convenient and natural to associate semantics at the operator level.

2) The verification conditions are given in conjunctive form. There are two ways to accumulate the effect of statements in forming the verification conditions: substitution and conjunction. Conjunction is more general since it is commutative while substitution may be performed forward (preassertion to postassertion) or backward (postassertion to preassertion). (See Good[GO] for a discussion of the two alternatives.) Conjunction seems more appropriate for APL since the substitution of a lengthy one-liner simply doesn't advance the verification process. The semantics of the operators are sufficiently complex that significantly different proof techniques will be required (see Chapter 5).

3) The fact that operations are partial is dealt with directly in the verification conditions by introducing the constraints for every operator. The sequential form of the implication operator \( \Rightarrow \) makes it possible to avoid the explicit development of a partial function logic.

4) Throughout the proof, it was assumed that the identities
resulting from assignment correspond to the values occurring during execution of the control segment. In other words, the mechanism of state vectors has been suppressed.

5) Function calls can be introduced into the formalism with a few additional considerations. Take the case of a dyadic function defined by the header \( \forall z \leftarrow \text{if} \quad \text{with}\quad PF \quad \text{and} \quad QF \quad \text{as preassertion and posassertion, respectively, and assume that} \ F \quad \text{has been verified. Assume for the moment that} \ F \quad \text{has no side effects. If statement}\quad S(L_j)\quad \text{were} \quad R3 \leftarrow R2 \ F \quad R1, \quad \text{where} \quad R3,R2,R1 \quad \text{are unique names, then}

\[
\begin{align*}
C_J & \iff C(J-1) \land (A(J-1) \lor PF') \\
A_J & \iff A(J-1) \land QF' \\
\text{where} \quad PF' & \iff S(R1; B; S(R2; A; PF)) \\
\text{and}\quad QF' & \iff S(R3; z; S(R2; A; S(R1; B; QF)))
\end{align*}
\]

The soundness of this rests on the fact that \( C_J \) guarantees that the entrance condition for \( F \) is satisfied and therefore the call of \( F \) will terminate normally and \( QF' \) will then hold for the result of executing \( F \).

Side effects introduce a complexity of handling names. Assume \( C \) is a nonlocal variable influenced by executing \( F \). For the verification condition to be sound, \( QF \) must state that side effect on \( C \). \( C \) then must be treated as a parameter and there must be a naming convention that distinguishes between initial and final values of \( C \). Then when side effects are considered, \( PF' \) would include substitutions of the current name for \( C \) in the verification conditions for the occurrences of initial names for \( C \) in \( PF' \). Similar substitutions for the initial value of \( C \) would occur in \( QF' \) along with an updating of the alteration counter and the substitution of the generated unique
name for all noninitial names for C in QF'' . The updated unique name would then be used as the current name for C in the process of continuing to form the verification conditions. For example, PF might be \( P1(A;B) \wedge P2(C) \) and QF be \( (P3(A;B) \supset (C' \rightarrow C)) \wedge (P4 \supset P5(C')) \) where C and C' designate initial and final values of C. Then if C.13 were the current name for C at the point of encountering the call on F, PF would be \( P1(R2;R1) \wedge P2(C.13) \) and QF would be \( (P3(R2;R1) \supset (C.14 \leftrightarrow C.13)) \wedge (P4 \supset P5(C.14)) \). C.14 would become the current name for C. There are other ways of implementing this process. Since the mechanism for generating verification conditions has not been thoroughly treated in this thesis, a clearer presentation may be Hoare[H02] or Elspas, et al.[EL3].

There are two important considerations with respect to side effects in APL. First, a syntactic mechanism for declaring side affected variables would be required as well as a convention for referring to initial and changed occurrences of the variables within the output predicate. Second, the soundness proof for verification conditions would be dependent on the fact that all side effects had been properly declared. Otherwise, it would be possible to obtain a proof for a verification condition corresponding to a control segment which could not be verified.
EXAMPLE OF INFORMAL VERIFICATION OF AN APL PROGRAM

Consider again the following problem:

Given a numerical vector \( M \) produce a vector \( N \) which is a left justified cyclic rotation of \( M \), i.e. the leftmost nonzero element of \( M \) if any, is the first element of \( N \) and the cyclic order is preserved. The program was explained in Chapter 1.

\[
N = ((M \neq 0) \land 1) \odot M
\]

We will use this simple program to illustrate several aspects of verification while carrying out an informal proof of correctness. Many of these aspects will be dealt with in more detail in the remainder of the thesis. We will depart from the standard formalism wherever it is convenient yet clear that the form is equivalent with the verification condition formalism.

Since this is an one-line program, only an input and an output predicate are necessary. There are several possible statements to make about this program: that \( N \) is a vector, that the order of elements in \( N \) is a cyclic permutation of the elements of \( M \), and that \( N \) starts with the leftmost nonzero element of \( M \). For now, choose the partial statement of correctness about the leftmost element

\[
N[0] \neq 0
\]

What input predicate is necessary to make this true of the result, \( N \)? The input predicate must sufficiently restrict the input values so that three conditions hold:

1. The program does not halt with an error in executing any operation.
2. The program does not loop infinitely.
3. The output predicate is true for the result of the program.

In order to devise an input predicate which guarantees that no errors will occur during execution of the program, we must know the restrictions on operands for each program operator and whether the operands will satisfy these restrictions. We can examine the successive operations and work backward to systematically develop the partial input predicate. \( M \neq 0 \) can be performed regardless of the shape of \( M \), but \((M \neq 0) \land 1 \) requires that the left operand (the result of \( M \neq 0 \)) be either a vector or a single-element shape. Now, the shape of \( M \neq 0 \) is identical to the shape of \( M \) so \( M \) must be either a vector or a single element shape. The final operation, the rotation, has two forms, one where the left operand has a single element and the other where the left operand has multiple elements and a shape which corresponds to the shape of the right operand. The former case holds since the result of \((M \neq 0) \land 1 \) is a scalar. Another restriction for the rotation is that the left operand be an integer and this is satisfied since the result of the \( \land \) operation is always a nonnegative integer. So the following input predicate will guarantee that the program can be executed:

\[
(1 = \text{pp} M) \lor (1 = \text{p}, M)
\]

Now consider the form of the output predicate which also must be an executable APL expression. \( N[0] \) requires that \( N \) be a nonempty vector. The rotation operation gives \( N \) the same shape as \( M \) so adding
the constraints on $M$ to the input predicate gives
\[(I=0pM)\land(0\neq pM)\]

The "correctness" statement is an incomplete correctness statement which asserts that "if $M$ is not all zeros, then $N[0]\neq 0\]". So this must be incorporated in the input predicate as $\neg\land/M=0$ or as $\lor/M\neq 0$. This restriction subsumes the nonempty restriction $(pM)\neq 0$ since $\land/0\rightarrow 1$. Therefore the input predicate derived from considering the restrictions on operands and the intent of the output predicate is
\[(I=0pM)\land \lor/M\neq 0\]

and we're going to prove that the program is correct with respect to that input predicate and the output predicate $N[0]\neq 0$

We know that the program will terminate without error from the construction of the input predicate and the fact that the program has no loops. The proof that the output predicate will be true requires the specification of some system for giving such a proof as well as suitable semantics for the operators of the program. This will be given more formally in Chapters 3 and 5. A brief argument is as follows:

\[N[0]\rightarrow M[(pM)(M\neq 0)\land 1]\]
semantics of $\phi$
\[\equiv M[(M\neq 0)\land 1]\]
lemma from number theory: $(R\land pM)\rightarrow (R=(pM)\cap R)$
\[(M\neq 0)(M\neq 0)\land 1]\equiv 1 \text{ from a property of the operator }\land$
Therefore $(M[(M\neq 0)\land 1] \neq 0)$ and $N[0] \neq 0$.

We can make the following observations from the example so far:
1. There were several possible ways to state correctness and we chose only one partial statement to prove. The proof that $N$ is a cyclic permutation of $M$ is simply a statement of the semantics of the rotation operator. The case where $M$ is all zeros gives $N$ as identical to $M$ regardless of the cyclic rotation.
2. The input predicate was developed by noting the necessary restrictions on input so that the operations of the program and the output predicate were all executable. This suggests that it might be possible to generate the input condition directly from the semantics of the program operators. Indeed this is possible and is discussed in Chapter 4.
3. It was quite easy to express the input and output predicates in APL. For example, to state the: $M$ is not all zeroes is simply $\lor/M\neq 0$. Quantifiers could also express the same thing: "there exists a subscript $i$ of $M$ such that $M[i]\neq 0". The $\land/$ and $\lor/$ operators are analogs of universal and existential quantifiers.
4. There was no explicit induction in the proof that the output predicate followed from the input predicate and the semantics of the operators. However, the proof required that the semantics of the operators be given as expressions from which it was possible to prove lemmas leading to equivalence with other expressions.

Now, keeping the same input-output relation, consider a similar program with the search $(M\neq 0)\land 1$ strung out into a loop (in flow chart form)
The proof of this program is much more complicated. First we must prove termination since the program might loop infinitely. But it is easily seen that \( I \) increases from 0 to \( pM \) and then the loop is exited either when \( I \) reaches \( pM \) or \( M[I] \neq 0 \). Floyd's method (see [FL]) of assigning monotonically decreasing functions to all loops is a more formal approach (let the function be \((pM)-I\)).

Since the program has a loop we must have an inductive assertion to help prove the input-output relation as well as the restrictions on the operations. There are several choices for location of the inductive assertion and the location will determine the form of the assertion. Let

\[
P \iff (I=ppM) \land V/M \neq 0
\]

Then the locations and respective assertions are

1. \( (I \in 1+pM) \land (\land/0=1M) \land P \)
2. \( (I \in pM) \land (\land/0=1M) \land P \)
3. \( (I \in pM) \land (\land/0=1M) \land P \)
4. \( (I \in pM) \land (\land/0=1M) \land P \)

The first conjunct of each assertion states the range and mode of \( I \) (necessary to prove that \( I \) is a legal subscript for \( M \)). The second conjunct summarizes the effect of the loop so far, that none of the elements in the initial sequence of \( M \) has been nonzero. The initial condition occurs as the third conjunct of the assertions and must be included to prove both that the loop exit will be from the condition \( M[I] \neq 0 \) for some \( I \) and that \( M \) is a vector and therefore the indexing \( M[I] \) is permissible.

Consider just the assertion 1 and the verification conditions for that inductive assertion. Each verification condition must state that the assertion at the start of the segment guarantees that all operations along the segment can be performed and that the first assertion along with the result of executing the segment will imply the second assertion. Each verification condition will be associated with an informal proof.

Let

\[
Q \iff (I \in 1+pM) \land (\land/0=1M) \land P
\]
VERIFICATION METHODOLOGY
EXAMPLE—LEFT JUSTIFICATION OF A VECTOR

vc1: (entrance to loop)
from \((1=p\cdot M) \land (\forall/M \neq 0) \land (I \leftarrow 0)\) prove
\((I \leftarrow 1) \land M \land (\land/0 = I \uparrow M) \land P\)
Substituting 0 for I in the consequent gives \((0 \land I \lhd p M)\) which is true
and \((\land/0 = I \uparrow M)\) which is 1 since 0 \uparrow M is the empty vector. The initial
predicate was preserved since M is unchanged.

vc2: (going around the loop) Let
\(S \land Q \land (I < p M)\)
Prove
\((S \land (I \leftarrow 1) \land M \land (\land/0 = I \uparrow M) \supset (I \leftarrow 1) \land M)\)
The first conjunct of the verification condition states that the
conditions for performing the indexing operation are satisfied by the
initial predicate and the inductive assertion and the branch
condition. The second conjunct states that the assertion will be true
for the new value of I' when control reaches that assertion.

The proof rests on a few operator properties (which will not be proved
here).

a) \((I \leftarrow 1) \land M \supset (0 \land I) \land M\) and therefore \((I \leftarrow 1) \land M \supset I \lhd p M\)

b) \((I \lhd p M) \supset (I = I \uparrow M) \supset (I \leftarrow 1) \land M\)

c) \((\land/0 = I \uparrow M) \land (M) \supset (\land/0 = I + 1) \land M\)

vc3: (top exit from the loop) Let
\(S \land Q \land (I > p M)\)
Prove \((S \land \text{CONDITIONS FOR } I \neq M) \land \)
\((S \land (N \leftarrow I \neq M) \land \text{CONDITIONS FOR } N[0]) \land \)
\((S \land (N \leftarrow I \neq M) \supset N[0] = 0)\)

But actually S is false since
\((I \lhd p M) \supset (I > p M)\) and \((\land/0 = (p M) \uparrow M) \supset (\land/0 = M)\).
\((\land/0 = M)\) and \((\forall/M \neq 0)\) are contradictory.

vc4: (lower exit from the loop) Let
\(S \land Q \land (I < p M)\)
Prove \((S \land \text{CONDITIONS FOR } M[I]) \land \)
\((S \land (N \leftarrow I \neq M) \land \text{CONDITIONS FOR } N[0]) \land \)
\((S \land (N \leftarrow I \neq M) \supset N[0] = 0)\)
The proof for the first conjunct is similar to that in vc2. The
second conjunct is all right since I is a scalar, as discussed for the
first program. Since \((p N) \leftarrow p M\) and \((\forall/M \neq 0) \supset 0 \rightarrow p M\), \(N[0]\) is defined.
Finally,
\(N[0] \rightarrow M[I](p M)[I]\) by the semantics of \&
\(\rightarrow M[I]\) since \(I \neq p M\) using the same lemma as in the one-line version
and \(M[I] \neq 0\) by the branch condition.

Comparing the loop version of the program with the one-liner version,
we observe:

1. There were several choices of location (and therefore form)
for the inductive assertion. Part of the assertion must cover
the explicit indexing operation \(M[I]\) showing that \(I\) is a legal subscript
and that \(I = p M\). The effect of the loop was summarized in the
expression \( \land \exists \theta \equiv \langle 1 \rangle \) which is again an analog of a quantified expression: "for all subscripts \( J \) such that \( J < I \), \( M[J] = 0 \)." The initial predicate had to be carried along as part of the inductive assertion in order to block the inconsistent path in vc3.

2. The process for deriving the initial predicate from the loop version would be much more complicated since the paths of control must be considered.

3. The verification conditions had many parts since each operation had to be shown executable as well as the proofs of the assertions. There was much more to prove about the second program that came directly from the property of the \( \land \) operator of the one-line version. We did not follow strictly the rules for constructing verification conditions, but took several short-cuts in writing down the statements to be proved. For example, the branch conditions were written \( I > p M \) instead of \( I = p M \) and \( I < p M \) instead of \( O = (I > p M) \). This use of short-cuts is consistent with the informal manner of verifying programs, though the reader should be easily convinced that the format for verification conditions has been followed.

4. Verification condition 3 corresponds to an inconsistent control segment.

The example illustrates a number of issues of verification which must be further developed:

1. The requirements for a language for stating assertions (such as those involving quantifiers)

2. The generation of partial assertions and initial predicates which guarantee the absence of operation errors.

3. The definition of the language in a form suitable for use in a deductive system for proving verification conditions.

4. The specification of the deductive system

Though the programs were quite simple, the proofs were rather complicated even in the informal style used here. The question now is: To what extent can the informal process be mechanised or assisted mechanically?
PRAGMATIC QUESTIONS ABOUT PROGRAM VERIFICATION

Underlying the theory of program verification are several basic pragmatic questions that are worth considering in general and which we will attempt to answer and/or illustrate within the context of verification of APL programs.

1. What is being verified-a program or assertions about a program? In the strictest sense, neither! Verification is a proof of consistency between assertions about a program and the program itself, or more precisely, between the semantics of the assertions and the semantics of the program according to the semantic definitions of the assertion and programming languages. Of course, the proof itself must be carried out in some system which has been rigorously specified. So from one point of view, program verification is simply an exercise in applying deductive rules to some formulae developed by giving an interpretation to assertions and program statements.

According to the definition of correctness, a program is verified only with respect to some input-output relation, although it is quite conceivable to verify a program for more than one input-output relation, for example when the input and proofs break into cases. On this level, it is being verified that the programmer (or any person directing the verification) knows what the program does and can express that knowledge.

Usually there is an abstraction of a program called an algorithm. Algorithms are commonly given in an informal language with the emphasis on order and nature of steps to be performed. A program,
however, also involves the representation of the data manipulated by the algorithm. It sometimes is not clear what is being verified—the algorithm or the program.

2. What is an appropriate language for making assertions about programs?

The most commonly used assertion language has been predicate calculus simply because it seems best to reduce the verification problem to formulae suitable for input to a mechanical theorem prover for which there exist well-known proof techniques (resolution). On the other hand, there are certain classes of programs for which specialized proof techniques may exist, such as the Presburger arithmetic variants discussed by King[K1,KF] and Cooper[CO1], and the assertion language might be restricted for that class. The issue is the matching of power of expression of the assertion language with the power of the deductive system. An example where difficulty arises is when a recursively defined predicate or function is required to state the correctness of a program, in which case the proof techniques become quite different, as will be seen in an example in Appendix D.

The approach used here is to have the same assertion language as programming language, a decision for which there are several pros and cons. Using the same language may result in similar errors occurring in the assertion as in the program. On the other hand, the use of different languages requires that there be two separate language definitions. The choice would not occur for those programming languages which, unlike APL, lack the analog of quantifiers (which
seems to be a necessity) and the useful mathematical operators like plus-reduction (the sigma function). Another disturbing problem would be the conflict of precedence rules of APL with those of almost every known language.

3. What are the requirements for a definition of a programming language to be precise enough to make possible the verification of programs written in that language?

Of course, the answer depends on choice of language, but in a broader sense it involves consideration of the features of the language, where features fall into classes such as control structure, parameter passing, block structure, data declaration, and storage allocation. Language manuals are notoriously imprecise and while many language definitional methods have been developed and used, none has achieved universal acceptance. Indeed, most definitional methods handle some features well and others poorly. So we must ask what form a language definition must take to be suitable for program verification and what possible restrictions must be placed on use of features in a programming language such that its programs are verifiable.

4. The complete verification of even the simplest programs can become an overwhelming task when carried out in complete detail. Are there alternative, or partial, methods of verifying programs?

The most viable alternative is the "structured programming" approach of Dijkstra[DJ] where the emphasis is on the construction of a program in such a way that the correctness is not in doubt. The
standard verification approach is to take a program and augment it with assertions that presumably were in the mind of the programmer while the program was taking form. In the structured approach, the program originates almost at the level of assertions and is successively refined, maintaining the intent of the assertions, until an executable program is obtained. The final program is then "intuitively" correct and requires no formal proof as long as each of the refinements was sufficiently convincing.

APL programs which make extensive use of the structured operators are structured at a very high level in the sense that certain complex operations need not be refined to a very primitive level. The operators represent certain standardized loop formats for which known properties can be stated and used in proofs (such properties will be given in Chapter 5), similar in a sense to Hoare's[HO1] axioms. Thus some APL programs display some of the characteristics of structured programs: high level operations and standardized control structures.

The major difficulty with the assertion method is that frequently there is far more detail than realized that must be asserted in order to complete a verification. Making the assertions can be a difficult, burdensome, and error-prone task simply through the amount of detail and redundancy with program content. Perhaps it is possible to relieve at least part of the burden of making assertions by identifying certain program properties that require verification and developing specialized analytic techniques for creating assertions which must hold for those properties to be verified. These assertions must then be supplemented by the necessary conditions for proving the
correctness statement.

SUMMARY

Previous work in program verification was reviewed with the emphasis on the method of inductive assertions and the extent to which it has been used and developed. Several important issues were raised regarding the usefulness and difficulty of the method. The importance of language definition in program verification was stressed.

The formal basis for verification of APL programs by the inductive assertion method has been given. The critical difference in formalism from previous work is the shift of emphasis from control flow to operators. This is manifested by the extension of the definition of correctness and the construction of verification conditions to include the possibility that a program may terminate in an error condition. Furthermore, it was necessary to introduce the representation of a program in decomposed form since it will be advantageous to present the semantics of the language at the level of individual operators. Rules were given for constructing verification conditions from which it would be possible to prove that a program is executable and produces the desired result.

It now remains to show that the necessary components of a verification system for APL do exist.

1. A definition of the APL operators will be presented in Chapter 3. This would be a trivial component for most modern programming languages which usually have only arithmetic operators.
defined for only scalar values. We have chosen to ignore some aspects of APL such as function calls and input-output and to simplify the conditional transfer statement.

2. APL has been suggested as the basis for an assertion language and it must be determined whether restrictions might be placed on the description of properties of APL programs or whether restrictions might have to be placed on the assertion language. It was seen in the example that analogs of quantifiers exist as operators, but it will also be useful to introduce specific (bounded) quantifiers. This will be further discussed in Chapter 5.

3. The deductive system for proving verification conditions must be specified. It is beyond the scope and goals of this thesis to undertake the complete specification or partial implementation of such a system, for reasons which will be discussed in Chapter 5. Instead, a number of examples will illustrate the types of proof techniques and a system for verifying some assertions related to constraints is described in Chapter 4.
CHAPTER 3
FORMAL DEFINITION OF THE APL OPERATORS

INTRODUCTION

We have seen that verification of APL programs will require a precise description of the operators in a form suitable for use in a specialized deductive system for proving verification conditions. In the language definition given here, each operator description will consist of expressions for

1) the restrictions on operands for the operation to be executable,

2) the shape of the result of the operation,

3) the value of a single element of the result.

Some operators are chosen as primitive and described in various ad hoc but generally acceptable forms with the above expressions of all remaining operators given in terms of these primitives and of the operands.

These expressions will serve as axioms in the deductive system to be discussed in Chapter 5. The expressions 3), for the value of a single element, will lead to a useful specialized deductive step called destructuring, where the description of a single element of a multi-operator expression is developed from the expressions for single elements of operands. It is then natural to permit quantification over subscripts using destructuring. It will also be shown that several properties of the operators can be derived from the definition as theorems in the deductive system and then used extensively in
There are several ways in which the term "primitive" is being used. In this chapter, the primitives are

1) The p (shape) and , (ravel) APL operators, whose meanings can only be described informally and in English

2) The notion of vector shape, again described informally

3) The notion of indexing of a vector, leading to notation for indexing values of arbitrary shape

4) The APL operators for scalar operations, reduction over a vector, index generation, catenation, and representation of a scalar.

In terms of these notations and informally described operators, it is possible to state the shape and element expressions for all the remaining "nonprimitive" operators and the extensions of reduction, scalar operations, and representation of a scalar to higher dimensional shapes. Those expressions are given in Table 3.2. Within the deductive system, however, the indexing notations and the APL operators for reduction, index generation, catenation, and representation are not considered primitive since these are defined in terms of scalar operators and conditional expressions for which the deductive system has axioms. This point will be clearer when the deductive system is discussed in Chapter 5.

The definition consists of the following parts:

1) a description of values of variables
2) the primitive operators for indexing
3) primitive APL operators
4) tabular definition of remaining operators
5) constraints on operands
The APL operators have been well described in the reference manual by Pakin[PA], the formal description by Lathwell[LM], and the thesis by Abrams[AB]. The definitional form used here is motivated by the needs of verification not of implementation or instruction. Since the language has been in continuous evolution from Iverson’s original notation through the most recent versions of APL\360 and on some fine points of the operators there is no consensus in the above references, the definition is slightly idealized with respect to any particular implementation or description. Several major simplifications are assumed throughout the definition:

1) Only 0 origin is considered.
2) Character data is ignored.
3) No operands are ever elided.
4) Strict right to left evaluation is assumed.
5) The random number operators are omitted.

NOTATION AND CONVENTIONS

Two forms of identity operators are in use throughout. ↔ is intended to assert that the expressions on both sides will evaluate in the usual APL right to left rule to exactly the same value. <=> denotes that the symbol (s) on the left is defined by the expression on the right, i.e., the expression on the left is an abbreviation for the expression on the right. These operators have a precedence lower than APL operators. A⇒B and B⇒C will be abbreviated A⇒B⇒C.

Since the expressions are defined in terms of APL operators and TRUE and FALSE are represented in APL by 1 and 0, respectively, the
definition will follow the APL convention.

In some cases, conditional expressions are used to define operators and will obey the usual rule

\[ IF \ A \ THEN \ B \ ELSE \ C \ \langle=> \]

\[ B \ IF \ A=1 \]

\[ C \ IF \ A=0 \]

UNDEFINED, OTHERWISE.

Axiom systems for manipulating conditional expressions have been developed and will be discussed further in Chapter 5. Conditional expressions usually lead to case analysis in the deductive system.

VALUES

An APL value has two aspects: shape and elements. The SHAPE aspect refers to the dimensionality (scalar, vector, etc.) and size of each dimension. The ELEMENT aspect refers to the individual elements of the value. The vector shape will be considered primitive since for ALL values these two aspects may be obtained in the form of vectors.

The APL operator \( \rho \) selects the shape aspect of a value. The notion of shape may be factored into RANK (number of dimensions) and SIZE (number of permissible subscripts of each dimension). Rank is the length, or shape, of the shape vector while size is given by the elements of the shape vector.

EXAMPLE: Let \( \Lambda \) be the matrix

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{pmatrix}
\]

its shape, i.e. \( \rho/\Lambda \), is the vector 2,3. The rank of \( \Lambda \), i.e. \( \rho\rho/\Lambda \), is
2 and its size is 2 rows by 3 columns.

Another APL operator, \( \text{RAVEL} \), obtains a vector of the elements of a value in row-major order. In the example above,

\[
\text{A} \leftrightarrow 1,2,3,4,5,6
\]

A value \( \text{A} \) is a vector if \( \pi \text{A} \leftrightarrow \text{pA} \). \( \text{pA} \) is then a vector of length 1 whose single element gives the number of elements of \( \text{A} \).

In many cases, APL does not distinguish between a scalar and a one-element array of any rank. The additional convention will be followed in the meta-notation, especially for the identity operator \( \leftrightarrow \) where if \( \text{A} \) and \( \text{B} \) are single-element objects then \( \text{A} \leftrightarrow \text{B} \) is true if \( \text{A} \) and \( \text{B} \) have the same element regardless of shape. In all other cases, this asserts that the element and the shape aspects of both sides are identical.

Another special case is the empty vector which has as its shape the vector \( ;0 \) indicating that there are no elements. Similarly, an empty array has a \( 0 \) as an element of its shape vector indicating that one dimension has no elements.

**PRIMITIVE OPERATORS FOR INDEXING**

The shape and elements aspects of values are related through the indexing operation. Since the vector shape is considered primitive, it will be assumed that indexing of a non-empty vector is understood in the following sense:

---

*: A one-element array (also termed a UNIT) has a shape vector composed only of 1's.
The syntax is \( A[S] \) where \( A \) is a vector and \( S \) a scalar. \( S \) must be a legal subscript; that is, its value must be within the range 0 to \((\rho A)-1\) and \( S \) must be an integer. The result is the scalar which is the \( S \)th element of \( A \).

The general APL indexing operator permits subscripts to be of arbitrary shape and will be described later.

It has been found useful to introduce a primitive operator for indexing a single element of a value of arbitrary shape and it is this SIMPLE-INDEXING operator which occurs in the expressions for semantics of the operators. The syntax is \( A[V] \) where \( A \) is of arbitrary shape and \( V \) is a vector of length \( \rho P A \) which represents a subscript. If \( A \) is a scalar or one-element value then \( V \) is ignored and the result is \( A \). Otherwise the operation is reduced to indexing \( A \) by the value \( (\rho A)_V \) where \( V \) will be defined more precisely later (as the innerproduct, in the mathematical sense, of \( V \) and a coefficient weighting vector obtained from \( \rho A \) and can be informally understood to convert from the representation of \( V \) in a number system with base \( \rho A \) to a base 10 value. \( V \) must represent a legal subscript for \( A \), meaning that for each \( I \), \( V[I] \) is an integer with value in the range 0 to \((\rho A)[I]-1\).

\[
A[V] <\to \begin{cases} 
  I=\rho A \text{ THEN } (A)[0] \text{ ELSE } \\
  \text{IF } ((\rho P A)-(\rho V))\land((0(\rho A)(0(\rho V))))\land((0(\rho A)(0(\rho V))))\land((0(\rho A)(0(\rho V)))) \\
  \text{THEN } (A)((\rho A)_V) \text{ ELSE UNDEFINED }
\end{cases}
\]

It is also necessary to have a SLICING operator which selects all elements along a specified coordinate. The syntax is \( A[[B]V] \) where \( B \) is a scalar designating the selected coordinate and \( V \) is a subscript
vector of length $(\rho \Lambda)^{-1}$.

$\text{IF } I =_p A \text{ THEN } A[(B)V] \leftrightarrow (A)[0] \text{ ELSE }$

$pA[(B)V] \leftrightarrow (pA)[B]$  

$A[(B)V]][I] \leftrightarrow A[\text{INSERT}(B;V;I)]$  

\text{FOR ALL}

\text{INTGERS } I \text{ SUCH THAT } (0 \leq I) \wedge (I < (pA)[B])$

where

$p\text{INSERT}(A;B;C) \leftrightarrow \langle (p,B) \rangle$

$(\text{INSERT}(A;B;C))[I] \leftrightarrow \text{IF } I = A \text{ THEN } C \text{ ELSE } B[I-1]$

\text{FOR } I <_{(pB)} \text{INSERT}(A;B;C)$

\text{INSERT} and two other operators (written as functions) perform simple operations on vectors: $\text{INSERT}(A;B;C)$ inserts $C$ as the $A$-th element; $\text{SUBST}(A;B;C)$ replaces the $A$-th element of vector $B$ by $C$;

$\text{DELETE}(A;B)$ deletes the $A$-th element of vector $B$.

$p\text{SUBST}(A;B;C) \leftrightarrow \langle (p,B) \rangle$

$(\text{SUBST}(A;B;C))[I] \leftrightarrow \text{IF } I = A \text{ THEN } C \text{ ELSE } B[I]$

\text{FOR } I <_{(pB)} \text{SUBST}(A;B;C)$

$p\text{DELETE}(A;B) \leftrightarrow \Theta_{(p,A)}$

$(\text{DELETE}(A;B))[I] \leftrightarrow \text{IF } I < A \text{ THEN } B[I] \text{ ELSE } B[I+1]$

\text{FOR } I <_{(pB)} \text{DELETE}(A;B)$

\-------------------

In terms of regular APL indexing the following identities hold.

$A[V] \leftrightarrow A[V[0];\ldots;V[-1+\rho\Lambda]]$

$A[(B)V] \leftrightarrow A[V[0];\ldots;V[-1+B[-1]]; V[B];\ldots;V[-2+\rho\Lambda]]$

$\text{INSERT}(A;B;C) \leftrightarrow B[0];\ldots;B[A-1];C[B[A]];\ldots;B[-1+\rho\Lambda]$

$\text{SUBST}(A;B;C) \leftrightarrow B[0];\ldots;B[A-1];C[B[A]];\ldots;B[-1+\rho\Lambda]$

$\text{DELETE}(A;B) \leftrightarrow B[0];\ldots;B[A-1];B[A+1];\ldots;B[-1+\rho\Lambda]$
APL PRIMITIVE OPERATORS

MONADIC AND DYADIC SCALAR OPERATORS

These operators are defined sufficiently well elsewhere (Pakin[PA], Lathwell[LM]), and many have standard mathematical meanings. Since this thesis is not concerned with verification at the level of arithmetic expressions, the detailed definitions are omitted here. See Table 3.1 for a list of these operators.

REDUCTION ON A VECTOR

Let V be a vector V[0],...V[N] and D a dyadic scalar operator. The reduction of V by D is the result of accumulating the application of D to V, e.g. +/0 1 2 +=0+/1 2+=0+/1+=0+1+2

D/V<>EMPTY VECTOR
D/V[0],...V[N]<=>IF 0=pV THEN THE IDENTITY FOR D(IF ANY) ELSE IF 1=pV THEN V[0] ELSE V[0]D D/V[1],...V[N]

Reduction over a scalar S is defined as

D/S<> D/S S

INDEX GENERATION

The index generator, with form \S, produces a vector which is a sequence of integers from 0 to S-1. Where I=p,S and S\geq0 and S is an integer

I<>S
I<>S<>I
I<>S<>I FOR (0\leq I) \land (I\lt S) AND INTEGER I
FORMAL DEFINITION
PRIMITIVE OPERATORS

Note that if $S \rightarrow 0$ then no integers $I$ can satisfy the subscript expression and $\vec{S}$ is the empty vector. Therefore $\vec{0}$ will be used as notation for the empty vector.

CATENATION

If $((I=_{pp}V 1) \lor (I=_{pp}V 1)) \land ((I=_{pp}V 2) \lor (I=_{pp}V 2))$ i.e., if $V 1$ and $V 2$ are vectors or single-element values, then

$\vec{pV 1},V 2 \leftrightarrow (\vec{p},V 1)+_{p},V 2$
$\neg_{pp}V 1,V 2 \leftrightarrow 1$
$(V 1,V 2)(\{I\}) \leftrightarrow IF (0<_{I}) \land (I<_{p},V 1) THEN V 1(I) ELSE$
$IF (p,_{V 1}) \land (I<_{p},V 1) \lor (p,_{V 2}) THEN V 2(I-_{p},V 1)$

REPRESENTATION OF A SCALAR

If $V$ is a vector and $S$ is a scalar then $V \uparrow S$ is the representation of $S$ in the number system with radix $V$. Since $V$ may contain an arbitrary mixture of positive and negative real numbers, the operator will be defined recursively. $R$ is the result and $T$ contains intermediate values.

$pR\leftrightarrow pV$
$ppR\leftrightarrow I$
$T[pV]\leftrightarrow S$ and for $(0<_{I}) \land (I<_{p}V)$
$R(I) \leftrightarrow IF V(I)=0 THEN T[I+1] ELSE V(I)\uparrow T[I+1]$
$T[I] \leftrightarrow IF V(I)=0 THEN 0 ELSE (T[I+1]-R[I]-A[I])$

For example if $V \leftrightarrow 2.3$ and $S \leftrightarrow 5$ then
$T[2]\leftrightarrow 5$
$R[1]\leftrightarrow 3;5\leftrightarrow 2 T[1]\leftrightarrow (5-2)-3\leftrightarrow 1$
$R[0]\leftrightarrow 2;1\leftrightarrow 1 T[0]\leftrightarrow 0$
therefore $R \leftrightarrow 1.2$.

The use of this operator will require more explanation. It obeys the

---

*From here on, if the subscript expression is unsatisfiable then the result is empty.*
relation \( (T[0] \cdot V[0] \cdot W[0]) = S - R + \cdot W \) where \( W \) is a weighting vector giving the positional coefficients for the number system with \( V \) as radix, where \( + \cdot \) is the mathematical inner product. In the above example, \( W = 3,1 \). The above relation can be formally derived from the definition using the techniques in Chapter 5 and that proof is given in Appendix C. The proof illustrates the use of induction and the manipulation of conditional expressions.

This operator introduces the notion of number systems into the APL language and consequently leads to number-theoretic proofs. The operator is especially important in the context of this definition for the role it plays in describing subscripting. When the shape of the result of an operation is taken as the radix, the successive subscripts in row-major order can be generated via the representation operator, with the elements of the result then computed in that order using the element expressions. It will be shown exactly how this takes place in the later discussion of the definition. Representation is included as a primitive because of its nonstandard definition rather than its use in defining other operators.

NONPRIMITIVE OPERATORS

Each primitive has a prescribed role to play in the semantic expressions for the remaining operators. The scalar operators provide the basic capacity for arithmetic. Reduction introduces a simple form of iteration with an accumulation of results; with the max and min scalar operators this provides a form of selection. The basic construction operator, index generation, is used to create indices
which are then subject to transformation and selection. Representation provides the basis for generating subscript vectors from scalars. Catenation combines two objects from which selection can then be made.

The definition is organized into levels of operator definition, as shown below. The primitives are at level (0) and any operator at level(i) is defined by operators at levels (0)-(i-1).

(0) COMPONENT SELECTION: p,
SCALAR OPERATORS AND VECTOR REDUCTION: OP S, S OP S, OP/V
CATENATION: V,V
INDEX GENERATION: \S
REPRESENTATION OF A SCALAR: V\S

(1) A OP A', OP A, OP[I[S] A, A\tA'
V\tA, V\tA', \phi[S] A, A\phi[S] A'
(2) OP\S[A], A OP A', A OP.OP A', \kappa A, A\kappa A'
(3) A\kappa A', A/[S]A'
(4) V\kappa A, V\kappa A, A\{A\theta,...;\theta N}, V\kappa A, \delta V, \emptyset V
(5) V\S[A]

Table 3.2 gives the expressions for shape and element aspects of the result of an operation, R. The constraint expressions are given in Table 3.3. Some examples will be given to show how the primitives are used in the semantic expressions for other operators.

COMPRESS

R→A/B   \rho R\leftrightarrow;+/A   R[V] \leftrightarrow B[[V\leftrightarrow\Lambda] \leftrightarrow 1+\nu B]\A\leftrightarrow<0 1 1 0 0 1
C<->0 1 2 3 4 5
+/A \leftrightarrow 0 1 2 2 2 3 1+\nu A \leftrightarrow 1 2 3 4 5 6
\rho R \leftrightarrow ,3 \quad \rho R \leftrightarrow 1
R[0] \leftrightarrow C[1/\hat{0} 0 0 0 0 \times 1 2 3 4 5 6]=C[1]=1
R[1] \leftrightarrow C[1/0 1 0 0 0 0 \times 1 2 3 4 5 6]=C[2]=2
R[2] \leftrightarrow C[1/0 0 1 1 0 0 \times 1 2 3 4 5 6]=C[5]=5
R \leftrightarrow 1 2 5
The expression uses the scan and max-reduction operators to search for the \((v+1)\)-th occurrence of a 1 in \(A\), giving the index for the \(v\)-th element of the result.

**RANKING**

\[ R \leftarrow A \setminus B \quad \rho R \leftarrow \rho B \quad R[V] \leftarrow |\{(A, B[V]) = B[V]\} \setminus 1 + \rho, A \]

The compression produces the indices of \(A\) where \(B[V]\) occurs, if any, concatenated with \(\rho A\). The \(1/\) then returns the least such index or \(\rho A\).

**GENERAL INDEXING**

\[ R \leftarrow A[B_0; \ldots; B_N] \quad \rho R \leftarrow (\rho B_0), \ldots, (\rho B_N) \]

The general indexing operation is broken into several indirect steps. \(V\) is transformed into \(V'\) which can be used to select elements from the subscripts without regard to the subscript shapes, thus producing \(V''\) which actually is the subscript identifying the single element. The relation

\[ A[B_0; \ldots; B_N] \rightarrow (\rho B_0), \ldots, (\rho B_N)\rho A[B_0; \ldots; B_N] \]

explains the steps.

\[
A \leftarrow (2,3)_{\rho 16} \leftrightarrow \begin{array}{c} 0 \ 1 \ 2 \\ 3 \ 4 \ 5 \end{array} \\
B_0 \leftarrow 1 \quad B_1 \leftarrow (2,2)_{\rho 13} \leftrightarrow \begin{array}{c} 0 \ 1 \\ 2 \ 0 \end{array} \\
\rho R \leftrightarrow (10),2,2 \leftrightarrow 2,2 \\
V' \leftrightarrow (1,4 \setminus (2,2)1V \\
V'' \leftrightarrow (\rho B_0)\{V'[0]\},(\rho B_1)\{V'[1]\} \\
V \quad V' \\
----- \\
0 0 \ 0 0 \\
0 1 \ 0 1 \\
1 0 \ 0 2 \\
1 1 \ 0 3 \\
\rho R \leftrightarrow 2,2 \\
R \rightarrow \begin{array}{c} 3 \ 4 \\ 5 \ 3 \end{array} \]
TAKE

\[ R \leftrightarrow A \uparrow B \quad pR \leftrightarrow \downarrow A \quad R(V) \leftrightarrow B\{V + (A \{}0\{A + pB)\} \]

The expression \((A \{}0\{A + pB\) forms an offset to be added to each subscript of the result to obtain the appropriate element of \(B\).

\[ A \leftrightarrow 2 \quad B \leftrightarrow (2,3)p \leftrightarrow 6 \]

\[ pR \leftrightarrow 2,2 \]

\[ V' \leftrightarrow (A \{}0\{A + pB\leftrightarrow 0 \quad 1 \times 4 \quad 1 \leftrightarrow 0 \quad 1 \]

\[ V \quad V' \leftrightarrow V + V'' \quad B(V') \]

\[ \begin{array}{c|c|c}
0 & 0 & 1 \\
0 & 1 & 2 \\
1 & 0 & 4 \\
1 & 1 & 5 \\
\end{array} \]

\[ R \leftrightarrow 1 \quad 2 \quad 4 \quad 5 \]
### TABLE 3.1
SCALAR OPERATORS

<table>
<thead>
<tr>
<th>MONADIC FORM</th>
<th>DYADIC FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>fA</td>
<td>f</td>
</tr>
<tr>
<td><strong>PLUS</strong></td>
<td>+</td>
</tr>
<tr>
<td><strong>NEGATIVE</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>SIGNUM</strong></td>
<td>x</td>
</tr>
<tr>
<td><strong>RECIPROCAL</strong></td>
<td>÷</td>
</tr>
<tr>
<td><strong>CEILING</strong></td>
<td>[</td>
</tr>
<tr>
<td><strong>FLOOR</strong></td>
<td>]</td>
</tr>
<tr>
<td><strong>EXPONENTIAL</strong></td>
<td>=</td>
</tr>
<tr>
<td><strong>NATURAL</strong></td>
<td>°</td>
</tr>
<tr>
<td><strong>LOGARITHM</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MAGNITUDE</strong></td>
<td></td>
</tr>
<tr>
<td><strong>FACTORIAL</strong></td>
<td>!</td>
</tr>
<tr>
<td><strong>ROLL</strong></td>
<td>?</td>
</tr>
<tr>
<td><strong>PI TIMES</strong></td>
<td>÷</td>
</tr>
<tr>
<td><strong>NOT</strong></td>
<td>~</td>
</tr>
<tr>
<td></td>
<td>∧</td>
</tr>
<tr>
<td></td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>⊼</td>
</tr>
<tr>
<td></td>
<td>algorithms</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td>#</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TABLE 3.2
OPERATOR DEFINITIONS

* denotes primitive operators for which the definition is repeated or extended to higher dimensions. \( p \) and , are defined in the section on values. \( {} \) is the primitive indexing operation. \( D \) represents a scalar operator. The scalar operators are listed in Table 3.1 and are not further defined here. The first column gives the form of an operation for which \( R \) is the result while column 2 gives the shape of \( R \) and column 3 gives the value of a single element \( R[V] \) where \( V \) is any legal subscript. 0 origin is in effect.

<table>
<thead>
<tr>
<th>( R )</th>
<th>( pR&lt;&gt; )</th>
<th>( R[V]&lt;=&gt; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D/A )</td>
<td>( \rho A )</td>
<td>( D A[V] )</td>
</tr>
<tr>
<td>( A/) D B</td>
<td>IF ( I=\rho A ) THEN ( \rho B ) ELSE ( \rho A )</td>
<td>( A[V] D B[V] )</td>
</tr>
<tr>
<td>( A,B )</td>
<td>( (p/A)+\rho B )</td>
<td>IF ( V&lt;\rho A ) THEN ( A[V] ) ELSE ( B[V-\rho A] )</td>
</tr>
<tr>
<td>( \varLambda )</td>
<td>( \varLambda )</td>
<td>( V )</td>
</tr>
<tr>
<td>( A\cap B )</td>
<td>( (p/A)\rho B )</td>
<td>( A\cap B{\text{DELETE}(0;V)]}V{0} )</td>
</tr>
<tr>
<td>( D/(!A)B )</td>
<td>( \text{DELETE}(A;\varLambda)B )</td>
<td>( D/B{A;V} )</td>
</tr>
<tr>
<td>( A\cap B )</td>
<td>( \varLambda )</td>
<td>( B{V+(A&lt;\varLambda)\varLambda+\varLambda B} )</td>
</tr>
<tr>
<td>( A\cup B )</td>
<td>( (\rho B)-\varLambda )</td>
<td>( B{V+(A\geq\varLambda)\varLambda } )</td>
</tr>
<tr>
<td>( \Phi(!A)B )</td>
<td>( \rho B )</td>
<td>( B{\text{SUBST}(A;V_{(A;B)}[(pB)[A]-1+V(A)]) )</td>
</tr>
<tr>
<td>( A\Phi[B]C )</td>
<td>( \rho C )</td>
<td>( C{\text{SUBST}(B;V_{(pC)[B]V{B+\varLambda{\text{DELETE}(B;V)))}} )</td>
</tr>
<tr>
<td>( D\backslash(!A)B )</td>
<td>( \rho B )</td>
<td>( D{1+V(A)}B{A)\text{DELETE}(A;V) )</td>
</tr>
<tr>
<td>( A\cdot D B )</td>
<td>( (p/A)\rho B )</td>
<td>( A{(P/A){W}D B{(p/A){W} )</td>
</tr>
</tbody>
</table>
| \( A D, D' B \) | \( \text{DELETE}(1+pp/A;\rho A) \) | \( D/A\{1+pp/A\} \)
| \( \text{DELETE}(0;\rho B) \) | \( D/\{0\} \) | \( \text{DELETE}(0;\rho B) \)
| \( A\Delta B \) | \( \text{DELETE}(0;\rho B) \) | \( W+\cdot B\{\{0\} V\} \) WHERE \( W<=>\Phi;\backslash;1\varLambda \)
| \( A\rho B \) | \( \varLambda \) | \( B\{(pB)\varLambda A;V\} \cap R \)
| \( (B)\{(p,B)\varLambda A;V \) |
| \( A/B;C \) | \( \text{SUBST}(B;pC;+/A) \) | \( C\{\text{SUBST}(B;V_{(p\rho A)+\cap\varLambda)\cdot1+\varLambda \} \) |
### FORMAL DEFINITION

#### OPERATOR TABLES

<table>
<thead>
<tr>
<th>( A \backslash B )</th>
<th>( \rho B )</th>
<th>( {/((A,B(V))=B(V))/1+\rho,A } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \backslash (B;C) )</td>
<td>( \text{SUBST}(B;C;pA) )</td>
<td>( (C;{([B]\text{DELETE}(B;V)),0) } ) ( [(A/\rho,A)V[B]] )</td>
</tr>
<tr>
<td>( A(B_0,...,BN) )</td>
<td>( (\rho B_0),...,(\rho BN) )</td>
<td>( A(V^\prime) ) ( \text{WHERE} ) ( V^\prime \leq (([\rho,B_0],...,[\rho BN] \cap (\rho R) \cup V^V) ) ( (\rho R) \cup V ) ( \text{WHERE} ) ( V^\prime \leq (([\rho,B])\cap V^V) )</td>
</tr>
<tr>
<td>( \Psi A )</td>
<td>( ({-2</td>
<td>ppA]\downarrow \rho A}) ) ( \Phi({-2</td>
</tr>
<tr>
<td>( A \backslash B )</td>
<td>( (\rho R)(V)\downarrow ((/A=1)/\rho B) ) ( \text{WHERE} ) ( I \in 1+I/A )</td>
<td>( B{V(A)} )</td>
</tr>
<tr>
<td>( A \subset B )</td>
<td>( \rho A )</td>
<td>( \vee/A{V}=B )</td>
</tr>
<tr>
<td>( \Delta A )</td>
<td>( \rho A )</td>
<td>( ([P=\rho A] \mid [V \rightarrow /P&gt;A] ) ( \text{WHERE} ) ( P \leftrightarrow ([/(V+1)\leq/+A \rightarrow A]/A )</td>
</tr>
<tr>
<td>( \Psi A )</td>
<td>( \rho A )</td>
<td>( ([P=\rho A] \mid [V \rightarrow /P&lt;A] ) ( \text{WHERE} ) ( P \leftrightarrow ([/(V+1)\leq/+A \rightarrow A]/A )</td>
</tr>
</tbody>
</table>
As mentioned earlier, it is also necessary to give for each operator the restrictions on operands which must be satisfied for the operation to be executable. These constraints fall into three classes—shapes, modes, and bounds. Shape constraints usually require that there be some correspondence of shape between two operands or that a single operand be restricted to a particular shape. Various operators require that operands be integers (as opposed to real) values. Indexing requires that the values of indices fall within a range of legal subscripts. Domain constraints are restrictions on values for scalar operators and will not be discussed further here. The constraints are discussed separately then summarized for all operators in Table 3.2.

Shape Constraints

Shape constraints are conveniently described in terms of a few primitives:

\[ \text{UNIT}(Y) \iff I = \rho Y \leftrightarrow I = \neg \rho Y \]
\[ \text{LINEAR}(Y) \iff \text{UNIT}(Y) \lor (I = \rho Y) \]
\[ \text{COMPATIBLE}(Y1; Y2) \iff \begin{cases} \text{IF UNIT}(Y1) \lor \text{UNIT}(Y2) \text{ THEN TRUE} \\ \text{ELSE IF } (\rho Y1) = (\rho Y2) \text{ THEN } \land (\rho Y1) = (\rho Y2) \\ \text{ELSE FALSE} \end{cases} \]
\[ \text{DIM.COMPATIBLE}(Y1; Y2; C1; C2) \iff \begin{cases} \text{UNIT}(Y1) \lor \text{UNIT}(Y2) \lor \\ (\rho Y1)(C1) = (\rho Y2)(C2) \end{cases} \]

Operators such as the index generator and those with designated coordinates require a single-element object as an operand.
Compression, expansion, ranking, cationation, and others specify that an operand be a vector or unit. The dyadic scalar operators are performed on an element by element basis so compatibility of elements is required, where compatibility means that if one operand is a unit it will be paired with every element of the other operand and if neither element is a unit the pairs are formed by subscripting both operands by the same subscript.

Mode and Bounds Constraints

The selection class of operators impose constraints on the range and mode of operands. Expansion and compression require boolean left operands to indicate the binary choice of elements. Coordinate operands must be integer and within the range of dimensions of the appropriate operands. Indices must be nonnegative integers and bounded by the range of subscripts on the corresponding coordinate.

Modes will be divided into real, integer, nonnegative integer, and boolean denoted REAL, AINT, NINT, and BOOL. Bounds consist of lower and upper bounds on values.

These two forms of constraints overlap in the sense that values which are constrained to be nonnegative integers are required to be integers AND to be greater than or equal to 0. The redundancy will be useful in Chapter 4. Many scalar operators also have restrictions on mode and range, but these constraints will be ignored at present, though considered somewhat in Chapter 4.
**TABLE 3.3**

**SUMMARY OF CONSTRAINTS**

*denotes primitives --- denotes don't care or not applicable

$M(A)$ where $M$ is a mode denotes that $A$ is of mode $M$.

<table>
<thead>
<tr>
<th>R</th>
<th>SHAPE</th>
<th>MODE</th>
<th>BOUNDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ast$</td>
<td>$D/A$</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$\ast$</td>
<td>$A/D/B$</td>
<td>COMPATIBLE($A;B$)</td>
<td>---</td>
</tr>
<tr>
<td>$\ast$</td>
<td>$A,B$</td>
<td>LINEAR($A$)</td>
<td>LINEAR($B$)</td>
</tr>
<tr>
<td>$\ast$</td>
<td>$\forall A$</td>
<td>UNIT($A$)</td>
<td>NINT($A$)</td>
</tr>
<tr>
<td>$\ast$</td>
<td>$A\exists B$</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$\ast$</td>
<td>$D/[A]B$</td>
<td>UNIT($A$)</td>
<td>NINT($A$)</td>
</tr>
<tr>
<td>$\ast$</td>
<td>$A1B$</td>
<td>LINEAR($A$)</td>
<td>AINT($A$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(p,A)_{pp}B$</td>
<td>---</td>
</tr>
<tr>
<td>$\Phi[A]B$</td>
<td>UNIT($A$)</td>
<td>NINT($A$)</td>
<td>$A \subset {ppB$</td>
</tr>
<tr>
<td>$\Phi[B]C$</td>
<td>UNIT($B$)</td>
<td>NINT($B$)</td>
<td>$B \subset {ppC$</td>
</tr>
<tr>
<td></td>
<td>LINEAR($A$)</td>
<td>AINT($A$)</td>
<td>$\forall (A;B) \subset DELETE(B;ppC$</td>
</tr>
<tr>
<td>$\exists A$</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$\Phi[B]C$</td>
<td>LINEAR($A$)</td>
<td>NINT($A$)</td>
<td>$\forall A \subset {l/A \subset A$</td>
</tr>
<tr>
<td>$D\setminus[A]B$</td>
<td>UNIT($A$)</td>
<td>NINT($A$)</td>
<td>$A \subset {ppB$</td>
</tr>
<tr>
<td>$A/D/B$</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$A/D1.D2. B$</td>
<td>DIM.COMPATIBLE</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>$(A;B; I_{pp}/A;0)$</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$A1B$</td>
<td>DIM.COMPATIBLE</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>$(A;B; I_{pp}/A;0)$</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$A/PB$</td>
<td>LINEAR($A$)</td>
<td>NINT($A$)</td>
<td>---</td>
</tr>
<tr>
<td>$A/[B]C$</td>
<td>LINEAR($A$)</td>
<td>BOOLEAN($A$)</td>
<td>$B \subset {ppC$</td>
</tr>
<tr>
<td></td>
<td>( U \mid B )</td>
<td>( A \backslash B )</td>
<td>( A \backslash (B \backslash C) )</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>( A \lor B )</td>
<td>( U \mid B )</td>
<td>( L \mid A )</td>
<td>( L \mid (A \cup C \setminus B) )</td>
</tr>
<tr>
<td>( A \backslash B )</td>
<td>( L \mid A )</td>
<td>( L \mid (A \cup C \setminus B) )</td>
<td>( B \mid { A_0 } )</td>
</tr>
<tr>
<td>( A \backslash (B \backslash C) )</td>
<td>( L \mid (A \cup C \setminus B) )</td>
<td>( B \mid { A_0 } )</td>
<td>( B \mid { A_0 } )</td>
</tr>
<tr>
<td>( \Pi(B_0; \ldots; B_N) )</td>
<td>( N \mid (B_0; \ldots; B_N) )</td>
<td>( N \mid (B_0; \ldots; B_N) )</td>
<td>( N \mid (B_0; \ldots; B_N) )</td>
</tr>
<tr>
<td>( A \subset B )</td>
<td>( B \mid { A_0 } )</td>
<td>( B \mid { A_0 } )</td>
<td>( B \mid { A_0 } )</td>
</tr>
<tr>
<td>( \Phi(A) )</td>
<td>( N \mid (B_0; \ldots; B_N) )</td>
<td>( N \mid (B_0; \ldots; B_N) )</td>
<td>( N \mid (B_0; \ldots; B_N) )</td>
</tr>
<tr>
<td>( \forall(A) )</td>
<td>( N \mid (B_0; \ldots; B_N) )</td>
<td>( N \mid (B_0; \ldots; B_N) )</td>
<td>( N \mid (B_0; \ldots; B_N) )</td>
</tr>
</tbody>
</table>
DEFINITION OF ASSIGNMENT

One further definition must be completed. Values may be assigned to subscripted portions of arrays, where the syntax is $A(B0;...;BN杀手)C$. The constraints are described by

$$S.COMPATIBLE(X;Y)\iff (I=X/Y)\land ((pX)=(pY)) \land (X=Y)$$

and the constraints are $S.COMPATIBLE((pB0),...pBN;C)$. The result will be denoted as $A'$ and

$$pA'\iff pA$$
$$A'[V]\iff IF (\forall I) I[B1 THEM C]ELSE A[V]$$
where $V'[I]\iff (p,B1)-1+(p,B1)\land V[I]$.
$$V''\iff (pC)\land ((p,B0),...pBN)\land V'$$

The complexity of this expression is due to the possibility that a subscript appears twice in $B0;...;BN$ in which case the value in that position is the element in $C$ such that its subscript corresponds to the last occurrence of the subscript for $A$.

Example: $A(0 0 1;0)杀手9 10 11$

$$A\iff (2,2)\land t\quad B0\iff 0 0 1\quad B1\iff 0$$

<table>
<thead>
<tr>
<th>$V$</th>
<th>$V'$</th>
<th>$V''$</th>
<th>$A'[V]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>1 0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>0 1</td>
<td>---</td>
<td>---</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>2 0</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>1 1</td>
<td>---</td>
<td>---</td>
<td>3</td>
</tr>
</tbody>
</table>

$A'\iff$

$$10 1$$
$$11 3$$

DERIVED PROPERTIES

The definition has provided a precise description of every operator in a standard form reducible to a set of primitives. It is possible to derive from the definition certain properties which we expect to hold for the operators. These properties have a two-fold
purpose: to assure the reader that the definition is correct and to provide the basis for verifying further properties of APL programs. As the proofs require specification of the deductive system, they are discussed later in Chapter 5 and listed in Table 5.2.

DISCUSSION OF THE DEFINITION

This definition has evolved from an effort to define APL in the Vienna Definition Language[LW] by another Carnegie-Mellon graduate student, Larry Snyder, and the author. The size and unwieldiness of the definition, which included not just the operators but also expression interpretation, lexical analysis, function calls, and workspace commands, prohibited its completion and publication. However, the definition of the operators provided the fundamental insight into the underlying importance of indexing. The original VDL definition had VDL routines which were shared by almost all operators and accomplished roughly the primitive indexing operators described here. The major failing of this definition effort was that the most interesting and unique feature of APL, the structured operators, was obscured by the unfamiliar notation of VDL, the detail of describing the entire APL system, and the mistaken attempt by the authors to "program" the definition in a language better suited for expressing information structures and transformations than lengthy, complicated algorithms.

VDL represents one style of language definition, commonly referred to as operational or COMPUTATIONAL, and characterized by the reliance on state descriptions and state transition functions. Given
a basic interpretation mechanism and formalism for describing states and transitions, the meaning of a program is the computation of a language-specific interpreter for that program. In a VDL definition, this involves giving a description of a state, of an initial state loaded with a program and its data, and of the instructions and functions which drive the state transition. Lathwell's[LM] description of APL is also computational in form, with APL as the base language and interpretation mechanism.

In contrast with the computational definitional form is that suggested by Hoare[HO1], Ashcroft[AS], and Floyd[FL1] and referred to as DENOTATIONAL. The form may be axioms, rules for constructing verification conditions, or functions and equations relating functions. The concern is more with the description of the result of executing various language constructs than with the implementation of the constructions.

Which method is more appropriate for use in verification? Clearly, any nondenotational definition pushes the program verification problem one step back to the language for definition of the language in which the program is written since assertions must be proved about the computation of the basic definition mechanism. This doesn't reduce the value of computational definitions, which may be more natural than denotational definitions, but only requires that it be possible to derive the semantics of the particular language features critical for verification, as in Allen[AL2].

The definition of the APL operators has been presented here
primarily in denotational form (the major exception being the
representation operator) but it is also computational if the basic
interpretation cycle is taken to be (informally):

1) Fetch the operands and check whether they satisfy the
   constraints. If not, signal an error.

2) Otherwise, compute the expression for the shape of the
   result and

3) Generate all subscripts for the result and compute the
   elements of the result, \( R_i \), by evaluating the definitional
   expression for a single element. This is equivalent (in mixed
   ALGOL and APL form) to

   \[
   \text{FOR } I=0 \text{ STEP 1 WHILE } I<r \text{ DO}
   \]
   \[
   [V=(0R)i; \quad R[V]+\text{DEFINITIONAL EXPRESSION}]
   \]

   (This illustrates the use of the representation operator for
   subscript generation.)

4) Store the shape and element vectors of the result.

The definitional expressions for the APL operators will play the
role of axioms upon which will be based a deductive system for proving
properties of APL programs. Several operators used conditional
expressions, axioms for which will be discussed in Chapter 5. The
derived properties of the operators are theorems in this system which
will be used as the basis for proving further properties of programs.

The definition given here has much in common with other
descriptions of APL. The thesis by Abrams[AB] develops normal forms
for certain APL expressions as the basis for a machine tailored to
execute APL efficiently. The reference manual of Pakin[PA] describes
each operator with shape and constraint expressions and examples of
the various shapes of operands.
SUMMARY

The definition of APL is the basis for verification of APL programs which will be further discussed in Chapters 4 and 5. A few APL and other operators form a basis of primitives for defining the remaining operators in a standard form. All operator definitions consist of expressions in the extended APL for the restrictions on the operands and for the shape and element aspects of the result of operation. Chapter 4 will discuss the mechanical verification of constraints and Chapter 5 will use the definition to prove assertions about APL programs.
CHAPTER 4
VERIFICATION OF CONSTRAINTS

INTRODUCTION

The major concern of this thesis is the study of the nature of verification for a language with powerful operators in which it is possible to write programs with relatively few loops. It was pointed out that though the operators of APL are very powerful, they are generally meaningful only when the operands satisfy certain constraints which were precisely stated in the definition in Chapter 3. Furthermore, APL permits great flexibility in mode and shape. As a result, it was necessary to extend the standard formalism for verification conditions to include the partial operations of APL along with the complex semantics of some of the operators. Part of the task of proving a verification condition is showing that all operations along a control path are executable, as required by the definition of correctness. It will be shown in this chapter that it is possible to develop stronger techniques in which partial assertions are generated while performing this part of the verification proof.

This chapter first discusses the rationale and goals of the approach to constraint verification. A series of procedures for dealing with constraint verification are outlined and the basic underlying procedure is described in detail. The implemented system will perform syntax analysis then expression decomposition (as discussed in Chapter 2) on any input program. The constraint
verification procedures then produce a list of unresolved constraints or an indication of errors. The output of the system is a tabular representation of the decomposed program with the constraint list. Annotated examples illustrate the procedure. Finally the performance of the system is discussed and the method is evaluated.

ERRORS IN APL PROGRAMS

Many programmers find that debugging is sometimes not just a matter of getting their program to produce the right answer but to produce any answer at all. Even though the program may be correctly structured, there are usually problems due to mistyping or to misunderstanding some language details. Frequently, programs are written for some "typical" data where it might not yet be clear what the special cases are. In verification, it is tacitly assumed that a program has been through this debugging phase and that the programmer is fairly confident of its correctness, although there is no reason why verification cannot replace debugging someday. The technique to be put forth here is described as part of a verification system but it could also be useful for the analysis of a program at some point in the debugging phase.

It might be worth considering some of the kinds of problems which crop up in debugging and must also be taken into consideration in verification, particularly for APL.

1. Domain errors

Each scalar operation is a function which has a domain over which it is defined. The classical example that occurs in almost every
programming language is division by 0 (ironically, 0 divided by 0 is defined as 1 in APL). For a program to be completely correct, the operands of every operation must fall into the domain of the operator. This thesis will ignore this kind of error since the techniques are quite different and would involve some complicated mathematics and numerical analysis.

2. Array Bounds

There are at least two common causes for this error: either an expression for a subscript is written incorrectly and computes an invalid subscript or a variable used as a loop control is used incorrectly beyond the scope of the loop.

3. Undeclared Attributes

The term ATTRIBUTES refers to the properties of a variable which affect its use within a program. In the context here, the attributes refer to the shape (scalar, vector, etc.), mode (real, boolean, etc.) and bounds (range of values). This error occurs in a different form in the many languages where all variables must be restricted in size, rank, and mode by declarations and a compiler catches the errors. However, APL is one of several languages (others are JOSS, LCC) where the major reason for the language's existence is its flexibility, making it suitable for use in a conversational environment where efficiency of execution is traded for style of program development. Furthermore, many of the operators have a strictly dynamic character, e.g. expansion requires that the size of the right operand equal the number of ones in the left operand. In a language with undeclared
attributes, the aspects of the value of a variable include not just its actual elements but also its size, and rank and possibly, mode. Therefore, it is fair to make assertions about mode and shape and the deductive system must be prepared to deal with these assertions.

It should be noted that the APL structured operators subsume much of the verification of subscripts as well as removing potential sources of errors. For example, a loop to add element-by-element two vectors would require an inductive assertion and three verification conditions with subscript verification in at least one. In King's system [KII1], subscripted variables presented a major problem because of profusion of cases. The same basic theoretical problems remain in APL but the use of structured operators vastly reduces the frequency of occurrence.

GOALS AND RATIONALE

The goal of this Chapter is to report on the development and implementation of a method for verifying assertions about shape and mode and for verifying that a program will execute properly. Besides constituting part of the APL deductive system, the method has the side benefit of being able to generate preassertions which, if satisfied, guarantee that no errors will occur and that postassertions are satisfied. Thus the method relieves the programmer of the burden of making assertions about some of the properties of the program. The left justification example of Chapter 2 indicated the basic goals and nature of this method.
What reason is there to believe that the method can make any significant reduction of verification effort? After all, the shape is a completely dynamic property of a program since the left operand of the reshape operator can be any APL expression. There are three reasons which motivate both the study of the general problem and the development of specific techniques:

1. Most programs have some fixed data representation (scalar, vector, matrix, etc.) dictated by the structure of the problem. This data representation should be used consistently throughout the program resulting in redundancy of constraints. In one sense, the goal is to extract the representation information packed into the operator structure of the program and check that all variables are consistently and correctly used with respect to that representation.

2. APL operators can be classified according to the way they influence the three attributes of interest—shape, mode, and bounds:

   a) Some operators originate information. For example, \(1\times\) produces a nonnegative integer vector.

   b) Many operators preserve attributes, e.g. the rearrangement operators.

   c) Other operators transform the attributes. For example scalar operators transform mode and bounds. Indexing transforms shape (of Indexed value).

3. There is a certain style to APL programs that extensively use the structured operands. A common practice is to synthesize an object with certain desired properties then perform transformations on the object or use the object to transform other objects.
EXAMPLE: \((X=\Lambda)/\Lambda\)
The process illustrated here is "Create the characteristic vector \((X=\Lambda)\) which marks the position in \(\Lambda\) where \(X\) occurs. Create the vector of indices \(\Lambda/\Lambda\). Use the characteristic vector to select the positions of \(\Lambda\) where \(X\) occurs." In another language, this process might be described by the loop

\[
R \leftarrow 0 \\
\text{FOR } I \leftarrow 0 \text{ STEP 1 WHILE } I < \Lambda \text{ DO} \\
\quad \text{IF } \Lambda[I] = X \text{ THEN } R \leftarrow R, I
\]

The APL expression constructs intermediate objects—the characteristic vector and the index vector—then performs selection using compression. Each intermediate object is easily and clearly defined by the operator used in its construction. Furthermore, those intermediate objects have the same shape and the characteristic vector is obviously boolean.

Horrible as it sounds, APL encourages inefficient coding and this is a great boon to verification. The less complicated the flow of control, the less the need for assertions and the fewer verification conditions. However, there is no reason why optimized programs can't be verified if a two stage approach is taken: first write the program in a language that has good verification properties and system and verify it in that language and then translate that program to a more efficient form or a different programming language showing that the translation preserves correctness.

OUTLINE OF THE METHOD

A procedure will be given for developing from an expression a PRE-CONDITION (partial assertion) which if satisfied guarantees that every operator in the expression can be properly executed. The pre-condition might be described as "reduced constraints" since it is formed by examining the constraints of the operations of the expression, discarding those which are satisfied from known properties of the operands and previous constraints. Once this procedure is
available, it will be shown how to use it for programs with assertions and no branches, programs with branches and assertions, and programs with branches and no assertions. After the procedures are outlined, the details will be given and illustrated. The constraints appear in tables 4.2-4.4. The reader may find it useful to skim the outline and the accompanying descriptions, then fill in the details by studying the annotations of the examples.

Consider first the simplest case - a one-line program with no assertions. The goal is to obtain an initial condition which guarantees that no errors will occur during execution of the program.

**EXAMPLE:** Returning to the left justification example of Chapter 2, the constraints were formed in the following way

\[ M \neq 0 \]

\( M \) and 0 must be compatible (as defined in Chapter 3) and this is satisfied since 0 is a unit. The shape attribute of the result is \( \rho M \) and the mode is boolean.

\[ (M \neq 0) \& 1 \]

The shape of \( M \neq 0 \) is constrained to linear but since \( \rho M \) is the shape of \( M \neq 0 \) the shape of \( M \) must be linear. The shape attribute of the result is the shape of the right operand 1. The mode attribute is nonnegative integer.

\[ ((M \neq 0) \& 1) \oplus M \]

The two forms of the constraints of the rotate operator depend on the shape of the left operand - either a unit or a shape equal to the shape of the right operand missing the specified coordinate. The former case holds here. The mode constraint on the left operand to integer is satisfied by the nonnegative integer attribute of 1.

The only remaining constraint is \( \text{LINEAR}(M) \). All other constraints were satisfied by the attributes of the operands.

This leads to the following description of the basic process of constraint verification: Process each of the successive subexpressions of the decomposed program performing the following two steps for each while maintaining as much information on attributes as possible.
1. Constraint Application

There are three possibilities for the constraints of the operation being considered:

a) Satisfied by the information about the operands. Then discard the constraint (it has been verified). In the example, COMPATIBLE(\textit{M};0) was satisfied by the unit shape attribute of 0.

b) Contradicted by the information about the operands. Signal that the entire verification condition is false.

c) Neither satisfied nor contradicted. There is insufficient information about the operands. The constraint must eventually either be satisfied or covered by the initial condition if the verification condition is to be true, so assume that the constraint is true from that point on (i.e. add it to the present information), also making any additional deductions about other constraints that are warranted. In the example, the linearity of \textit{M} \neq 0 was assumed and transferred to the linearity of \textit{M}.

2. Attribute determination

Given what information there is about the attributes of the operands, compute the attributes of the result of the operation. In the example, for the result of (\textit{M} \neq 0)\textsubscript{1} the shape attribute was scalar and the mode attribute was nonnegative integer.

After all subexpressions have been processed the possibilities are

1) All constraints were resolved.

2) Constraints remain which can be satisfied due to constraints made after processing the subexpression from which the original constraint arose. That is, the constraints can be reduced.

3) No other constraints can be satisfied.

It is possible to distinguish between case 1 and cases 2 and 3 simply by keeping a record of the unresolved constraints. But distinguishing
between 2 and 3 will require another pass. The second pass computes those attributes which it now has the information for and didn't have on the first pass and again tries to satisfy each remaining constraint. Much simplification can also be performed and may assist in satisfying constraints. However, the simplification of constraints may also lead to new constraints and lead to further passes. For the example above, the constraints would be unchanged through a second pass, leaving as the only constraint $LINEAR(M)$.

There are several properties of this procedure which should be proved to show that it does accomplish the verification. 1) No constraint should ever be discarded that is not satisfied or would conflict with another constraint. 2) All constraints left unresolved should be necessary. 3) The procedure should be complete in the sense that all constraints are reduced as far as possible within the system.

Call this algorithm SLC (Straight Line for Constraints). The details will be given in a later section. SLC will be used as the basis for other procedures SLA, BA, and BC in the following ways.

SLA (Straight Line with Assertions)

As discussed in Chapter 2, assertions are written in APL and thus may have partial operators. The assertions (if any) will be processed by SLC yielding a set of constraints which make the assertion executable. In the left justification example of Chapter 2, the assertion $N[0] \neq 0$ requires that $(l = p \circ N) \land (0 < p N)$.

Then the assertion will be broken down into individual properties
of input variables according to a process called EXTRACTION ("extracting" the assertions from the expression). These extracted assertions will have the same form as expressions in the constraints and will be treated as constraints on initial variables.

Example: If the assertion \((v/M\neq 0)\land (l=ppM)\) were subject to extraction, then \(l=ppM\) would be mapped into a constraint on the shape of \(M\). The assertion \(v/M\neq 0\) would be ignored since it says nothing about the attributes of interest.

The decomposed statements of the program can then be processed under the constraints developed from the form and the content of the pre-assertion. Finally the post-assertion is reached and the constraints for its execution must be satisfied. Sub-assertions are then extracted from it and treated as constraints to be satisfied.

Each of these procedures SLA and SLC results in a modified verification condition. The terms used informally in the outline of the method can be made precise by referring to components of the verification condition. The "constraints" are the expressions given by the formal definition. That part of the operator semantics having to do with shapes constitutes part of the attribute "information" gathered by SLC. The remaining part of the "information" consists of mode and bounds attributes which can be derived from the definition. These algorithms represent a partial deductive system for the verification of correctness. The deductive system only "understands" shape, mode, and bounds constraints and its goal is to reduce those constraints to other problems which could then be solved by further appropriate deductive systems or incorporated into initial assertions.

The SLC and SLA algorithms operate on a verification condition in
the formalism of Chapter 2 to produce a modified verification condition of the form

\[ II \wedge (II \Rightarrow (P \land A_2 \land \ldots \land A(N-1) \Rightarrow Q \land G)) \]

where \( II \) is a conjunction of the constraints reduced to input variables
- \( P \) is the preassertion
- \( G \) is a conjunction of constraints on intermediate values of the expression
- \( Q \) is the postassertion

Thus, the constraints have been reduced to two formulae \( II \) and \( G \). All the constraints on input variables have been combined into \( II \) and the intermediate constraints have been combined into \( G \). The result is a modified verification condition which looks more like those of Floyd, where the operations were considered to be total. For straight-line programs the effect has been to strengthen the input-output predicates assigned by the human verifier so that operation errors will be avoided.

For the example from Chapter 2,

\begin{align*}
[1] & A \land 1 < A \\
[2] & II' \Rightarrow A \\
[3] & AI \land p/A
\end{align*}

the verification condition reduces to

\[ II \wedge (II \Rightarrow ((1 < A) \land (I' \Rightarrow A)) \Rightarrow I' < p/A)) \]

where \( II' \Rightarrow (1 = p/A) \lor (1 = p/A) \) and \( G \Rightarrow \text{TRUE} \).

BA (Branching with Assertions)

Form the verification conditions in the usual way and perform SLA for each. The modified verification conditions have now been produced. What about \( II' \)? It must be considered in forming the verification condition preceding the present one along the control path being verified. If there are no loops, it will be possible to push all reduced constraints back to the entrance. But if there are
loops, it will be necessary to further process the modified verification conditions.

BC (Branching without Assertions, only Constraints).

The goal is simply to generate an initial condition to cover the constraints. This might be used just to get the ball rolling at the start of a verification with the intention of putting on the complete assertions later. It could also be used as a preliminary check that the program is correctly typed and doesn't suffer from any shape errors. The first step will be to break the program into segments by identifying the assertion places as the entrance, exit, and every merge (where two paths converge). Then the BA procedure is applied with TRUE as both the pre-assertion and postassertion.

EXAMPLE: Using the second form of the left justification program of Chapter 2

Before presenting the details of the algorithm, the nature of the attributes and constraints should be reconsidered. The attributes and constraints of interest are shape, mode, and bounds, while scalar domain constraints are ignored. Shape constraints have to do with errors of rank and size. Mode constraints cover the errors where an operand must have values which are boolean, nonnegative integer, or integers. Bounds constraints cover the range of values required for subscripting and coordinate selection.

As mentioned previously, the operators can be grouped into classes according to how they affect these attributes—preservation, transformation, and origination. Table 4.1 gives the breakdown of operators into these classes.

Table 4.2 gives the attributes and constraints for shapes. The functional notation, e.g. \( LINEAR(A) \) is used to mean that \( A \) must satisfy the definition of linear constraint given in Chapter 3. Tables 4.3 and 4.3.1 list bounds attributes for those operators for which meaningful bounds expression can be obtained. The remaining operators are ignored in the system. Table 4.4 and 4.4.1 give the meaningful mode attributes and constraints for the nonscalar and some scalar operators.
### TABLE 4.1

<table>
<thead>
<tr>
<th>CONSTRAINT</th>
<th>OPERATORS WHICH ORIGINATE</th>
<th>OPERATORS WHICH PRESERVE</th>
<th>OPERATORS WHICH TRANSFORM</th>
<th>OPERATORS WHICH CONSTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>$T , p_1 , \lor_1$</td>
<td>$R , C , ,_1 , p_2$</td>
<td>$S , E , ,_2$</td>
<td>$S , E , S , C , M , N$</td>
</tr>
<tr>
<td>Bounds</td>
<td>$T , \lor_1$</td>
<td>$C , R , ,_1 , p_2$</td>
<td>$S , E , ,_2 , \lor$</td>
<td>$N$</td>
</tr>
<tr>
<td>Shape</td>
<td>$\lor_1 , p_2 , \lor_1$</td>
<td>$S , T , \phi$</td>
<td>$\phi$</td>
<td>$ALL , BUT$</td>
</tr>
</tbody>
</table>

**Operator classes (according to breakdown in Chapter 2).**

<table>
<thead>
<tr>
<th>Code</th>
<th>Class</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>SCALAR</td>
<td>$+,-,...$</td>
</tr>
<tr>
<td>$E$</td>
<td>EXTENDED SCALAR</td>
<td>ABOVE APPLIED ELEMENT BY ELEMENT \ AND COMPOSED</td>
</tr>
<tr>
<td>$T$</td>
<td>SEARCH</td>
<td>$\lor_2 , \phi , \lor_1$</td>
</tr>
<tr>
<td>$R$</td>
<td>REARRANGEMENT</td>
<td>$\phi , \phi , \lor$</td>
</tr>
<tr>
<td>$M$</td>
<td>CONSTRUCTION</td>
<td>$,2 , \lor_1 , \lor_2$</td>
</tr>
<tr>
<td>$C$</td>
<td>SELECTION</td>
<td>$\lor_1 , \lor_1 , /$</td>
</tr>
<tr>
<td>$N$</td>
<td>COORDINATE</td>
<td>$\phi , \phi , \lor , D/$</td>
</tr>
</tbody>
</table>

The class code is used wherever the category applies to every operator in the class. The 1 or 2 following the operator designates its monadic or dyadic use.
### Table 4.2
SHAPE CONSTRAINTS AND ATTRIBUTES

<table>
<thead>
<tr>
<th>Operator</th>
<th>Expression for shape of result</th>
<th>Expression for rank of result</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>A * A</td>
<td>( pA \cdot ppA )</td>
<td>1</td>
<td>---</td>
</tr>
<tr>
<td>A * A</td>
<td>( pA \cdot ppA )</td>
<td>1</td>
<td>---</td>
</tr>
<tr>
<td>A * B</td>
<td>((pA_1+pA_2) )</td>
<td>1</td>
<td>LINEAR(A)</td>
</tr>
<tr>
<td>A * B</td>
<td>((pA_1+pA_2) )</td>
<td>1</td>
<td>LINEAR(B)</td>
</tr>
<tr>
<td>A * A</td>
<td>( pA )</td>
<td>1</td>
<td>UNIT(A)</td>
</tr>
<tr>
<td>A * A</td>
<td>( pA )</td>
<td>1</td>
<td>UNIT(A)</td>
</tr>
<tr>
<td>A * A</td>
<td>( pA )</td>
<td>1</td>
<td>UNIT(A)</td>
</tr>
<tr>
<td>A * A</td>
<td>( pA )</td>
<td>1</td>
<td>UNIT(A)</td>
</tr>
<tr>
<td>A * A</td>
<td>( pA )</td>
<td>1</td>
<td>UNIT(A)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(A)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(B)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(A)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(B)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(A)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(B)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(A)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(B)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(A)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(B)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(A)</td>
</tr>
<tr>
<td>A * B</td>
<td>( pA_1+pA_2 )</td>
<td>1</td>
<td>LINEAR(B)</td>
</tr>
</tbody>
</table>
TABLE 4.3
BOUNDS CONSTRAINTS AND ATTRIBUTES

Bounds are represented by a tuple
[lower bound, upper bound]

\( \text{COMBINE}(A;B) \)
is described in Appendix B under bounds constraints.

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>ATTRIBUTE</th>
<th>CONSTRAINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pA )</td>
<td>( 0 \leq pA )</td>
<td>---</td>
</tr>
<tr>
<td>( ,A )</td>
<td>( \text{BOUNDS}(A) )</td>
<td>---</td>
</tr>
<tr>
<td>( A,B )</td>
<td>( \text{COMBINE}(A;B) )</td>
<td>---</td>
</tr>
<tr>
<td>( \lambda A )</td>
<td>( [0, \lambda + A] )</td>
<td>---</td>
</tr>
<tr>
<td>( \Lambda )</td>
<td>( \text{BOUNDS}(B) )</td>
<td>---</td>
</tr>
<tr>
<td>( A\backslash B )</td>
<td>( \text{COMBINE}(C;A) )</td>
<td>---</td>
</tr>
<tr>
<td>( A[BO,...;BN] )</td>
<td>( \text{BOUNDS}(A) )</td>
<td>( \forall I ) ( \wedge /{IF}c(pA)[I] )</td>
</tr>
<tr>
<td>( A\uplus B )</td>
<td>( \text{BOUNDS}(B) )</td>
<td>( \forall I ) ( A[I]B{1+(pB)[I] )</td>
</tr>
<tr>
<td>( A\downarrow B )</td>
<td>( \text{BOUNDS}(B) )</td>
<td>( \forall I ) ( (A[I])B{1+(pB)[I] )</td>
</tr>
<tr>
<td>( A/{B}C )</td>
<td>( \text{BOUNDS}(C) )</td>
<td>---</td>
</tr>
<tr>
<td>( \phi(A)B )</td>
<td>( \text{BOUNDS}(B) )</td>
<td>( A\in \pp B )</td>
</tr>
<tr>
<td>( A\phi{B}C )</td>
<td>( \text{BOUNDS}(C) )</td>
<td>( B\in \pp C )</td>
</tr>
<tr>
<td>( \Sigma A )</td>
<td>( \text{BOUNDS}(A) )</td>
<td>( \forall A \in [1/A] )</td>
</tr>
<tr>
<td>( A\backslash B )</td>
<td>( \text{BOUNDS}(B) )</td>
<td>( \forall /{1/A} \in A )</td>
</tr>
<tr>
<td>( A\setminus B )</td>
<td>( [0,pA] )</td>
<td>---</td>
</tr>
<tr>
<td>( A\setminus B )</td>
<td>( [0,1] )</td>
<td>---</td>
</tr>
<tr>
<td>( \delta A )</td>
<td>( [0,\delta + pA] )</td>
<td>---</td>
</tr>
<tr>
<td>( \Psi A )</td>
<td>( [0,\Psi + pA] )</td>
<td>---</td>
</tr>
<tr>
<td>( D/(A)B )</td>
<td>---</td>
<td>( A\in \pp B )</td>
</tr>
<tr>
<td>( A\cdot D )</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>( A\cdot D1.D2 )</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>( D\setminus A)B</td>
<td>---</td>
<td>( A\in \pp B )</td>
</tr>
</tbody>
</table>
TABLE 4.3.1

SCALAR OPERATOR BOUNDS

Let the bounds be

\[
\begin{array}{c}
[L_A; U_A] \\
[L_B; U_B]
\end{array}
\]

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>BOUNDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; \lor \neq \not \equiv \neq \geq \leq \rangle \langle \leq \rangle \neq |</td>
<td></td>
</tr>
<tr>
<td>\langle \leq \rangle \langle \leq \rangle \langle \leq \rangle \langle \leq \rangle \langle \leq \rangle</td>
<td></td>
</tr>
<tr>
<td>+A</td>
<td>[L_A; U_A]</td>
</tr>
<tr>
<td>A+B</td>
<td>[L_A+L_B; U_A+U_B]</td>
</tr>
<tr>
<td>-A</td>
<td>[-U_A; -L_A]</td>
</tr>
<tr>
<td>\times A</td>
<td>[-1;1]</td>
</tr>
<tr>
<td>\Lambda A</td>
<td>[0;\Omega]</td>
</tr>
<tr>
<td>\Lambda</td>
<td>B</td>
</tr>
<tr>
<td>?A</td>
<td>[0;A+1]</td>
</tr>
<tr>
<td>A?B</td>
<td>[0;B-1]</td>
</tr>
</tbody>
</table>
## TABLE 4.4
MODE CONSTRAINTS AND ATTRIBUTES

Modes are assumed to be ordered as follows:
\[ BOOL < NINT < AINT < REAL. \]

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>MODE OF RESULT</th>
<th>CONSTRAINTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \land A )</td>
<td>NINT</td>
<td>---</td>
</tr>
<tr>
<td>( \lor A )</td>
<td>MODE(A)</td>
<td>---</td>
</tr>
<tr>
<td>A B</td>
<td>MODE(A)( \land )MODE(B)</td>
<td>---</td>
</tr>
<tr>
<td>( \forall A )</td>
<td>NINT</td>
<td>NINT(A)</td>
</tr>
<tr>
<td>( A \setminus {B} )</td>
<td>MODE(B)</td>
<td>NINT(A)</td>
</tr>
<tr>
<td>A( \langle B; \ldots ; B N \rangle )</td>
<td>MODE(C)</td>
<td>BOOL(A)( \land )NINT(B)</td>
</tr>
<tr>
<td>A( \uparrow B )</td>
<td>MODE(B)</td>
<td>AINT(A)</td>
</tr>
<tr>
<td>A( \downarrow B )</td>
<td>MODE(B)</td>
<td>AINT(A)</td>
</tr>
<tr>
<td>A( \setminus \cup B )</td>
<td>MODE(C)</td>
<td>BOOL(A)( \land )NINT(B)</td>
</tr>
<tr>
<td>( \phi{A} )</td>
<td>MODE(B)</td>
<td>NINT(A)</td>
</tr>
<tr>
<td>A( \setminus \cap B )</td>
<td>MODE(C)</td>
<td>NINT(B)( \land )AINT(A)</td>
</tr>
<tr>
<td>A( \setminus \exists B )</td>
<td>MODE(B)</td>
<td>NINT(A)</td>
</tr>
<tr>
<td>A( \setminus \times B )</td>
<td>MODE(B)</td>
<td>---</td>
</tr>
<tr>
<td>A( \setminus B )</td>
<td>NINT</td>
<td>NINT(A)</td>
</tr>
<tr>
<td>A( \setminus B )</td>
<td>BOOL</td>
<td>---</td>
</tr>
<tr>
<td>( \delta A )</td>
<td>NINT</td>
<td>---</td>
</tr>
<tr>
<td>( \nabla A )</td>
<td>NINT</td>
<td>---</td>
</tr>
<tr>
<td>D( \left\langle{A}B \right\rangle )</td>
<td>SEE TABLE 4.4.1</td>
<td>NINT(A)</td>
</tr>
<tr>
<td>A( \setminus D B )</td>
<td>SEE TABLE 4.4.1</td>
<td>---</td>
</tr>
<tr>
<td>A( D1,D2 B )</td>
<td>SEE TABLE 4.4.1</td>
<td>---</td>
</tr>
<tr>
<td>D( \setminus A)B</td>
<td>SEE TABLE 4.4.1</td>
<td>NINT(A)</td>
</tr>
<tr>
<td></td>
<td>MONADIC</td>
<td>OPERATOR</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td>\textit{f A}</td>
<td>\textit{f}</td>
</tr>
<tr>
<td>\textit{MODE(A)}</td>
<td>+</td>
<td>\textit{MODE(A)} \textit{MODE(B)}</td>
</tr>
<tr>
<td>\textit{AINT MODE(A)}</td>
<td>-</td>
<td>\textit{AINT MODE(A)} \textit{MODE(B)}</td>
</tr>
<tr>
<td>\textit{AINT}</td>
<td>\times</td>
<td>IF \textit{MODE(A)}=\textit{BOOL} \lor \textit{MODE(B)}=\textit{BOOL} THEN BOOL ELSE \textit{MODE(A)} \textit{MODE(B)}</td>
</tr>
<tr>
<td>\textit{REAL}</td>
<td>+</td>
<td>\textit{REAL}</td>
</tr>
<tr>
<td>\textit{REAL}</td>
<td>\times</td>
<td>\textit{REAL}</td>
</tr>
<tr>
<td>\textit{REAL}</td>
<td>\bullet</td>
<td>\textit{REAL}</td>
</tr>
<tr>
<td>IF MODE(A) \texttimes NINT THEN \textit{MODE(A)} ELSE IF MODE(A) = AINT THEN NINT ELSE REAL</td>
<td>\textit{MODE(A)} \textit{MODE(B)}</td>
<td></td>
</tr>
<tr>
<td>IF MODE(A) \texttimes NINT THEN NINT ! ELSE REAL</td>
<td>NINT</td>
<td></td>
</tr>
<tr>
<td>IF MODE(A) \texttimes NINT THEN MODE(A) ELSE AINT</td>
<td>\textit{MODE(A)} \textit{MODE(B)}</td>
<td></td>
</tr>
<tr>
<td>IF MODE(A) \texttimes NINT THEN NINT ! ELSE NINT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textbf{BOOL}

\textbf{CONSTRAINTS}

\begin{align*}
\text{\&\&\&} & \text{ BOOL(A) \&\&\& BOOL(B)} \\
\text{\textasciitilde} & \text{ BOOL(A)} \\
! & \text{ NINT(A) \&\&\& NINT(B)}
\end{align*}
Discussion of the SLC algorithm and its implementation

The algorithm accomplishes a form of theorem proving in the sense that the constraints on operators can be proved consistent or inconsistent or reduced to problems more appropriate for another deductive system. The "theorem" that the system is proving is of the form

$$VC ⊢ C'$$

where $C$ is the set of constraints of the operators, $C'$ is a subset of $C$ and $VC$ is a verification condition which supplies the "context" in which the constraints occur. Actually the procedure will do much more: it finds the $C'$.

The system can be best described as the "evaluation" of the constraint expressions in an environment consisting of the symbolic forms of the attributes of the subexpressions of the verification condition being proved. The algorithm is able to operate in an environment where some attribute information is missing by making multiple passes using created names for some variable attributes not yet known. The output is a reduced set of constraints for which the system has no evaluation or theorem-proving skills. Examples of unresolvable constraints are sets of inequalities (possibly nonlinear) resulting from subscripts and equalities requiring induction proofs which arise from the expansion operator and from compatibility constraints.

The entire programmed system consists of the decomposition algorithm, built-in routines for the constraints as put forth in the
formal definition, some simple deduction rules derived from the definition, and simplification rules for some operators appearing in expressions.

The implemented system has numerous subroutines. Since the amount of detail in all these routines would be overwhelming the system is described in the following sections:

1) Decomposition algorithm
2) Major steps of the algorithm
3) General description of constraint routines and details of one routine
4) Expression simplification
5) Examples (Annotated)
6) Performance and system detail
7) Appendix A-Remaining constraint routines
8) Appendix B-More examples

Decomposition Algorithm (Preprocessor)

As was mentioned in Chapter 2, it is easier to verify APL programs when their expressions have been broken down to the single-operator level. The decomposition algorithm is given an expression and produces a tabular representation of the expression, having performed the change of variable discussed in Chapter 2. Each table entry consists of

"name" - current name of variable assigned to (concatenation of name and alteration counter)
"Subexpr" - variable of single-operator subexpression or OMEGA (if initial variable)

-------------------
*All references to fields in table entries are quoted.
EXAMPLE: \( N \oplus (M \neq 0) \wedge 1 \oplus M \)

<table>
<thead>
<tr>
<th>Name</th>
<th>Subexpr</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R \cdot 0 )</td>
<td>( M \cdot 0 \neq 0 )</td>
</tr>
<tr>
<td>( R \cdot 1 )</td>
<td>( R \cdot 0 \cdot 1 )</td>
</tr>
<tr>
<td>( N \cdot 1 )</td>
<td>( R \cdot 1 \land M \cdot 0 )</td>
</tr>
</tbody>
</table>

Entries are arranged in the order in which the subexpressions would be executed.

Details of the SLC Algorithm

PURPOSE: Given the output from the decomposition algorithm applied to a series of assignment statements, develop the reduced constraints which, if satisfied, guarantee that the statements will execute properly.

VARIABLES

\( OMEGA \) -denotes an unknown value
\( ET \) -expression table produced by the decomposition algorithm
\( AT \) -Attribute Table
"name"-same as corresponding entry in \( ET \)
"shape"-(partially) simplified expression from table 4.2.
"partial shape"-marking whether unit or linear or neither
"rank"-expression from Table 4.2
"mode"-of corresponding "name" (computed from table 4.4)
"bounds"-of corresponding "name" (computed from table 4.3)
"constraints"-unresolved constraints

Step 1: (pass 1) For each successive entry in \( ET \) create the entry in \( AT \)
"name"-same as \( ET \)
"shape"-simplification applied to the shape expression from table 4.2
"partial shape"- (see the discussions for unit and linear constraints)
"rank"-simplification of expression from table 4.2
"bounds"-set to \( OMEGA \)
"mode"-compute from table 4.4
"constraints"-result of test and application of each constraint

Step 2: (pass 2) For each entry in \( AT \)
-Simplify shape and rank expressions
-Recalculate mode and, if mode was constrained and determined mode does not satisfy that constraint then leave a constraint message to that effect.
-Compute bounds from table 4.4 and if constrained, make bounds obey that constraint
-Recalculate partial shape and check if shape satisfies any
constraints on partial shapes
-For each unresolved constraint, test if it is now satisfied.
Simplify those constraints with expressions, removing any that evaluate to true, and if any equality constraints are solved make the necessary shape constraints.

Step 3: If no new constraints were applied then quit.

Step 4: (pass 2+i)
Note: Only shape constraints could be applied in Step 2.
Repeat the actions of step 2 that have to do with shape. Simplify all bounds expressions. Simplify relations in unresolved constraints and apply any shape constraints resulting from solution of equality constraints. Return to step 3.

Comments on Routines used by SLC

Rather than give code or pseudo-code, the numerous system routines will be discussed in the context of individual constraints. For each constraint, it will be necessary to know how to REPRESENT the constraint, TEST if the constraint is satisfied, APPLY the constraint, COMPUTE the ATTRIBUTES associated with the constraint, SIMPLIFY the constraints and attributes, OBEY CONSTRAINTS that have been applied, DEDUCE some simple properties of the constraints and attributes, and justify the CORRECTNESS of the handling of the constraint. The discussions of each constraint are broken into the above categories. The unit constraint is given here as an example and the remaining constraints appear in Appendix A.

Several routines must work with incomplete information, e.g. with the shape of a scalar expression or with an unknown mode which is preserved. When this occurs, it is convenient to create a name which can be manipulated in place of that unknown attribute. These names will be pointed out in examples.
Note that constraints are entered directly into the AT in pass 1 whenever possible so that the attributes which are constrained are indistinguishable from attributes that were computed for the satisfaction of later constraints. However later passes recognize whether a constraint has been applied. Input variables are entered in the AT with created names for shape and rank expressions so that constraints can be applied to them. Many routines have simple deduction rules which are deduced from the definition whenever not immediate from the constraint definitions.

THE UNIT CONSTRAINT

REPRESENTATION- mark in "partial shape" entry in AT [1,019]**

TEST- satisfied if constant- scalar or one-element vector variable-marked as such in AT or has 0 as value of "rank" entry in AT [1,021] created name-denoting an unknown rank.

APPLY- if the test fails then Constant-error!
Variable-Mark in its "partial shape" entry in AT. [3,004,083] If the "shape" of the variable is the created name for the "shape" of another variable (i.e. its shape was preserved) then apply to that variable also. [5,004-005] If the variable is the result of a scalar expression then constrain each operand to be a unit. Created name-if name is r/l then constrain l=PP/l (see deduction U0 below). [2,011]

COMPUTE ATTRIBUTE- mark in "partial shape" entry if -0 is value of "rank" entry in AT [1,019] -"shape" expression is a vector of 1's [2,046] -"subexpr" is a scalar expression with only unit operands [4,005] -"shape" is preserved and the variable with preserved "shape" is a unit (see deduction rule U1) [5,090] [1,043]

SIMPLIFICATION - reduce "shape" expression to a vector of n 1's when "rank" is known to be n and the variable is constrained to be a unit. [2,010]

---

***Brackets refer to example (1) and line number (019) where the point is illustrated.
CONSTRAINT VERIFICATION
SLC ALGORITHM

OBEY CONSTRAINTS - If a variable is constrained to a unit, then check if the "shape" expression is constant and nonunit and report an error, if so. If the "rank" is known but the "shape" is unknown then perform the simplification.

DEDUCTIONS
U0: UNIT(pA) => l=pp/l
U1: UNIT(lA) \& (pA \leftrightarrow pB) => UNIT(B)
U2: (see discussion of compatibility)

CORRECTNESS - The unit attribute of a variable is acquired in three ways:
- constraint
- unity of "subexpr"ession
- unity of preserved shape

Conflict of constraint and attributes are detected in the second and successive passes when the "partial shape" attribute has been marked as constrained to a unit and the "shape" expression is checked for possible violation.

Unit constraints are applied only on the first pass and therefore used if needed. Unit attributes are computed at each pass so any unit constraint that can be satisfied will be.

EXPRESSION SIMPLIFICATION

Expressions occur in the contexts of equality relations for constrained shapes and in bounds expressions. Simplification occurs at the level of operators in many ways-identity operands are eliminated; scalar and one-element objects are identified; ELEM (the simple-indexing operation), and DELETE are applied to subexpressions: concatenations of units are converted to vectors; evaluation occurs wherever possible; and many others. The same scheme is used throughout all simplifications- an operator occurring in an expression generates a call to a routine which attempts to perform the operation, and failing that tries to simplify the operation, and then failing that returns the symbolic form of the call. For example, A,10 would generate a call to a routine for catenation which would recognize the
empty vector and return just \( \lambda \).

Other forms of simplification occur in expressions with scalar operators:

1. \( A-B \) is converted to \( A+B^{-1} \)  
\(-A \) is converted to \( A^{-1} \)

2. For commutative operators, the operands are normalized by ordering the operands according to occurrence in the symbol table with constants last. For example, \( 1+A+B \) might be normalized to \( A+B+1 \) or \( B+A+1 \) depending on when \( A \) and \( B \) were first referenced.

3. When two expressions are added, an attempt is made to combine terms or remove the term if coefficients add to 0.

4. Simplification is applied only to expressions which are assumed to be integer so relations of the form \( A \leq B \) can be normalized to \( A+1 \leq B \). Then the arithmetic simplification is invoked by converting \( A \leq B \) to \( 0 \leq B-A \).

\( 1 \leq A+B+1 \) is normalized to \( 2 \leq A+B+1 \) then to \( 0 \leq A+B+1-2 \) then simplified to \( 0 \leq A+B-1 \).
GUIDE TO EXAMPLES:

The examples appear with decreasing amount of annotation and with some parts missing (this is noted when it happens). The examples are numbered C1-C8 to distinguish them from the example ordering of Chapters 5 and Appendix D. As this is the direct output of the system on a teletype, some notation must be adopted. Operators have associated (mnemonic) 2-letter keywords preceded by a period, for which the translation table is given in Appendix E.

All examples have the form (with some parts omitted).

1. Output messages of the first pass actions
2. Table after first pass
3. Constraints after first pass
4. Output messages of second pass
5. Table after second pass
6. Constraints after second pass.

No example given here requires more than 2 passes.

Example C1 is the one-line left justification example discussed previously.

Example C2-Expand.text- illustrates several forms of constraints in a fairly simple program (the complete verification is not given).

Example C3-Outside Ace- is verified completely in Appendix D and illustrates the SLA procedure.

Example C4-Histogram- illustrates constraint application in a postassertion.

Example C5-Magic Square- illustrates mode constraints and transitive unitization.

Example C6-Quadratic equation- is a long one-liner in which a shape is computed.

Example C7-Hamming Codes- shows how two different forms of the same program come up with about the same constraints.

Example C8-Matrix Inverse- gives the BC procedure applied to a very large example.
Complete Annotations on program LEFT JUSTIFICATION

The program is \( N = ((M \neq 0) \& 1) \& M \).

001 Segment number-1 (and only)
002 reduce compatibility list to \( M.0 \) since 0 will not affect the shape of the result
003 "FROM" indicates the name of the entry producing the constraint
004-006 using the deduction \( \text{LINEAR}(R$.1) \land (P\text{R$.1} \lor M.0) \Rightarrow \text{LINEAR}(M.0) \)
007-024 Table after pass 1
   Each entry contains in order "name", "subexpr", "shape expression", "rank expression", "partial shape", "mode", and "constraints", though if any subentry has OMEGA as value then the field is not printed.
009 entry for \( M.0 \), an input variable, denoted by OMEGA as subexpr value
010 created names for shape and rank
011 linear constraint from 004 above
012 initial mode REAL indicates unconstrained mode
014 shape acquired through compatibility of \( M.0 \) with 0.
015 linear constraint from line 005 above
016 BOOL mode because operator is \#  
018 empty vector for shape of a scalar
019 unit since \( pR$.2 \lor P \)
020 NINT mode since operator is \( l \)

021 0 substituted for elided coordinate operand (computed from MONUS \( ppM.0 \)).
022 shape of \( M.0 \) preserved.
023 linear since \( M.0 \) is linear.
024 created name indicating mode preserved.
025-027 list of unresolved constraints
027 Only remaining constraint is on linearity of input variable \( M.0 \).
030 Actions during 2nd pass-none
031-049 Table after pass 2 actions
032-039 no change
040 bounds computed from operator \#.
041-044 no change
045 bounds computed from operator 1 according to table 4.4
046-048 no change
049 preserved mode of \( M.0 \) used.
051-052 unresolved constraints -same as for pass 1.
001 SEGMENT 1
002 COMPATIBILIZING M.Ø. TO M.Ø
003 FROM RS.1
004 LINEARIZED M.Ø
005 LINEARIZED RS.1
006 FROM RS.1
007 ~ SEGMENT 1 AT-ET
009 M.R = OMEGA
010 SHAPE: - RHO M.Ø RANK: - RHO M.Ø
011 LINEAR
012 MODE: REAL
013 RS.1 = M.Ø NE Ø
014 SHAPE: - RHO M.Ø RANK: - RHO M.Ø
015 LINEAR
016 MODE: BOOL
017 RS.2 = RS.1 .10 I.
018 SHAPE: - 10 Ø RANK: Ø
019 UNIT
020 MODE: INT
021 N.1 = RS.2 .RV [0.]M.Ø
022 SHAPE: - RHO M.Ø RANK: - RHO M.Ø
023 LINEAR
024 MODE: MODE M.Ø
025
026 CONSTRAINTS
027 LINEAR M.Ø
028
029 END OF PASS 1
030
031 SEGMENT 1 AT-ET
033 M.R = OMEGA
034 SHAPE: - RHO M.Ø RANK: - RHO M.Ø
035 LINEAR
036 MODE: REAL
037 RS.1 = M.Ø NE Ø
038 SHAPE: - RHO M.Ø RANK: - RHO M.Ø
039 LINEAR
040 MODE: BOOL
041 BOUNDS: [-1.1]
042 RS.2 = RS.1 .10 I.
043 SHAPE: - 10 Ø RANK: Ø
044 UNIT
045 MODE: INT
046 BOUNDS: [-1.1 RHO M.Ø]
047 N.1 = RS.2 .RV [0.]M.Ø
048 SHAPE: - RHO M.Ø RANK: - RHO M.Ø
049 MODE: REAL
050
051 CONSTRAINTS
052 LINEAR M.Ø
053
054 END OF PASS 2
055

CONSTR.AINT VERIFICATION
EXAMPLE OF LEFT JUSTIFICATION
EXPLANATION: Assume that \( \text{STAR} \) is a unit and that \( \text{TEXT} \) is a vector. \( \text{EXPANDTEXT} \) inserts \( \text{STAR} \) between every adjacent pair of elements of \( \text{TEXT} \). The program is

\[
T\{1+2^{-1}\cdot(1+2^0\cdot\text{TEXT})\cdot p1.0\}\text{TEXT}
\]

For example, if

\( \text{TEXT} \leftarrow 0 \ 5 \ 9 \ 4 \ 2 \) and \( \text{STAR} \leftarrow 7 \)

After the program is executed \( T \leftarrow 0 \ 7 \ 5 \ 7 \ 9 \ 7 \ 4 \ 7 \ 2 \)

Partial annotation (covering features not occurring in example 1.)

004 constraint from \( \text{RESHAPE} \) operator. Note that the linearity constraint was satisfied.

010-013 deduction \( \text{UNIT}(p\text{TEXT}0) \Rightarrow l=p\text{TEXT}0 \)

Simplification \( \text{UNIT}(p\text{TEXT}0) \Rightarrow 1 \Rightarrow p\text{TEXT}0 \)

016-017 constraints from subscripting -bounds and mode

028 mode constrained to nint

033 \( \text{MIPUS} \ p1 \) is an abbreviation for \( 0^{-1}+p1 \) and has an associated simplification routine.

034 symbolic form of \( \text{SUBST} \) since \( p\text{TEXT}0 \) had not yet been constrained when the expression was formed. It will be simplified on the second pass.

047 mode constrained to NINT

056 bounds constraint

060 shape compatibility of subscripts and assigned value \( S.C_{\text{COMPATIBLE}}(p\text{R}.7,p\text{R}.4) \Rightarrow (1=+/p\text{R}.4) \lor (p\text{R}.7=+/p\text{R}.4) \)

062-066 unresolved constraints

064 constraint on input variable is equivalent to

\( (+/p\text{R}.2) = (p\text{TEXT}0)(0^{-1}+p\text{TEXT}0) \)

069-075 Actions during 2nd pass

069 semi-bounds computed from mode constraint to NINT

071 semi-bounds computed from constraint to NINT

073 simplification of expressions in bounds

Constraint was originally \( 0;ELEM(0;P1.1) \neq -1 \) which simplified to \( 0;ELEM(0;P1.1) \neq -1 \) since \( l=++T.1 \) and then simplified to \( 0;ELEM(0;P1.1) \neq -1 \) using the fact that \( pT.1=+/pR.1 \Rightarrow (2-2\cdot p\text{TEXT}0)-1 \). The bounds computed for \( pR.6 \) were \( 0;ELEM(0;P1.1) \neq -2 \) and for \( 2\cdot pR.6 \) the bounds were \( 0;ELEM(0;P1.1) \neq -4 \) and for \( 1+2\cdot pR.6 \) the bounds were \( 0;ELEM(0;P1.1) \neq -3 \). The latter bounds are "tighter" since \( 0 \leq l \) and \( (2\cdot p\text{TEXT}0)^{+3} < (2\cdot p\text{TEXT}0)^{+1} \).

082 and 089 note how the expression was normalized

108 unresolved mode constraint

126-135 unresolved constraints

Two different kinds of processing are required to resolve the constraints further:

\( (+/pR.2) = p\text{TEXT}0 \) states that the length of the vector \( \text{TEXT}0 \) equals the number of 1's in the expression \( ^{+1} + 2 \cdot p\text{TEXT}0 \cdot p1.0 \) and would require an induction proof, the techniques for which are discussed in Chapter 5. \( l=++\text{TEXT}0 \) is a constraint on the input. \( S.C_{\text{COMPATIBLE}}(...) \) is also a constraint on the input. Both input constraints are what would be expected from the stated purpose of the program. The semi-bounds can be stated as \( (p\text{TEXT}0)^{+2} \) and \( (p\text{TEXT}0)^{+1} \) and obviously the second implies the first although the system doesn't have that kind of information built in.
CONSTRAINT VERIFICATION
EXAMPLE C2-EXPANDTEXT

061 SEGMENT 1
062 COMPATIALIZING (-1.2) RHO TRUE TEXT.8 TO "RHO TEXT.8"
063 FROM R1.1
064 MODE OF R1.1 TO NINT
065 FROM R1.2
066 LINEARIZED R1.2
067 FROM R1.3
068 COMPATIALIZING (-1.) RHO TRUE TEXT.8 TO "RHO TEXT.8"
069 FROM R1.5
070 SHAPE OF R1.5 TO 1.
071 RANK OF "RHO TEXT.8" TO 1.
072 MODE OF R1.5 TO NINT
073 FROM R1.6
074 COMPATIALIZING ... R1.6 TO R1.6
075 FROM R1.7
076 MODE OF R1.7 TO NINT
077 BOUNDS OF R1.7 TO { detained ( (B.) RHO T.1 )=(-1.)}
078 FROM T.2 (R1.7 )
079
080 SEGMENT 2: AT-IT
081 TEXT.8 - OMEGA
082 SHAPE: RHO TRUE TEXT.8 RANK: 1.
083 LINEAR
084 MODE: REAL
085 R1.1 = (-1.3)R1.2 RHO TRUE TEXT.8
086 SHAPE: "RHO TRUE TEXT.8" RANK: 1.
087 LINEAR
088 MODE: NINT
089 R1.2 = R1.1 RHO TRUE 1.
090 SHAPE: R1.1 RANK: "RHO R1.1"
091 LINEAR
092 MODE: HINO
093 T.1 = R1.2 (MONUS "RHO TRUE TEXT.8") TEXT.8
094 SHAPE: SUBST (MONUS "RHO TRUE TEXT.8") "RHO TEXT.8" "RHO R1.2"
RANK: "RHO TRUE TEXT.8"
095 MODE: TRUE TEXT.8
096 CONSTRANTS: R-=(/= (0.,3) R1.2 ) (DETAINED (MONUS "RHO TRUE TEXT.8") "RHO TEXT.8") (-1.)
097
098 STAR.8 - OMEGA
099 SHAPE: "RHO STAR.8" RANK: "RHO TRUE STAR.8"
100 MODE: REAL
101 R1.4 = STAR.8
102 SHAPE: "RHO STAR.8" RANK: "RHO TRUE STAR.8"
103 MODE: MODE STAR.8
104 R1.5 = (-1.) RHO TRUE TEXT.8
105 SHAPE: 1. RANK: 1.
106 UNIT
107 MODE: NINT
108 R1.6 = (-10 R1.5
109 SHAPE: R1.5 RANK: 1.
110 LINEAR
111 MODE: NINT
112 R1.7 = 1.; R1.6
113 SHAPE: "RHO R1.6" RANK: 1.
114 LINEAR
115 MODE: NINT
116 BOUNDS TO ( detained ( (A.) RHO T.1 )=(-1.)
117 T.2 (R1.7 ) = R1.4
118 SHAPE: "RHO T.1" RANK: "RHO TRUE TEXT.8"
119 MODE: TRUE T.1
120 CONSTRANTS: 5. COMPATIBLE ((-RHO R1.7 ) "RHO R1.4")
121
122 CONSTRANTS
123 TEXT.8 RANK TO 1.
124 R-=(/= (0.,3) R1.2 ) (DETAINED (MONUS "RHO TRUE TEXT.8") "RHO TRUE TEXT.8") (-1.)
125 5. COMPATIBLE ((-RHO R1.7 ) "RHO R1.4")
126
127 EU OF PASS 1
CONSTRAINT VERIFICATION
EXAMPLE C2-EXPANDTEXT

847 BINDING (.RHO TEXT.0)@, TO (1.,OMEGA )
848 FROM Rs.1
849 BINDING (.RHO TEXT.0) TO (1.,OMEGA )
850 FROM Rs.5
851 TIGHTENED BOUNDS ((.RHO TEXT.0)@,(@-2))
852 SATISFIED MODE Rs.7
853 FROM Rs.7
854 SEGMNT 1 AT-ET
855 TEXT.O = OMEGA
856 SHAPE: (.RHO TEXT.0) RANK: 1.
857 MODEL: REAL
858 S:R.1 - (-1.)@,.RHO TEXT.0
859 SHAPE: .J.
860 RANK: 1.
861 LINEAR
862 MODE: MINT
863 CONSTRAINTS: MODE Rs.1 TO MINT
864 S:R.2 - Rs.1,.RHO OMEGA
865 SHAPE: ((.RHO TEXT.0)@,-(-1)) RANK: 1.
866 LINEAR
867 MODEL: BOOL
868 BOUNDS: (8.,1.)
869 T.1 - Rs.2 (.RHO TEXT.0)
870 SHAPE: ((.RHO TEXT.0)@,-(-1)) RANK: 1.
871 MODEL: REAL
872 CONSTRAINTS: S:=(/=S)Rs.2补偿 ((.RHO TEXT.0)@,-(-1))
873 STAR.0 - OMEGA
874 SHAPE: (.RHO STAR.0) RANK: .RHO STAR.0
875 MODEL: REAL
876 Rs.4 = STAR.0
877 SHAPE: (.RHO STAR.0) RANK: .RHO STAR.0
878 MODEL: REAL
879 Rs.5 - (-1.)*RHO TEXT.0
880 SHAPE: .J.
881 RANK: 1.
882 UNIT
883 MODEL: MINT
884 CONSTRAINTS: MODE Rs.5 TO MINT
885 S:R.6 - .10 Rs.5
886 SHAPE: (.RHO TEXT.0)@,(-1)
887 linear
888 MODEL: MINT
889 CONSTRAINTS: MODE Rs.5 TO MINT
890 S:R.7 - Rs.6,.RHO Rs.6
891 SHAPE: (.RHO TEXT.0)@,(-1)
892 MODEL: REAL
893 CONSTRAINTS: .S.COMPATIBLE ((.RHO Rs.7),.RHO Rs.4 )
894
895 CONSTRAINTS
896 TEXT.O RANK TO 1.
897 MODE Rs.1 TO MINT
898 S:=(/=S)Rs.2补偿 ((.RHO TEXT.0)@,(=-2))
899 MODEL: REAL
900 CONSTRAINTS: S:COMPATIBLE ((.RHO Rs.7),.RHO Rs.4 )
901 SEMI-BOUNDS
902 S:=(/=S)Rs.2 (.RHO TEXT.0)@, (1.,OMEGA )
903 (.RHO TEXT.0)@, (1.,OMEGA )
904 END OF PASS 2
THE SLA ALGORITHM

The goal is to produce reduced constraints as in SLC, except that assertions will be associated with the control segment. As was mentioned in Chapter 2, assertions are written in APL and an assertion can be true only if it is executable. Since an assertion is an APL expression, its constraints can be formed by the SLC process. Those constraints are considered as implicit in the assertion and they must be satisfied for the assertion to be true.

But an assertion also involves predicates about the program which must be interpreted. We are concerned only with predicates about the shape, mode and bounds of variables so there must be a way of extracting those predicates from the expression. The predicates must also have a form which can be processed by the deductive system, i.e. each predicate must be translatable into the form of one of the constraints that appears in the definition. Several syntactic restrictions have been adopted:

<table>
<thead>
<tr>
<th>FORM IN ASSERTION</th>
<th>CORRESPONDING CONSTRAINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT X</td>
<td>UNIT</td>
</tr>
<tr>
<td>LINEAR X</td>
<td>LINEAR</td>
</tr>
<tr>
<td>X EQS E</td>
<td>SHAPE</td>
</tr>
<tr>
<td>X = E</td>
<td>SHAPE</td>
</tr>
<tr>
<td>X &lt; E</td>
<td>BOUNDS</td>
</tr>
<tr>
<td>X &gt; E</td>
<td>LOWER BOUNDS</td>
</tr>
<tr>
<td>X &lt;= E</td>
<td>LOWER BOUNDS</td>
</tr>
<tr>
<td>X &gt;= E</td>
<td>UPPER BOUNDS</td>
</tr>
<tr>
<td>X UB E</td>
<td>UPPER BOUNDS</td>
</tr>
<tr>
<td>NINT X</td>
<td>MODE</td>
</tr>
<tr>
<td>BOOL X</td>
<td>MODE</td>
</tr>
<tr>
<td>AINT X</td>
<td>MODE</td>
</tr>
</tbody>
</table>

where E is an expression and where X may be a variable name or an expression of the form, pY or ppY where Y is a variable.
UNIT, LINEAR, NINT, BOOL, MINT are written as monadic functions and EQU, LB, UB as dyadic functions, all of which are assumed to produce a boolean value.

The syntactic restrictions given above are severe but sufficient to demonstrate the system. However, the restriction to conjunctive form would have to be removed in practice since its effect is to force the input-output relation to be broken into cases. It also makes it impossible to process the negation of some branch conditions.

Steps of the SLA algorithm

Given a control segment, with decomposed expressions
1. Perform SLC on the preassertion
2. Extract and apply the constraints of the preassertion
3. Process program statements
   3a) For assignment statements, perform SLC.
   3b) For branches perform SLC, then constraint extraction and application.
3. Process program statements
4. Perform SLC on the postassertion, then apply its extracted constraints

PROCEDURES BA AND BC

The remaining two procedures involve programs with branches.
Their purpose was described in the outline of the method. In the case of a program with assertions, the verification conditions can be formed directly from the control segments. For programs with no assertions, every merge can be designated as a place for the assertion TRUE. Then control segments can be formed. Since every loop must have a merge, the conditions for the inductive assertion method are satisfied.
Branches are treated as expressions just like assertions for extraction purposes. However, an inconsistency detected in applying an extracted constraint does not mean that the verification condition is false, only that the control segment is inconsistent.

It is then possible to apply SLA to every verification condition to obtain the reduced constraints for each. If all constraints are verified, then it remains only to prove the remaining parts of the assertions. But SLA only affects individual verification conditions not the program as a whole. Several possibilities exist for procedures which will further assist in the verification:

1. Try to combine verification conditions along control paths (joining)
2. Try to combine constraints on initial variables of all verification conditions with the same preassertion and of those with the same postassertion. (merging)
3. Try to detect what is constant throughout the program or loops. (invariances)
4. Develop constraints on function calls

To some extent all are possible mechanically, and indeed, are used by humans in their creation of assertions.

All three possibilities fail within the context of the system proposed here for the same reason: it is impossible to detect inconsistent control segments either through extracting from branches or proving the inconsistency of extracted constraints. The implemented deductive system only handles constraints. A more complete but informal and unimplemented deduction system will be discussed in Chapter 5. An example was given in Chapter 2 where the
verification condition was inconsistent because $\vee M \neq 0$ and $\wedge M = 0$
occurred in conjunction. This could not be recognized by the system
described here. Even if this were not the case, all possibilities
would suffer to some extent from combinatorial difficulties because
there are too many possible paths and constraints, especially if
nested loops were occur.

On the other hand, the human verifier probably has the overall
view of the program required to develop the assertions further.
Therefore, why not ask him to select from the constraints developed by
the system those that should be in the assertions? The implemented
system accomplishes the SLC and SLA procedures. These could be used
in the development of an interactive system (which has not been
implemented) for dealing with all forms of APL programs. There are
several advantages to this approach:

1. The program may be wrong in the first place and it would be
best to abort at this early point. The feedback of preconditions may
point out something the programmer had overlooked or simply that a
typing error had been made. Unfortunately, in practice, verifications
would have to be debugged just as programs are.

2. With the information available the user may be better able to
make assertions, e.g. in selecting places where the assertion will be
simplest or in selecting constraints. Having even some information
about the state of the variables at merges or assertion points is much
better than none.
3. The user will not have to go through the tedious and error-prone task of entering complicated predicates since the system can print out a constraint and wait for his response as to selection of the constraint.

4. The left justification program with a loop of Chapter 2 was an example where it was necessary to preserve the initial predicate in the inductive assertion. This can easily be done here, again by simply selecting constraints.

The disadvantages are that there may be many constraints to be understood (but then how would the inductive assertions have been developed in the first place?).

Each of these possibilities will be discussed in more detail.

1. Joining two verification conditions

If it is assumed that verification conditions are in the form accomplished by the first pass of SLA, then it is possible to change the verification condition without harm. The biggest problem is renaming variables, but the underlying implementation uses pointers into tables so this can be accomplished. It only remains then to combine the constraints on initial variables of the second verification condition with attributes of the first verification.

The second pass of SLA can then be performed with the effect being the same as if the statements of both verification conditions had been in the same control segment.

2. Merging two verification conditions.
The constraints on the initial variables of one verification condition can easily be applied to another, retaining the capability of detecting inconsistencies. However it is not as easy to merge constraints on intermediate variables since the naming may be different.

3. Invariance detection

It is quite reasonable to expect that some variables would not change their attributes very much (e.g. loop control variables and input variables). Therefore, one routine could inspect all variables of control segments entering a node and determine whether a particular variable has the same attribute in all segments. If so, that attribute can be "fixed" as part of the assertion and all verification conditions related to that assertion can be simplified with respect to that attribute. Since there may be many attributes and variables this could be highly productive. Note that the routine might need to join two verification conditions in order to obtain attributes of a variable (in particular, control segments of a loop).

4. Function calls

The basic procedure can be used to develop partial assertions on functions by regarding every call on a function like an input variable. The constraints for all calls are then conjoined into a predicate which must be implied by the output predicate of the function. Working backward through the function using that generated predicate might strengthen the function's input predicate. If the input predicate for the function has been developed, then the relevant constraints can be applied to the actual parameters of the function.
call. The basic difference between functions and operators is simply that operator constraints are fixed whereas function constraints must be separately developed and integrated into its calling functions.

Given the above tools, the user experienced in interactive programming would probably have little difficulty selecting an assertion.

CODING METHODS AND PERFORMANCE

The system is written in PPL[ST], an extensible, interactive language developed at Harvard for the PDP-10. The system consists of the following parts:

1. Syntax analyzer
2. Output conversion and print routines
3. Decomposition algorithm
4. Constraint verifier

The structure of the decomposition algorithm and constraint verifier is approximately as described in this chapter. The routines named in the description have corresponding routines in the implementation, although the correspondence may be one to many, especially in the case of simplification of expressions. Altogether there are approximately 200 functions, averaging about 10 lines each.

ET and AT are treated as one large table. A symbol table contains strings and pairs of integers used to represent created and generated names. A parallel table contains pointers into the ET-AT table.

The lexical and syntax analyzer was developed during the earlier
work on the formal definition in VDL and was recoded in PPL. The syntax analyser is recursive.

PPL has extensibility of data structures and operators. Few operators were defined in this system, however the data structure definition facility was exceedingly useful. The table entries and expressions were represented as structures with field names as described here. The tables were variadic sequences, in PPL's parlance. Program constants were represented using the built-in data types: pointers were represented as PPL integers, constants as PPL reals, and strings as PPL strings. PPL has built-in tests for types of values and the tests made it possible to write many routines almost like a case statement with a series of tests for type followed by the appropriate action for the test.

The running performance of the system was astounding, in a negative sense. PPL is interpretive and the system ran slow enough to make the author one of the year's top ten users of the C-MU PDP-10. But the value of a language like PPL must be measured in other terms than speed:

1. PPL, like APL, is a system with its own editor and file storage commands, making it possible to work entirely within PPL.

2. PPL, like APL, also has trace and stop commands for debugging. Little output conversion is necessary to print intermediate results.

3. PPL is a remarkably stable system. Only one system error occurred in the eight months of development and that was due to inadvertent erasure of the assignment operator by the author.

4. The variability of data structures, heterogeneous lists for example, is immensely useful.

5. The system's slow performance was tolerable as long as the PDP-10 was lightly loaded.
PPL was chosen over APL for its data structure capabilities. Since the system was intended only for experimentation by a single user (the author) debugging efficiency was far more important than execution efficiency, determining the choice over a language like BLISS. Other alternatives would be SAIL (but not interactive enough) and LISP (too restrictive in syntax and data structures).

Though PPL was never intended to be used in the way this system demanded, the author feels that languages like PPL can be a great boon in developing prototype, single-user systems like the one described here. It is interesting to compare the approach used here where a highly inefficient system was developed and experimented with with that of King's[K11]. He mentions that most theorems can be proved within 10 seconds but that it took a month to excise the lexical and syntactic "front-end" of the IBM 360/67 ALGOL compiler and adapt it for use in his verifier. Inefficiency of performance is often preferable to long development periods, especially in the scope of a thesis.

Rough Statistics on the System

Space 60K program+1 example
Time to translate and load the system-7 minutes (but PPL also has a command to restore an entire environment).
One-liner examples given here
  Lexical and syntax analysis-25 seconds-5 minutes
  Constraint verification-30 seconds-10 minutes

As can be seen from the examples, almost all constraints are removed or reduced in correct programs. The remaining constraints clearly require distinctly different proof techniques such as solution of systems of inequalities or the mathematical properties of the
scalar operators. The proportion of unreduced to reduced constraints usually decreases with increasing number of operators because the operators are highly redundant.

SUMMARY AND CONCLUSIONS

A collection of algorithms for verifying that the constraints on operations are satisfied and that assertions about shape and mode are true have been presented. The basic algorithms, SLC and SLA, form part of the deductive system of the APL verifier. The algorithms for programs with branches, BA and BC, form verification conditions which the user could manipulate interactively.

It is interesting to consider again the form of this "deductive system". The algorithm has certain features—extensive bookkeeping, multiple passes, built-in rules (some derived) from the definition, and simplification—which obviate the need for ever making long chains of deductions. The verification task was analyzed to the extent of establishing the appropriate times for making certain simple steps (constraint application and attribute computation). The appropriate time for performing a step is when the necessary information is available to perform the step in the simplest way, when the effect of the step can be used safely, and when the information provided by the step can be used for performing other steps. This seems to summarize the essence of a specialized deductive system: The task at hand is analyzed to determine the necessary and sufficient information to support a temporal order of events. A general-purpose system would be faced with performing a large number of useless steps and maintaining
The method described here also has the interesting feature of requiring very little of the user. In this restricted sense, the system can generate assertions. The method was developed in response to a particular language feature, undeclared attributes and partial operators, but it does illustrate that it is possible to extract an immense amount of useful information from a program. Perhaps a better term than verification can be applied to the process described here—"semantic checking", the capability of detecting the presence or absence of certain semantic errors in a program. This point will be discussed further in the conclusions to the entire thesis.

The success of the described techniques decreases with the amount of control flow in a program, but it has been pointed out how an interactive verifier based on the SLC algorithm would be useful.

One should keep in mind, regarding the verification techniques of this chapter and the next, that if the programs used as examples were written in languages lacking APL's operators an immense amount more of work would be involved, both in writing and in verifying the programs.
CHAPTER 5
PROVING ASSERTIONS ABOUT APL PROGRAMS

INTRODUCTION

One of the purposes of studying the verification of APL programs was to determine how the APL operators influence the difficulty and nature of stating and proving properties of APL programs. The formal definition of the operators provided a succinct and uniform description of the more complicated operators and it was suggested that certain useful properties of those operators could be derived from the definition. In this chapter, we describe some informal (nonmechanical) techniques for proving general assertions about APL programs and then indicate some steps toward mechanization of those techniques. But first, it must be determined what characteristics are expected of the assertion language for stating the properties of APL programs.

APL AS AN ASSERTION LANGUAGE

To a large extent, the verification of programs is determined by the nature of the assertion language, both in terms of the range and ease of stating the assertions and of the ensuing proof. The question was discussed in Chapter 2 and APL was suggested as an assertion language. Very little consideration of this question has appeared in the literature, primarily because the state of the art is at the level of informal treatment so one is free to state anything that one can
prove. However, APL provides a good framework for examining the question in more depth.

London[LO4] suggests that most assertions will fall into the patterns

\[
\text{EXPRESSION RELATION OPERATOR EXPRESSION}
\]

joined by the logical connectives. The operator might be sigma, pi, or, and, max, or min and the relation could be either a set or an arithmetic relation. The operator could be bounded or modified. Interestingly enough, APL has all of the above operators and relations plus several others. Of these, the bounded form of an operator is of most concern. The reduction operator supplies the basic mechanism in that the "bounds" are the subscripts of the value being reduced (given implicitly since there is no subscript variable). For example, where \( \lambda \) is a vector

\[
S = +/\lambda
\]

is the same as

\[
\text{LENGTH}(\lambda)-1 \quad \sum_{I=0}^{\text{LENGTH}(\lambda)-1} \lambda[I]
\]

and

\[
S = +/J\uparrow\lambda
\]

is the same as

\[
J-1 \quad \sum_{I=0}^{J-1} \lambda[I]
\]

Since the "bounds" in the APL reduction operator are implicit, the question arises: Is there ever a need for explicit variables in the use of a bounded operator? One case is when a form of bounded quantification is required. Suppose one wants to assert that in a vector \( \lambda \) each element occurs only once. In APL this might be stated in several ways:
a) \( \land / (A \land I) = \land P / A \) states that "for subscript \( I \) of \( A \) the first occurrence of \( A[I] \) in \( A \), i.e. \( (A \land A[I]) \) is at \( I \) and if every occurrence is the first then the uniqueness property holds".

b) \( \land / I = + / A \cdot = A \) states that "the number of occurrences of \( A[I] \) in \( A \), i.e. \( S \geq (+ / A \cdot = A)[I] \cdot + / A[I] = A \) is 1 for each element, i.e. \( I = S \) ".

c) \( (X \sqsubset A) \rightarrow I = + / X = A \) states "any element of \( A \), say \( X \), occurs once and only once in \( A \) ".

But there are further ways of stating this property which might be useful.

d) \( \forall (X \sqsubset A) \rightarrow I = + / X = A \) is related to version c) with the quantifier \( \forall \) and quantified variable \( X \) and domain of quantification \( A \) all given explicitly.

e) \( \forall (V \sqsubset A)^{r} \land (V^{\prime} \sqsubset A)^{r} \vee A[V] \neq A[V^{\prime}] \) has two quantified variables \( V \) and \( V^{\prime} \), both quantified over the domain of subscripts \( \sqsubset A \) of \( A \). This states that "for every pair of elements of \( A \), \( A[V] \) and \( A[V^{\prime}] \), either the indices are identical \( (V = V^{\prime}) \) or the elements are different \( (A[V] \neq A[V^{\prime}]) \) ".

Now consider more complicated variations of the same statement, e.g. that in a matrix, \( M \), no two rows are identical. The quantified form seems most useful:

\[
\forall (I \sqsubset (\rho M)[0]) \vee (J \sqsubset (\rho M)[0]) \exists (K \sqsubset (\rho M)[I]) M[I;K] \neq M[J;K]
\]

Therefore, it is proposed that assertions may contain these bounded quantifiers with the syntax \( Q[V \sqsubset B] E \) where

- \( Q \) is \( \exists \) or \( \forall \)
- \( V \) is a variable to be treated as having a scalar shape
- \( \sqsubset \) is the APL operator
- \( B \) is an APL expression denoting the domain of \( V \)
- \( E \) is any APL expression.

The quantifiers will behave as one might expect- \( \exists \) as a finite disjunction and \( \forall \) as a finite conjunction. The operators would be forced to obey a number of standard rules of predicate calculus, such as scope and change of bound variables and conversion to prenex form. See Mendelson[ME] or any other standard logic book for the precise statement of these rules.
Is there ever a need for unbounded quantification? The domain of input variables is tacitly universally quantified over real numbers (or characters if that simplification is removed). There appears to be no occasion in program verification where one would want to make assertions about anything except the values of the program variables and quantification over subscripts should suffice for that.

One problem with APL as an assertion language was discussed in Chapter 2 where the partial operators were dealt with in forming verification conditions. The need to make assertions when the rank of variables is not known can present problems. The simple-indexing operator described in Chapter 3 presents a possible way out since simple-indexing can be used throughout a quantified expression with the quantified variable being a subscript vector rather than a scalar. The same syntax for quantification can be used perhaps with the extension of meaning of \( \varepsilon \) so that subscript vectors can be used. Abrams[AB] suggests the following operator definitions (denoted here by names rather than symbols):

- **ODOMETER(\( A \))** where \( A \) is a vector of nonnegative integers, yields a matrix \( M \) of \( \times / A \) rows and \( \rho A \) columns where \( M[I;] \leftrightarrow (A)^I \).

- **\( A \rel E L T B \)**, where \( A \) is a vector and \( B \) is a matrix with \( \rho A \) columns, yields 1 if \( A \) is a row of \( B \) otherwise 0.

Then one might write \( \forall[V \rel E L T \text{ODOMETER } \rho A] E \) to denote that \( V \) ranges over all possible subscripts for \( A \).

Another operator which might be useful is the if-then operator (denoted by \( \Rightarrow \)) defined as

\[ \text{IF } A \Rightarrow 1 \text{ THEN } B \text{ ELSE TRUE} \]

so that \( B \) must be evaluated only if \( A \) is true. An example might be
in an assertion of the form

\[(\theta=\mathbb{P}(A) \Rightarrow A=1) \land ((1=\mathbb{P}(A) \Rightarrow A[0]=1)\]

Since there are usually ways around this problem it is not clear where \( \Rightarrow \) is really necessary in practice.

Experience has shown that APL extended by bounded quantification has the necessary operators to make succinct, readable assertions. It is not uncommon to have assertions that are larger than the program on which they are used. Of course, no prohibitions are placed on using in assertions any other operators than the ones just mentioned since verification is really the proof of consistency of assertions and program code.

Two problems do arise in the use of APL, or any other language, for assertions. First, as London[LO4] mentions, it is useful to "parameterize" assertions when a single assertion pattern is frequently used. In effect, this means having a macro facility within the assertion language. Second, there are no provisions for declaring functions and then using function calls in assertions. An example will be given in appendix D where it appears impossible to state the correctness of a program without using a recursively defined function. It is interesting that this will lead to a proof method that appears better than inductive assertions for that particular problem.

THE APL DEDUCTIVE SYSTEM

It was beyond the scope of this thesis to develop the complete deductive system for APL verification. The proof techniques that will be discussed evolved during the experience of proving (manually) that
several APL programs were correct. These techniques certainly must appear in any deductive system for APL. Since the purpose of this chapter is to sketch the basis for a deductive system and then to illustrate it in examples, we will simply stop at the point where sufficient justifications for individual steps of the proofs can be enumerated and assume that these steps can be further formalized. See Table 5.1 for a list of the forms of deductive steps. Standard logic and number theory are assumed and the special forms of steps will be discussed separately.

The identity relation $\leftrightarrow$ discussed earlier will frequently be used with the intention that the identity follows from the basic rules of the deductive system. $=$ (the APL scalar operator) is sometimes used as a special case of $\leftrightarrow$ where both expressions are restricted to evaluate as scalars. $\leftrightarrow$ is retained to denote the use of definition.

The type of proof differs markedly from others in the literature. In King[KI], substitution of assignments into assertions in forming verification conditions led to large formulae to which was applied a collection of specific routines for simplification and determining inconsistency of equations. The formalism for developing verification conditions for APL uses conjunction, not substitution, for reasons discussed in Chapter 2. The proofs given here make extensive use of matching expressions for the semantics of the operators resulting in generation of subgoals requiring lemmas for specific operator properties. Overall, most proofs have the form of standard mathematical proofs.
### TABLE 5.1
APL DEDUCTION SYSTEM

<table>
<thead>
<tr>
<th>TYPE OF STEP</th>
<th>DISCUSSION</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>Chapter 3</td>
<td>$R \leftarrow A \land 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$p R \leftrightarrow p 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$R \leftrightarrow { (B=(A,B))/\lor 1 \land p, A }$</td>
</tr>
<tr>
<td>Quantification</td>
<td>Chapter 5</td>
<td>$\land /A \equiv \forall [V \in p/A] A[V]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lor /A \equiv \exists [V \in p/A] A[V]$</td>
</tr>
<tr>
<td>Logic</td>
<td>Any standard logic book</td>
<td>$\forall V ( \neg A) \land B \leftrightarrow A \lor B$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\forall [V \in E1] \forall [V \in E2] E \leftrightarrow $</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\forall [V \in E2] \forall [V \in E1] E$</td>
</tr>
<tr>
<td>Destructuring</td>
<td>Chapter 5</td>
<td>$(X= A) \land B { V } \leftrightarrow$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(X{ V }= A{ V }) \land B{V }$</td>
</tr>
<tr>
<td>Induction</td>
<td>Chapter 5</td>
<td>Proof for $(\land /A \equiv 0, 1) \supset (+/A) \in p/A$</td>
</tr>
<tr>
<td>Number theory</td>
<td>Standard mathematics</td>
<td>Proof for $(R \in M) \supset (R</td>
</tr>
<tr>
<td>Derived property</td>
<td>Chapter 5, Table 5.2</td>
<td>$(X \in A/B) \supset \lor/(X=B) \land A$</td>
</tr>
<tr>
<td></td>
<td>Appendix C (Proofs)</td>
<td></td>
</tr>
<tr>
<td>Lemma</td>
<td>Proof may or may not</td>
<td>$(\land /A = 0) \supset 0 =+/A$</td>
</tr>
<tr>
<td></td>
<td>be given with its appearance</td>
<td></td>
</tr>
<tr>
<td>Case analysis</td>
<td>Chapter 5</td>
<td>$IF \ A(I)=0 THEN T(I)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$ELSE A(I) \lor (I)$</td>
</tr>
<tr>
<td>Convention</td>
<td>Chapter 3</td>
<td>$WHERE (I= p, A) \land (I=p, B)$</td>
</tr>
<tr>
<td></td>
<td>Appendix E</td>
<td>$A=B \supset A \leftrightarrow B$</td>
</tr>
</tbody>
</table>
Definition

The shape and element expressions given in the formal definition of Chapter 3 appear as proper axioms in the deductive system. In forming verification conditions, an assignment statement \( V \leftarrow E \) resulted in an identity of the form \( V' \leftarrow E \) where \( V' \) is the adjusted name. It is assumed that the program expressions have been decomposed so that \( E \) would have at most one operator. For example, the statement \( N \leftarrow R \phi M \) would perhaps result in the identity \( N.l \leftarrow R.0 \phi M.0 \) in the verification condition. The axioms for the semantics of \( \phi \) would lead to

\[
\begin{align*}
\rho N.l & \leftarrow \rho M.0 \\
N.l(V) & \leftarrow M.0 (\{\rho M.0\}V + \rho R.0)
\end{align*}
\]

and furnish the basis for proofs about the properties of \( N \).

Quantification

The operators \( \land / \) and \( \lor / \) are analogs of the quantification operators \( \forall \) and \( \exists \), respectively, and can be directly translated by the rules (where \( A \) is a boolean vector)

\[
\begin{align*}
\land / A & \leftrightarrow \forall (V \in \rho A) A(V) \\
\lor / A & \leftrightarrow \exists (V \in \rho A) A(V)
\end{align*}
\]

The conventions for 1 (true) and 0 (false) are in effect here.

The element expressions of the formal definition are actually quantified: If \( R \) is the result of an operation, then the definitional element expressions assert that

\[
\forall (V \in \text{ELT ODOMETER } \rho R) R(V) = \text{ELEMENT EXPRESSION}
\]

So quantification is one way of relating assertions to the definition of the operators.
Conditional Expressions

There are several formalizations of the properties of conditional expressions, e.g. McCarthy[MC]. Where \( P \rightarrow A, B \iff \text{IF } P \text{ THEN } A \text{ ELSE } B \), the axioms are

\[
\begin{align*}
(P \rightarrow A, A) & \rightarrow A \\
(1 \rightarrow A, B) & \rightarrow A \\
(0 \rightarrow A, B) & \rightarrow B \\
(P \rightarrow 1, 0) & \rightarrow P \\
(P \rightarrow (P \rightarrow A, B), C) & \rightarrow (P \rightarrow A, C) \\
(P \rightarrow A, (P \rightarrow B, C)) & \rightarrow (P \rightarrow A, C) \\
((P \rightarrow Q, R) \rightarrow A, B) & \rightarrow ((P \rightarrow Q \rightarrow A, B), (R \rightarrow A, B)) \\
(P \rightarrow (Q \rightarrow A, B), (Q \rightarrow C, D)) & \rightarrow ((A \rightarrow (P \rightarrow A, C), (P \rightarrow B, D))
\end{align*}
\]

Other axioms for conditional expressions are useful, e.g. where \( D \) is a dyadic scalar operator,

\[
X \ D \ (IF \ P \ THEN \ A \ ELSE \ B) \iff IF \ P \ THEN \ X \ D \ A \ ELSE \ X \ D \ B
\]

For the purposes of the informal deductive system to be used here, we will usually break a proof into cases wherever conditional expressions occur.

Destructuring

The definition gives an expression for a single element of the result of one operation. The expression for a single element of the result of an expression is obtained by a process called "destructuring." DESTRUCTURING is the recursive substitution of the element expressions for operands to obtain an element expression for the result of a multi-operator expression.
Example 1: Consider the expression \( \forall (X=B) \land A \) where \( \{0=\varphi X \land (1=\varphi A) \land (1=\varphi B) \land ((\varphi A) \leftrightarrow (\varphi B)) \} \). In decomposed form with semantic expressions this is

\[
\begin{align*}
R\$.0 &\leftarrow X=B & pR\$.0 &\leftrightarrow pB & R\$.0\{V\} &\leftrightarrow X\{V\}=B\{V\} \\
R\$.1 &\leftarrow R\$.0 \land A & pR\$.1 &\leftrightarrow pA &\leftrightarrow pB & R\$.1\{V\} &\leftrightarrow R\$.0\{V\}\land A\{V\} \\
R\$.2 &\leftarrow \forall / R\$.1
\end{align*}
\]

Then

\[
R\$.1\{V\} \leftrightarrow R\$.0\{V\}\land A\{V\} \leftrightarrow (X\{V\}=B\{V\})\land A\{V\} \leftrightarrow (X=B\{V\})\land A\{V\}
\]

Example 2: Let \( (I=\varphi A) \land (0=\varphi W) \land (W \in \varphi A) \).

Then \( ((\varphi / (W \in A)) \rightarrow \land\{1+\varphi A\} Z) \)

\[
\begin{align*}
\leftrightarrow &((\varphi / (W \in A)) \rightarrow \land\{1+\varphi A\} Z) \\
\leftrightarrow &((\varphi / (W \in A)) (Z) \rightarrow (\land\{Z\} \rightarrow (1+\varphi A) Z) \\
\leftrightarrow &((\varphi / (W \in A)) (Z) \rightarrow (\land\{Z\} \rightarrow (1+\varphi A) Z)) \\
\leftrightarrow &((\varphi / (W \in A)) (Z) \rightarrow (\land\{Z\} \rightarrow (1+\varphi A) Z)) \\
\leftrightarrow &((\varphi / (W \in A)) (Z) \rightarrow (\land\{Z\} \rightarrow (1+\varphi A) Z))
\end{align*}
\]

Each step uses the element expressions of the definition of the operators and the definition of \( \bot \).

In order to prove a property of the result of a multi-operator expression, the property is usually proved for an arbitrary element and hence, destructuring comes into use. While the semantic expressions use the simple-indexing operation \( \bot \) we will frequently reduce this immediately to the standard APL indexing \( \bot \) wherever the ranks of variables are known.

Induction Rules

Reduction is a primitive operation which is defined by simple recursion. The use of reduction within a program or assertion or an element expression may require induction proofs. Induction on the length of a vector is common enough to result in the following rules:

INDUCTION RULES: Let \( P(X) \) represent an expression in \( X \) where \( X \) is a vector.

Complete induction: if \( P(Y) \leftrightarrow 1 \) for all vectors \( Y \) such that \( (\varphi Y) < (\varphi X) \) implies that \( P(X) \leftrightarrow 1 \) then for all vectors \( X \), \( P(X) \leftrightarrow 1 \).

Mathematical induction: if \( P(0) \leftrightarrow 1 \) and \( P(X) \leftrightarrow 1 \) implies \( P(Z, X) \) where \( Z \) is a scalar, then for all vectors \( X \), \( P(X) \leftrightarrow 1 \).
Note that the scalar \( \mathcal{Z} \) is concatenated on the left of \( X \) to accommodate the definition of the reduction operator. An example of an induction-requiring proof is

\[
(\forall / \Lambda (0,1) \exists(+) / \Lambda) \phi / \Lambda.
\]

Derived Properties

While the formal definition provides a precise standard form for the semantics of the operators, in many cases it was found useful to derive properties of the operators and then use those derived properties in proofs. Many of the derived properties have the characteristic that objects are regarded as sets, not as ordered arrays. In some cases, the derived properties simply state rather obvious characteristics of the operators but is it shown that these properties do follow within the deductive system from the definitional expressions. The representation operator is an instance where the derived property is especially useful since the definition is given as recursive conditional expressions.

Several properties are proved in Appendix C and Table 5.2 lists these properties. Two properties will be given here for illustration.

Example 1: Reduction property 21.

Let \((I = \phi A) \wedge (\forall / \Lambda (0,1))\). Then \(v / \Lambda \mapsto 1c / \Lambda\).

Proof:

\[
\begin{align*}
\forall / \Lambda \mapsto & \exists [V \in \phi A] \Lambda[V] \text{ quantification} \\
\mapsto & \exists [V \in \phi A] I = \Lambda[V] \text{ property of } = \\
\mapsto & \exists [V \in \phi A] (I = \Lambda[V]) \text{ destructuring} \\
\Rightarrow & \forall / I = \Lambda \text{ quantification} \\
\Rightarrow & 1c / \Lambda \text{ definition of } c
\end{align*}
\]
Example 2: Compression Property

The compression property states the criterion for an element $X$ to be in the result of $A/B$ where $A$ and $B$ are vectors, namely, that $X$ occur at some position of $B$ where there is a $1$ in the corresponding position of $A$. The lemma develops a relation between the positions of $1$'s in $A$ and the positions of selected elements of $B$. The proof of the property then follows by simple substitution and destructuring.

Compression lemma:
Let $(I=_{pp}_A)\land(\langle Z\rangle_{p_A})\land lW'\in p_A$.
Let $CW'\leq((+/W'\uparrow A)+/Z+1)\uparrow A$. Then $(A[IW']=1)=(I[IW'=1/CW')]$.

PROOF: The following four statements hold for $CW'$.

(1) $\forall[Z_{<\in A}]\; CW'[Z]=(/+/W'\uparrow A)+/Z+1)\uparrow A$ THEN $Z+1$ ELSE $0$.

by destructuring (see the example in the discussion) and therefore
$\forall[Z_{<\in A}]\; CW'[Z]=/+/W'\uparrow A)+/Z+1)\uparrow A$ THEN $Z+1$ ELSE $0$.

(2) By reduction property 3 (see appendix C)
  i) $\langle I/[CW']\rangle CW'$
  ii) $(I/[CW')]\leq CW'$

(3) $\forall[Z_{<\in A}]\; CW'[Z]=/+/W'\uparrow A)+/Z+1)\uparrow A$ THEN $Z+1$ ELSE $0$.

PROOF: $(I/[CW')]\leq CW'$ by (1) and therefore $(+/W'\uparrow A)+/Z+1)\uparrow A$ THEN $I/[CW']$.

Lemma: Let $(I=_{pp}_A)\land(\langle Z\rangle_{p_A})\land(0=_{pp}X)\land(0=_{pp}Y)\land(X\geq Y)$.

Then $(+/X\uparrow A)=/+/Y\uparrow A$.

Proof (not given) by induction and take property 3.

  if $Z<1/CW'$ then $(+/I/[CW']\uparrow A)+/Z+1)\uparrow A$ by the lemma and therefore

$CW'[Z]=Z+1$.

  if $Z>1/CW'$, then $CW'(Z-1)=0$. Otherwise $CW'(Z-1)=Z$ and $Z>1/CW'$

contradicting 2).

Case 2: $(I/[CW'])=0$ then by 2) $\forall[Z_{<\in A}]\; CW'[Z]=0$

(4) $(+/W'\uparrow A)=+/W'\uparrow A)/A[IW]$ Proof (not given) by take property 3 and
the definition of reduction.

Consider the possibilities for $W$ and $I/[CW']$.

Case 1: $W>1/[CW']$.

$CW'[W-1]=0$ by (3) contradicting $(I/[CW'])=0$. Therefore $W\leq I/[CW']$.

Case 2: $W<1/[CW']$.

$CW'[W]=W+1$ by (3) and therefore $(+/W'\uparrow A)+/W+1)\uparrow A$ by (1).


Case 3: $W=1/[CW']$.

PROVING ASSERTIONS ABOUT APL PROGRAMS

APL DEDUCTIVE SYSTEM

It is now proved that
i) \( \langle W/C \rangle \supseteq A[W]=0 \) and \( A[W]=1 \supseteq W=I/C \)
ii) \( \langle W=I/C \rangle \supseteq A[W]=I \) and \( A[W]=0 \supseteq W \neq I/C \)

Therefore \( A[W]=I \supseteq W=I/C \)

COMPRESSION PROPERTY
Let \( (1=ppA) \land (1=ppB) \land (\land \forall (0,1) \land ((pA)=pB) \land (0=ppX)) \).
Then \( (X \land A/B) = \land / (X=I) \land A \)

PROOF:
\( \land / (X=I) \land A \rightarrow \exists [W \subset \land A] (X \land B[W]) \land A[W] \) quantification
\( \leftrightarrow \exists [W \subset \land A] (X \land B[W]) \land A[W] \) destructuring
\( \leftrightarrow \exists [W \subset \land A] (X \land B[W]) \land W=I/((+/W \amp A) \lor + \land A) \land + \lor A \) compression lemma
\( \leftrightarrow \exists [W \subset \land A] X=B[I/((+/W \amp A) \lor + \land A) \land + \lor A] \)
\( \land A \rightarrow \exists [W \subset \land A] (X \land B[V]) \land A[V] \) definition
\( \leftrightarrow (X \land A)/B \)
TABLE 5.2
DERIVED PROPERTIES

COMPRESSION PROPERTIES: Assume

(1=ppA) \land (1=ppB) \land (0=ppX) \land (\wedge/A\in\{0,1\}) \land (pA\iff pB)

Lemma: If (C\wedge/vpA) \land C\wedge/v>pA
then (Af[v]=1) \iff (Af[v]=C\wedge/vpA)

Property: (X\wedge/vpA) \iff (X\wedge/vpA) \iff X\wedge/vpA

Corollary 1: (X\wedge/vpA) \iff X\wedge/vpA

Corollary 2: \forall [V<\wedge/vpB]/[V]=B[V]<\wedge/vpB

Corollary 3: \forall [V<\wedge/vpB]/[V]=B[V]<\wedge/vpB

Corollary 4: (X\wedge/vpA) \iff (X\wedge/vpA) \iff X\wedge/vpA

RANKING PROPERTIES: Let the following hold throughout:

(1=ppA) \land (0=ppB)

Property (1): P\in\{1+pA\}

Property (2): \sim B\in P\wedge A

Property (3): (A\wedge/B)(P)=B

Property (4): (\sim A\wedge/B)(P)=pA)

Property (5): (B\wedge/A)(P)=pA)

Property (6): (B\wedge/A)(P)=P\wedge A

Property (7): (A\wedge/B) \iff (\sim B\wedge P\wedge A) \iff P\wedge A

EXPANSION PROPERTIES: Let the following hold throughout:

(0=ppX) \land (1=ppA) \land (1=ppB) \land (\wedge/A\in\{0,1\}) \land ((\wedge/A)=pB)

Expansion property: (X\wedge/vpA) \iff (X\wedge/vpA) \iff (X\wedge/vpA) \iff (X\wedge/vpA)

Corollary: (A\wedge/B)(I) \iff IF A(I) THEN B[I\wedge A] ELSE 0

CATENATION PROPERTY: Let (0=ppX) \land (1=ppA) \land (1=ppB).

Then (X\wedge/vpA) \iff (X\wedge/vpA)

TAKE PROPERTIES:

Property 1: Let (X\wedge/vpA) \iff (X\wedge/vpA). Then (X\wedge/vpA) \iff A[V]

Property 2: Let (I=ppV) \iff (I=ppV) \iff (I\wedge/vp1\wedge V). Then

I[V\wedge/vp1\wedge V] \iff (I=ppV) \iff (I\wedge/vp1\wedge V)

Property 3: Let (W<\wedge/vpA) \iff (I=ppA). Then

(O[I\wedge/vpA]) \iff (W<\wedge/vpA)

REDUCTION PROPERTIES:

Property 1: Where D is an associative dyadic scalar operator,

(D\wedge/v=) D\wedge/v D (D\wedge/v D)

Property 2:

2i) \wedge/vA \iff 1\wedge/vA

2ii) \wedge/v\sim 0 \iff \wedge/v\sim 0

Property 3:

ia) (I=ppA) \iff (I=ppA)

ib) (I=ppA) \iff (I=ppA)

REPRESENTATION PROPERTY: Let R<\wedge/vA\wedge B. Then

(A\wedge/v[0-W[0]]-R<\sim W=0)

where W<\wedge/v\phi \wedge/v 1\wedge/v A
GUIDE TO THE EXAMPLES

Two examples are included here and several more appear in appendix D. The examples illustrate the general aspects of program verification such as correctness specification and the particular aspects of APL program verification of assertion language, program structure, formal definition and proof methods. The proofs are given informally where if the step of a proof is standard logic or mathematics then justification is omitted. Lemmas are frequently given without proof wherever the proof is similar to previously given proofs or follows directly from definitions. Since these proofs were not checked in any mechanical fashion, it is not inconceivable that there are mistakes in the proofs.

For brevity, the verification conditions will not be written down, but reference will simply be made to the included program statements. However the rules for construction of verification conditions still hold. The constraint verification aspects of several examples are discussed in Chapter 4 and Appendix B. Most programs have appeared in the APL literature with the exception of example 5 which is due to the author and example 4 which is an APL-ized version of a previously verified program.

Example 1 gives the details of the proof of the left justification one-liner introduced in chapter 2.

Example 2-union of sets-compares a one-line and a loop version of the same program for two correctness statements.
Example 3-Magic Square-shows the usefulness of destructuring for program analysis.

Example 4-Outside ac emphasises data representation and quantification. It was previously verified in ALGOL form in London[LO3].

Example 5-Hamming codes-illustrates the usefulness of the APL operators for describing complex mathematical processes. For every phase of the problem there happens to be an APL operator which accomplishes exactly that phase. The proof of correctness then shows exactly how each operator performs its specified function.

Example 6-Prime numbers-shows the construction of intermediate values with specific properties.

Example 7-Deal-illustrates some difficulties with assignments into arrays.

Example 8-Postfix translator-sketches the problem and proof where a recursive function definition mechanism is required in the assertion language and where a different proof technique from inductive assertions seems appropriate.
EXAMPLE 1: LEFT JUSTIFICATION

Returning to the example discussed in chapter 2, left justification of a vector, it is now possible to give the detailed steps of the proof. The program is

\[
\begin{align*}
(1) & \; A(\sqrt{M+i})\wedge (I \equiv pM) \\
(2) & \; N \Rightarrow ((M\neq 0) \land 1) \land M \\
(3) & \; R N(0) \neq 0
\end{align*}
\]

The decomposed form of statement 2 is

\[
R 0 \leftarrow M \neq 0 \\
R 1 \leftarrow R 0 \wedge \\
N \leftarrow R 1 \land M
\]

1. \(N(V) \leftarrow M((pM)(R1+V))\) definition of \(\phi\)
2. \(N(0) \leftarrow M((pM)(R1))\) substitute 0 for \(V\) in 1 and simplify.
3. \(\sqrt{M\neq 0} \leftarrow 1 < M \neq 0\) reduction property 2i).
4. \(\sqrt{M\neq 0}\) pre-assertion
5. \(1 < M \neq 0\) modus ponens, 3, 4.
6. \((1 < M \neq 0) \Rightarrow R 1 < pM\) ranking property (5)
7. \(R 1 < pM \Rightarrow ((pM)(R1)) = R1\) lemma using number theory (not proved)
8. \((pM)(R1)) = R1\) modus ponens, 6, 7.
9. \(M((pM)(R1)) = M(R1)\) substitution of equalities, 8.
10. \((1 < M \neq 0) \Rightarrow (M \neq 0) \Rightarrow R1=1\) ranking property (6)
11. \((M \neq 0) \Rightarrow 1 \Rightarrow (M[R1]) \Rightarrow 1 \Rightarrow M[R1] \neq 0\) destructuring and properties of =
12. \(M[R1] \neq 0\) modus ponens, 5, 10, 11.
13. \(N(0) = M[R1]\) transitivity of = 2, 9.
14. \(N(0) \neq 0\) substitution of equalities, 13, 12.

The proof structure can readily be seen in terms of subgoals:

1. - 2. Obtain the expression for \(N(0)\) as an element from \(M\).
3. - 9. Simplify that subscript expression to \(M[R1]\)
10. - 12. Show that \(M[R1] \neq 0\)
13. - 14. Deduce that \(N(0) \neq 0\)

Notice that the derived properties for ranking were important for a straightforward proof.
EXAMPLE 2: UNION OF SETS

PROBLEM: Representing a set as a vector or a scalar, form \( U \), the "union" of two sets \( A \) and \( B \).

SOLUTION: We will consider only one way of forming the union - by concatenating all elements of \( B \) which are not in \( A \) with \( A \). Two forms of this program will be verified - one using only APL operators in a single expression, and the other using a loop in place of one APL operator.

Now, what does it mean to "verify" this program? The following properties of the union operator should certainly hold:

1. The result \( U \) should be a vector or a scalar.
2. The members of \( U \) should be members of either \( A \) or of \( B \) and all members of \( A \) or \( B \) should be members of \( U \).
3. If there is no repetition of elements in \( A \) or in \( B \), then there is no repetition of elements in \( U \).

We can express these properties in APL as

1. \( I \uparrow \uparrow U \)
2. \( (X \in U) = (X \in A) \lor (X \in B) \)
3. \( (X \in A) \Rightarrow 1 + / X = A \lor (X \in B) \Rightarrow 1 + / X = B \Rightarrow (X \in U) \Rightarrow 1 + / X = U \)

The two programs which will be verified are

1. \( U \leftarrow A, (\sim B \in A) / B \)
2. \( \leftarrow \)
   1. \( U \leftarrow A \)
   2. \( I \leftarrow 0 \)
   \( \leftarrow \)
   3. \( I \uparrow B \) \( T \) \( \rightarrow 4. \ EXiT \)
   \( F \)
   \( \leftarrow \)
   7. \( I \leftarrow I + 1 \) \( T \) \( \rightarrow 5. \ B[I] \in A \)
   \( F \)
   \( \leftarrow \)
   6. \( U \leftarrow U, B[I] \)

PROOF OF ASSERTION 1: Omitted (would use standard techniques of chapter 4)
PROOF OF ASSERTION 2:

\[(X \in U) = (X \in A) \lor (X \in B)\]

PROGRAM 1: The following derived properties of the operators are used:

\[(X \in Y, Z) \Rightarrow (X \in Y) \lor X \in Z\] catenation property

\[(X \in Y / Z) \Rightarrow (X \in Z) \land Y\] compression property

\[(X \in U) \Rightarrow (X \in A) \lor X \in (~B \in A) / B\] catenation property

\[\Rightarrow (X \in A) \lor (X = B) \land ~B \in A\] compression property

\[\Rightarrow (X \in A) \lor 3(V \in \{B \mid (X = B[V]) \land ~B \in A\}\]

quantification and destructuring

\[\Rightarrow (X \in A) \lor 3(V \in \{B \mid (X = B[V]) \land ~X \in A\}\]

substitution

\[\Rightarrow (X \in A) \lor (~X \in A) \land 3(V \in \{B \mid (X = B[V])\}\]

property of quantifiers

\[\Rightarrow (X \in A) \lor (~X \in A) \land (X \in B)\]

definition

\[\Rightarrow (X \in A) \lor (X \in B)\]

PROGRAM 2 - Use the inductive assertion

\[P: (I \in 1 + PB) \land ((X \in U) = (X \in A) \lor (X \in 1 + PB) \land ~X \in A)\]

at the start of the loop. The first conjunct \(I \in 1 + PB\) uses techniques similar to those of King[K11] and is omitted here.

**vc1:** (statements 1-2) Substitute 0 for \(I\) and \(A\) for \(U\).

\[(X \in A) \Rightarrow (X \in A) \lor (X \in 0 \in PB) \land ~X \in A\]

\[\Rightarrow (X \in A) \lor (X = 0) \land ~X \in A\]

definition of \(\top\) and \(\land\)

\[\Rightarrow (X \in A) \lor (X \in 0) \land ~X \in A\]

definition of scalar operator extensions to vectors

\[\Rightarrow (X \in A) \lor 0\]

\[\Rightarrow \top\]

**vc2:** (statements 3-4)

\[(X \in U) \Rightarrow (X \in A) \lor (X \in 1 \in PB) \land ~X \in A\]

\[\Rightarrow (X \in A) \lor (X \in 1 \in PB)\]

\[\Rightarrow (X \in A) \lor (X \in B)\]

using \(I = PB\), the condition for the path.

**vc3:** (statements 3-5-7)

Show \((X \in U) = (X \in A) \lor (X \in 1 + PB) \land ~X \in A\) from

\[(X \in U) = (X \in A) \lor (X \in 1 \in PB) \land ~X \in A \land (B[1] \in A)\]

**Lemma:** \((X \in 1 + PB) \lor (X \in B[1])\)

**Proof:** omitted (uses take property 3 and reduction property 1).

\[(X \in U) \Rightarrow (X \in A) \lor ((X \in 1 \in PB) \lor (X \in B[1])) \land ~X \in A\]

using the lemma

\[\Rightarrow (X \in A) \lor (X \in 1 \in PB) \land ~X \in A \lor (X \in B[1]) \land ~X \in A\]

\[\Rightarrow (X \in A) \lor (X \in 1 \in PB) \land ~X \in A\]

since \(B[1] \in A\)

**vc4:** (statements 3-5-6-7). Prove that

\[(X \in U) = (X \in A) \lor (X \in 1 \in PB) \land ~X \in A \lor (U \in U) \lor (U \in B[1]) \lor (U \in U) = (X \in A) \lor (X \in 1 \in PB) \land ~X \in A\]

**Proof:**

\[(X \in U) \Rightarrow (X \in U) \lor (X \in B[1])\]

\[\Rightarrow (X \in A) \lor ((X \in 1 \in PB) \land ~X \in A \lor (X \in B[1]) \land ~X \in A\]

\[\Rightarrow (X \in A) \lor (X \in 1 \in PB) \land ~X \in A \lor (X \in B[1]) \land ~X \in A\]
PROVING ASSERTIONS ABOUT APL PROGRAMS
EXAMPLE 2-UNION OF SETS

\[(X \land \neg X \land (X \lor 1 \lor B))\] using the lemma above

**PROOF OF ASSERTION 3:**

Let \( P(Z) \Leftrightarrow (X \land Z) \lor I \lor /X = Z \)

**Proof (not given) using reduction property 2i.**

**Lemma 1:** \((\neg Q \lor \neg R \lor R) \Leftrightarrow (\neg Q \lor \neg R) \lor R \)

**Proof (not given) using lemma 1.**

**Lemma 3:** \((/X = Y, Z) = (/X = Y) \lor (/X = Z) \)

**Proof (not given) using reduction property 1.**

**Lemma 4:** \((/X = B) = (/X = Y) \lor (/X = B) \)

**Proof (not given) using the compression property.**

**Lemma 5:** \((/X = B) \lor B \Leftrightarrow /X = A \)

**Proof (not given) using the compression property 2i and reduction property 1.**

**PROGRAM 1:** Use assertion 2 to break into cases. Assume \( P(A) \land P(B) \).

**Case 1:** \( X \notin A \). Therefore \( /X = A \) from \( P(A) \).

\[ (+/X = U) \Leftrightarrow (+/X = A) \lor (+/X = B) \]

**Lemma 3**

\[ \equiv \lor (+/X = B) \]

**Proof (omitted) using reduction property 2i and reduction property 1.**

**PROGRAM 2:** \( \nu = \nu(B[I]) \) prove \( P(A) \land P(B) \land P(U) \land P(U') \).

**Case 1:** \( X \notin U \).

**Further case analysis using assertion 2.**

**Case 1:** \( X \in A \)

\[ \neg B[I] \in A \] by the branch condition. Therefore \( X \neq B[I] \) and \( 0 = (X = B[I]) \)

**Case 1b:** \( (X \lor B[I]) \land /X = A \)

By lemma 6 \( 0 = (X = B[I]) \).

Therefore \( X \notin U \lor (X = B[I]) = 0 \) and \((+/X = U') \Leftrightarrow (+/X = U) \)

**Case 2:** \( X = B[I] \).

by the contrapositive of case 1,

\[ X = B[I] \Rightarrow (X \land /X = 0) \Rightarrow (+/X = U) \]

and \((+/X = U') \Leftrightarrow X = B[I] \Leftrightarrow I \)
TOWARD A MECHANICAL VERIFIER FOR APL

Previous efforts in mechanical program verification have emphasized the automatic generation of verification conditions (Good[GO], King[K11]) and theorem-proving for a subclass of programs (integer arithmetic in King[K11]). The formalism and generation of verification conditions has been deemphasized in this thesis since the goal was to produce an understanding of the problems and potential of verification of APL programs before attempting mechanization. The constraint verification system of Chapter 4, however, does represent a partial solution to the task of mechanically verifying APL programs.

There is much that can be observed from the specification of the deductive system and its application to the verification proofs for the example programs:

1. A number of "packages" are required: logic, number theory (especially integer arithmetic for subscripts), induction, conditional expression manipulation, and operator simplification.

2. Two specialized rules have been identified: quantification and destructuring.

3. The derived properties are exceedingly useful in giving straightforward verification proofs by avoiding constant reference to the definitional expressions.

4. The proofs for the derived properties are much more subtle and mathematical than several of the verification proofs which rely more on pattern matching between the assertions and the definitional
preassertion.

Example 2 requires several lemmas which could be "upgraded" to the status of derived properties. It has the interesting feature that assertion 2 is used as the basis for case analysis for the proof of assertion 3. The proof also requires extensive logic manipulation and simplification. While neither of the programs is conceptually difficult, the proofs are still somewhat lengthy and intricate.

Currently, there are several systems under development which might provide a reasonable basis for the construction of a mechanical theorem prover for APL. QA4[RWD], for example, provides data and control structures for expressing the search and inference mechanism that would be required in level c) above. Presumably, the specialized packages could be builtin and provisions could be made for a growing, partitioned data base of derived properties and lemmas. Such a specialized verification system is under development by Elspas, et al.[EL3] at Stanford Research Institute. Building an APL verifier from scratch would be an immense task, but the use of such a language specialized for theorem proving would bring the project into the range of possibility.

SUMMARY

This chapter presented the basis for stating and proving general assertions about APL programs. Explicit quantification was introduced as necessary for the assertion language and deductive system. The formal definition is integrated into the deductive system as axioms. The various proof techniques for manipulating the definitional
expressions were presented. Though the deductive system has been left at the informal stage, several examples have been verified using the system as it stands. The next steps toward mechanization were discussed.

The most useful and interesting aspect of this informal deductive system is the class of derived properties that have been stated and proved for several of the operators. The derived properties for an operator reflect the various reasons for a programmer's use of the operator in developing a program. Therefore it is natural to expect that the derived property will be critical for the proof of the program. This characteristic of the APL deductive system is probably the most important for comparison with that of languages without structured operators. The power of the operators for succinct expression of many diverse program constructions can lead to the succinct proof of the properties of those constructions.

Another interesting aspect of the deductive system is the level at which induction occurs. The structured operators are defined by implicit loops yet induction is rarely needed except for instances of use of the reduction operator and even there derived properties or simple induction proofs often suffice.

As stated in the introduction, the emphasis of the thesis has been on structured operators and how they influence program verification. It is necessary to distinguish the class of problems for which the APL structured operators can be successfully and extensively used. This class is characterized by having operations
that can be uniformly applied to arrays, i.e. the same basic operation can be performed on every element of an array. Examples of programs without this property are efficient sorting algorithms, complex sequential processes, and backtracking algorithms. However, the class does include a wide variety of programs which are search-oriented or mathematically-based. The examples verified here are all of this class.

For the class of programs for which APL is well-suited, the gains in verification are:

1. Ease in understanding the solutions to a problem in terms of APL operators, resulting in a program which directly implements the solution. The high degree of structure then gives the program a good chance of being correct and verifiable. There is also the concomitant of focusing verification efforts on the statement of solution to the problem (which presumably is well understood) and making a few simple statements as to how the APL operators implement various parts of the solution. For example, the lemma \((R\setminus p M) \supseteq R = (p M)\) \(R\) of the left justification example expresses the effect of the rotation operator in the particular case where the \(R\) is a legal subscript for \(M\).

2. Suppression of subscripts and trivial loops resulting in fewer assertions than in other programming languages and the opportunity for tackling the verification of larger problems, e.g. the Hamming codes example in appendix D.

3. The potential for developing and using derived properties of the operators in verification proofs, drawing from and adding to a
data base of known (previously proved) properties of the language which are applicable to a wide class of problems.

However, there were also some drawbacks to the approach to verification through APL:

1. The partial operators required special treatment, although this was shown to be mechanizable.

2. The deductive system and nature of proof turned out to be beyond the range of present mechanical theorem-proving capabilities, although much of this is due to the mathematical character of the class of programs for which APL is designed. That is not to say that it will be impossible to develop a verifier for APL, only that it will be a difficult task requiring a suitable programming language base and extensive planning. Unfortunately, there did not appear to be any isolatable subclasses of APL programs.

3. The informal use of the deductive system requires a deep understanding of the structured operators.

It is not easy to draw objective conclusions from the comparison of programs with and without structured operators since the verification effort was expended by the author rather than any mechanical system. The left justification and union of sets examples have been given in two forms: one-liner and with one of the operators replaced by a loop. Example 4-outsideace- reproduces the proof given by London[LO3]. The study of these examples shows that the loop versions require inductive assertions with many clauses and the proofs
of verification conditions are still fairly complex. The one-line versions have shorter, simpler, and more transparent proofs because they can exploit the derived properties. One of the features of the APL operators, the deductive system, and the verification examples given here is the frequent use of quantification. King's system resorted to a single heuristic in dealing with quantifiers, though this was adequate for verifying several programs. It would be expected that quantification is the boundary beyond which proofs change character.

It is important to re-emphasize the high degree of structure which can occur in APL programs and can be handled by the proof techniques given here. The structured operators embody several specific loop formats which facilitate the expression of common mathematical and programming constructs. For example,

1. Extensions and compositions of scalar operators occur frequently in numerical applications. The special proof techniques are destructuring and the derived properties for reduction.

2. Ranking (\(\text{t}\)) and compression (\(\text{/}\)) are common programming constructs. Ranking provides for searching and its derived properties state the various possibilities for the results of that search. Compression is used for selection and its derived properties express various bases on which that selection can be made. Similar statements can be made for take, drop, expansion, and catenation.

3. The \(\wedge/\) and \(\vee/\) operators, which are analogs of quantification, and the membership (\(\in\)) operation facilitate common assertion
constructs and make APL viable as an assertion language. As part of the language, they also make certain kinds of programs, such as the outside ace example, particularly easy to verify, since the program is so similar to the assertion.
SUMMARY AND CONCLUSIONS

SUMMARY

The thesis has studied the nature of verification for programs in a particular language, APL, which has a large and powerful set of operators. The fact that the operations of a program are partial and that attributes are undeclared has led to an extension of the formalism for describing the verification requirements and to the development of a mechanical procedure for handling the verification of correctness with respect to constraints and of assertions about attributes. In order to verify general assertions about APL programs, a particular form of definition of the APL operators was presented and used as the basis for several specific, informal proof techniques. A side issue has been the choice of an assertion language.

We can examine how the thesis has answered the questions posed in the introduction.

1. What does it mean to say that a program is correct, especially when there are conditions under which the program operations may not be defined? Chapter 2 gave an appropriate definition of program correctness considering partial operators and went on to develop a formalism for verification by inductive assertions under this definition.

2. What are the requirements on the description of a language such that programs in the language can be verified in some rigorous fashion? Some restrictions on statement forms were imposed in Chapter 2. Chapter 3 gave a definition of the APL operators
which was used as the basis for some informal proof techniques which were discussed in Chapter 5 and illustrated in examples there and in appendix D. The identification of derived properties was especially useful.

3. What forms do assertions take and, consequently, what is a suitable language for making assertions? It was shown in Chapter 5 how APL can serve as an assertion language. The examples show its adequacy and usefulness, although the problem of recursively defined predicates in assertions was also mentioned.

4. What techniques, informal and mechanical, should or must be available to verify the programs? To what extent can verification be mechanized? Chapter 4 developed a mechanical system for the constraint part of the verification task which was able to make a significant reduction in the demands on the human verifier. No attempt was made to develop a mechanical theorem prover for general assertions about APL programs although specific proof techniques were identified in Chapter 5 and the informal proofs in the appendices illustrate the feasibility of proving the correctness of APL programs in that manner.

CONCLUSIONS

It is possible to draw conclusions from this research on three different levels: the approach to verification through a single programming language, the specific study of the operators of APL, and the general view of the scope of program verification. We will
SUMMARY AND CONCLUSIONS

discuss these areas separately indicating some places where future work is possible.

Relation of Program Verification to Programming Languages

At the basis of the verification process is the definition of programming languages. Language definition methods have been developed for the purpose of documentation, design, comparison, and implementation of languages. Perhaps verification should be added to this list, since such a requirement imposes some interesting restrictions on language description. This thesis furnishes an instance where the requirements of verification deeply influenced the form of language definition, although admittedly, the APL operators are quite novel and the definition of the operators is a highly specialized part of the whole language definition process. The descriptions of Pakin[PA] and Lathwell[LM] (and the common knowledge of every APL programmer) rest on the basic notion of subscripting yet the point was not pressed to the extreme until here where the standard form of description of all operators became necessary. Similarly, the distinction between primitive and nonprimitive operators was not exploited except to some extent in Abrams[AB].

Verification also provides a basis for evaluating a language or particular language features. If the techniques for verifying the properties of programs related to particular language features can not be developed, then how will it be possible to adequately describe the feature such that programmers can use it without error?
Related to the issues of definition and description of a language is the issue of language analysis, both before and after development of the language. For example, Iverson's design of the original notation was in response to the need for expression of common mathematical concepts in a programming format. The APL operators (based on Iverson's notation) have been analyzed from several different perspectives—Iverson[IV2], Abrams[AB], and this thesis. We have shown here that the operators can be described in such a way that reasonably short proofs can be given, especially when the derived properties are used.

The operators have at least partially captured the essence of programming, which is the flow of control through a series of prescribed steps, by the specification of the uniform application of simple steps to EVERY element of data structures as represented by arrays. (This uniformity is illustrated in the definition given here where the description of a single element of the result suffices to describe the entire result.) The fact that many interesting and highly complex programs can be represented succinctly in APL leads to the hope that extensive analysis of common program structures may produce this type of "packaging" of control flow in other languages.

Extensions of the APL Operators

APL enthusiasts have long been intrigued by the influence of the APL structured operators on their programming, manifested in conceptual simplicity and ease of expression of so many common programming tasks. This thesis has shown that in the verification of
APL programs these same traits are reflected in the derived properties.

It is natural then to ask on the one hand whether the set of operators can be extended and how much control flow can be expressed in operators, and, on the other hand, what general techniques learned about the operators can be applied to loops which cannot be expressed as operators. This will serve as the departure for some research to follow up this thesis. The basic goal is to express the uniformity of effect of some loops as transformations on subscripts and on elements selected by the use of uniformly generated subscripts. Perhaps a different set of primitives will be necessary. Along with this comes the need to identify loops with forms similar to the operators studied and to try to extend the proof techniques to these loops.

It can be claimed that APL programs which make extensive use of structured operators are highly inefficient, although there is no apparent reason why APL expressions cannot be optimized to a high degree. Perhaps the destructuring proof technique discussed in Chapter 5 could serve as a basis for optimization of some kinds of expressions.

Structured operators can have an immense impact on the nature of verification. The suppression of much simple control flow provides the opportunity to concentrate verification effort on the control flow characteristic to the problem being considered. This thesis only sketched some informal proof techniques and gave a number of examples which can serve as a challenge to mechanical theorem provers. It is
hoped that some progress can be made toward developing a specialized
theorem prover (possibly interactive) capable of handling the class of
problems expressible in structured operators.

APL data structures are restricted to arrays. Some attempts have
been made to extend APL data structures without conclusive results.
While this thesis does not venture any suggestions in that area, it
would appear useful to explore the development of structured operators
for a variety of data structures.

Another aspect of the structured operators was the fact that the
operators are partial. Chapter 4 has shown a technique whereby the
operator constraints can be mechanically reduced and checked. This
should lead to the conclusion that partial structured operators
present no real difficulty.

Scope of Verification Possibilities

Program verification is not yet well enough developed to be
considered a normal programming activity. The following chart may
serve as a classification of existing technology:
SUMMARY AND CONCLUSIONS

<table>
<thead>
<tr>
<th>debugging</th>
<th>formal*</th>
<th>* informal here means &quot;operating totally under control of human intuition and skill&quot; whereas formal means &quot;operating under pre-determined procedures&quot;.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>informal*</td>
<td>formal*</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>various</td>
<td>semantic</td>
</tr>
<tr>
<td></td>
<td>debugging</td>
<td>checking</td>
</tr>
<tr>
<td></td>
<td>systems</td>
<td></td>
</tr>
<tr>
<td>verification</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>structured</td>
<td>Floyd-King</td>
</tr>
<tr>
<td></td>
<td>programming</td>
<td>formalism and</td>
</tr>
<tr>
<td></td>
<td>London-type</td>
<td>mechanization</td>
</tr>
<tr>
<td></td>
<td>proofs</td>
<td></td>
</tr>
</tbody>
</table>

3 and 4 have been discussed earlier. For 1, the various debugging systems that are in use appear to operate at the programmer's discretion and include such features as traces, stops, dumps, etc. An interesting gap exists for 2. Surprisingly little work has been done in finding ways of formally analyzing programs for the purpose of feedback to the programmer. It is very difficult to imagine initially what kind of feedback information might be useful. Memory maps and symbol tables are examples and frequently point out spelling and typing errors. Elspas, et al[EL1] point out other "semantic checks" which a compiler might perform, such as locating useless statements, intrinsic loops, and nonreachable statements.

It is suggested that the procedure described in Chapter 4 is another example of this type of activity in the sense that the feedback consists of certain conditions the input to a program should meet for the program to execute properly. Of course, it is up to the programmer to interpret these conditions and determine the relation of
the conditions to his specifications for the program.

The goal of this type of activity—which we might call "formal debugging" according to the classification in the above chart—is to obtain information about a program from the structure or meaning of the program WITHOUT executing it. Verification is currently viewed as an analytic technique while debugging is viewed as sampling. Perhaps, as in Chapter 4, the same techniques can be applied for both goals.

In the broadest sense, the purpose of this kind of activity is to extract useful information purely from the structure of control flow and operators of a program, which can then be used to debug or to informally justify the correctness of a program short of complete verification. The one-liners used as examples to illustrate the constraint verification procedure show that a significant amount of information can be extracted from the programs regarding the form of their input. The goal of "formal debugging" would be to produce more information of this kind. Another example would be to produce a set of "test cases" for a program where the "testing" must be defined in some informal sense. Simply to distinguish that the program may behave differently on one test case than another may be sufficient and certainly could be interesting. The approach through structured operators shows real promise since expressions could be more manageable than general control although eventually the techniques should be extended as indicated above.

To summarize, it is hoped that the research of this thesis will be continued in several ways: concentration on identifying useful
control flow forms (with perhaps inclusion in languages as operators) and the development of standard verification methods for those control forms; the development of specialized theorem proving strategies for programs with structured operators; and the identification and development of special tools, such as preconditions for execution and test cases, for extracting useful information from programs for the purpose of debugging and partial verification.

One constant source of amazement throughout the progress of this thesis has been the deeper complexity of apparently well-understood programs. The determination of the statement of correctness and the ensuing proof often reveal facets of the programming language or the problem that would otherwise have remained unrecognized. Perhaps the greatest payoff from the efforts in program verification could be the greater appreciation of programming as a mental phenomenon and of programs as objects worthy of study for their own sake. The extent to which program verification becomes a practical reality sometime in the future is important but hopefully the view of correct programming as an important discipline and the acquisition of the necessary skills for the learning of this discipline will be enhanced by the current research in program verification.
PREFACE TO APPENDICES

Appendix A contains the additional constraints of the verification system as discussed in Chapter 4. These explanations should be read in order to completely understand the examples although the annotation to the examples should illustrate and explain several points sufficiently. Throughout the discussion of constraints references are made to particular examples and line numbers which illustrate the point.

Appendix B gives a number of additional examples of the output of the constraint verification system. The examples appear with decreasing amount of annotation so that only items which have not appeared in previous examples are noted. The complete verifications for some examples are discussed in more detail in Chapter 5 or Appendix D.

Appendix C contains several derived properties of the operators. The most important properties are proved but these proofs in turn use some lemmas for which proofs are not given (for space reasons). The properties were selected both to illustrate the proof techniques and for use in the examples. The derived properties are listed in Table 5.2. Table 5.1 gives the steps of the deductive system. Where the justification for a step in a proof is a derived or nonobvious property of the operator, that justification is given. Otherwise the justification is ordinary logic or mathematics.

Appendix D gives the verifications of several examples in addition to those in Chapter 5. Again the proofs are given informally with some lemmas left unproved.

Appendix E contains a summary of notation and conventions.
APPENDIX A
CONSTRAINT VERIFICATION

THE UNIT CONSTRAINT

REPRESENTATION- mark in "partial shape" entry in AT [1,019]**

TEST- satisfied if
constant- scalar or one-element vector
variable- marked as such in AT or has 0 as value of "rank" entry in AT [1,021]
created name- denoting an unknown rank.

APPLY- if the test fails then
Constant- error!
Variable- Mark in its "partial shape" entry in AT. [3,004,083] If the "shape" of the variable is the created name for the "shape" of another variable (i.e., its shape was preserved) then apply to that variable also. [5,004-005] If the variable is the result of a scalar expression then constrain each operand to be a unit.
Created name- if name is p/l then constrain l=pp/l (see deduction U0 below). [2,011]

COMPUTE ATTRIBUTE- mark in "partial shape" entry if
- 0 is value of "rank" entry in AT [1,019]
- "shape" expression is a vector of 1's [2,046]
- "subexpr" is a scalar expression with only unit operands [4,005]
- "shape" is preserved and the variable with preserved "shape" is a unit (see deduction rule U1) [5,090] [1,043]

SIMPLIFICATION- reduce "shape" expression to a vector of n 1's when "rank" is known to be n and the variable is constrained to be a unit. [2,010]

OBEY CONSTRAINTS- If a variable is constrained to a unit, then check if the "shape" expression is constant and nonunit and report an error, if so. If the "rank" is known but the "shape" is unknown then perform the simplification.

DEDUCTIONS
U0: UNIT(p/A) ⇒ l=pp/l
U1: UNIT(A) ∧ (p/A→p/B) ⇒ UNIT(B)
U2: (see discussion of compatibility)

CORRECTNESS- The unit attribute of a variable is acquired in three ways:
- constraint
- unity of "subexpr"ession
- unity of preserved shape

Conflict of constraint and attributes are detected in the second and successive passes when the "partial shape" attribute has been marked as constrained to a unit and the "shape" expression is checked for

**brackets refer to example (1) and line number (019) where the point is illustrated.
possible violation.

Unit constraints are applied only on the first pass and therefore used if needed. Unit attributes are computed at each pass so any unit constraint that can be satisfied will be.

LINEAR CONSTRAINT

REPRESENTATION: mark in "partial-shape"

TEST-satisfied if
constant- (constants can only be vectors or scalars)
variable- marked as linear or as unit or the "rank" is 0 or 1 [2,022]
created name- either for the shape or rank

APPLY-mark in "partial-shape" entry in AT. [1,004,034]
If the "shape" is a created name for a variable other than the one being constrained, then apply the constraint to that variable also. [1,004-005]
If the variable is the result of a scalar expression, then constrain each operand in the expression to be linear.

COMPUTE ATTRIBUTES -Mark in "partial-shape" entry if any of the following occur:
-"rank" known as 0 or 1 [1,018]
-"shape" expression is a vector of 1's
-the "subexpr" is a scalar expression and all operands are linear [2,027]
-"shape" expression is that of one of the operands of a scalar expression and the operand is linear. [2,054]
-"shape" is the created name for the shape of a variable that is linear (i.e., a linear shape is preserved). [1,022]

SIMPLIFICATION-if the "rank" is constrained to N>1 then change the "shape" expr to a vector of N 1's

OBEY CONSTRAINTS- if constrained to a unit, check if the "shape" expression is constant and nonlinear and signal an error, if so.

DEDUCTIONS
L0: LINEAR(A)\(\Rightarrow\)LINEAR(B)
L1: (See compatibility discussion)

CORRECTNESS- argument is similar to unit correctness
COMPATIBILITY

REPRESENTATION - List of values for which compatibility remains to be shown.

TEST - It is shown below that the compatibility constraints of all the operations of a scalar expression are satisfied if the operands are pair-wise compatible and that units and duplicate shapes can be removed from further consideration. The test for compatibility does exactly what the derived property suggests: a list of operands is formed from which units and duplicate-shaped operands are removed. If the list is empty after the removal then all operands were units and compatibility is satisfied. \[ 4,004 \] If only one operand remains, then the shape of the result is the shape of that operand and compatibility is satisfied. \[ 1,002,014 \] If it is possible to prove that two operands are nonunits and have nonidentical shapes then compatibility cannot occur and an error is signaled.

APPLY - The list computed by the test, if not satisfied, consists of the remaining operands for which compatibility is yet to be shown. The list will represent the constraint.

COMPUTE ATTRIBUTES - If compatibility is not satisfied, then the shape of a scalar expression is not known. In that case, a created name is used for the "shape" and "rank" expressions. \[ 7,004 \] Also, a created name is used as the "shape" if the operands are units for which the "rank"s are not known. If the compatibility test reduces to a single operand, the "shape" of the result is the shape of that operand.

OBEY CONSTRAINTS and SIMPLIFICATION - On passes after the first, the removal process is applied to the list computed on the first pass and left as a constraint. \[ 7,116 \] If the list becomes reduced to 0 or 1 elements, the shape and rank expressions are computed and entered into AT. Whenever the created name that formerly stood for the unknown shape appears, the recently computed shape will be used. Should nonunit and nonidentical shapes be detected, an error is signaled.

CORRECTNESS - Conflicts will be detected during the test for removal of elements from the list of operands. However, it would be difficult to find cases where this could occur within this limited system since there are few ways to detect inequality of expressions. The compatibility list is reprocessed at every pass so all removals will be made and removals are made only by the test so no constraints are lost.

DERIVED PROPERTIES

Definition - compatibility

\[ \text{COMP}(A;B) \leftrightarrow \text{UNIT}(A) \lor \text{UNIT}(B) \lor (pA \leftrightarrow pB) \]

Definition - result, compatibility

\[ R\text{COMP}(A;B) \leftrightarrow \text{UNIT}(A) \lor (pA \leftrightarrow pB) \]

( \( R\text{COMP} \) is used to state the relation between the result of an operation and the operands).

The following properties follow directly from the above definitions:

CPO: \( \text{COMP}(A;B) \leftrightarrow \text{COMP}(B;A) \)
APPENDIX A
CONSTRAINT VERIFICATION

CP1: If \( R \leftrightarrow A \neq B \) where \( D \) is a dyadic scalar operator and \( \text{COMP}(A;B) \) then \( \text{RCOMP}(A;R) \wedge \text{RCOMP}(B;R) \)

CP2: \( \text{RCOMP}(A;R) \equiv \text{COMP}(A;R) \)

CP3: \( \text{RCOMP}(A;R) \wedge \text{RCOMP}(R;T) \Rightarrow \text{RCOMP}(A;T) \)

CP4: \( \text{RCOMP}(A;R) \wedge \text{RCOMP}(R;T) \Rightarrow \text{RCOMP}(A;T) \)

Theorem: Let \( E \) be an expression with only scalar operations. Define the OPERANDS OF \( E \) to be all variables and constants appearing in \( E \). Then

i) if \( A \) is an operand of \( E \), then \( \text{RCOMP}(A;E) \)

ii) the compatibility constraints for all operations of \( E \) are satisfied if the operands of \( E \) are pair-wise compatible.

Proof: Since \( p \in m \) \( E \leftrightarrow pE \), where \( m \) is a scalar operator, the monadic operators can be ignored.

Proof by induction of \( n \), the number of operations performed in an expression. Let \( RK \leftrightarrow R1 \) \( DN \) \( RJ \) where \( R1 \) and \( RJ \) are the result of evaluating the operands of \( DN \), the \( n \)th operation to be performed. Let \( A1, \ldots, ANI \) be the operands of \( RI \) and \( B1, \ldots, BNJ \) be the operands of \( RJ \).

By the induction hypothesis \( \text{COMP}(AL;AM) \) for all \( AL, AM \) and \( \text{COMP}(BL;BM) \) for all \( BL, BM \). Also \( \text{RCOMP}(AL;RI) \) and \( \text{RCOMP}(BL;RJ) \) for all \( AL, BL \).

Assume \( \text{COMP}(RI;RJ) \).

\( \text{RCOMP}(RI;RK) \wedge \text{RCOMP}(RJ;RK) \) by CP1.

\( \text{RCOMP}(AL;RK) \) and \( \text{RCOMP}(BL;RK) \) for all \( AL \) and \( BL \) by CP3.

\( \text{RCOMP}(AL;RI) \wedge \text{COMP}(RJ;RI) \) by CP4 and

\( \text{RCOMP}(BM;RJ) \wedge \text{COMP}(RI;AL) \) by CP4.

Therefore all operands of \( RI \) and \( RJ \) are pair-wise compatible.

Assume pair-wise compatibility. \( RI \) is either a unit in which case compatibility of \( DN \) is satisfied immediately, or \( RI \) is a non-unit and \( pRI \leftrightarrow pA' \) for some operand \( A' \). Then if \( RJ \) is not a unit, there is some operand \( B' \) such that \( pRI \leftrightarrow pB' \). But \( \text{COMP}(A';B') \) so \( pA' \leftrightarrow pB' \) and \( \text{COMP}(RI;RJ) \).

Definition: \( ALL . \text{COMP}(L1, \ldots, LN) \leftrightarrow \) for all \( L1, L2, \text{COMP}(L1;L2) \)

Two further useful properties are

CP5: if \( L1 \) is a unit then

\( ALL . \text{COMP}(L1, \ldots, LN) \leftrightarrow \text{ALL .COMP}(\text{DELETE}(L1;L)) \)

since \( \text{COMP}(L1;L2) \leftrightarrow \text{TRUE} \) for all \( L1, L2 \).

CP6: if \( pL1 \leftrightarrow L1 \) then

\( ALL . \text{COMP}(L1, \ldots, LN) \leftrightarrow \text{ALL .COMP}(\text{DELETE}(L1;L)) \)

since \( \text{COMP}(L1, K1;L) \leftrightarrow \text{COMP}(K1;L) \) for all \( L1 \).

CP5 and CP6 permit the reduction of compatibility constraints.

The deductions mentioned in the unit and linear constraints are (where \( E \) is the result of a scalar expression)

U2: if all the operands of \( E \) are units then \( E \) is a unit.

L2: if all the operands of \( E \) are linear, then \( E \) is linear.

\* For example, if \( E \leftrightarrow A + (B - C) \cdot 3.5 \) the operands of \( E \) is the list \( A, B, C, 3.5 \).
OTHER SHAPE CONSTRAINTS

REPRESENTATION - Several constraints require that shapes satisfy some equation of the general form \( A + B \) where \( A \) and \( B \) represent arbitrary expressions which may occur as "shape" or "rank" attributes (from table 4.2).

TEST - satisfied if \( A + B \) reduces to 1 (using the expression simplification discussed in Chapter 4). [3,014] [2,057]

APPLY - if, after simplification, the equality cannot be solved for a created name representing either a rank or a shape, then attach the constraint as an equality relation. [2,036] If the relation can be solved, then the unknown "shape" can be replaced in AT by the solved value if the value is either constant or a vector of constants and units or a created name. [4,007] If the entry in AT corresponding to the created name has an expression as value, the equality of that expression and the solved-for value must hold.

If the "shape" entry in AT is the created name of another variable the constraint is applied to that variable also.

COMPUTE ATTRIBUTES - "shape" expressions are computed from table 4.2 and inserted in AT in pass 1 with simplification being applied. [2,034] "Rank" expressions are obtained from table 4.2 and also simplified. On successive passes, simplification is reapplied. [2,094]

SIMPLIFICATION - Each operator and each primitive defined list operator (SUBST, ELEM, DELETE) have associated simplification routines. See the discussion of expression simplification. [2,094]

OBEY CONSTRAINTS - On passes after the first the equality relations remaining as constraints are simplified and another attempt is made to solve or satisfy the equation. Should the equality reduce to 1 the constraint is removed. Should it reduce to 0, an error is signaled. Otherwise, an attempt is made to apply the constraint as above.

CORRECTNESS - Constraints are discarded only when the equality relation holds or when the equation can be solved for the shape or rank and then inserted into AT. Whether unresolved constraints can be further processed depends on the power of the expression simplification system. Since all "shape" and "rank" expressions are reevaluated at every pass, simplification will always occur if possible. Also constraint expressions are re-evaluated and thus additional constraints may evolve.

D.COMpatibility

\[ \text{D.COMpatible}(A; B; C; D) \iff \text{UNIT}(A) \cup \text{UNIT}(B) \cup (pA)(C) - (pB)(D) \]

REPRESENTATION - functional form of the above constraint in "constraint".
APPENDIX A
CONSTRAINT VERIFICATION

TEST- first for unity of either operand then for equality as described above. [3,014]

APPLY- If it is possible to detect that neither operand is a unit, the equality could be applied as described above. Otherwise, leave in "constraints" [6,019]

OBEY CONSTRAINTS- The test is repeated. [6,104]

MODE CONSTRAINTS

REPRESENTATION - Modes are represented by integer variables BOOL, NINT, AINT, REAL which obey the relation BOOL ≤ NINT ≤ AINT ≤ REAL.

TEST - a constraint to mode $M$, one of the four types above, is satisfied if $M$ is < the mode of the value being constrained. [1,021]

The modes of constants can be computed. [2,029].

APPLY- Let $M$ be the mode constraint. If the "mode" in AT is a created name then replace it by $M$. [2,004,028] If the created name refers to a variable other than the one being constrained, then apply the constraint to that variable also. If "mode" is not a created name then replace it by $M$.

COMPUTE ATTRIBUTE - Using tables 4.4 and 4.4.1, compute the mode. [1,020] [5,062] On pass 1, enter the mode into AT only if it is BOOL or NINT. [1,016] Otherwise create a name if the mode is not preserved and enter either the created name or the created name for the preserved mode. [1,024] On the second pass, if the mode has been constrained, say to $M$, and the computed mode satisfies the constraint, then enter the computed mode [2,074] and if the computed mode does not satisfy then leave the constraint $M < C$, where $C$ is the created name for the mode of the variable. [2,086] If the mode is not constrained, then enter the computed mode. If the mode has been preserved, then enter the mode of the variable from which the mode was preserved. [1,049]

OBEY CONSTRAINTS- The computed attributes above describe the situation on pass 2. On successive passes, mode constraints are not processed since no change could occur. That is, mode constraints are reduced as far as possible on pass 2.

CORRECTNESS- Since two passes are required anyway, it was decided to do the mode constraining on the first pass and the mode computation on the second pass. Constraints are discarded either by satisfaction on the first or the second pass. The constraints that remain after pass 2 cannot be further processed since no constraints were made during that pass. Frequently the modes reduce to constraints on bounds (to be discussed).
CONSTRAINTS ON BOUNDS

REPRESENTATION: \([L;U]\) where \(L\) and \(U\) may be either expressions or \(OMEGA\), denoting that the bound is unknown. [2,017,134] This permits constraining one bound but not the other resulting in a SEMI-BOUND.

TEST: Divide into separate tests for upper and lower bounds. Given bounds \([L;U]\) to be constrained to \([L';U']\)

- **TEST-UPPER:** satisfied if
  \[ U' \leq OMEGA \text{ or } (U' \leq OMEGA) \land (U' \leq OMEGA) \land U' \leq U' \]

- **TEST-LOWER:** satisfied if
  \[ L' \leq OMEGA \text{ or } (L' \leq OMEGA) \land (L' \leq OMEGA) \land L' \leq L \]

APPLY: When a variable is being constrained, the constraints become associated with the variables, according to the Table 4.3. [2,119] However, when a constant is being constrained the constraints become associated with the expressions occurring as bounds. [6,003] Given bounds \([L;U]\) to be constrained to \([L';U']\)

- **CONSTRAIN-UPPER:** if \(U' \leq OMEGA\) then the new bounds are
  \[ [L;U'] \text{ if } U' \leq OMEGA \]
  \[ [L;U'] \text{ otherwise} \]

- **CONSTRAIN-LOWER:** if \(L' \leq OMEGA\) then the new bounds are
  \[ [L';U] \text{ if } L' \leq OMEGA \]
  \[ [L';U] \text{ otherwise} \]

When both upper and lower bounds exist, the bounds are entered in AT. [2,073][2,119] A separate list is kept of semi-bounds. [2,134-135]

SIMPLIFICATION: \(I\) and \([\ ]\) have associated simplification routines as discussed in Chapter 4. For example, \(E_1|E_2\) where \(E_1\) and \(E_2\) are expressions, tries \(E_1 < E_2\) which in turn tries \(0 < E_2 - E_1\) opening up the possibilities for simplification of arithmetic expressions. [2,073]

COMPUTE ATTRIBUTES: No bounds are computed during the first pass, mainly because the bounds could not be resolved without simplification, which doesn't occur until the second pass. However all bounds constraints are applied on the first pass. [2,017] On pass 2, the bounds are computed according to Table 4.3. [1,045] However, for some operators, the bound expressions are not easily determinable and have been ignored. When a BOOL mode attribute is computed the bounds attribute becomes \([0;1]\) [1,039] and when NINT is computed for a mode attribute the bounds become \([0;OMEGA]\). [2,069] Bounds on constants are computed in the obvious way, for example 2 is bounded by \([2;2]\) [2,092] and the vector \(1,2,3\) is bounded by \([1;3]\).

OBEY CONSTRAINTS: Since no bounds are computed on the first pass, any bounds in AT at the start of pass 2 are constraints. When bounds are computed and a constraint is existent, the tests (upper and lower) are performed. Should the tests not be satisfied the constraints are applied, first upper then lower. If no constraints exist or the tests succeed the computed bounds are entered directly into the table.
DEDUCTIONS: The constraint applications use the $\lceil$ and $\lfloor$ functions so that constraining the upper to $U\lceil U$ guarantees that $U''\leq UU'$. This takes care of the case where it is impossible to determine whether $UU'$ through simplification.
OUTSIDE ACE

This problem is described in appendix D, example 4. It is used here to illustrate the SLA procedure. The program is

\[
\begin{align*}
[1] & A \land ((p\text{H}A\text{N}D) \text{E}Q\text{S} \cdot 13) \land (K \cdot 4) \land (\text{UNIT } K) \\
[2] & \text{ACE} \leftarrow \text{H}A\text{N}D \cdot 13 \cdot (K \cdot \text{SUIT}\text{S}) / \text{SUIT}\text{S} \cdot 4 \\
[3] & \text{A TRUE}
\end{align*}
\]

002 F$.$2 and F$.$3 are created names to stand for the value returned by the pseudo-functions UNIT and EQS which are explained in chapter 4. These functions are assumed to return a boolean scalar as value. The first step is to process the preassertion as a scalar expression.

004 The constraint UNIT K is extracted and applied.

006 The conjunct must be a unit for the whole scalar expression to be a unit.

007 The bounds of K.0 are constrained to be the bounds of the expression 14.

009-010 the shape of H\text{AND}0 becomes equivalent to the vector 13 and the rank is computed from the shape of that vector.

014 The \textit{COMPRESSION} operator requires a d\text{compatibility test, which is satisfied here since }pR$.5\rightarrow4\rightarrow\text{SUIT}\text{S.1}

018-078 AT after first pass is omitted

Constraints after first pass are omitted.

No actions on second pass.

081-111 the decomposition, result of application of SLC, and the extraction all applied to the preassertion

112-140 the decomposition and result of application of the SLC procedure to the assignment statement

142-147 All constraints are from the preassertion (the system does not distinguish the origin of constraints).
APPENDIX B-CONSTRAINT VERIFICATION
EXAMPLE C3-OUTSIDE ACE

**001 SEGMENT I**

**002 COMPATIBILIZING FS.8,RS.8,FS.1 TO RS.8**

**003 FROM RS.3**

**004 UNITIZE FS.8**

**005 FROM FS.1**

**006 UNITIZE FS.8**

**007 BOUNDS OF FS.8 TO (0,3.3)**

**008 FROM K.6.EXP RS.1**

**009 SHAPE OF K.6 TO .13.**

**010 RANK OF K.6 TO 1.**

**011 FROM FS.2**

**012 COMPATIBILIZING K.8.SUITS.1 TO .SUITS.1**

**013 FROM RS.5**

**014 DCOMPATIBILIZING RS.5.SUITS.1 .S.6. SATISFIED**

**015 FROM RS.6**

**016 COMPATIBILIZING IS.7,RK.6 TO RS.6**

**017 FROM RS.7**

**018 SEGMENT I AT-Z7**

**019 K.6-Omega**

**020 SHAPE: RHO K.6 RANK: RHO RHO K.6**

**021 UNIT**

**022 NODE: REAL**

**023 NODES: (1,1.3)**

**024 F3.1-(UNIT.6 K.6)**

**025 SHAPE: .10 .8. RANK: 8.**

**026 UNIT**

**027 NODE: BOOL**

**028 RS.1-.10 4.**

**029 SHAPE: .4. RANK: 1.**

**030 LINEAR**

**031 NODE: INT**

**032 NODES: (8,3.3)**

**033 RS.8-K.8.EXP RS.1**

**034 SHAPE: RHO K.8 RANK: RHO RHO K.8**

**035 UNIT**

**036 NODE: BOOL**

**037 HAND.8-Omega**

**038 SHAPE: .13. RANK: 1.**

**039 LINEAR**

**040 NODE: INT**

**041 NODES: (1,1.3)**

**042 F3.2-(RHO HAND.8)E.8,.13.**

**043 SHAPE: .10 .8. RANK: 8.**

**044 UNIT**

**045 NODE: BOOL**

**046 RS.3-F3.2 ARS.2 AS.1.**

**047 SHAPE: RHO K.8 RANK: RHO RHO K.8**

**048 NODE: BOOL**

**049 NODES: (8,1.1)**

**050 SUITS.1-.10 4.**

**051 SHAPE: .4. RANK: 1.**

**052 LINEAR**

**053 NODE: INT**

**054 NODES: (8,3.1)**

**055 RS.5-K.8.WE SUITS.1**

**056 SHAPE: .4. RANK: 1.**

**057 LINEAR**

**058 NODE: BOOL**

**059 NODES: (8,1.1)**

**060 RS.6-RS.5/(0.15SUITS.1)**

**061 SHAPE: */(8)RS.5 RANK: 1.**

**062 LINEAR**

**063 NODE: INT**

**064 NODES: (8,1.1)**

**065 RS.7-.13.45.6**

**066 SHAPE: */(8)RS.5 RANK: 1.**

**067 LINEAR**

**068 NODE: INT**

**069 NODES: (8,3.9)**

**070 RS.8-HAND.8.EXP RS.7**

**071 SHAPE: .13. RANK: 1.**

**072 LINEAR**

**073 NODE: BOOL**

**074 NODES: (8,1.1)**

**075 ACE.1-5R/8.8RS.8**

**076 SHAPE: .10 .8. RANK: 8.**

**077 UNIT**

**078 NODE: BOOL**

**079 CONSTRAINTS**

**080 K.6 BOUNDS TO (0,3.3)**

**081 UNIT K.6**

**082 HAND.8 SHAPE TO .13.**

**083 RANK . RANK TO 1.**

**084**
This program illustrates the SLA procedure applied to a postassertion. The program is

1. A TRUE
2. $HISTOGRAM \leftarrow X' \div i/X$
3. $A = ppHISTOGRAM$

The matrix $HISTOGRAM$ might be used to print out a histogram of the values of $X$. For example, if $X \leq 4 2$,

$HISTOGRAM \leftarrow$

1 0 0 0
1 1 1 1
1 1 0 0

and $"[HISTOGRAM]$ would print as

□ □ □ □
□ □ □ □
□ □ □ □

$ppHISTOGRAM$ is computed from Table 4.2 as

$(ppX) + ppR$.2 $\div (ppX) + 1$

Therefore the postassertion equality $2 = ppHISTOGRAM$ yields the equality $I = ppX$. The other constraint $MODE R$.1 TO NINT requires that $(/X) \geq 0$. 
APPENDIX B-CONSTRAINT VERIFICATION

EXAMPLE C4-HISTOGRAM

001 SEGMENT 1
002 UNITIZED RS1
003 MODE OF RS1 TO MINT
004 FROM RS2
005 COMPARING RS1-RHODA HISTOGRAM 1 TO .19 S.
006 FROM RS4
007 RANK OF HISTOGRAM 1 TO 2.
008 RANK OF X.0 TO 1.
009 FROM RS4-RHODA HISTOGRAM 1
010 011 SEGMENT 1 AT-ET
012 X.0 = ONEGA
013 SHAPE = RHOD X.0 RANK 1.
014 LINEAR
015 MODE = REAL
016 RS1 = .CE/(MONUS .RHOD X.0 X.0)
017 SHAPE = DELETE (MONUS .RHOD X.0 , RHO X.0)
018 RANK 1 MONUS .RH
019 UNIT
020 MODE = MINT
021 RS2 = .10 RS1
022 SHAPE = .RS1 RANK 1.
023 LINEAR
024 MODE = MINT
025 HISTOGRAM 1 = X.0 X.0 .RS2
026 SHAPE = {RHOD X.0 , RS1}
027 MODE = BOOL
028 RS4 = .2 .RHOD HISTOGRAM 1
029 SHAPE = .10 R.
030 RANK 8.
031 UNIT
032 MODE = BOOL
033 CONSTRAINTS
034 X.0 RANK TO 1.
035 END OF PASS 1
036 037 SEGMENT 1 AT-ET
038 X.0 = ONEGA
039 SHAPE = RHOD X.0 RANK 1.
040 LINEAR
041 MODE = REAL
042 RS1 = .CE/(RH.X.0)
043 SHAPE = .10 R.
044 UNIT
045 MODE = MINT
046 CONSTRAINTS: MODE RS1 TO MINT
047 048 RS2 = .10 RS1
049 SHAPE = .RS1 RANK 1.
050 LINEAR
051 MODE = MINT
052 MONUS = (0 .RS1 +(-1.))
053 HISTOGRAM 1 = X.0 X.0 .RS2
054 SHAPE = {RHOD X.0 , RS1}
055 MODE = BOOL
056 MONUS = (R.1)
057 RS4 = .2 .RHOD HISTOGRAM 1
058 SHAPE = .10 R.
059 RANK 8.
060 UNIT
061 MODE = BOOL
062 CONSTRAINTS
063 X.0 RANK TO 1.
064 MODE RS1 TO MINT
065 END OF PASS 2
066
MAGIC SQUARE

This problem is explained in more detail in example 3 of appendix D. The program is

\[(X+2) \oplus IOV \oplus 0 \oplus (IOX+1X) \oplus (X,X)_{P\mid X+X}\]

This example has very few constraints generated since the rotation constraint is met twice by having a vector left operand and finally with a unit left operand. All constraints are satisfied on the basis of the mode and unity of the input variable X. Note that the mode constraints in 006 and 012 are satisfied during the second pass.

The first pass AT is omitted.
APPENDIX B-CONSTRAINT VERIFICATION
EXAMPLE C5-MAGIC SQUARE

001 SEGMENT I
002 COMPATIBILIZING X.0,X.0 TO X.0
003 FROM Rs.1
004 UNITIZED Rs.1
005 UNITIZED X.0
006 MODE OF Rs.1 TO MINT
007 FROM Rs.2
008 MODE OF X.0 TO MINT
009 FROM Rs.3
010 COMPATIBILIZING X.0,X.0 TO X.0
011 FROM Rs.7
012 MODE OF Rs.7 TO MINT
013 FROM Rs.6
046 CONSTRAINTS
058 X.0 MODE TO MINT
051 UNIT X.0
052 END OF PASS 1
054 SATISFIED MODE Rs.1
055 SATISFIED MODE Rs.7
057 SEGMENT 1 AT-ET
058 X.0 = OMEGA
059 SHAPE: RHO X.0 RANK: RHORD X.0
060 UNIT
061 MODEL MINT
062 Rs.1 = X.0 RHO X.0 RANK: RHORD X.0
063 SHAPE: RHO X.0 RANK: RHORD X.0
064 UNIT
065 MODEL MINT
066 Rs.2 -> 10 Rs.1
067 SHAPE: X.0 RHO X.0 RANK: 1.
068 LINEAR
069 MODEL MINT
070 BOUNDS: [0..(X.0 X.0)))->(-1.1)]
071 M.1 = (X.0 X.0),NO Rs.2
072 SHAPE: X.0 X.0 RANK: 2.
073 MODEL MINT
074 BOUNDS: [0..(X.0 X.0) )->(-1.1)]
075 IDX.1 = 10 X.0
076 SHAPE: X.0 RANK: 1.
077 LINEAR
078 MODEL MINT
079 BOUNDS: [0..X.0 )->(-1.1)]
080 Rs.5 = IDX.1 RHO (1.1M.1
081 SHAPE: X.0 X.0 RANK: 2.
082 MODEL MINT
083 BOUNDS: [0..(X.0 X.0) )->(-1.1)]
084 Rs.6 = IDX.1 RHO (0.1Rs.5
085 SHAPE: X.0 X.0 RANK: 2.
086 MODEL MINT
087 BOUNDS: [0..(X.0 X.0) )->(-1.1)]
088 Rs.7 = RH X.0 X.0
089 SHAPE: RH X.0 RANK: RHORD X.0
090 UNIT
091 MODEL MINT
092 SHAPE: MINT RHO X.0 X.0 RANK: 2.
093 MODEL MINT
094 BOUNDS: [0..(X.0 X.0) )->(-1.1)]
095 CONSTRAINTS
096 X.0 MODE TO MINT
099 UNIT X.0
100 END OF PASS 2
QUADRATIC EQUATION

This program (from [LE]) illustrates the use of a computation of shape. The program is

\[ S \leftarrow (R \leftarrow \Phi[0](2-F), 2, 2) \times G, G \leftarrow 1, F, (F + D > 0), -1 \times \]

\[ Z \leftarrow (E - M[1]), (D - ((M[1] > 2) - 4) \times / M[0, 2]) < 0.5 \Rightarrow / M[0, 0] \]

The program computes the roots of a quadratic equation \( ax^2 + bx + c \) using the well known formula

\[-b + \sqrt{(b^2 - 4ac)} \]

\[ \frac{\text{----------------}}{2a} \]

The coefficients are given in order in the input vector \( M \). The output may take the form

\[ r \quad s \]

\[ t \quad u \]

where \( r + si \) and \( t + ui \) are the imaginary roots, or

\[ [r \quad s] \]

where \( r \) and \( s \) are the roots depending on the value of the determinant

002-003 constraints from subscripts
002 semi-bounds from \( 0 < pM \leq 1 \leq pM \)
004 semi-bounds again increasing to \( 3 \leq pM \)
012 simplification-scalars combined into a vector
021-096 AT after first pass omitted
099 d.compatibility satisfied through simplification
160-161 bounds and modes combined

Comment: this program shows how much information can be obtained from a program solely from the way in which the operators are used. That is, even though a program may have many constraints, the operators combine in such a way that the constraints of any operation are usually satisfied by the attributes of its operands.
APPENDIX B-CONSTRAINT VERIFICATION
EXAMPLE C6-QUADRATIC EQUATION

081 SEGMENT 1
082 RANK OF M.8 TO 1.
083 BINDING .RHO M.8 TO [.1..0.543]
084 FROM Rs.1
085 BINDING .RHO M.8 TO [.3..OMEGA]
086 FROM Rs.3
087 COMPATIBILIZING Rs.5 ,R.4 ,Rs.4 TO .10 S.
088 FROM Rs.8
089 COMPATIBILIZING D.1 ,.5 TO .10 S.
090 FROM Rs.7
091 COMPATIBILIZING (Rs.9 ,Rs.7 ,Rs.9 TO .10 S.
092 FROM Rs.10
093 COMPATIBILIZING D.1 ,.5 TO .10 S.
094 FROM Rs.11
095 COMPATIBILIZING Rs.9 ,Rs.1 TO .10 S.
096 FROM Rs.16
097 MODE OF Rs.17 TO MINT
098 FROM Rs.15
099 DCOMPATIBILIZING R.1 ,Z.1 ,R.3 ,S.
100 FROM Rs.8

097 CONSTRAINTS
100 M.8 RANK TO 1.
101 DCOMPATIBLE ,R.1 ,Z.1 ,R.3 .
102 SEMI-BOUNDS
103 .RHO M.8 (3..OMEGA )
104 END OF PASS 1
105 DCOMPATIBILIZING R.1 ,Z.1 ,R.3 ,S. SATISFIED
106 FROM S.1
107 SEGMENT 1 AT-ET
108 M.8 = OMEGA
109 SHAPE ,RHO M.8 RANK: 1.
110 LINEAR MODE: REAL
111 Rs.1 = M.8 (0..0.1)
112 SHAPE ,Z. RANK: 1.
113 LINEAR MODE: REAL
114 Rs.2 = /(/R.3R.1)
115 SHAPE .10 S. RANK: 8.
116 UNIT MODE: REAL
117 Rs.3 = M.8 (0..0.2)
118 SHAPE ,Z. RANK: 1.
119 LINEAR MODE: REAL
120 Rs.4 = /(/R.3R.3)
121 SHAPE .10 S. RANK: 8.
122 UNIT MODE: REAL
123 Rs.5 = M.8 (1..)
124 SHAPE .10 S. RANK: 8.
125 UNIT MODE: REAL
126 D.1 = (Rs.5 +R.2) - .95S.
127 SHAPE .10 S. RANK: 8.
128 UNIT MODE: REAL
129 Rs.7 = .AU D.1 ) .5
130 SHAPE .10 S. RANK: 8.
131 UNIT MODE: REAL
CONSTRAINT VERIFICATION

EXAMPLE C6-QUADRATIC EQUATION

148 Rs.6 - M.6 [1]
149 SHAPE: 10 R.  RANK: 5.
150 UNIT.
151 MODE: REAL
152 Rs.9 + Rs.9
153 SHAPE: 10 R.  RANK: 8.
154 UNIT
155 MODE: REAL
156 L.1 + (Rs.9, Rs.7) RANK: 2.
158 LINEAR
159 MODE: REAL
160 BOUNDS: [8, 1.]
161 Rs.10 - F.1, F.1; (-1).
162 SHAPE: .3.  RANK: 1.
163 LINEAR
164 BOUNDS: [(-1), 1.]
165 SHAPE; 4.  RANK: 1.
166 LINEAR
167 BOUNDS: [(-1), 1.]
168 SHAPE; 4.  RANK: 1.
169 LINEAR
170 MODE: MINT
171 BOUNDS: [8, 1.]
172 Rs.15 - G.1, Rs.14
173 SHAPE; .8.  RANK: 1.
174 LINEAR
175 MODE: AINT
176 BOUNDS: [(-1), 1.]
177 Rs.16 - G.-F.1
178 SHAPE; 10 R.  RANK: 8.
179 UNIT
180 MODE: AINT
181 BOUNDS: [1, 2.2]
182 Rs.17 + Rs.18, Rs.2.
183 SHAPE; .3.  RANK: 1.
184 LINEAR
185 MODE: MINT
186 BOUNDS: [1, 2.2]
187 Rs.18 - Rs.17, Rs.15
188 SHAPE; Rs.16, Rs.2.  RANK: 1.
189 MODE: AINT
190 BOUNDS: [(-1), 1.1]
191 Rs.1 - Rs.1, Rs.18
192 SHAPE; Rs.16, Rs.2.  RANK: 1.
193 MODE: AINT
194 BOUNDS: [(-1), 1.1]
195 S.1 + S.1, Rs.1
196 SHAPE; Rs.16, Rs.2.  RANK: 2.
197 MODE: REAL
198 CONSTRAINTS
199 M.0 RANK TO 1.
200 M.0 BOUNDS
201 MODE M.0 (3.0MEGA)
202 END OF PASS 2
203
HAMMING CODES

This problem is described fully in example 5 of Appendix D. Two different one-line versions of the problem are verified for constraints here. The programs are

\[ B = Y \vee T \leftarrow P_2 \phi((K_2) \tau S) \land Y \leftarrow ((\sim P_2 \leftarrow (S \leftarrow 1 \uparrow N \backslash K) \times 2 \leftarrow 1 \lor (\sim (2 \times L \times N \backslash N) \times L \times N \leftarrow \{2 \times N\}) \leftarrow M \right) \]

\[ B \leftarrow (P_2 \phi \leftarrow M \leftarrow (K_2) \tau (\sim P_2) / S) \lor (\sim P_2 \leftarrow 0 \leftarrow (S \leftarrow 1 \uparrow N \backslash K) \times 2 \leftarrow K \leftarrow ((N \leftarrow 1 \uparrow N \leftarrow (p \times M) \uparrow 1)) \leftarrow M \]

The output of the constraint verification system is not annotated since nothing new happens in this example. Only the output of the first form is given here. However it is interesting to note the similarities and differences in constraints. The constraints for version 2 are

1. \( M.0 \) MODE TO BOOL
2. \( M.0 \) RANK TO 1
3. \( 0 = (p \times M.0) \times (+/\sim P_2.1)^{-1} \)
4. \( D \).COMPATIBLE(M.0,R$$.18,0,1.)
   where R$.18 <= (K_2) \tau (\sim P_2.1) / S.1
5. \( 0 = K.1 + (+/P_2.1)^{-1} \)

1. \( M \) is constrained to be a boolean vector in both versions.

[7,211-212]

2. In version 1, \( K \) is computed in a closed form containing a number of scalar operators about which the constraint verification system knows nothing. Consequently the constraint on \( K \) [7,215,218] is not reduced. However, in version 2, \( K \) is computed via the \( 1 \) operator and therefore its mode is known.

3. The expansion operator introduces a shape constraint which is unsolvable within the system.

[7,214] \( \leftrightarrow (p \times M) = +/\sim P_2 \)
[7,215] \( \leftrightarrow (+/P_2) = K \)

The constraints require a derived property of the compression operator which will yield \( p \times P_2 \leftarrow (+/\sim P_2) + (+/P_2) \). The proof is completed in appendix D, example 5.
APPENDIX B-CONSTRAINT VERIFICATION
EXAMPLE C7-HAMMING CODES

SEGMENT 1
COMPATIBILIZING S.1,N.1 TO N.1
FROM RS.2
COMPATIBILIZING M.1,N.1,LN.1,N.1,LN.1 TO N.1,LN.1
FROM RS.3
UNITIZED N.1
SHAPE OF N.1 TO .1.
RANK OF M.1 TO 1.
SHAPE OF LN.1 TO .1.
MODE OF K.1 TO MINT
FROM RS.4
COMPATIBILIZING S.,RS.,4 TO ,RS.,4
FROM RS.5
MODE OF RS.,6 TO MINT
FROM RS.7
COMPATIBILIZING 1.,RS.,7 TO ,RS.,7
FROM RS.8
COMPATIBILIZING RS.13,Y.1,I,..,S. SATISFIED
MODE OF RS.13 TO MINT
MODE OF M.6 TO MINT
MODE OF Y.1 TO MINT
FROM RS.9
COMPATIBILIZING Y.1,.,T.1 TO ,T.1
FROM RS.10
BINDING LN.1 TO (((((S.4+LN.1)+LE LN.1+MHO M.8)<>(-1))<>((MHO M.8)+LE B.3)(-1)),OMEGA)
COMPATIBILIZING M.1,B.3+LN.1,N.1,LN.1,LN.1 TO LN.1
FROM LN.1
SATISFIED MODE RS.6
COMPATIBILIZING M.1,K.1 TO ,K.1
FROM RS.6
COMPATIBILIZING M.1,K.1 TO ,K.1
FROM RS.6
SEGMENT 1 AT-ET
N.8 = OMEGA
SHAPE RHO M.8
RANK: 1.
LINEAR
MODE: MINT
UNIT
MODE: MINT
LN.1 = CE B.3+LN.1
SHAPE T.1
RANK: 1.
UNIT
MODE MINT
R.1 = (N.1,LE B.3)(S.4+LN.1)+LE LN.1+MHO M.8)<>LN.1
SHAPE T.1
RANK: 1.
UNIT
MODE MINT
CONSTRAINTS: MODE K.1 TO MINT

R.8 = -10 K.1
SHAPE LN.1 = ((S.4+LN.1)+LE LN.1+MHO M.8)+(MHO M.8)+LE B.3(-1))
LINEAR
MODE: MINT
BOUND: [S.8, LN.1 = ((S.4+LN.1)+LE LN.1+MHO M.8)+(MHO M.8)+LE B.3(-1))]
R.5 = M.3+RS.4
SHAPE LN.1 = ((S.4+LN.1)+LE LN.1+MHO M.8)+(MHO M.8)+LE B.3
RANK: 1.
LINEAR
MINT
R.6 = N.1.K.1
SHAPE T.1
RANK: 1.
P-IDEA MINT
R.7 = -19 RS.6
SHAPE LN.1 = ((MHO M.8)+(S.4+LN.1)+LE LN.1+MHO M.8)+(MHO M.8)+LE B.3(-1))
LINEAR
P-MINT
L-MINT
L-MINT
L-MINT
L-MINT
APPENDIX B-CONSTRAINT VERIFICATION
EXAMPLE C7-HAMMING CODES

161 5.1 = 7.
162 SHAPE LN1 + (RHO M,O) + (S+LN1) LE LN1 + Rho M.O) + (RHO M.
+0) LE D. RANK 1.
163 LINEAR
164 MODE: MINT
165 BOUNDS: (1.1+LN1 + (RHO M.O) + (S+LN1) LE LN1 + RHO M.O) + (RHO M.
+0) LE D. RANK 1.
166 D1 = S1.1 EP RS.1
167 SHAPE LN1 + (RHO M.O) + (S+LN1) LE LN1 + RHO M.O) + (RHO M.
+0) LE D. RANK 1.
168 LINEAR
169 MODE: BOOL
170 BOUNDS: (2.1.)
171 RS.1 = SPE.1
172 SHAPE LN1 + (RHO M.O) + (S+LN1) LE LN1 + RHO M.O) + (RHO M.
+0) LE D. RANK 1.
173 LINEAR
174 MODE: BOOL
175 BOUNDS: (2.1.)
176 T.1 = RN.10 + (R.S.1)
177 SHAPE LN1 + (RHO M.O) + (S+LN1) LE LN1 + RHO M.O) + (RHO M.
+0) LE D. RANK 1.
178 LINEAR
179 MODE: BOOL
180 CONSTRAINTS: (R.(RHO M.O) + (S.RS.1)) = 1.
181
182 RS.10 = X.1. RO S.
183 SHAPE LN1 + (S+LN1) LE LN1 + RHO M.O) + (RHO M.O) LE 2.
RANK 1.
184 LINEAR
185 MODE: MINT
186 BOUNDS: (2.1.)
187 RS.1 = RS.10 + RN.1
188 SHAPE LN1 + (S+LN1) LE LN1 + RHO M.O) + (RHO M.O) LE 2.
LN1 + RHO M.O) + (S+LN1) LE LN1 + RHO M.O) + (RHO M.O) LE 2.
RANK 2.
189 MODE: BOOL
190 RS.14 = RS.12 + WE.LY.1.
191 SHAPE LN1 + (S+LN1) LE LN1 + RHO M.O) + (RHO M.O) LE 2.
RANK 1.
192 LINEAR
193 MODE: BOOL
194 RS.15 = RR (R.S.)RS.14
195 SHAPE LN1 + (S+LN1) LE LN1 + RHO M.O) + (RHO M.O) LE 2.
RANK 1.
196 LINEAR
197 MODE: BOOL
198 T.1 = FG.1 + (G.1)RS.15
199 SHAPE LN1 + (RHO M.O) + (S+LN1) LE LN1 + RHO M.O) + (RHO M.
+0) LE D. RANK 1.
200 LINEAR
201 MODE: BOOL
202 CONSTRAINTS: (G.1) LN1 + (S+LN1) LE LN1 + RHO M.O) + (S+(R.P.S.
1)) (1.1) = (RHO M.O) LE 2.
203
204 D.1 = X.1. ON T.1.
205 SHAPE LN1 + (RHO M.O) + (S+LN1) LE LN1 + RHO M.O) + (RHO M.
+0) LE D. RANK 1.
206 LINEAR
207 MODE: BOOL
208 BOUNDS: (2.1.)
209
210 CONSTRAINTS
211 RO MODE TO BOOL
212 RO RANK TO 1.
213 MODE X.1 TO MINT
214 R.(RHO M.O) + (S.RS.10) #1.1.
215 R.1(R.1) + (S+LN1) LE LN1 + RHO M.O) + (RHO M.O) #1.1) = (R.
216 RO M.O #1.1.
217 BEGIN BOUNDS
218 LN.1 = ((S+LN1) LE LN1 + RHO M.O) #1.1) = (RHO M.O) LE
L.2) = (1.1).DLL.
219 END OF MSS 2.
220
APPENDIX B-CONSTRAINT VERIFICATION
EXAMPLE C8-MATRIX INVERSE

MATR rverse

This program is as "real-life" as any to be found in APL. The
program computes the inverse of a matrix by Gaussian elimination and
appeared in Falkoff and Iverson[FI7].

[1] \rightarrow 3 IF (2=S)∧=⟨/A
[2] \rightarrow 0
[4] A←((S,P),0)∧A
[7] P[0,I]←P[I,0]; A[0,0]←A[I,0; ]
[8] A[0,0]←A[0,0]; 1=A[0,0]
[9] A←A-((0,S-1)×A[S])×A[0,0; ]
[10] A←Φ[0] 1/4 A
[12] \rightarrow 5 IF 0<K×K-1

The program has a loop, therefore the BC procedure is applicable.
The merge is 5, therefore the segments are 1-3-4, 5-6-7-8-9-10-11-12,
5-6-7-8-9-10-11-12-13, 1-2. Only the first two segments are given
here since the example is very long.

Segment 1-Annotation
004 Extraction from branch and constraint of p,A to 2.
005 created names for elements of shape vector (the names are treated as
nonnegative integer scalar variables.)
006-048 first pass AT omitted.
097 constraint requires reduction property 1 from appendix C,
(⟨p,A⟩[I]=(+/R$5,0)×+⟨p,A⟩[I]=(+/R$5,0)=+⟨p,A⟩[I]=(p,A)[I])

Segment 2
Several constraints [485-488, 497-498, 506] would be reduced if
S.1=(p,A)[I] were in the assertion before line 5. Similarly with
(pA)[I]=/A[1] the constraints [501,502] would have to be pushed back
to the input predicate. [492] and [446] appear to be due to bugs in
the system. Considering the amount of code in the segment, there are
remarkably few constraints. Compared with the number of steps that
would appear in a program in a language, without structured
operators, the verification effort would be significantly reduced for
this program. The verification was not completed because of the
difficulty of stating the inductive assertions and the anticipated
size of the proof. This example might stand as a challenge for the
completion of the proof in APL or in any other programming language.
APPENDIX B-CONSTRAINT VERIFICATION
EXAMPLE C8-MATRIX INVERSE

881 SEGMENT 1
882 COMPARABLE AIX (.RHO A.B .RS:1 TO .IO B.
883 FROM RS:2
884 RANK OF A.B TO 2.
885 FROM E.+RHO A.B
886 END OF PASS 1
887
888 SEGMENT 1 AT-AT
889 A.B = OMEDA
890 SHAPE (.RHO A.B(0) .RHO A.B(I)) RANK: 2.
891 MODEL REAL
892 RS:1 = +/(0.0) RHO A.B
893 SHAPE .IO B. RANK: 2.
894 UNIT
895 MODE: DOOL
896 RS:2 = +(0.0) RHO A.B
897 SHAPE .IO B. RANK: 2.
898 UNIT
899 MODE: DOOL
900 RS:3 = +(0.0) RHO A.B
901 SHAPE .IO B. RANK: 2.
902 UNIT
903 MODE: DOOL
904 RS:4 = +(0.0) RHO A.B
905 SHAPE [.RHO A.B(0) .RHO A.B(I)
906 MODEL REAL
907 CONSTRAINTS: R=(.RHO A.B(0) .RHO A.B(I))
908 END OF PASS 2
APPENDIX B-CONSTRAINT VERIFICATION
EXAMPLE C8-MATRIX INVERSE

181 SEGMENT 8
182 COMPATIBILIZING S.I,I. TO S.I
183 FROM R.S8
184 LINEARIZED S.1
185 LINEARIZED R.S8
186 MODE OF R.S8 TO WINT
187 FROM R.S9
188 LINEARIZED R.S9
189 FROM R.S10
190 MODE OF S.I TO WINT
191 RANK OF A.I TO S.
192 BOUNDS OF S.I TO \( R_8 \times (\text{RH} A.I) (1) \) \( (-1) \)
193 FROM R.S10 \( (\text{RH} A.I) (1) \) \( (1) \)
194 UNIFIED K.I
195 MODE OF K.I TO WINT
196 FROM R.S11
197 BOUNDS OF R.S1 TO \( R_8 \times (\text{RH} A.I) (1) \) \( (-1) \)
198 BINDING \( (\text{RH} A.I) (1) \) TO \( (1, \Omega) \)
199 FROM R.S12
200 BOUNDS OF I.I TO \( R_8 \times (\text{RH} A.I) (1) \) \( (-1) \)
201 BINDING \( (\text{RH} A.I) (1) \) TO \( (1, \Omega) \)
202 FROM R.S16
203 RANK OF P.1 TO I.
204 BOUNDS OF I.I TO \( R_8 \times (\text{RH} A.I) (1) \) \( (-1) \)
205 BINDING \( \text{RH} P.1 \) TO \( (1, \Omega) \)
206 FROM R.S17
207 BOUNDS OF I.I TO \( R_8 \times (\text{RH} A.I) (1) \) \( (-1) \)
208 BINDING \( \text{RH} P.1 \) TO \( (1, \Omega) \)
209 FROM R.S18
210 COMPATIBILIZING R.S19 R.S18 TO R.S19
211 FROM R.S20
212 COMPATIBILIZING S.I,I. TO S.I
213 FROM R.S21
214 MODE OF R.S23 TO WINT
215 FROM R.S24
216 LINEARIZED R.S24
217 FROM R.S25
218 COMPATIBILIZING R.S25 R.S22 TO R.S25 R.S22
219 FROM R.S26
220 COMPATIBILIZING A.A R.S27 TO A.A R.S27
221 FROM R.S28
222 BINDING \( \text{RH} P.2 \) TO \( (1, \Omega) \)
223 FROM R.S29
224 COMPATIBILIZING S.K S.K TO \( S.K \)
225 FROM R.S30
226 END OF PASS 1
227 BINDING S.I TO \( (1, \Omega) \)
228 BOUNDS OF R.S6 TO \( R_8 \times (\text{RH} A.I) (1) \) \( (-2) \)
229 FROM R.S8
230 BINDING S.K TO \( (1, \Omega) \)
231 TIGHTENED BOUNDS \( R_8 \times (\text{RH} A.I) (1) \) \( (-1) \)
232 FROM R.S11
233 BINDING \( (\text{RH} P.1) (1) \) \( (1) \)
234 TIGHTENED BOUNDS \( R_8 \times (\text{RH} A.I) (1) \) \( (-1) \)
235 FROM R.S12
236 BOUNDS OF R.S23 TO \( R_8 \times (\text{RH} A.I) (1) \) \( (-2) \)
237 FROM R.S26
238 COMPATIBILIZING R.S25 R.S22 TO R.S25 R.S22
239 FROM R.S29
240 COMPATIBILIZING R.S27 R.S24 TO R.S27 R.S27
241 FROM R.S26
242 COMPATIBILIZING A.A R.S27 TO A.A R.S27
243 FROM R.S31
APPENDIX C
DERIVED PROPERTIES OF THE OPERATORS

COMPRESSION: Assume \((I=ppA)\land (0=ppB)\land (\land /A<0,1)\land ((pA)\leftrightarrow pB)\).

Lemma: If \((CW\land pA)\land (CIW<+/?W\uparrow A)\land (\land /A=I+\land /p)\)
then \((A[WI]=I)\land (W=/?/CW)\).

Property: \((XA/B)\equiv (\land /X=A)\land /A)\)

COMPRESSION COROLLARIES:

Corollary 0: \((XA/B)\equiv \exists [W\land pA] \land [W] \land (X=B[WI])\)

Proof: Quantification then destructuring of the compression property

Corollary 1: \((XA/B)\Rightarrow X<\land /B\)

Proof: \((XA/B)\Rightarrow \exists [W\land pA] \land [W] \land (X=B[WI])\) corollary 0

\Rightarrow \exists [W\land pA] \land (X=B[WI])

\equiv X<\land /B

Corollary 2: \(\forall [W\land pA] \land [W] \Rightarrow (A/B)\land [W] \land /A/B\)

Proof:

\(\land [WI]\Rightarrow \land [W] \land (B[WI]=B[WI])\)

\Rightarrow \exists [W\land pA] \land [W] \land (B[WI]=B[WI])

\equiv (B[WI]=(A/B)\land [W] \land /A/B)\) corollary 0

Corollary 3: \(\forall [W\land pA] \land [W] \Rightarrow (A/B)\land [W] \land (\land /A/B)=B[WI]\)

Proof: \((A[WI]=I)\Rightarrow W=I/((\land /A/B)=I+\land /p)\) compression lemma

\Rightarrow \exists [W\land pA] \land [W] \land (X=A) \land [W] \land (X=B)\)

Corollary 0

\Rightarrow \exists [W\land pA] \land [W] \land (X=A) \land [W] \land (X=B)\)

Corollary 4: \((XA/\land pA)\Rightarrow (XA/\land pA)\land [X]\)

Proof: \((XA/\land pA)\)

\Rightarrow \exists [V\land pA] \land (X=V) \land [V] \land [V] \land (X=A) \land [V] \land (X=B)\)

Corollary 0

\Rightarrow \exists [V\land pA] \land (X=V) \land [V] \land (X=A) \land [V] \land (X=B)\)

RANKING PROPERTIES:

Let \((I=ppA)\land (0=ppB)\). By the definition, letting \(C<=>((A=B)/A+\land /p)\)

\(P<=>(A/B)<=>/C\)

(1) \(P\land /pA\)

\((P=/>/C)\Rightarrow P\land C\) reduction property 2

\(\Rightarrow P\land /pA\) Compression corollary 1
(2) \( \sim B \triangleleft P \triangleleft A \)
\( (B \triangleleft P \triangleleft A) \Rightarrow (B \triangleleft P \triangleleft A, B) \text{ using take property 2} \\
\Rightarrow \exists [V \triangleleft P] B = (A, B)[V] \text{ take property 1} \\
\Rightarrow \exists [V \triangleleft P] (V, A)[V] \subseteq C \text{ compression corollary 2} \\
\Rightarrow \exists [V \triangleleft P] C \text{ destructuring} \\
\Rightarrow \exists [V \triangleleft P] (V \triangleleft C) \wedge (0 < V) \wedge (V < P) \text{ definition of } V \triangleleft P \\
\Rightarrow \exists [V \triangleleft C] V < P \text{ definition of } \exists \\
\Rightarrow P \neq I / C \\
\text{Therefore } P = I / C \Rightarrow \sim B \triangleleft P \triangleleft A \\

(3) \( (A, B)[P] = B \)
\( P = I / C \Rightarrow P \triangleleft C \text{ reduction property 3} \\
\Rightarrow \exists [V \triangleleft I + P / A] (P = V) \wedge (A, B)[V] = B \text{ compression corollary 0} \\
\Rightarrow (A, B)[P] = B \\

(4) \( \sim B \triangleleft A = (P = \neq A) \)
\( \sim (B \triangleleft P \triangleleft A) \text{ from (2)} \\
\Rightarrow \sim (B \triangleleft P \triangleleft A) \wedge (P = \neq A) \iff \sim B \triangleleft A \\

(5) \( B \triangleleft A = (P \triangleleft P / A) \)
\( B \triangleleft A = (P \triangleleft P / A) \text{ from (4)} \\
\Rightarrow P \triangleleft I + P / A \text{ from (1)} \\
\Rightarrow P \triangleleft I + P / A \iff (P \triangleleft P / A) \wedge (P = \neq A) \text{ catenation property} \\
\text{Therefore } (B \triangleleft A) = P \triangleleft P / A \\

(6) \( (B \triangleleft A) \triangleright A[P] = B \)
\( (B \triangleleft A) \triangleright P \wedge P / A \text{ from (5)} \\
\Rightarrow P \triangleright (A, B)[P] = B \text{ compression corollary 4} \\
\Rightarrow A[P] = B \text{ catenation definition} \\

(7) \( (A[P] = B) \wedge \sim (B \triangleleft P \triangleleft A) \Rightarrow P = A \triangleleft B \\
(A[P] = B) \triangleright (A, B)[P] = B \text{ catenation definition} \\
\Rightarrow P \triangleright C \text{ compression properly} \\
\Rightarrow \sim (B \triangleleft P \triangleleft A) \Rightarrow \nexists [V \triangleleft P] B = (A, B)[V] \\
\Rightarrow \nexists [V \triangleleft P] B \subseteq (A, B)[V] \text{ catenation definition} \\
\Rightarrow \nexists [V \triangleleft P] - V \subseteq C \text{ compression corollary 4} \\
\Rightarrow \nexists [V \triangleleft P] (V \subseteq C) \wedge (0 < V) \wedge (V < P) \\
\Rightarrow P = I / C \\


d

**Expansion Property:**

Let \( (0 = \# P) \wedge (I \triangleleft P / A) \wedge (I \triangleleft P / B) \wedge (\wedge / A \wedge 0, 1) \wedge (\varnothing / A \triangleleft P / B) \)

Then \( (A \triangleleft R) \triangleleft B) \wedge (X \triangleleft R) \wedge (X \triangleleft 0) \wedge (\sim / A) \)
APPENDIX C
DERIVED PROPERTIES OF THE OPERATORS

Proof: \( (X \land B) = \exists [V \land P] X = (B, 0) \land (A \land P) \land V \) definition and quantification

Case 1: \( V \land A \land P \). Then \( (A \land P) \land V \land X \) by ranking property (4).
Therefore \( \exists [W \lor B] (X = B(W)) \land V = (A \land P) \land V \) and \( X \land B \).

Case 2: \(~V \land A \land P\). Then \( (A \land P) \land V = +/A \) and \( X = 0 \).
\((X = 0) \land ~V \land A \land P \) \( \iff (X = 0) \land ~A[V] \) by compression corollary (4)
and \( (X = 0) \land V \) \( \sim A . \)

Corollary: \( (A \lor B)[I] \Rightarrow IF \ A[I] THEN B(+/I \land A) ELSE 0 \)
\( A[I] \Rightarrow (A \land P[I]) [+/I \land A] = 1 \) compression corollary 3
\( \\
(\exists (A \land P[I]) [+/I \land A] = 1 \) ranking property (7)
\( \exists [V \land (A \lor B), I] \Rightarrow (B, 0) \) \((A \land P[I]) \land I \) definition
\( \\
\Rightarrow B(+/I \land A) \) since \( I \land A \land P \)
\( \sim A[I] \Rightarrow (A \land P[I]) [+/I \land A] \) compression corollary 4
\( \\
\Rightarrow (A \land P[I]) [+/I \land A] = 0 \) ranking property (4)
\( \\
\Rightarrow (A \land P[I]) [-/I \land A +/P B \) therefore \( (B, 0) \) \((A \land P[I]) \land I \) = 0

CATENATION PROPERTY:

\( (X \land A, B) \Rightarrow (X \land A) \lor (X \land B) \)

\( ((X \land A) \lor (X \land B)) \Rightarrow \exists [V \land (A, B)] X = A[V] \lor \exists [V \land B] X = B[V] \)
\( \\
\Rightarrow \exists [V \land (A, B)] X = A[V] \lor \exists [V \land (A, B) \land V_0] X = B[V_0] \)
\( \\
\Rightarrow \exists [V \land (A, B)] IF V \land A \land B \) THEN \( A[V] = X \)
\( ELSE B[V_0] = X \)
\( \\
\Rightarrow \exists [V \land (A, B)] IF V \land A \land B \) THEN \( A[V] \) ELSE \( B[V_0] \)
\( \\
\Rightarrow \exists [V \land (A, B)] X = (A, B)[V] \)
\( \\
\Rightarrow X \land (A, B) \)

TAKE PROPERTIES

Take property 1: Let \( P \land I \) and \( I = _P A \). Then
\( (X \land P \land I) \Rightarrow \exists [V \land P] X = A[V] \)

Proof: \( X \land P \land I \Rightarrow X \land P \land I \) definition
\( \\
\Rightarrow \exists [V \land P \land I] X = (P \land I)[V] \) quantification
\( \\
\Rightarrow \exists [V \land P] X = A[V] \land (P \land I) = \land P \land I \)
\( \\
\Rightarrow \exists [V \land P] X = A[V] \)

Take property 2: Let \( V \) and \( V_2 \) be vectors and \( I \) a scalar where
\( I \land I \land (V_1, V_2) \), then
\( I \land V_1, V_2 \Rightarrow (I \land I \land (V_1, V_2)) (V_1, V_2) \)

Proof follows by destructuring and breaking into cases \( I \land I \land (V_1, V_2) \) and
\( I \land (V_1, V_2) \land I \land (V_1, V_2) . \)
Similarly, \( I(V(I+V)) = (I(I+V))V \)

Take property 3: Let \( (W \cap A) \wedge (1 \cap p) \).
Then \( (W + l) \cap A \equiv (W \cap A) \cap [W] \).

Proof: Let \( I \cap W + l \).

\[
(W \cap A)(W^2) = IF \ I \cap W THEN (W \cap A)(W^2) ELSE UNDEFINED
\]

\[
IF \ I \cap W THEN (I \cap W)(I) \quad ELSE \ I \cap W
\]

and \( p(W + l) \cap A = (p \cap W)(A) + (p \cap A) \).

**REDUCTION PROPERTIES**

Reduction property 1: Let \( D \) be an associative scalar operator with identity and let \( V \) be a vector. Then for all \( I \cap I + pV \)

\[
(D/V) = (D/1V)(D/(I-1)1V)
\]

Proof: by complete induction on the length of \( V \). Assume true for all vectors with length \( \leq pV \). Consider \( X/V \).

Case 1: \( I=0 \).

\[
(D/0X,V)(D/0X,V)
\]

\[
(D/0X,V) = (D/0X,V)
\]

Case 2: \( (I \neq 0) \cap I \cap pX,V \)

\[
((D/I1X,V)(D/I1X,V)
\]

\[
(D/I1X,V) = (D/I1X,V)
\]

\[
(D/I1X,V) = (D/I1X,V)
\]

Reduction property 2: Let \( (I=1p) \wedge (A \cap 0, I) \). Then

\[ 2i) \cap A \equiv 0 \wedge A \]

Proof: see example 1 of derived properties of chapter 5

\[ 2ii) \cap A \equiv 0 \wedge A \]

Proof: similar to 2i).

Reduction property 3: Let \( (I=1p) \wedge (A \leq p) \). Then

\[ i) (I/A) \wedge A \]

\[ ii) (I/A) \wedge A \]

Proof:

\[ i) \text{Assume true for } A \text{ and consider } X/A. \]

\[ (I/X)A = X/I/A \text{ by the definition of reduction} \]

Case 1: \( X \wedge /A. \) Then \( X=I/X/A \) and \( X \wedge X/A \)

\[ \wedge X \wedge A = (X \wedge X) \wedge X /A \]

\[ \wedge \wedge X /A \]

\[ \wedge I \text{ since } X \wedge I /A \]
Case 2: \( (I/\lambda)>X \). Then \( (I/X,\lambda)\vdash i/\lambda \) and \( (I/\lambda)\vdash \lambda \) by the induction hypothesis and therefore \( (I/X,\lambda)\vdash X,\lambda \). Also \( (I/\lambda)>X \) therefore \( \forall(I/X,\lambda)\vdash X,\lambda \).

**REPRESENTATION PROPERTY:** Let \( R<=>\lambda \tau B \) where \( \lambda (I=\lambda A)\land (0=\lambda B)\land (I<\lambda A) \). Then \( 0=\lambda (\lambda (A[I];W([0])B-R^+\cdot \lambda W) \) where \( W<=>\lambda \cdot \lambda 1 \lambda A \).

Proof: by lemma 2 below and the definition of the residue operator.

**Lemma 1:** \( W[I]=A[I+1]-W[I+1] \) where \( J<\lambda 1+\lambda A \).

**Proof:** Let \( U<=>1,\lambda \lambda A \)

\[-U[I]<=>IF \quad I=0 \quad THEN \quad 1 \quad ELSE \quad (\phi \lambda A[I-1]) \quad definition \quad of \quad catenation\]

\[-IF \quad I=0 \quad THEN \quad 1 \quad ELSE \quad (\lambda A[I+1]) \quad definition \quad of \quad \phi \]

\[-IF \quad I=0 \quad THEN \quad 1 \quad ELSE \quad \lambda A[I+1] \quad definition \quad of \quad \downarrow \]

\[W[I]<=>\lambda (\phi \lambda A[I])\]

\[-(\phi \lambda U)[(\lambda U)-1+I] \quad definition \quad of \quad \phi \]

\[-\lambda U[I] \quad U \quad definition \quad of \quad \lambda \]

\[-\lambda U(I-I+1) \quad U(I-I+1) \quad definition \quad of \quad \lambda \]

\[-\lambda U[I+1] \quad A[I+1] \]

**Lemma 2:** \( (T[I];A[I];W[I])=B-R^+\cdot \lambda W \) where \( T \) and \( R \) are given in the definition.

**Proof by reverse induction:** \( (T[I];A[I];W[I])=B-(R[R]) \cdot \lambda W \)

Let \( I=(\lambda A)-1 \). Then \( R[I]=IF \quad A[I]=0 \quad THEN \quad B \quad ELSE \quad A[I]B \)

\[(T[I];A[I];W[I])\]

\[-(I-I-1) ;\lambda (\lambda A)-1] ;0 \quad THEN \quad 0 \quad ELSE \]

\[-(B-R[\lambda A]-1) ;\lambda \lambda (\lambda A)-1] ;\lambda (\lambda A)-1 ;\lambda W[\lambda A]-1 \]

\[-IF \quad A[I-1]=0 \quad THEN \quad 0 \quad ELSE \]

\[-(B-R[\lambda A]-1) ;\lambda \lambda (\lambda A)-1] \quad since \quad W[\lambda A]-1 \]

\[-IF \quad A[I-1]=0 \quad THEN \quad B-R[\lambda A]-1 \quad ELSE \quad B-R[\lambda A]-1 \]

\[-B-R[\lambda A]-1 \quad property \quad of \quad conditional \quad expressions \]

\[-B-\lambda R[\lambda A]-1 ;\lambda W[\lambda A]-1 \]

\[-B-\lambda (\lambda A)-1 ;\lambda R[\lambda A]-1 \]

Assume the lemma true for \( 0<I<(\lambda A)-1 \). Prove for \( I-1 \).

**Case 1:** \( A[I]=0 \). Then \( R[I]=T[I] \) and \( T[I]=0 \).

\[(T[I]-1) ;\lambda (I-1)]W \]

\[-(\lambda T[I]-1) ;\lambda [I-1] ;W[I-1] \cdot (I-1) ;\lambda W[I] \]

\[-(\lambda T[I]-1) ;\lambda [I-1] ;W[I-1] \cdot (I-1) ;\lambda W[I] \]

**Lemma 1** by \( \lambda 1 \lambda W \).

\[-B-\lambda (I-1) \cdot (I-1)]W \]

**Case 2:** \( A[I]=0 \). Then \( T[I]=T[I]-1 ;\lambda (I-1) \cdot (I-1) \)

\[(T[I]-1) ;\lambda [I-1] ;W[I-1] \cdot (I-1) \]

\[-(\lambda T[I]-1) ;\lambda [I-1] ;W[I-1] \cdot (I-1) ;\lambda W[I] \]

\[-(\lambda T[I]-1) ;\lambda [I-1] ;W[I-1] \cdot (I-1) ;\lambda W[I] \]

\[-(\lambda T[I]-1) ;\lambda [I-1] ;W[I-1] \cdot (I-1) ;\lambda W[I] \]

**Lemma 1** by \( \lambda 1 \lambda W \).
EXAMPLE 3: MAGIC SQUARE

PROBLEM: There is a well-known algorithm for creating a magic square—a matrix all rows, columns, and diagonals of which add up to the same number. The following APL program (from [PA], p. 81) accomplishes roughly the same procedure as that algorithm. It consists of three rotations, the first lining up the columns, the second the rows, and the third the diagonals.

PROGRAM: \[ S \leftarrow (N \times 2) \phi IOV \phi \left[0\right] \left(\phi IOV \leftarrow \downarrow N\right) \phi M \leftarrow (N, N)_{\leftarrow N \times N} \]

INPUT PREDICATE: \( \text{UNIT}(N) \land (N \geq 0) \land (1 \leq 2|N) \)

CORRECTNESS PREDICATE: \((\land / S \Rightarrow / S / SQUARE) \land (\land / S \Rightarrow / / \text{IOV} \times / SQUARE) \land ((/ 0 \text{IOV} \times / SQUARE) = S \Rightarrow / 0 \prime \text{SQUARE})\)

EXAMPLE: \(N=3\) (Ignore the matrices to the right for the moment)

\[
M \leftarrow \begin{pmatrix}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8
\end{pmatrix}
\]

\[
IOV \phi M \leftarrow \begin{pmatrix}
0 & 1 & 2 \\
0 & 0 & 1 \\
2 & 0 & 1
\end{pmatrix}
\]

\[
IOV \phi [0] IOV \phi M \leftarrow \begin{pmatrix}
0 & 0 & 1 \\
1 & 1 & 2 \\
2 & 2 & 0
\end{pmatrix}
\]

\[
\left(I \leftarrow 2 \phi \right) \text{IOV} \phi [0] \text{IOV} \phi M \leftarrow \begin{pmatrix}
2 & 1 & 0 \\
0 & 2 & 1 \\
1 & 0 & 2
\end{pmatrix}
\]

The program might be verified if we can establish each of the properties of the rotations as described. More insight is gained when we look at the representation of each element of the matrix written in the base \(N, 2\). These are the matrices given to the right. With the exception of the diagonals, the numbers in any position of the representation for a row or column are a permutation of \(1, 3\). Therefore a possible way of stating the rotation effect is in terms of these permutations. But it becomes very difficult to express the rotation effects after the first rotation.

An alternative is to work only with the final value of the expression, the variable \(\text{SQUARE}\).

\[ \text{SQUARE} \leftarrow \ldots \leftarrow (N, N)_{\leftarrow N \times N \times N \leftarrow N \times 2}, N(N \times 1 \times \leftarrow N \times 2), N \times (N \times 0 \times N \times 1 \times \leftarrow N \times 2) \]

This expression is derived from the semantics of the operators by
deconstructing as follows:
\[ M(V_0; V_1)\rightarrow (N\cdot N)(V_0; V_1) \rightarrow (N\cdot N)(V_0; V_1) \]
\[ (I\cdot O\cdot V\cdot M)(V_0; V_1) \rightarrow M(V_0; [V_1 + I\cdot O\cdot V]) \]
\[ \leftrightarrow M(V_0; N\cdot [V_1 + V_0]) \]
\[ \leftrightarrow (N\cdot N)(V_0; N\cdot [V_1 + V_0]) \]
and similarly for the other rotations.

Two lemmas make it possible to verify all the sums in the same way. \( N, A, B \) are scalars.

Lemma 1: \( (N\cdot A + N\cdot B)\rightarrow N\cdot A + B \)

Proof (not given) using the definition of the residue operator.

Lemma 2: \( (+/N\cdot A + B\cdot N)\leftrightarrow (+/N)\leftrightarrow (N\cdot (N-1))\cdot 2 \)

when \( N \) is odd and \( B \cdot 1, 2 \) or \( (B=3)\land (N\cdot A)=1 \)

Proof by cases for \( B \cdot 1, 2 \) where it is shown that the expression induces a permutation on \( N \) and for \( B=3 \) it is shown that 3 cycles occur.

Under lemma 1, \( SQUARE(V_0; V_1)\leftrightarrow (N\cdot N)(V_0; N\cdot [V_1 + R]) \cdot (N\cdot [V_1 + V_0 + 2\cdot V]) \cdot (2\cdot V_1 + 2\cdot R) \)

where \( R<=[N-2] \).

Each sum over rows, columns, and diagonals, gives an instance of lemma 2. For example, the main diagonal
\[ (0\ 0\ SQUARE)(V_0)\rightarrow SQUARE(V_0; V_0)\leftrightarrow (N\cdot N)((N\cdot R + 2\cdot V_0)\cdot N(2\cdot R) + (3\cdot V_0)) \]

and
\[ +/0\ 0\ SQUARE\leftrightarrow (N\cdot N\cdot (R + 2\cdot N)\cdot (+/N)(2\cdot R) + 3\cdot N) \]

expanding the definition of 1
\[ \leftrightarrow (N-1\cdot (N-1))\cdot 2 \]
\[ \leftrightarrow (N\cdot (N-1))\cdot 2 \]

All the other sums follow in the same way: Use lemma 1 to simplify the destructured expression and then substitute \( N \) for the subscript over which the reduction is applied and then use lemma 2.

COMMENTS

The correctness predicate explicitly relates the sums of all diagonals, rows and columns. Note that APL permits the identification of the diagonals by the expressions \( 0\ 0\ SQUARE \) and \( 0\ 0\ SQUARE \) and the column and row sums, respectively, by \( +/\cdot 0\ SQUARE \) and \( +/\ SQUARE \).

The first attempt at verification concentrated on the properties of the individual rotations but once the two lemmas were discovered it seemed easier to work only with the final result.

It is interesting to note that three implicit loops could be compressed into a single loop when deconstructuring was applied.

The proof of executability is given in appendix B.
EXAMPLE 4: OUTSIDE ACE

This example appeared in London[LO3] and is carried out here for comparison with that verification. The program is to be given a vector representing a bridge hand and a scalar representing a suit and is to determine whether the hand has an ace of another suit than the one given as the parameter.

PROGRAM: \(ACE \leftarrow v/\text{HAND} \in 13 \langle K \# \text{SUITS} \rangle/\text{SUITS} \leftarrow v\)

REPRESENTATION OF DATA AND PREDICATES:

\(S\) is a suit \(\iff S \in 4\)
\(X\) is an ace \(\iff X \in 13 \cdot 14\)
\(X\) is the ace of suit \(K\) \(\iff X = 13 \cdot K\)

INPUT PREDICATE: \((K \in 4) \land (0 = p K) \land (13 = p \text{HAND})\)

POSSIBLE CORRECTNESS STATEMENTS:

1. \(ACE = 3 \langle Y \in 4 \rangle (Y \# K) \land (13 - Y) \in \text{HAND}\)
2. \(ACE = 3 \langle Z \in 13 \cdot 14 \rangle (Z \# K) \land Z \in \text{HAND}\)
3. \(ACE = 3 \langle X \in \text{HAND} \rangle (X \in 13 \cdot 14) \land (X \# K)\)

Prose expressions of these correctness statements are:

1. "there exists a suit, \(Y\), such that \(Y\) is not the suit \(K\) and the ace of suit \(Y\) is contained in \(\text{HAND}\)"

2. "there exists an ace, \(Z\), such that \(Z\) is not the ace of suit \(K\) and \(Z\) is contained in \(\text{HAND}\)"

3. "there exists a member of \(\text{HAND}\), \(X\), which is an ace but is not the ace of suit \(K\)"

See appendix B for the details of showing that the program executes without error. It is possible to reduce the program to a form where subscripts are quantified.

\(ACE \leftarrow v/\text{HAND} \in 13 \langle \text{SUITS} \# K \rangle/\text{SUITS}\)

\(\leftarrow 3 \langle V \mid \in 4 \rangle \text{HAND} \in (13 \cdot V) \in 13 \langle \text{SUITS} \# K \rangle/\text{SUITS}\)

\(\leftarrow 3 \langle V \mid \in 4 \rangle \text{HAND} \in 13 \langle \text{SUITS} \rangle/\text{SUITS}\)

Each correctness statement has a form with quantification over subscripts:

1. \(ACE = 3 \langle V \mid \in 4 \rangle 3 \langle V \mid \in 4 \rangle \text{HAND} \in (13 \cdot V) \in 13 \langle \text{SUITS} \rangle/\text{HAND} \in 13 \langle V \rangle\)
2. \(ACE = 3 \langle V \mid \in 4 \rangle 3 \langle V \mid \in 4 \rangle \text{HAND} \in (Z = 13 \cdot V) \in (Z \# K) \land (Z \in \text{HAND} \in 13 \langle V \rangle)\)
3. \(ACE = 3 \langle V \mid \in 4 \rangle 3 \langle V \mid \in 4 \rangle \text{HAND} \in (X \in \text{HAND} \in 13 \langle V \rangle) \land (X \# 13 \cdot K)\)
The equivalence of each of these correctness statements with the quantified form of the program can be proved in a few steps by matching expressions and using transitivity of equality and the relation \((A\cdot B)\neq (A\cdot C)\Rightarrow B\neq C\).

London's proof is repeated here (with modification for 0 indexing) for comparison with the APL version.

Boolean procedure \texttt{OUTSIDEACE(SUIT)}; value \texttt{SUIT}; integer \texttt{SUIT};
begin integer \texttt{I,K};
\begin{verbatim}
OUTSIDEACE:= FALSE;
  comment 1: OUTSIDEACE=FALSE;
  for \texttt{I:=0} step 1 until \texttt{12} do begin
    comment 2: OUTSIDEACE=TRUE IFF
      SOME \texttt{HAND[J]}, \texttt{J=0,...,I-1}, IS AN OUTSIDE ACE;
      for \texttt{K:=3} step -1 until \texttt{0} do begin
        comment 3: OUTSIDEACE=TRUE IFF SOME \texttt{HAND[J],J=0,...,I-1},
        IS AN OUTSIDE ACE
        OR \texttt{HAND[J]} IS THE \texttt{I}TH OUTSIDE ACE, \texttt{L=K+1,...,3};
        if \texttt{HAND[I]=13\cdot K} and \texttt{K\#SUIT}
          then OUTSIDEACE:=TRUE;
        comment 4: OUTSIDEACE=TRUE IFF
          SOME \texttt{HAND[J]}, \texttt{J=0,...,I+1}, IS AN OUTSIDE ACE
          OR \texttt{HAND[J]} IS THE \texttt{I+1}TH OUTSIDE ACE, \texttt{L=K,...,3};
        end \texttt{K LOOP};
      end \texttt{I LOOP};
  end \texttt{comment 6: OUTSIDEACE=TRUE IFF}
  SOME \texttt{HAND[J]}, \texttt{J=0,...,I}, IS AN OUTSIDE ACE;
end \texttt{OUTSIDEACE};
\end{verbatim}

The proof is:
"Comment 1 is reached only from the immediately preceding assignment statement, and that statement verifies comment 1. Comment 2 may be reached either from comment 1 or from comment 5. In the former case, \texttt{I=0} and the range of \texttt{J} is empty; hence comment 2 says \texttt{OUTSIDEACE} is false and this is so from comment 1. In the latter case comment 2 follows from comment 5 noting the appropriate change in \texttt{I} caused by the for statement on \texttt{I}. Comment 4 is reached only from comment 3 and the if statement. Since the aces are represented by \texttt{13\cdot SUIT} the Boolean expression is true iff \texttt{HAND[I]} is the \texttt{K}th ace and an outside ace. Accordingly \texttt{OUTSIDEACE} is set true if \texttt{HAND[I]} is an outside ace and otherwise is not changed. Hence comment 3 holds for \texttt{I=K},
i.e., comment 4 holds. Comments 3, 5, and 6 follow by similar arguments which are omitted. Comment 6 is the lemma to be proved. OUTSIDEACE terminates since I and K are changed only in the for statements and hence the if statement is executed precisely 52 times. Q.E.D.

Assertions 2–6 all contain quantification which must tediously be repeated. The quantifications express the conditions for the existence of the OUTSIDEACE in that part of the hand so far surveyed by the outer loop and the possible suits of the inner loop. The inner loop corresponds to the construction (K#4)/SUITS and the outer loop to the construction √/HANDc... The APL program requires as an assertion, the single quantification corresponding to comment 6. Furthermore, there are several precise variations of this statement. The APL program is almost a direct statement of the assertion, making the proof a matter of simple matching.
EXAMPLE 5: HAMMING CODES

PROBLEM: Error-detecting and -correcting codes are used to provide communication over noisy channels. One of the best-known coding schemes is that of R.W. Hamming.[HA]

Consider the transmission of n-bit messages. Hamming's method encodes the n-bit message as an n+k-bit binary sequence, where the extra k bits provide for error detection and correction in any of the n+k positions of the sequence. A decoder can then map a transmitted n+k-bit sequence into a n-bit message and a k-bit sequence indicating the detection and position of error.

Example: n=4, k=3.

Let M1 M2 M3 M4 be the message to be transmitted as the sequence X1 X2 X3 X4 X5 X6 X7. The following equations are used in the encoding where \( \oplus \) is the exclusive-or sum modulo 2 operator.

\[
\begin{align*}
X1 &= M1 \oplus M2 \oplus M4 \\
X2 &= M1 \oplus M3 \oplus M4 \\
X3 &= M1 \\
X4 &= M2 \oplus M3 \oplus M4 \\
X5 &= M2 \\
X6 &= M3 \\
X7 &= M4
\end{align*}
\]

Assume the received message is Y1 Y2 Y3 Y4 Y5 Y6 Y7. The decoder computes D3 D2 D1 where

\[
\begin{align*}
D3 &= Y4 \oplus Y5 \oplus Y6 \oplus Y7 \\
D2 &= Y2 \oplus Y3 \oplus Y6 \oplus Y7 \\
D1 &= Y1 \oplus Y3 \oplus Y5 \oplus Y7
\end{align*}
\]

If one and only one digit is transmitted incorrectly, say YJ \( \neq XJ \), then D3 D2 D1 will give the binary representation of J and will be 0 0 0 if no errors occur.

Generalizing to a code for n-bit messages, an additional k bits are required to point to any of the n+k bits of the encoding which might be in error. The sufficient condition is

\[
(2^K) > N+K+1
\]

A general method for the equation assignment is:

1. Use the positions numbered by powers of 2 for check bits.

2. Assign the bits of the original message in order to the remaining positions.
The check position equations are formed by grouping the binary representations of the positions by occurrence of powers of 2:

\[
\begin{array}{cccccccc}
1234567 & \text{BIT POSITIONS} \\
0011 & P \\
0113 & X \\
1015 & X \\
1117 & X \\
\hline
0102 & P \\
0113 & X \\
1106 & X \\
1117 & X \\
\hline
1004 & P \\
1015 & X \\
1106 & X \\
1117 & X \\
\end{array}
\]

Position 1 serves as a parity check for positions 1,3,5,7 of the encoding (and positions 1,2,4 of the message), and similarly, for positions 2 and 4.

Let \( M \) be the \( n \times k \)-bit sequence to be transmitted. Let \( U \) be a \( K \) by \( N+K \) matrix where column \( I \) is the binary representation of \( I \) in \( K \) places. The sequence is constructed to satisfy the matrix equation

\[ Z = U \# \& M \]

where \( Z \) is a vector of length \( K \) of all zeroes.

The received message may be represented as \( M+E \), where \( E \) is an error vector with 1's in every position where an error occurs. Then

\[ D <= U \# \& (M+E) \leftrightarrow (U \# \& M) \& (U \# \& E) \leftrightarrow 0 + U \# \& E \]

If \( E \) is all zeroes (no errors in transmission) then \( \& / D = 0 \) and if \( E \) has a 1 in position \( I \) then \( E \) selects the binary representation of \( I+1 \) from \( U \).

SOLUTION: The encoder can be written as a one-line APL program. Two variations will be given with a complete proof of correctness of the first solution.

\[
\forall B \lnot ENCODEV1 M
\]

[1] \( B \leftarrow Y \lor T \oplus P 2 \{\Phi (K_2) \{S\} \} \lor Y \leftarrow \{\neg P 2 \oplus \{S \leftarrow 1 \cap N+K\} \times i \}
\]

\[
K \leftarrow \{N \times 2 \} \{\{2 \times L N \} \} \times L N \times N \times \{120 \cap N \cap p M\} \} \} \}
\]

\[
\forall B \lnot ENCODEV2 M
\]

[1] \( B \leftarrow P 2 \{\Phi / \} / (K_2) \{1 \} (- P 2) / S \} \lor
\]

\[
(\neg P 2 \oplus 0 \{S \leftarrow 1 \times N+K\} \times \{K \} \times \{N \times 2 \} \times \{N \cap p M\} \} \} \}
\]
The only conditions not satisfied from that process are: The basic constraint verification is discussed in appendix B. The only conditions not satisfied from that process were 

\[ (+/\sim P2) = PM \]

and 

\[ (+/P2) = K \]

Since a property of the / operator is \((+/\sim P2 + (+/P2))\) so \(P2 \leftrightarrow N+K\) only one condition need be shown.

**Lemma 1.1:** \((+/P2) = K\)

**Proof:** Using lemma 3, \(P2[I] = 1\) iff \(I+1\) is a power of 2. The number of powers of 2 which are less than or equal to \(N+K\) is \(1 + 2 + N + K\) which is less than \(K\). Assume \(I < K\). Then \(2 + I < N + K\) by the construction of \(K\). Therefore the number of powers of 2 which are less than \(N+K\) is \(K\).

**Lemma 2:** \((2^K) > N + K\)

\[ K \sim (N < 2) + ((2 \times I \cdot N) \times (N + N) + I N < 2 + N) \]
APPENDIX D
EXAMPLE 5-HAMMING CODES

Case 1: N=1 or N=2 obvious
Case 2: N>3
K=(2sLN)(LN+N)+I, N<12sN
K<N since (2sN)>2sN.
At least 12sN bits are required to represent N and at most I+(2sN) bits
will be required to represent N+K. Therefore LN contains the minimum
number of bits and the expression (2sLN)(N+LN) adds 1 if another bit is
necessary.

Lemma 3: P2[I]=1 iff (2s2[I]+I)=I+1 for I<N+K
Proof: P2=S<2sK where S=I+1
I<N+K=S(I+1)<N+K
(I+1)<2sK implies P2[I]=1
If (2s2[I]+I)=I+1 then
(I+1)<2sK implies P2[I]=1
P2[I]=1 implies (I+1)<2sK implies P2[I]=1

Lemma 4: If U(I)=[K+2] then for I<N+K and then \( \land/(U^R, \land B)=0 \).
Proof: Y=[(I+1)-P2]\M
From the expansion property Y[I]=IF P2[I] THEN 0 ELSE M[I+1]_P2
Note that U=[(K+2)]
Let T=[P2]\W where W=[U^R, \land Y]
W[I]=[U^R, \land Y][K-1+I]+[U[K-1+I]_Y]

The following properties of the operators are used (without proof).
Let A and C be boolean vectors of the same length.
1) \( \forall/C=2I/C \)
2) \( ((+/A/C)=+/(\neg A)/C)=+/C \)
3) \( (+/A/CvC2)=+/A/C\land A/C2 \)
4) \( (+/A/C\land C2)=+/A/C\land A/C2 \)
5) \( \land/0=(\neg A)/A\land C \)

The variables of the program have the properties:
6) \( ((P2[U]/[I]+I)=(I+1)=2s(K-1+I)) \)
Proof: \((P2/[I]/[I]+I)=U/[I], P2/[N+K] \)
U[I]+I=[K+2] \rightarrow I+1+I=2sK
U[I]+I=1 \rightarrow I+1=K-1+I \rightarrow I+1=K-1+I
Therefore P2/[U[I]+I] has 1 only in the position K-1+I and
((P2/[U[I]+I])=(P2/[B])=IF J-K-1+I THEN (P2/[B][J]) ELSE 0
7) \( +/(/P2)/[I]+(/P2)/Y+/U/[I]_Y \)
Proof: \((-P2[I])=Y[I]=I=0 \rightarrow Y[I]\land U[I]=0 \)
Finally the goal is to show that for \( I \subseteq N \setminus K \), \( \neq /U[I;] \wedge B \neq 0 \).

\( \neq /U[I;] \wedge B \)

\( \leftrightarrow 2[(+/P_2/U[I;] \wedge B) \leftrightarrow (\sim P_2)/U[I;] \wedge B \text{ properties 1 and 2)}

\( \leftrightarrow 2[(+/P_2/U[I;] \wedge (P_2/B)) \leftrightarrow (\sim P_2)/U[I;] \wedge B \text{ property 3)}

\( \leftrightarrow 2[(P_2/B)/(K-1+I) \leftrightarrow (\sim P_2)/U[I;] \wedge B \text{ property 6) and 7)]

\( \leftrightarrow (P_2/B)/(K-1+I)/(\sim P_2)/U[I;] \wedge B \wedge 0 \text{ properties 2) and 4)}

\( \leftrightarrow (P_2/B)/(K-1+I)/(\sim P_2)/U[I;] \wedge (\sim P_2/Y) \wedge 0 \text{ Reduction property 1) and property 5) above)}

\( \leftrightarrow W[/K-1+I] \neq 2]/U[I;] \wedge Y \text{ Lemma 4)}

\( \leftrightarrow W[/K-1+I] \neq 2]/U[I;] \wedge Y \text{ } \leftrightarrow (\neq /U[I;] \wedge Y) \neq /U[I;] \wedge Y \leftrightarrow 0 \text{ )}

Theorem: Let \( BR \) be the received message.

\( D = ([K_2]_1 U \neq \wedge B \neq 0) \)

i) (0 errors) \( \neq /B=BR \Rightarrow D = 0 \)

ii) (1 error) \( \neq /B\neq BR \Rightarrow B[D] \neq BR[D] \)

Proof:

i) by lemma 4 \( \wedge (U \neq \wedge BR) = 0 \) therefore \( D = 0 \)

ii) \( I = \neq /B\neq BR \Rightarrow \exists V \in I \wedge B \wedge (B \neq BR) = (V = \wedge B) \)

\( BR \neq = B \neq BR \text{ Therefore } BR \neq B \neq (V = \wedge B) \text{ for some } V \in \wedge B \)

\( \neq /U[I;] \neq /U[I;] \wedge BR \leftrightarrow /U[I;] \wedge B \neq (V = \wedge B) \)

\( \leftrightarrow (\neq /U[I;] \neq /U[I;] \wedge B \neq (V = \wedge B) \)

\( \leftrightarrow 0 \neq /U[I;] \wedge B \neq (V = \wedge B) \)

\( \leftrightarrow U[I;V] \leftrightarrow ((K_2]_1 V + 1)(I)] \text{ )}

Therefore \( U \neq \wedge B \leftrightarrow ([K_2]_1 V + 1 \text{ and } D \leftrightarrow V \)

Then \( (B \neq BR)[D] = 1 \text{ and } B[D] \neq BR[D] \)
EXAMPLE 6: PRIME NUMBERS

The following program (from [AB], p. 176) computes a vector containing all primes less than \( N \), where \( N \) is a nonnegative integer scalar.

\[
PRIMES \leftarrow \{2\} + \{0\} = ION : ION / ION \leftarrow 1 + \!\!\!\!\!\!N
\]

The output predicate that will be used (there are others) is

\[
(X < PRIMES) \Rightarrow \forall [Y < 2 + \!\!\!\!\!\!0 | X - 2] \, 0 \neq Y | X
\]

which is equivalent to

\[
(X < PRIMES) \Rightarrow (2 < Y) \land (Y \leq X - 1) \land (Y = |Y|) \Rightarrow 0 \neq Y | X
\]

It is useful to decompose and then destructuring the decomposed expressions for reference in the proof.

\[
ION \leftarrow 1 + \!\!\!\!\!\!N \\
R0 \leftarrow ION : ION \\
R1 \leftarrow R0 \\
R2 \leftarrow [0]R1 \\
R3 \leftarrow 2 \rightarrow R2
\]

\[
ION \leftarrow V \leftarrow 1 + \!\!\!\!\!\!N \\
R0 \leftarrow R0 \leftarrow N, N \\
R1 \leftarrow R1 \leftarrow N, N \\
R2 \leftarrow R2 \leftarrow N \\
R3 \leftarrow R3 \leftarrow N
\]

\[
(X < PRIMES) \Rightarrow \exists [V \in 1] \, (X = ION[V]) \land R3[V] \ \text{compression lemma}
\]

\[
\Rightarrow \exists [V \in 1] \, (X = V \leftarrow 1 + \!\!\!\!\!\!N) \land (2 = + / 0 = (1 + \!\!\!\!\!\!N) | V + 1)
\]

\[
\Rightarrow (2 = + / 0 = (1 + \!\!\!\!\!\!N) | X) \land (X \in 1 + \!\!\!\!\!\!N)
\]

Case 1: \( N \leq 1 \). \( (+ / 0 = (1 + \!\!\!\!\!\!N) | X) < 2 \) therefore \( 0 = p PRIMES \).

Case 2: \( N > 2 \)

\( 2 = + / 0 = (1 + \!\!\!\!\!\!N) | X \)

\[
\Rightarrow 2 = (0 = X) \ast (+ / (X - 2) \ast 1 \ast 0 = (1 + \!\!\!\!\!\!N) | X) + (0 = X) \ast + / 0 = X \ast (1 + \!\!\!\!\!\!N) | X \ \text{using reduction property 1}
\]

\[
\Rightarrow 2 = 1 + (+ / (X - 2) \ast 1 \ast 0 = (1 + \!\!\!\!\!\!N) | X) + 1 \ast 0
\]

using the definition of 1

\[
\Rightarrow 0 = + / (X - 2) \ast 1 \ast 0 = (1 + \!\!\!\!\!\!N) | X
\]

\[
\Rightarrow + / (X - 2) \ast 1 \ast 0 = (1 + \!\!\!\!\!\!N) | X
\]

Lemma: Proof (omitted by induction)

\[
(\land / / (0, 1) = (0 = + / (0, A) \ast (0 = + / (0, A) \ast (0 = + / (0, A)
\]

\[
\Rightarrow V[V \in 1, X = 2] \Rightarrow (0 \neq (1 + \!\!\!\!\!\!N) | X) | V + 1) \ \text{take property 1}
\]

\[
\Rightarrow V[V \in 1, X = 2] \Rightarrow 0 \neq (V + 2) | X \ \text{destructuring}
\]

\[
\Rightarrow V[V \in 1, X = 2] \Rightarrow 0 \neq Y | X
\]
EXAMPLE 7: DEAL OPERATOR

The following program is used as the definition of the APL deal operator in the definition of APL by [LM]. It illustrates some techniques and problems in using explicit subscripts.

The deal operator, written \( A?B \), returns a vector of length \( A \) composed of random numbers chosen from \( B \) without repetition. The correctness statement for the operator is

\[
A \backslash (A \backslash 1 \backslash B) \backslash (B \geq 1) \backslash (NINT \ B) \\
R \leftarrow A ? B \\
A \backslash (\bigvee (R \backslash I) = R \backslash I) \backslash (A = R) \backslash (I = p R)
\]

The statement \( R \leftarrow A ? B \) will be replaced by the program which defines the operator and the program will be shown to satisfy the correctness statement.

1. \( A(A \backslash 1 \backslash B) \backslash (B \geq 1) \backslash (NINT \ B) \)
2. \( R \leftarrow B \)
3. \( I \leftarrow 0 \)
4. \( J \leftarrow I + 2 \backslash B - I \)
5. \( (A(A \backslash 1 \backslash B) \backslash (B \geq 1) \backslash (NINT \ B)) \backslash \\
\bigvee _{R \backslash I} (A = R) \backslash (I = p R) \backslash (J \leq p R) \Rightarrow (p R) \sim B \)
6. \( R[I, J] \leftarrow R[I, J] \)
7. \( \rightarrow 4 \text{ IF } I > I + 1 \)
8. \( R \leftarrow A \backslash p R \)
9. \( (\bigvee (R \backslash I) = R \backslash I) \backslash (A = R) \backslash (I = p R) \)

The program uses the random number generator \( ?Y \) which is defined by the expression \( X = ?Y \times X \backslash 1 \) where \( Y \) is a scalar.

Let VC1 by the verification condition for statements 1-2-3-4-5, VC2 for statements 5-6-7-4-5, VC3 for statements 5-6-7-8-9. The various forms of the uniqueness property \( \bigvee (R \backslash I) = R \backslash 1 \) were discussed in chapter 5. It can be proved as a lemma that

\[
\bigvee (R \backslash I) = R \backslash 1 \Leftrightarrow \forall [V \backslash I \backslash p R] \forall [V \backslash 2 \backslash p R] (V \backslash 1 = V \backslash 2) \Rightarrow R \backslash [V \backslash 2]
\]

VC1 and VC3 are easily shown using straightforward properties of the operators and the quantified form of the uniqueness predicate. VC2 is of most interest since it involves a permutation of elements of an array, a problem discussed in Kirp [KI]. The clauses of the inductive assertion will be proved separately:

\( (A \backslash 1 \backslash B) \backslash (B \geq 1) \backslash (NINT \ B) \) are preserved around the loop since \( A \) and \( B \) are unchanged.

\( (I + 1) \backslash p R \) holds from \( A(A \backslash 1 \backslash B) \backslash (I \geq 1) \backslash (A < B) \) and \( (I + 1) \backslash A \), the branch condition, and \( I \leq p R \geq (0 \backslash I) \backslash I \backslash p R \), the inclusive assertion.
(J \leq R) holds from
\[ J \leftrightarrow (I+1)+(I+1) \Rightarrow \neg (I+1) \land B \land (I+1) \land \\
\neg (I+1) \land J \land (I+1) \land \\
\neg (I+1) \land J \land (I+1) \]

Now, it is necessary to examine the semantics of \( R[J,J] \leftrightarrow R[J,J] \) to get
the rest of the assertion.

Letting \( R' \) be the new \( R \),
\[ R'[V] = \begin{cases} \\
\text{IF } V < (J,J) \text{ THEN } R[I,J][1-(\Phi[J,J])R] \text{ ELSE } R[V] \\
\text{IF } V < (J,J) \text{ THEN } R[I] \text{ ELSE } R[J] \text{ ELSE } R[V] \\
\end{cases} \]

\( \land / R' \leq B \) comes from \( \land / R' \leq R \) from the definition above and \( \land / R' \leq B \)
from the inductive assertion.

\( \land / (R' \land R) \leq B \) uses the quantified form and, using the lemma, breaks
into cases for \( R' \).

Case 1: \( V \in J,J \)

Case 1 a: \( V = J \)

Show
\[ \forall(V \in R) \begin{cases} \\
(W = J) \land R'[J] \land R'[W] \\
\end{cases} \land \\
(V \in R) \begin{cases} \\
W \in J \text{ THEN } W = J \text{ THEN } R[I] \\
\text{ELSE } R[J] \text{ ELSE } R[W] \\
\end{cases} \land \\
\forall(W \in R) \begin{cases} \\
(W = J) \land (W = J) \land (R[I] \neq R[I]) \land \\
\end{cases} \land \\
\neg (W \in J,J) \land (R[I] \neq R[W]) \\

and all the cases work out so the quantification holds. The
remaining cases are similar.

COMMENT: Case analysis is as much required here as in the problems
discussed by King[11]. However, there is the possibility for
"packaging" some properties of this common programming construct,
interchange of elements of a vector. It involves the notion of
permutation: if the vector of subscripts to the left is a permutation
of the vector of subscripts to the right and all subscripts occur
uniquely, then the result of the assignment is a permutation of the
elements of the expression being assigned. Then it would be possible
to prove that there are certain properties, such as uniqueness, which
are preserved under permutation.
EXAMPLE 8: POSTFIX TRANSLATION

This example is included to illustrate some difficulties that can arise in stating the correctness of procedures which are most naturally defined recursively. The problem is to verify a program which translates a syntactically correct string in an infix expression language to its postfix form. An APL function which does this is given below. ALPH is a string of characters to be used as variables. PREC is a function which determines the relative precedence of two operators, and PARENS is a function which determines if a pair of operators are matching parentheses.

Let S be a global variable containing the string to be translated.
The functions are written in recursive conditional expression form for clarity and brevity.

\[
\text{POST}(S;I;Z;ST) \leftrightarrow
\begin{align*}
\text{IF } I = S & \text{ THEN } Z, ST \\
\text{ELSE IF } S[I] \in \text{ALPH} & \text{ THEN } \text{POST}(S;I+1;Z,S[I];ST) \\
& \text{ELSE POP}(S;I;Z;ST)
\end{align*}
\]

\[
\text{POP}(S;I;Z;ST) \leftrightarrow
\begin{align*}
\text{IF } \text{PREC}(S[I];ST[0]) & \text{ THEN } \text{POST}(S;I+1;Z,S[I];ST) \\
& \text{ELSE IF } \text{PARENS}(S[I];ST[0]) \text{ THEN } \text{POST}(S;I+1;Z;ST[0];1\downarrow ST) \\
& \text{ELSE POP}(S;I;Z;ST[0];1\downarrow ST)
\end{align*}
\]

\[
\text{POSTFIXEVAL}(S;I;ST) \leftrightarrow
\begin{align*}
\text{IF } I = S & \text{ THEN } S \\
\text{ELSE IF } S[I] \in \text{ALPH} & \text{ THEN } \text{POSTFIXEVAL}(S;I+1;S[I];ST) \\
& \text{ELSE POSTFIXEVAL}(S;I+1;\text{OP}(S[I];ST[1];ST[0]);2\downarrow ST)
\end{align*}
\]

where \( \text{OP}(\text{RATOR};\text{RAN}D;2;\text{RAN}D;1) \) evaluates the dyadic operator.

A correctness statement for this problem is "the postfix string when evaluated by a postfix interpreter produces the same result as the infix string when evaluated by an infix interpreter." In other words,

\[
\text{POSTFIXEVAL} \cdot \text{POSTFIXTRANSLATION} \leftrightarrow \text{INFIXEVAL}.
\]

There are at least two choices for induction:

1. Show that the sequence of operator applications is the same for both interpreters.

2. Show that the stacks used by both interpreters are in correspondence throughout the interpretations.
For the second choice the correctness statement is

\[ \text{POSTFIXVAL(POST(S;0;"";0;")\rightarrow INFIXEVAL(S;0;"";0;")} \]

and the induction statement is

\[ \text{POSTFIXEVAL(ZI;0;")\rightarrow INFIXOPERANDSTACK} \]
\[ \text{STI\rightarrow INFIXOPERATORSTACK} \]

where the variables \( STI \) and \( ZI \) are the values of the parameters of the postfix translation at the \( i \)th step.

The details of these proofs will not be carried out since the basic technique is simply symbol manipulation. However, the nature of the proof is of interest since it falls into the category of "parallel induction" as discussed by Wegner[WE] and Manna[MNV].

The basic problem here is that there is no nice way of giving an inductive assertion for the above program without defining the interpreter functions and once the interpreter functions have been defined parallel induction seems to be a more natural proof technique.
APPENDIX E
NOTATION AND CONVENTIONS

This manuscript was prepared using the XCRIBL system developed at Carnegie-Mellon University. The output device is an LDX, a Xerox graphic printer, which is controlled by a PDP-11. Files were prepared on the PDP-10 using XOFF, a version of DEC's RUNOFF with special commands for LDX control. The APL character set was developed especially for this thesis and some characters do not look exactly like those produced from the Selectric type ball. Several conventions are used with respect to the two type faces:

1. Since underlining was impractical, capitalization has been used for emphasis in some places. Also terms are capitalized in the context of their definition.

2. APL is used both as a programming language and as meta-notation throughout the thesis. Wherever the APL or italic type face is used, the APL and meta-notation rules are in effect.

3. Subscripts are written linearly. For example L₂ is written L₁ and L_{k-1} is written L(k-1).

4. Output from the constraint verification system is on a TTY. This forced the use of keywords for many APL symbols as given in table E.1

META-NOTATION

Meta-notation has been developed for the statement of verification conditions, the definition of the operators, and the assertion language. All meta-operators have lower precedence than the APL operators.

Identity Operators

= is reserved as the APL scalar operator

<> denotes that the right hand side is the defining expression for the symbols on the left, i.e. the left hand side is an abl! evaluation for the right hand side.

:\ denotes the equivalence of two expressions within the deductive system and the rules of APL evaluation.
APPENDIX E
NOTATION AND CONVENTIONS

True and False
1 and 0 are used for true and false, respectively. When \( B \) is an expression \( B=1 \) means "\( B \) is true" and \( B=0 \) means "\( B \) is false".

Conditional expressions

\[
\text{IF } A \text{ THEN } B \text{ ELSE } C \iff \begin{cases} B & \text{IF } A \leftarrow 1, \\ C & \text{IF } A \leftarrow 0, \\ \text{UNDEFINED, OTHERWISE.} & \end{cases}
\]

\( \iff \) is used to avoid some difficulties with partial operators. \( A \iff B \) means "if \( A \leftarrow 1 \) then \( B \) else true" in the sequential sense. The truth table is (where \( U \) means undefined)

\[
\begin{array}{c|cccc}
A \mid B & T & F & U \\
\hline
T & T & F & U \\
F & T & T & T \\
U & U & U & U \\
\end{array}
\]

The operator use is restricted throughout so that \( A \) is never undefined. The case of interest is where \( A \) is false and \( B \) is undefined and the sequential evaluation prevents consideration of \( B \).

Simple-indexing

\[
A[V] \iff \text{IF } I=p, A \text{ THEN } (A)[0] \text{ ELSE}
\begin{cases}
\text{IF } ((p=0) \land (0(V) \land I<pV) \lor (0(V) \land I<pV)) \land (V[I]<pA[I]) \text{ THEN} \\
(A)[((pA)1V)] \\
\text{ELSE UNDEFINED}
\end{cases}
\]

Slicing

\[
\text{IF } I=p, A \text{ THEN } A[(B)V] \iff (A)[0] \text{ ELSE}
\begin{cases}
pA[(B)V] \iff (pA)[B] \\
A[(B)V][I] \iff A[\text{INSERT}(B; V; I)] \text{ FOR ALL} \\
\text{ALL INTEGERS } I \text{ SUCH THAT } (0[I]<pA)[B]
\end{cases}
\]

Functional notation

In the three list operations below and the constraints a functional notation \( f(...) \) is used. Arguments are separated by semi-colons.

\[
\text{INSERT}(A; B; C) \iff B[0], \ldots, B[A-1], I[B[A]], \ldots, B[-1+pV] \\
\text{SUBST}(A; B; C) \iff B[0], \ldots, B[A-1], C[B[A+1]], \ldots, B[-1+pPB] \\
\text{DELETE}(A; B) \iff B[0], \ldots, B[A-1], B[A+1], \ldots, B[-1+pPB]
\]

\( ^{E1} \)

\(^{\land} \) is used as a prefix operator to denote a conjunction \( E_1 \land E_2 \land \ldots \)

The subscript range is omitted.
### Table E.1
TELETYPE OPERATOR CODES

<table>
<thead>
<tr>
<th>APL</th>
<th>TTY</th>
<th>APL</th>
<th>TTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.DE</td>
<td>.</td>
<td>.EN</td>
</tr>
<tr>
<td>l</td>
<td>.FL</td>
<td>l</td>
<td>.CE</td>
</tr>
<tr>
<td>c</td>
<td>.EP</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>'</td>
<td>.IO</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>p</td>
<td>.AB</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>*</td>
<td>.RO</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>~</td>
<td>.LO</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>↓</td>
<td>.NT</td>
<td>↑</td>
<td>.</td>
</tr>
<tr>
<td>≥</td>
<td>.DA</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>≤</td>
<td>.GE</td>
<td>≤</td>
<td>.</td>
</tr>
<tr>
<td>≠</td>
<td>.NE</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>V</td>
<td>.NG</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>A</td>
<td>.OR</td>
<td>^</td>
<td>&amp;</td>
</tr>
<tr>
<td>X</td>
<td>.LA</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>#</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.LG</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


