1979

Type determination in an optimizing compiler for APL

Robert D. Roeder

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation

Roeder, Robert D., "Type determination in an optimizing compiler for APL " (1979). Retrospective Theses and Dissertations. 7243.
https://lib.dr.iastate.edu/rtd/7243

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
ROEDER, ROBERT D.
TYPE DETERMINATION IN AN OPTIMIZING COMPILER FOR APL.

IOWA STATE UNIVERSITY, PH.D., 1979
Type determination
in an optimizing compiler for APL

by

Robert D. Roeder

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

Signature was redacted for privacy.
In Charge of Major Work

Signature was redacted for privacy.
For the Major Department

Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa

1979
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER II. A DATA BASE OF APL PROGRAMS</td>
<td>6</td>
</tr>
<tr>
<td>CHAPTER III. TECHNIQUES FOR APL TYPE DETERMINATION</td>
<td>11</td>
</tr>
<tr>
<td>CHAPTER IV. RESULTS</td>
<td>29</td>
</tr>
<tr>
<td>CHAPTER V. APPLICATION OF RESULTS</td>
<td>46</td>
</tr>
<tr>
<td>CHAPTER VI. CONCLUSION</td>
<td>59</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>62</td>
</tr>
</tbody>
</table>
CHAPTER I. INTRODUCTION

The Problem Area

The intent of this research is to evaluate type determination as a code optimization technique in a compiler for the programming language APL. The purpose of an optimizing compiler is the generation of an efficient target program - one which will run faster than and/or take less space than unoptimized object code. The criteria for assessing an optimization method are the cost of its implementation and the benefits it produces.

Code optimization is desirable for very high level languages, such as APL, which are designed to facilitate program development, rather than execution efficiency. If the execution time of very high level language programs could be sufficiently improved, then it becomes possible to capitalize on the programming convenience offered by the language. Foremost among APL features which sacrifice run time efficiency for programming flexibility is the absence of type declarations.

Type determination may be viewed as an attempt to supply type declarations for variables appearing in a program. A type declaration binds a variable to specified data attributes at compile time. This static fixing of type can bring about an increase in the execution speed of the target program in the areas of static type checking, type specific code, and storage management.
The goal of type determination is the derivation of information about the type attributes of program variables. The characteristics of a language which influence type determination are the data types it supports and the semantics of its operations with respect to those data types. The language definition specifies type rules for operations, which may place restrictions on acceptable operand types and designate a result type, both of which may enable type attributes to be deduced. The amount of type information obtained is measured by both the quantity and precision of the inferred attributes.

Efforts to enhance the performance of a programming language implementation must take into account the properties of the programs it is expected to process. For an existing language, compiler design may include an examination of a typical set of programs written in the language, which can suggest those language features conducive to optimization attempts. The results to be generated will utilize a collection of APL programs to direct the development of a type inference scheme for APL and to measure its effectiveness.

Problem Specification

The basic problem undertaken in this research is an appraisal of methodology for inferring the type attributes of quantities appearing in an APL program. Type analysis may be applied at three APL program levels: (1) a single statement (individual line), (2) a defined function (sequence of statements), and (3) a workspace (group of defined functions).
To discover the role of each of these levels in the derivation of type information, a type determination phase in an APL translator will be implemented for each of the three levels. An assemblage of APL workspaces will then be submitted to this phase so that the contribution of each level to the acquisition of type information can be empirically ascertained.

The results generated from the implementation will then be analyzed with regard to their applicability to code improvement. The extent of optimization achievable measured against the associated implementation cost will indicate the practicality of type determination as an optimization technique for APL.

Literature Review

Aspects of compiler construction may be found in Gries (26). Pratt (40) and Nicholls (36) both discuss the role of type and declarations programming language design. Code optimization techniques are described in Aho and Ullman (3, 4). Optimization of very high level languages is the subject of work by Schwartz (44, 45) with the SETL language. Details of an optimizing compiler for the language BLISS are given in Wulf, et al. (54).

Kaplan and Ullman (32) developed a general algorithm for type inference in a model of a typeless language. Jones and Muchnick (31) also developed a general scheme for type determination, though their work was directed toward the language TEMPO. Miller (35) applied the work of Kaplan and Ullman to type checking. Type determination for the SETL language was studied by Tenenbaum (52, 53) and Sharir (47).
Bauer and Saal (10) estimated the percentage of run time checking which could be eliminated by a static line-by-line analysis of APL programs.

Program flow analysis is an accessory to methods for type determination. Hecht (28) surveys strategies for both interprocedural and intraprocedural data flow analysis. The implementation of a program flow analysis strategy is detailed in Allen and Cocke (7). Allen (5, 6), Schwartz (46), and Barth (9) discuss interprocedural analysis. Flow analysis to assist software reliability is covered in Fosdick and Osterweil (21).

Study of language utilization as an aid to compiler design was pioneered by Knuth (33) in his analysis of FORTRAN programs. Robinson and Torsun (41) did an additional study on the use of FORTRAN. Bingham (11, 12) and Saal and Weiss (42) have examined characteristics of APL programs. PL/I has been the target of studies by Elshoff (16, 17, 18), while features of COBOL usage have been reported by Chevance and Heidet (14) and Salvadori, Gordon, and Capstick (43).

The original definition of APL can be found in Iverson (29). Texts dealing with the programming language APL are Polivka and Pakin (39) and Geller and Freedman (23). Some better known APL implementations are APL\360 (20, 38), APLSV (19), and York APL (48). Formal descriptions of APL have taken several forms: a primitive recursive semantics by Orgass (37), a description for program verification by Gerhart (25), and a description in APL by Lathwell and Mezei (34).
Proposals for optimization of APL have considered both hardware and software design. Abrams (1) designed a machine for APL which would make use of delayed evaluation of certain operations. Guibas and Wyatt (27) studied delayed evaluation for the compilation of APL programs. Ashcroft (8) imposed restrictions on the manner in which language features could be used as a means of facilitating the development of an APL compiler. Jenkins (30) also placed restrictions on language usage in order to estimate the gain in execution speed made possible by the availability of type information at compile time. Sykes (51) suggested how programs could be written more efficiently. Strawn (49) devised a method for the compile time parsing of the inherently ambiguous expression language for APL, a prerequisite for the implementation of many of the preceding optimizations.
CHAPTER II. A DATA BASE OF APL PROGRAMS

Introduction

A first step toward the evaluation of type determination as an optimization technique for APL was to examine a collection of APL programs; as Saal and Weiss (42) indicate, an attempt to "optimize the performance of an implementation should be aware of the nature of the programs with which it must deal." For type determination, this would involve analyzing those language features affecting type, in an effort to decide where effort should be expended in the accumulation of type information.

Two instruments were needed to obtain measurements relative to type determination for APL: a set of APL programs from which statistics could be accumulated, and an APL translator which could process the set of programs so that the statistics of interest could be gathered. The following section will describe the available tools and how they were employed to generate the desired results.

Generation of the Data Base

ISUAPL is a batch system for APL which contains most of the standard features of APL\360 and APLSV. (Those omitted deal with interactive use.) The compilation process in ISUAPL has two phases: lexical analysis and parsing/code generation. The generated code is referred to as pseudo-code, which takes the form of the instruction set for a stack APL machine. This pseudo-code is then processed by a software interpreter to effect program execution.
Because of the inherent ambiguity of the APL expression language, all statements cannot be completely parsed at compile time. ISUAPL accommodates the occurrence of an ambiguous situation by generating pseudo-code which will allow the interpreter to resolve the ambiguity and execute the correct code sequence.

APL/PIE, the APL Program and Information Exchange, was chartered at the State University of New York at Binghampton for the interchange of APL programs among the user community. One of the services provided by APL/PIE is the distribution of contributed workspaces, so a tape of all workspaces maintained by APL/PIE as of June 23, 1977 was obtained.

There were 147 workspaces on the tape, stored in an API/360 internal format. A program was written to transform this format into ISUAPL source code, so all 147 workspaces were submitted to it for conversion. One hundred thirty of the workspaces were successfully converted; the other 17 raised error conditions in the conversion program, and thus had to be discarded. Inspection of the source code for these 130 workspaces revealed some duplication. A number of them were made up of most if not all of the same defined functions. There were 51 duplicated workspaces, which were then eliminated, leaving 79 workspaces in the data base.

These 79 workspaces were then compiled under ISUAPL, which resulted in 57 parse errors. Of these, eight were due to the use of APLSV system variables, which were changed to appear as regular APL variables. The other 49 parse errors had an assortment of causes: duplicate labels within a defined function, missing parentheses, extraneous operators in expressions, assignments without target variables, and parameters
declared as local variables. Appropriate corrections were inserted, and the errors were thus rectified.

Examination of the successfully parsed workspaces disclosed several which were made up solely of character string constants that served to describe the contents of other workspaces. Also, within many of the workspaces were defined functions made up of character string constants which served to describe the contents of the workspace or the functions in it. (Many of these functions were standardly named DESCRIBE.) These functions performed no computation, so they were of no interest from the standpoint of optimization. There were 4 workspaces and 85 defined functions thus considered as documentation, so these were removed from the data base.

There were a total of 75 workspaces with 914 defined functions and 11,340 APL statements, other than function header lines, in the data base. The average number of functions per workspace was 12.2, and the average number of lines per function was 12.4. The 11,340 statements generated 104,220 ISUAPL pseudo-code instructions. A detailed description of the data base contents may be found in (13), in which comparisons with other studies of APL programs (11, 12, 42) establish the "present sample of programs as being truly representative of APL programs in general."

Ambiguity Resolution

Analysis of the data base was contingent on the removal of ambiguity in the pseudo code. Details of the nature of ambiguity in
APL may be found in (49); the topic will be discussed here in general terms so that a description of the method employed to resolve ambiguity in the pseudo-code may be given.

Ambiguity may arise in an APL expression because of the inherent ambiguity of the syntax and the dynamic binding strategy of the language. When an identifier declared as a local variable of a defined function is also the name of a defined function, a reference to that identifier may be bound to either the variable or the defined function. The calling chain in effect at run time decides the correct binding, so in the absence of any additional information, the expression containing such a reference cannot be parsed at compile time.

The additional information deemed sufficient for compile time ambiguity resolution was a list of the names of all defined functions within a workspace, along with their type - number of arguments accepted (zero, one, or two: in APL terms, a niladic, monadic, or dyadic function, respectively) and number of values returned (zero or one). The function type was needed because the syntax of a potentially ambiguous expression requires a specific function type for the reference to be actually ambiguous.

The strategy developed for ambiguity elimination was to compile and maintain a list of all defined functions within a workspace. Next, a pass through the pseudo-code for each function in the workspace was made to detect if any ambiguity was present. This process was then applied to all 75 workspaces in the data base.
There were a total of 3,019 ambiguities encountered in the pseudo-code for the workspaces. Of these, 2,945 involved a reference to an APL primitive function and 74 contained a reference to a defined function. In the former case, 2,920 of the references (99%) were resolved as a reference to the dyadic form of the primitive function: the other 25 required the monadic form of the primitive function. In the latter case, 55 of the references (74%) called for a dyadic defined function; the other 19 invoked a monadic defined function.

There were nine instances, appearing in five of the workspaces, of a local variable and a defined function of the same name. None of these references was ambiguous, however. The syntax of the expression in which the identifier was referenced required the function to be monadic and value-returning, and the function of that name was not. Hence, the entire data base was free of any actual ambiguity, so the pseudo-code could be altered to reflect the correct identifier binding.

The removal of ambiguity from the pseudo-code completed the creation of a data base suited to the gathering of type determination statistics. The pseudo-code for each workspace was generated; then a scan of the pseudo-code was made to resolve any ambiguities. The unambiguous pseudo-code was then processed, and the data of interest was recorded in a format which would facilitate its analysis. All of the data to be presented was acquired in this fashion.
CHAPTER III. TECHNIQUES FOR APL TYPE DETERMINATION

An overview of the APL language will now be presented. Emphasis will be placed on those language features bearing upon type determination, along with the introduction of APL terminology. Details of the language and its implementation may be found in (19, 20, 38, 39, 48).

Atomic data types in APL are single real numbers and individual characters, each of which is referred to as a scalar value. Integers are embedded within the real numbers, while Boolean values are represented by the integers 0 (false) and 1 (true). The only data structure is the homogeneous (either entirely numeric or entirely character) array, which may have any number of dimensions. An array is characterized by its rank, which is the number of dimensions, and its shape, which is the number of elements along each dimension. The lower bound on subscript ranges may be either zero or one.

APL has a large number of primitive operations, most of which can accept an entire array as an argument and can produce an array as a result. Each primitive operation takes the form of a function which will accept either one argument (written in prefix form) or two arguments (written in infix form), and will return a value. The symbols for primitive functions play a dual role in that the same symbol may represent both a monadic function and a dyadic function, depending upon the syntax of the statement in which it occurs.

The APL primitive functions are grouped into three classes: scalar functions, composite functions, and mixed functions. In general,
the result of a scalar operation takes its shape from the argument(s) it is supplied with, a mixed function yields a value whose shape is unrelated to that of its argument(s), and a composite operation is an extension to an array of a dyadic scalar function, and thus derives its shape requirements from those of the dyadic scalar function.

The result of a monadic scalar function is the same as that of its argument. While the result of a dyadic scalar primitive is governed by three rules: (1) if both arguments are scalar, the result is a scalar; (2) a scalar argument and an array argument yields an array, with the scalar extended to each element of the array; and (3) two array arguments deliver an array result, if they are conformable (have the same shape): the result has that common shape.

An APL statement is formed as an expression, which may contain primitive and defined functions, along with their arguments. A defined function originates with a header line, followed by a sequence of statements. The header line designates the function name, the names of the formal parameters (there may be zero, one, or two), whether or not the function returns a result, and a list of identifiers which are to be local variables of the function. Parameter passage is by value, with parameters treated as local variables of the called function. A function may be defined recursively. Referencing of nonlocal variables obeys the most recent association rule, meaning the calling chain is searched at run time to resolve the reference.

The sole means of altering the flow of control between the statements of a defined function is the branch operation, whose operand
expression must evaluate to either a label or a line number. Branching to a line number not existing in the defined function is treated as a function return. Flow of control between functions is by function call, which takes place whenever the function name appears in an expression.

Type Determination for APL

Implementation of type determination for APL was guided by established techniques for type inference (10, 31, 32, 35, 52, 53) and the characterization of APL programs and workspaces provided by the data base. The fundamental concept of type determination is that operations of a language enable type attributes of variables to be deduced. The use of a variable refers to its appearance as an operand of a language operation, while the assignment of a value to a variable is referred to as its definition. For a use of a variable, the restrictions on allowable argument types to an operation may permit type attributes to be deduced; for a definition, the type of the value to be assigned to a variable may allow its type attribute to be deduced.

Type attributes of variables may be derived from definitions and uses by two different methods: forward inferences and backward inferences. Forward inferences deduce (1) the type of the result of applying an operation to its argument(s), from the type semantics of the operation and (2) the type of an argument to an operation, from the restrictions imposed on the argument by the operation. Backward
inferences deduce type information about arguments to an operation from previously obtained information about the result type of the operation. The term operation applies to a language feature which either accepts an argument or yields a value. For APL, operations include primitive functions, defined functions, assignment, array accessing, branching, input and output.

As an example of type inference, consider the following sequence of statements.

\[ A + B + 1 \]
\[ C + ?A \]

From the first statement, the forward inferences are that \( A \) and \( B \) must be numeric, since addition requires a numeric argument and yields a numeric result. The forward inferences in the second statement are that \( A \) and \( C \) must be integers, since \( ? \) requires an integer argument and produces an integer result. (\( ? N \) for positive integer \( N \) is an integer pseudo-randomly selected from the integers 1 through \( N \).) The backward inference from this sequence is that \( B \) must be an integer: \( A \) is known to be an integer from the second statement, so it must be an integer in the first statement (from which it received its value). Given \( A \) is an integer in the first statement, then it can be inferred that \( B \) is an integer, since an integer result was produced by adding the integer 1 to it. While only domain was typed in this example, the inference methods are applicable to the deduction of rank and shape information also.
The above example shows that backward inferences involve more than a single statement to produce information about type. The additional context needed for backward inferences is the basic block (5, 29), a maximal grouping of program statements into sequentially executed units. Such a grouping occurs within a defined function in APL, which suggests that an intraprocedural level of typing would make supplemental type information available.

In the statement sequence

\[ A \leftarrow 1.5 \]
\[ B \leftarrow A \]
\[ A \leftarrow 1 \]

The use of A in the second statement is said to be chained to the definition of A in the first statement, since it is the value assigned in the first statement that is being used in the second statement. Type information from all uses chained to a particular definition may be combined to augment the type information for a variable. [This combining process is formally developed in the framework of lattice theory in (32)].

Multiple definitions of a variable act to complicate chaining. For sequential flow of control, a use is chained to its latest preceding definition. In the sequence of statements above, the use of A in the second statement is chained to the real value (1.5) assigned to A in the first statement, whereas any uses of A following the third statement will be chained to the integer value 1.
The branch operation can produce nonsequential flow of control in a defined function. If, after the above statements, a subsequent statement transferred control to the second statement, then the integer value of A, as well as the real value, could be chained to the use of A. The type information for A would then have to be combined from both definitions. Thus, intraprocedural type analysis hinges on the flow of control within a defined function.

An additional consideration for type determination is the interaction between defined functions within a workspace. Methods for interprocedural flow analysis (5, 6, 9, 46) account for the effects of the invocation of a defined function. The issuance of a function call may (1) define and/or use variables in the calling function, (2) pass in parameters which can contribute to type inference in the called function, and (3) return a value whose type attributes are known and available to the calling function.

In the sequence of statements

\[ A \rightarrow B \rightarrow F2 \rightarrow C \]

\[ D \rightarrow E \]

the call to function F2 (syntactically dyadic and value-returning) has the following ramifications for type analysis: (1) type information about parameters B and C may assist the derivation of type attributes in the called function, F2; on the other hand, the attributes of B and C may not be known at the time of the call, but may be inferred from usage in F2; (2) the type of the returned value may be deduced in F2, so the type of A after the call may be known; and (3) if E is a
global variable (accessible to F2), then its type in the second statement may be determined by F2, which in turn may determine the type of D. Such deductions are predicated on information about the calling relationships among functions and the scope of variables.

Because of the dynamic binding strategy of APL, variable referencing depends on the run time calling chain. In the functions below,

\[
\begin{align*}
\Delta F & \quad \Delta G; X \\
X & \leftarrow 0 \quad X & \leftarrow 1 \\
\Delta & \quad \Delta \\
H & \quad H \\
\Delta & \quad \Delta
\end{align*}
\]

there are two distinct variables named X: a global (not declared) variable X referenced in F, and a local variable X of function G. The reference to X in H can be to either of these: if the call to H in F is executed, the reference in H is to the global X, but when the call to G in F is executed, the reference in H is to the local variable X of G. The flow of control in G may be such that it is not possible to determine at compile time whether or not the call to H will be executed, which means the binding of X in H cannot be decided.

Characteristics of APL Programs Affecting Type Determination

The discussion of type determination methodology identified the factors involved in deducing type. The influence of each of these factors is dependent on the extent of its utilization. For
example, multiple definitions of a variable were observed to complicate chaining, but if no APL program contained a variable with more than one definition, chaining would present little problem. The following sections will deal with measures of APL programs which influence typing: definitions and uses, branching, scope, and calling relationships.

Definitions and Uses

Since definitions and uses are the basis for inferring type, their frequency of occurrence in the data base was measured. To further characterize this information, the scope of each variable within a function as distinguished by the compiler was recorded: local (declared in the function), nonlocal (not declared in the function), parameter (argument to the function), and return (value to be returned by the function). In the tables to follow, the abbreviations L, N, P, and R, respectively, are used for these scope categories.

Table 1 gives the distribution of occurrences of definitions. A prominent figure is the number of definitions per variable; on the average, there were 1.3 definitions per variable, and 88% of the variables had two or less definitions. From the standpoint of definition-use chaining, the percentage of single-definition variables (39.8) is significant, for all uses are chained to that single definition.

Scope must be taken into account when interpreting these figures, however. For nonlocals, the number of definitions means on a per function basis, so nonlocals are counted as separate variables in each function in which they occur. The data for the other scope
categories do reflect the actual number of definitions quite accurately. (This will be clarified when a more precise characterization of scope is given in an ensuing section.)

When scope is considered, the items of note are the single-definition local and return variables, and the zero-definition nonlocals and parameters. Fifty-four per cent of the local variables have one definition, so all uses may be chained to that definition. Fifty-six per cent of the return variables have a single definition;
Such a variable will return a single result type to its calling environment. The zero-definition nonlocals (54% of all nonlocals) are assumed to receive a value from another environment. The parameters not assigned to (68% of all parameters) are utilized as read-only values (which is to be expected, because they are passed by value), and not used as local variables within the function.

Table 2 shows the distribution of variable uses. There was an average of 3.2 uses per variable; 61% of the variables had two or less uses, and 73% had three or less. An "average" variable could thus be portrayed as having three uses to be chained to a single definition, which suggests that uses might contribute more than definitions in determining type.

When scope is considered for uses, two figures appear noteworthy. Forty-four per cent of the nonlocals have only a single use, which may suggest they are being used as input parameters to a function. This could be due to the restriction of two parameters passed to a function, as Saal and Weiss (42) have mentioned. Of the return variables, 63% are never used, indicating that they are just used to return a value, and not as local variables.

Flow of Control

Intraprocedural type determination needs to analyze the sequence of statement execution in order to correctly chain definitions and uses. The mechanism for altering the execution sequence within a defined function in APL is the branch operation, so its utilization
Table 2. Distribution of number of uses of a variable.

<table>
<thead>
<tr>
<th>Uses</th>
<th>Variables</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>679</td>
<td>11.7</td>
<td>125</td>
<td>240</td>
<td>1</td>
<td>363</td>
</tr>
<tr>
<td>1</td>
<td>1854</td>
<td>32.1</td>
<td>535</td>
<td>1008</td>
<td>236</td>
<td>75</td>
</tr>
<tr>
<td>2</td>
<td>974</td>
<td>16.9</td>
<td>392</td>
<td>374</td>
<td>176</td>
<td>59</td>
</tr>
<tr>
<td>3</td>
<td>701</td>
<td>12.1</td>
<td>271</td>
<td>206</td>
<td>132</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>432</td>
<td>7.5</td>
<td>199</td>
<td>143</td>
<td>82</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>248</td>
<td>4.3</td>
<td>99</td>
<td>86</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>216</td>
<td>3.7</td>
<td>115</td>
<td>67</td>
<td>26</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>147</td>
<td>2.5</td>
<td>66</td>
<td>38</td>
<td>35</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>91</td>
<td>1.6</td>
<td>41</td>
<td>24</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>72</td>
<td>1.2</td>
<td>45</td>
<td>20</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>&gt;9</td>
<td>393</td>
<td>6.8</td>
<td>203</td>
<td>98</td>
<td>91</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>18211</td>
<td></td>
<td>2031</td>
<td>2304</td>
<td>867</td>
<td>576</td>
</tr>
</tbody>
</table>

was studied. There were a total of 3732 branches in the workspaces (out of 11,340 lines), which means about one out of every three lines (32.9%) could result in a branch. Flow of control depends on the target of the branch, so arguments to the branch operation were examined.

Inspection of the branch statements exposed a variety of usage. Many branch targets could be readily discerned, as they contained either only a label or constant, or an expression containing one of
these in conjunction with a stylized usage. [Saal and Weiss (42) describe stylized branching.] However, there was a noticeable number of statements which had unclear means for forming a branch target. Some of these targets were not known at compile time, such as branching to the result of a function call, while others were part of a (usually complex) expression which used "clever" coding.

This was exemplified by a particular workspace, which consisted of a single function of only one statement: that statement contained a branch, along with five assignment operations, eight array accesses, references to 12 variables, and 20 uses of APL primitive functions. Thus, it was difficult to accurately measure the extent to which branch targets were known: an estimate would be 90%, in agreement with a figure reported by Saal and Weiss.

Based on the examination of branch statements, two conclusions were reached regarding the flow of control in APL programs. First, because of the frequency of branching (every third statement, on the average), there would be few statements within a sequentially executed group (basic block). Secondly, if a branch statement target was unknown, it could be treated as a transfer of control to any statement in the program: this would equate single statements with basic blocks. Both of these factors concurred with the statement of Bauer and Saal (10) that flow analysis was "feasible but the expected benefits seemed small compared to the effort entailed." Thus, it was felt that an alternative to flow analysis was in order, owing to the nature of branch operator usage in APL.
Consequently, a scheme for handling definition-use chaining in the absence of flow of control information was sought. The foremost issue was the occurrence of multiple definitions of a variable. If all of the definitions were of the same type, all uses could be chained to that type, but if the definitions were of differing types, flow analysis would be necessary for chaining.

The distribution of the number of definitions of a variable offers evidence that the likelihood of incompatible definitions is small, because variables average less than two definitions. Also, all uses of single-definition variables (which occur frequently) can be chained to that definition. As further evidence of compatibility, a speculation supported by examination of APL programs will be offered: many two-definition variables do not have their type changed by redefinition, as typified by a reassignment of the form \( A + A + 1 \).

Thus, it was assumed that all definitions and uses could be chained, that is, information gathered from all references to a variable could be combined to determine its type attributes. This technique would incorrectly chain a use to a definition only if multiple definitions assigned different type attributes to a variable; otherwise, chaining could take place since the attributes involved were compatible. All definitions for a variable would be checked to detect if type attributes were changed; if so, type inferences for that variable would be invalidated. The overall effect was to presume a strictly sequential flow of control, thereby ignoring the actual flow of control, which possibly was not determinable.
Calling Structure

The scope of variables in APL influences type determination in that an identifier may have different bindings in different defined functions. A reference to a variable in an APL workspace can be classified as either local, global, or nonlocal. A local reference within a defined function is bound to an identifier declared in that function. A global reference in a defined function is bound to a variable that is either not declared in the workspace or has no declaration in the run time calling chain. A nonlocal reference within a defined function is bound to a variable that is declared within another defined function which is part of the calling chain in effect at run time.

The frequency distribution of the three categories of variable references was accumulated for all workspaces in the data base. This compile time measure contained some chance of error, since scope may depend on the run time structure of the chain of function calls. To include all possible nonlocal references, it was assumed that any calling chain obtainable from the call structure of the workspace would exist at the time a variable was referenced. This could cause some global references to be considered as nonlocal, since the presence of a declared variable in a possible calling chain would always be counted as a nonlocal reference. This was done even though the reference would actually be global if that calling chain was not present when the reference was made.
For the potentially nonlocal references, the nesting depth of the calling chain from the reference to its declaration was recorded. For example, if F called G and a nonlocal reference in G was declared in F, it was considered one level deep in the calling chain. If F called G which in turn called H, and H contained a nonlocal reference to a variable declared in F, the reference was considered two levels deep in the calling chain. (In this latter case, if F could also call H, the reference was recorded as one level deep, so a nonlocal reference was counted only once and at the nearest level.)

Tables 3 and 4 display the scope analysis data. It can be seen from Table 3 that most local variables (74%) are "strictly" local. [Bauer and Saal (10) mention this as being useful for static APL analysis.] Consequently, there are few nonlocal references, and most of these (68%) are only one level distant in the calling chain. As for the other nonlocal references at deeper nesting levels, recall that these are only potentially that deeply nested, because the calling chain at that level may not be present when the reference is encountered. The scope analysis data thus indicates that variable references are for the most part either (strictly) local or global.

In conjunction with scope analysis, which required the structure of the calling chain, data was collected on the issuance of function calls. Eighteen of the workspaces contained no calls to defined functions; six of these workspaces consisted of only one function. There were a total of 1,379 function calls issued: 656 monadic, 514 dyadic, and 209 niladic.
Table 3. Scope distribution.

<table>
<thead>
<tr>
<th>Category</th>
<th>Occurrences</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>3347</td>
<td>74.1</td>
</tr>
<tr>
<td>Global</td>
<td>973</td>
<td>23.8</td>
</tr>
<tr>
<td>Nonlocal</td>
<td>95</td>
<td>2.1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4518</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Calling chain levels between nonlocal reference and declaration.

<table>
<thead>
<tr>
<th>Number of levels</th>
<th>Number of Nonlocal References</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65</td>
<td>68.4</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>13.7</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>15.8</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2.1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>95</strong></td>
<td></td>
</tr>
</tbody>
</table>

The distribution of calling functions and called functions is shown in Table 5. Approximately the same number of functions are either called or issue a call, but this number is less than half of the total number of functions. Also, approximately the same number of functions are either both called and issue calls, or are neither called nor issue any calls. This data suggests that calling
Table 5. Calling function and called function distribution.

<table>
<thead>
<tr>
<th>Category of functions</th>
<th>Number of Functions</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue calls</td>
<td>392</td>
<td>42.8</td>
</tr>
<tr>
<td>Invoked by call</td>
<td>403</td>
<td>44.0</td>
</tr>
<tr>
<td>Both call and called</td>
<td>161</td>
<td>17.6</td>
</tr>
<tr>
<td>Neither call nor called</td>
<td>132</td>
<td>14.4</td>
</tr>
</tbody>
</table>

Chains would not tend to be very deeply nested, since over half of the functions do not issue any calls.

The call graphs of function referencing relationships were analyzed for additional function call data. The call graphs of the workspaces are not all connected, as can be seen from the last row of Table 5. A total of 24 workspaces were connected; of these, 14 seemed to possess a certain regularity of structure. These 14 had a single (main) function which was never called, but did invoke other functions such that all functions could be reached from a calling chain emanating from this single function. One instance of direct recursion was found, and generation of all possible calling chains revealed no instances of indirect recursion in the workspaces.

Although the calling chain structure is not known at compile time, the call graph, which indicates potential calling chains, is. The supposition underlying the gathering of the scope data was employed so that interprocedural typing could proceed: any calling chain that could be formed from the call graph was assumed to be present.
The data previously given (Table 3) indicated that most variables (97.9%) can be statically scoped, so the type information for them will be valid regardless of the structure of run time calling chains. Variables whose scope is unknown at compile time can cause (1) less type information to be inferred for those globals which are assumed to be nonlocals, and (2) type information to be inferred for nonlocals which never are present in a calling chain, due to the treatment of a potential calling chain as an actual calling chain. These two causes would not seem to have a serious impact on interprocedural typing, due to the small number of variables (2.1%) possibly involved.

Summary

This chapter has described techniques that may be applied to the determination of type in APL. Measurement of language features which bear upon typing were obtained. These measurements suggested methods for overcoming some of the difficulties connected with type analysis of APL. The next step to be described is the implementation of these procedures to discover their effectiveness.
CHAPTER IV. RESULTS

Implementation

Type determination for APL was implemented as a phase of the ISUAPL translator. After the pseudo-code for a workspace was generated, the type determination module was initiated. Defined functions were processed in the order in which they appeared in the workspace; each defined function was processed sequentially, a line at a time.

As each operation was encountered, its type semantics were "executed:" operands were checked for proper type, and a result type was produced as specified by the type rules of the language definition. In conjunction with these actions, forward and backward inference procedures were applied to each variable operand of an operation. The type information thus acquired was entered in a descriptor maintained for each variable.

Statistics of Interest

In APL, the attributes of a variable are domain (atomic type), rank (number of dimensions), and shape (number of elements along each dimension). The domain attribute can be subdivided into character, real, integer, and Boolean. To assist type analysis, two additional types were provided for - general (attribute of variable not known) and varying (attribute of variable incompatible with previously inferred attribute). Rank and shape attributes are interrelated in that the rank of a variable is equal to the number of values in its shape. If shape is known, then so also is rank; in addition, the size (total
number of elements) is known, since it may be computed as the product of the shape values.

The scope of each variable was classified as either local, non-local, or global. (These categories are described in Chapter 1.) Local variables which were parameters and those which were returned values of a defined function were also distinguished. This information will aid in the interpretation of the results by providing a more complete profile of variable usage. (The abbreviations L, N, G, P, and R, respectively, are used for these scope categories in the tables to follow.)

Attribute and scope data for each variable was gathered for three hierarchical levels of type analysis: single line, defined function, and workspace. This would identify those factors responsible for supplying type information at each level. Also, a comparison of the results at each level would indicate the predominant influences on type determination.

Single Line Typing

The first level of type analysis was a single line of a defined function. Each appearance of a variable on a separate line was typed exclusive of references to that variable on other lines. Chaining of type information for a variable occurred only if it had multiple definitions and/or uses on the same line, in which case the information was merged.
An example of such a merging of type attributes would be a variable used both as an integer type and a real type on the same line. The domain attribute integer would be inferred in this instance: it is the more precise attribute, since it is included in the real type. Multiple definitions on a single line were checked to insure compatible attributes were assigned to the variable, in which case the variable typed as that compatible attribute; otherwise, the varying type was assigned.

Type analysis proceeded by examining each line of a defined function once, in sequential order. This may be viewed as assuming a worst case flow of control, as if every statement could transfer control to any other statement. For each line, type inferences were applied for each operation involving a variable. The actions of function calls were not analyzed, so global and nonlocal variables were not distinguished; the category nonlocal refers to both of these. Single line analysis was performed for each defined function in the data base.

Results of Single Line Typing

The results to be presented for single line typing count each occurrence of a variable on a separate line as a separate occurrence. Also, multiple definitions on the same line were counted separately. But all uses on a line referring to the same definition were merged, and this was counted as a single occurrence.

From Table 6, it can be seen that domain could be determined for 72.5% of variable occurrences. This figure is due mostly to the domain
Table 6. Distribution of domain attribute for single line typing.

<table>
<thead>
<tr>
<th>Type</th>
<th>Occurrences</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>8847</td>
<td>38.9</td>
<td>4485</td>
<td>2757</td>
<td>1144</td>
<td>461</td>
</tr>
<tr>
<td>General</td>
<td>6199</td>
<td>27.3</td>
<td>2212</td>
<td>1852</td>
<td>1343</td>
<td>792</td>
</tr>
<tr>
<td>Integer</td>
<td>4105</td>
<td>18.1</td>
<td>2772</td>
<td>867</td>
<td>258</td>
<td>208</td>
</tr>
<tr>
<td>Boolean</td>
<td>1878</td>
<td>8.3</td>
<td>1220</td>
<td>487</td>
<td>28</td>
<td>143</td>
</tr>
<tr>
<td>Character</td>
<td>1631</td>
<td>7.2</td>
<td>539</td>
<td>616</td>
<td>221</td>
<td>255</td>
</tr>
<tr>
<td>Varying</td>
<td>41</td>
<td>.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>22761</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

restrictions of both arguments to, and results produced by, APL operations: many operations accept and/or yield only numeric operands. This is exemplified by addition, which would type both the arguments and result as real (assuming no other information about them is known), since it is only defined for numeric values. Typing a variable as real does not necessarily mean it will take on a real value (for it may have integer values), but that the type real is all that may be inferred about the variable. The varying type represents multiple references to a variable on a single line which require differing (incompatible) attributes. The varying category does not indicate which attribute was changed at this level of analysis. Also, the scope of the variables which have incompatible attributes was not recorded.
With regard to scope, local variables (80%) and nonlocals (72%) are most often known, while parameters (55%) and return variables (57%) are least known. This could be anticipated at this level of typing: all type information about locals is confined to the function in which they occur, while parameters receive their type from the calling environment, and return values are used in the calling function.

Table 7 shows the distribution of variable ranks that were deduced by single line typing. (Rank zero is a scalar, rank one is a vector, and rank two is a matrix.) The rank attribute was inferred for 28.5% of variable occurrences. This figure can be accounted for by the rank rules of APL: few operations require a specific rank as an argument. Also, the result rank of an operation is often defined by the arguments to the operation, instead of being of fixed rank.

Vectors are the most frequent of the known ranks. This is probably due to the fact that most operations involving a fixed rank require a vector. As for scope, there is not much difference between the categories: nonlocal variables are slightly less known (25%) than are the other three categories (about 30% known). Parameter ranks show a preponderance of vectors and matrices (94% of known ranks), perhaps related to the restriction of at most two parameters to a defined function.

The size attribute, as given in Table 8, is known for only 10.3% of occurrences of variables. This figure is somewhat predictable, since even fewer APL operations require a specific size than do those requiring a specific rank. In addition, there are several operations
### Table 7. Distribution of rank attribute for single line typing.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Occurrences</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>16226</td>
<td>71.5</td>
<td>7919</td>
<td>4916</td>
<td>2136</td>
<td>1255</td>
</tr>
<tr>
<td>0</td>
<td>1411</td>
<td>6.2</td>
<td>834</td>
<td>408</td>
<td>49</td>
<td>120</td>
</tr>
<tr>
<td>1</td>
<td>4025</td>
<td>17.7</td>
<td>2249</td>
<td>906</td>
<td>529</td>
<td>341</td>
</tr>
<tr>
<td>2</td>
<td>1011</td>
<td>4.5</td>
<td>258</td>
<td>334</td>
<td>274</td>
<td>145</td>
</tr>
<tr>
<td>3</td>
<td>28</td>
<td>.1</td>
<td>0</td>
<td>20</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22701</td>
</tr>
</tbody>
</table>

### Table 8. Distribution of size attribute for single line typing.

<table>
<thead>
<tr>
<th>Size</th>
<th>Occurrences</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>20418</td>
<td>89.9</td>
<td>9899</td>
<td>5964</td>
<td>2925</td>
<td>1630</td>
</tr>
<tr>
<td>0</td>
<td>199</td>
<td>.9</td>
<td>62</td>
<td>64</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>1</td>
<td>1778</td>
<td>7.8</td>
<td>1127</td>
<td>442</td>
<td>63</td>
<td>146</td>
</tr>
<tr>
<td>2</td>
<td>36</td>
<td>.2</td>
<td>11</td>
<td>21</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>28</td>
<td>.1</td>
<td>16</td>
<td>10</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>.1</td>
<td>14</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>.1</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6-10</td>
<td>63</td>
<td>.3</td>
<td>43</td>
<td>15</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>11-20</td>
<td>68</td>
<td>.3</td>
<td>38</td>
<td>27</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>21-50</td>
<td>36</td>
<td>.2</td>
<td>20</td>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>&gt;50</td>
<td>41</td>
<td>.2</td>
<td>18</td>
<td>20</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22701</td>
</tr>
</tbody>
</table>
in APL which require a specific size but not a specific rank, but since size is defined in terms of rank (the product of the shape values), it was decided not to consider these as known sizes. Thus, if rank was unknown, then so also was size.

A noteworthy figure related to scope is the size attribute for parameters - 98% are of unknown size. The distribution of variable definitions indicated that parameters tended to be read-only: 68% were never assigned to, and they averaged less than one (.65) definition. This supports the notion that the major source of size information is the assignment of a known size to a variable.

The percentage of size one appears significant (78% of known sizes); of these, 79% are scalars, while the rest are single element arrays. The size zero figure represents the null vector, which is created as a constant or by a stylized usage. Since it often represents a special case for typing, it had to be recognized whenever possible; this may account in part for its frequency of occurrence.

Intraprocedural Typing

The next level of type analysis was the defined function. The single line typing method was extended by chaining all definitions and uses of a variable occurring within a defined function, that is, type information about a variable in a statement was merged with information obtained from other statements. This method was explained in the preceding chapter.

After all statements of a function were typed, intraprocedural processing was reiterated utilizing attribute information previously
deduced. This acts to propagate known attributes, and thus adds to the amount of type information obtained. This iterative process was repeated until the information stabilized, that is, subsequent passes produced no additional type information for any variable. Such intraprocedural processing was applied to each function in the data base.

Intraprocedural Results

The actions of function calls were not analyzed at this level. Thus, the nonlocal category again refers to both global and nonlocal variables, and a nonlocal was counted as a separate variable in each defined function in which it was referenced. An added measurement at this level is the number of iterations of intraprocedural typing needed for attribute information to stabilize.

As shown in Table 9, with intraprocedural typing the domain attribute was inferred for 82% of variable occurrences, approximately a 10% improvement from the single line results. The most noticeable improvement is the precision of known types, as displayed by the figures for integer and real types. Under single line typing, reals were 39% of known types and integers were 18%, but at this level, 29% of the variables are real and 27% are integer. The varying type represents all variables which took on incompatible attributes; the number of variables with incompatible domains was 176, or 66% of all incompatibilities.

Of the scope categories, locals are the most known and the most precise: 95% are known and approximately twice as many are integer as
Table 9. Distribution of domain attribute for intraprocedural typing.

<table>
<thead>
<tr>
<th>Type</th>
<th>Occurrences</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>1696</td>
<td>29.3</td>
<td>411</td>
<td>835</td>
<td>330</td>
<td>120</td>
</tr>
<tr>
<td>Integer</td>
<td>1569</td>
<td>27.2</td>
<td>958</td>
<td>383</td>
<td>129</td>
<td>99</td>
</tr>
<tr>
<td>General</td>
<td>1047</td>
<td>18.1</td>
<td>92</td>
<td>539</td>
<td>279</td>
<td>137</td>
</tr>
<tr>
<td>Character</td>
<td>642</td>
<td>11.1</td>
<td>138</td>
<td>295</td>
<td>89</td>
<td>120</td>
</tr>
<tr>
<td>Boolean</td>
<td>558</td>
<td>9.7</td>
<td>289</td>
<td>198</td>
<td>15</td>
<td>57</td>
</tr>
<tr>
<td>Varying</td>
<td>267</td>
<td>4.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>5779</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

are real. On the other hand, 24% of nonlocals are unknown and there are about half as many integers as reals. Return variable domains are known about as often as nonlocals are, while parameters are less often known (67%).

Table 10 displays the rank attributes which were deduced by intraprocedural type analysis. Rank is known for 57% of the variables, compared with 29% at the single line level. Vectors are the most frequent, as they constitute 56% of known ranks, while scalars are 26% and matrices are 12% of known ranks.

The ranks of 79% of the local variables were determined, in contrast with nonlocals and parameters, which have 42% and 37% of their ranks known, respectively. Parameters again are primarily vectors (68% of known ranks) or matrices (28% of known ranks). There were
Table 10. Distribution of rank attribute for intraprocedural typing.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Occurrences</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>2471</td>
<td>42.8</td>
<td>400</td>
<td>1314</td>
<td>532</td>
<td>225</td>
</tr>
<tr>
<td>0</td>
<td>868</td>
<td>15.0</td>
<td>501</td>
<td>271</td>
<td>12</td>
<td>84</td>
</tr>
<tr>
<td>1</td>
<td>1773</td>
<td>30.7</td>
<td>892</td>
<td>492</td>
<td>210</td>
<td>179</td>
</tr>
<tr>
<td>2</td>
<td>388</td>
<td>6.7</td>
<td>95</td>
<td>163</td>
<td>87</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>.2</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Varying</td>
<td>267</td>
<td>4.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5779</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

33 variables which had incompatible ranks, which is 20% of the total number of variables which had incompatible attributes.

The size of 33% of variables is known, as given in Table 11; the corresponding figure was 10.3% for single line typing. Size one is the most prevalent (65% of known sizes); 70% of these are scalars. Size zero (the null vector) is the second most frequent; consequently, small sizes account for most of the known sizes.

Scope follows the general trend at this level - locals are most often known (52%), followed by return variables (30%) and nonlocals (21%). Parameters differ little from the single line results; from 98% unknown to 97% unknown, which supports the notion of parameters as read-only variables as presented in the discussion of single line results. Fifty-eight of the variables with incompatible attributes were of different size, which is 35% of the incompatible variables.
Table 11. Distribution of size attribute for intraprocedural typing.

<table>
<thead>
<tr>
<th>Size</th>
<th>Occurrences</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>3873</td>
<td>67.0</td>
<td>913</td>
<td>1773</td>
<td>814</td>
<td>373</td>
</tr>
<tr>
<td>0</td>
<td>119</td>
<td>2.1</td>
<td>32</td>
<td>53</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>1236</td>
<td>21.4</td>
<td>775</td>
<td>333</td>
<td>24</td>
<td>104</td>
</tr>
<tr>
<td>2</td>
<td>67</td>
<td>1.2</td>
<td>41</td>
<td>20</td>
<td>.0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>39</td>
<td>.7</td>
<td>25</td>
<td>10</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>.4</td>
<td>14</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>.2</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6-10</td>
<td>39</td>
<td>.7</td>
<td>31</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>11-20</td>
<td>39</td>
<td>.7</td>
<td>26</td>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>21-50</td>
<td>24</td>
<td>.4</td>
<td>9</td>
<td>14</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>&gt;50</td>
<td>41</td>
<td>.7</td>
<td>13</td>
<td>24</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Varying</td>
<td>267</td>
<td>4.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5779</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The intraprocedural analysis was performed for a total of 914 defined functions. The type information for 742 of these (81%) did not change after the first pass. The type information for 155 of the functions (17%) stabilized after two passes. Fourteen of the functions were stable after the third pass, while the other three required four passes.
Interprocedural Typing

The final level of type analysis was the workspace. The intra-procedural level of typing was extended by chaining all definitions and uses for a variable occurring within any function of a workspace. Also parameters and return variables were typed both in the calling and called functions. Such analysis depends on the scope of variables, which depends on run time calling chains; the technique for handling this situation at compile time was described in the previous chapter.

The processing of a workspace proceeded by analyzing each of its defined functions, in order of appearance. Since the effect on type of each function is not completely known until all have been processed, additional passes through all functions were made until no type changes occurred for any of the variables in the workspace. The intraprocedural type analysis results indicated that two iterations were sufficient to capture type information for almost all (98%) defined functions, so within the iteration over all functions in a workspace, two iterations of intraprocedural typing were applied.

Interprocedural Data

Since scope was analyzed at this level, the categories of global and nonlocal could be distinguished. This would result in fewer variables being recorded, since globals were not treated as separate variables of each function in which they were referenced.

As depicted in Table 12, the domain attribute could be deduced for 86% of the variables encountered, which is about 4% better than
Table 12. Distribution of domain attribute for interprocedural typing.

<table>
<thead>
<tr>
<th>Type</th>
<th>Occurrences</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>G</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1534</td>
<td>34.1</td>
<td>1032</td>
<td>45</td>
<td>156</td>
<td>191</td>
<td>108</td>
</tr>
<tr>
<td>Real</td>
<td>1060</td>
<td>23.6</td>
<td>368</td>
<td>34</td>
<td>272</td>
<td>266</td>
<td>121</td>
</tr>
<tr>
<td>General</td>
<td>647</td>
<td>14.4</td>
<td>81</td>
<td>12</td>
<td>252</td>
<td>188</td>
<td>115</td>
</tr>
<tr>
<td>Character</td>
<td>527</td>
<td>11.7</td>
<td>148</td>
<td>7</td>
<td>138</td>
<td>110</td>
<td>124</td>
</tr>
<tr>
<td>Boolean</td>
<td>489</td>
<td>10.9</td>
<td>267</td>
<td>14</td>
<td>86</td>
<td>66</td>
<td>56</td>
</tr>
<tr>
<td>Varying</td>
<td>242</td>
<td>5.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4499</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the intraprocedural level. Precision shows a more marked improvement, as integers comprise 42% and reals 29% of known domains; the corresponding figures for intraprocedural typing were 35% and 37%. The varying type lists all instances of incompatible attributes; the actual count for domain was 141, or 58% of all variables with incompatible attributes.

Locals are most often known (96%), as was the case at other levels of type analysis. Global variables are the least often known (72%); direct mode may be responsible for this in some cases. (Direct mode refers to APL statements entered interactively. Several functions in the data base were observed to contain prompts for direct mode entry of values.) Since nonlocals are local variables of some function, they are often (90%) known. Parameters and return variables are less known (about 78%); these figures are likely caused by functions which neither issue calls nor are called.
Table 13 shows that the rank of 69% of the variables could be deduced, which is 12% more than the intraprocedural level. Locals are the most often known (80%); the other categories are less often known (52% for globals, 57% for parameters, 62% for return variables, and 64% for nonlocals). All of these, however, have more of an increase in percentage known than do locals, when compared with the intraprocedural level.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Occurrences</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>G</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>1399</td>
<td>31.1</td>
<td>379</td>
<td>40</td>
<td>433</td>
<td>350</td>
<td>197</td>
</tr>
<tr>
<td>0</td>
<td>892</td>
<td>19.8</td>
<td>524</td>
<td>29</td>
<td>153</td>
<td>94</td>
<td>92</td>
</tr>
<tr>
<td>1</td>
<td>1639</td>
<td>36.4</td>
<td>892</td>
<td>37</td>
<td>232</td>
<td>289</td>
<td>189</td>
</tr>
<tr>
<td>2</td>
<td>319</td>
<td>7.1</td>
<td>101</td>
<td>5</td>
<td>80</td>
<td>88</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>.2</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Varying</td>
<td>242</td>
<td>5.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Of the known ranks, vectors were the most frequent, with 53% of known ranks. Scalars were next with 29% of known ranks, while matrices were 10% of known ranks. There were a total of eight variables of rank 3 (out of 3,100 known ranks), and none of any rank greater than three. Parameters once again show a high proportion of vectors (61% of known ranks). Thirty variables had incompatible rank attributes, which was 12% of the total number of incompatible variables.
The size of 43% of the variables was inferred by interprocedural typing, as can be seen from Table 14; this is 10% better than the intraprocedural level. Locals are most often known (52%) and parameters are the least (24%). Of the variables of known size, 74% were of size one, and 66% of these are scalars; size zero was the next most frequent. No other sizes appeared with any pronounced frequency. Seventy one variables had a varying size attribute, which was 29% of all variables with incompatible attributes.

Table 14. Distribution of size attribute for interprocedural typing.

<table>
<thead>
<tr>
<th>Size</th>
<th>Occurrences</th>
<th>%</th>
<th>L</th>
<th>N</th>
<th>G</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>2656</td>
<td>57.0</td>
<td>913</td>
<td>64</td>
<td>615</td>
<td>620</td>
<td>353</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>2.5</td>
<td>34</td>
<td>2</td>
<td>32</td>
<td>10</td>
<td>33</td>
</tr>
<tr>
<td>1</td>
<td>1246</td>
<td>27.7</td>
<td>780</td>
<td>35</td>
<td>191</td>
<td>127</td>
<td>113</td>
</tr>
<tr>
<td>2</td>
<td>85</td>
<td>1.9</td>
<td>42</td>
<td>1</td>
<td>13</td>
<td>22</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
<td>1.0</td>
<td>24</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>.6</td>
<td>14</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>.3</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6-10</td>
<td>53</td>
<td>1.2</td>
<td>31</td>
<td>1</td>
<td>3</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>11-20</td>
<td>47</td>
<td>1.0</td>
<td>27</td>
<td>4</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>21-50</td>
<td>31</td>
<td>.7</td>
<td>9</td>
<td>1</td>
<td>13</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>&gt;50</td>
<td>34</td>
<td>.7</td>
<td>13</td>
<td>2</td>
<td>14</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Varying</td>
<td>242</td>
<td>5.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4499
For each of the 75 workspaces, the number of times the interprocedural typing procedure was applied before the attribute information stabilized was recorded. Thirty nine of the workspaces (52%) only required a single pass; it is likely that these contained few functions or function calls. Thirty three workspaces (43%) required two passes, while the other three had no changes after three passes. These three were observed to both contain many defined functions and issue many function calls.

Summary of Data

The principal results obtained from implementing type determination for APL are summarized below:

1. Each successive level of type analysis ameliorates the amount of type information that may be determined; consequently, the interprocedural level produces the most type information.

2. The attributes in order of most often determinable are domain, rank, and size: this follows the order of attribute restrictions of APL type rules.

3. Local variables have more of their attributes deduced than do the other scope categories; in addition, they are the most frequently occurring scope.

4. The majority of sizes that were inferred were small (zero or one); either most variables are of these sizes, or the situations in which these sizes arise lend themselves to the inference of size.

5. Few iterations are needed for type information to stabilize; this holds at both the intraprocedural and interprocedural levels.
A final item that warrants commentary is the assumption underlying the gathering of these results - consistency of variable usage with respect to type attributes. The interprocedural figure for incompatible variables (5.4% of all variables) offers evidence that variables tend to be used as a single type in APL programs. For those which do assume varying attributes, it would seem appropriate to eschew typing them. [The procedure for type analysis changes a variable of type varying to the general type, as suggested in (47), to avoid propagating the varying type.] The varying and general attributes are similar in that neither is usable as is information about known attributes, as will be discussed in the next chapter.
CHAPTER V. APPLICATION OF RESULTS

Introduction

The results presented in the foregoing chapter demonstrated that a substantial amount of attribute information for APL variables could be determined. The main areas in which type information may be applied to code improvement are: static type checking, less costly storage management, and the generation of type specific object code. One other area in which type information may contribute to implementation efficiency is the detection of program errors.

Type Checking

Type checking insures that operands conform to the type rules of the language operations. If type declarations are present, static type checking can be employed, since type attributes of variables are bound at compile time. Under dynamic type checking, each operation must, at run time, examine the attributes of its operands, since their bindings may change during execution; this requires the maintenance of a run time descriptor containing type information for each variable. As stated in (3), "techniques that attempt to replace dynamic checks by equivalent static ones are particularly important" (optimizations).

The advantage of static checking is that the type checks for an operation are performed once, at compile time, while under dynamic checking, they are performed for each execution of the operation. Compile time type checking decreases both the execution time and the space requirements of object code: the type check is not executed,
and code to perform the check is not present. Also, a run time
descriptor may not need to be maintained for variables whose type is
known, so the cost of space for the descriptor and time for its manipu-
lation are averted.

APL Type Checking

In APL, there are five type checks which may be performed:

(1) A value check verifies that a variable has been given a value,
as illustrated by the statement $A \leftarrow B + 1$, where a value must be
associated with $B$ at the time the statement is executed.

(2) A domain check tests if a value possesses the proper domain
attribute (such as numeric or character) for the operation that
is applied to it. In the above statement, $B$ must be numeric,
since the addition operation is not defined for character operands.

(3) A rank check is necessary when an operation calls for a fixed
number of dimensions, as in the statement $A[I] \leftarrow 1$; $A$ must be a
vector, since it is indexed with one subscript.

(4) A length check is made whenever a specific size is mandatory;
in the statement $A + B + 2 3 4$, $B$ must either be a scalar or a
vector of length three, since it is added to $2 3 4$, a vector of
size three.

(5) An index check is required to avoid reference to a nonexistent
array element. In the rank check example statement above, the
value of $I$ must be within the extent of the vector $A$.

Bauer and Saal (10) reported that 80% of APL run time type checks
could be eliminated by a static, line by line analysis of APL programs.
They used the methodology of forward and backward inferences to deduce type, along with noting situations in which a type check could be avoided. Examples of such situations are: (1) once a value, rank, or domain check is made for a variable appearing in a statement, it need not be made for subsequent references to that variable in that line; and (2) if one operand to a dyadic scalar operation is a scalar, then rank and length checks of the other operand can be deleted.

The percentage of each type check that could be eliminated was measured; the figures were domain (88%), value (88%), rank (80%), length (62%), and index (3%). Also it was found that the most frequent checks were eliminated most often. The reduction of value checking was observed to be a "trivial result; clearly intermediate expressions have values in APL and only variables need to be checked."

The salient differences between this study and the work of Bauer and Saal are the following:

(1) A single line was the only level of type analysis examined by Bauer and Saal, whereas three different levels of type determination were implemented in this study; this entailed development of methods for handling flow of control with compile time information at both the intraprocedural and interprocedural levels of typing.

(2) Type information was only used by Bauer and Saal to eliminate type checks, which was sometimes accomplished without actually knowing type attributes; this study sought to infer attributes of variables for application to both static typing and storage management.
(3) Bauer and Saal processed APL programs only to record the number type checks that could be eliminated, and they ignored possible syntactic ambiguity: this study implemented type analysis as a phase in an APL translator, and developed a strategy for resolving ambiguity at compile time.

(4) The sample of programs analyzed by Bauer and Saal consisted of 39 defined functions: this study analyzed a data base of 75 workspaces containing 914 defined functions.

The different approaches to type analysis between this study and Bauer and Saal, as given in item (2) above, means that the same kind of type information was not measured. However, it is possible to provide estimates of the percentage of type checks that can be eliminated by the methods of this study, so that these figures can be compared with the results of Bauer and Saal.

In order to relate type information about variables to the elimination of type checks, the operands to all operations in the data base were examined. There were a total of 71,539 operands, of which 18,211 (25%) were variables, 22,667 (32%) were constants (including labels), and the other 30,661 (43%) were the results of expressions (temporaries).

Since only variables require value checking, 75% of value checks (those for temporaries and constants) can be eliminated. This figure is due only to the nature of operand usage in APL programs, and does not rely on any type information. The corresponding figure reported by Bauer and Saal was 88%; their technique of eliminating
all but the first value check for multiple occurrences of a variable on a single line could be used in this work.

As for the other type checks, it would be necessary to have type information for temporaries, similar to that for variables, in order to calculate the percentage of each check that could be eliminated. Separate typing of temporaries was not done in this work, because of the large number of temporaries (30,661) involved; it was felt that this would excessively complicate the typing procedure. For estimation purposes, it seems proper to assume temporaries are known as often as are variables: they inherit their type attributes from constants, variables, and other temporaries, and they also act to determine variable types.

For each of the following categories of type check, the proportion of operands which are temporaries, constants, and variables, respectively, will be multiplied by the proportion of the particular attribute which is known and does not vary for each of these operand classes. This then will provide an estimate of the amount of each type check which may be performed statically.

For domain checking, the estimate of the amount of checks that can be eliminated is less accurate than for other categories, since precision must be taken into account. Typing a variable as real, for example, does not always mean a domain check can be avoided, since an integer may be required. Nonetheless, an estimate of the percentage of domain checking that can be eliminated is given by \((.81)(.43) + (1.)(.32) + (.81)(.25)\), or approximately 87%, which compares with
the 88% figure found by Bauer and Saal. For rank checking, the figures are 
(.64)(.43) + (1.)(.32) + (.64)(.25), or approximately 77%, which compares with the 80% reported by Bauer and Saal. For length checks, the estimate is 
(.38)(.43) + (1.)(.32) + (.38)(.25), or approximately 58%, which compares with 62% obtained by Bauer and Saal. For index checks, no figure is available from this study; however, it does not seem particularly important, since Bauer and Saal could eliminate only 3% of index checks. Thus, the methods of this study can be seen to yield results for the reduction of dynamic type checking comparable to those of Bauer and Saal.

Storage Management

Storage management deals with the technique by which space is made available to data objects manipulated by a program. The method of storage administration depends on the amount and duration of storage occupied by a data item, which has, as stated in (31), "significant effects on runtime efficiency." Storage management techniques can be classified as either static or dynamic; the latter classification includes two disciplines, stack and heap.

Static allocation is the simplest (and most time efficient) means of managing storage. If the sizes of all variables are given by type declarations, and no recursion is present, the amount of storage to be allocated can be determined at compile time. All space can then be statically allocated, and it will remain for the duration of program execution. Thus, no time costs are incurred at run time for storage management activities.
Stack-based allocation is the less expensive technique for dynamic storage management. Available storage for program data is set up as a sequential block of memory. Allocations take place from one end of this block, while deallocation is made in reverse order of allocation. This is the same order in which function calls and returns occur, so local variables of functions can be readily accommodated by this scheme. The amount of storage occupied by local variables must be fixed at function entry time, at which time allocation takes place; deallocation may then take place upon return from the function.

Heap-based storage management makes use of a heap, which is a block of storage, within which allocation requests are satisfied by partitioning the block into portions of the requested size. A heap is used when arbitrary-sized allocations are made at arbitrary points during program execution. Heap management involves keeping track of space available for allocation, deciding when coalescing may take place during deallocation, and minimizing fragmentation, which occurs when available space is divided into small blocks, none of which are large enough to satisfy an allocation request. All of these add to the expense of heap-based storage management.

Storage administration costs can be reduced by selecting the least expensive storage allocation scheme for each variable; the selection is based on determination of the storage requirements of the variable. Storage management techniques can be compared in terms of the number and duration of allocations for variables, and the type information requisite for utilization of the technique:
static - space allocated once at compile time, no deallocation; size known at compile time.

stack - space allocated at time of function entry, deallocation upon return from the function; size fixed at time of function entry.

heap - space allocated each time value assignment occurs, deallocation upon reassignment; size known at time of assignment.

APL Storage Management

In APL, storage must be provided whenever an operation produces a result value. The amount of storage depends on the operation and its operands. The duration of the value depends on whether the result is an intermediate value of an expression, or is assigned to a variable, in which case its duration depends on the scope of the variable, and whether or not the variable is reassigned. The lack of type declarations in APL means that, in general, neither the size nor the duration of a value can be determined prior to execution time, which dictates the use of a heap-based storage management scheme.

The overhead involved in APL storage management can be lessened by reducing the amount of heap activity, since it is the most costly method for managing storage. The APL data items requiring storage administration are variables, constants, and the results of expression evaluation (temporaries).

Global variables of known and fixed size can be managed statically. The proportion of variables which can be handled in this manner is the
proportion of globals (.24) times the proportion of globals whose size can be determined (.32), which is approximately 8% of all dynamic storage activity. Constants may also be handled statically in that space for them can be allocated as they are encountered during compilation. One consideration for constants is to avoid replication of values, that is, there should be only one copy of a particular constant, regardless of how many times it appears in a program.

Stack allocation can be used for temporaries and local variables whose size is known and does not vary. A stack is appropriate for temporaries, even though their storage needs are not known, because their values need not be retained after they are used. The proportion of variables which can be stack allocated is the proportion of locals (.74) times the proportion of locals of known size (.52), which is approximately 38% of all variables.

The amount of storage management for variables that can be removed from the heap, as calculated above, assumes that variables of all shapes are to be heap allocated, but it would be reasonable to exclude scalars from this figure. The facts that they occupy only a single storage location, and that 66% of variables of known sizes are scalar, suggest that their storage needs to be afforded special consideration. (The ISUAPL system treats scalars as a special case for allocation.)

Two issues related to storage management for APL were not addressed by this investigation, but they merit mention. First of all, there exist variables with the following two properties: (1) size not known, but (2) size known not to vary. Such variables could be
identified via type analysis methods by noting whether or not the 
operations which yield the value for a variable can change its size. 
Storage for such variables could be stack allocated because allocation 
would take place only once for local variables of a function, at 
the time the variable initially was assigned a value. The recognition 
and recording of such variables were not implemented in this study.

Another possibility for APL storage management would be the 
employment of a method of garbage collection. Storage allocation would 
be made from one end of memory, and reclamation (deallocation) would 
be attempted only when all available storage was exhausted. The 
appropriateness of such a method for APL was not approached in this 
study.

Type Specific Object Code

Static typing may enable the production of a more efficient object 
program by permitting type specific code to be generated. Without 
information about type, generic code must be generated, which must 
include testing of operand types in order to determine the specific 
operation to be performed. With type information, operand types 
are known at compile time, so only the code for the specific operation 
to be performed needs to be generated. (Most target machines contain 
instructions for type specific operations.)

The expression A + B illustrates the manner in which a generic 
operation can be translated to a type specific operation. If the 
domains of A and B are not known, then the generic addition code must 
test A and B, determine if a real or integer or mixed add is called
for, and decide how storage for the result is to be allocated. If, on the other hand, A and B are known to be integer, then an integer add instruction can be generated, and an integer storage location can be allocated.

Rank specific code is also of import for code generation. In the above example, if the ranks of A and B are unknown, then the generic addition code must also test A and B to decide the dimensionality of the add (scalar, vector, etc.). If the ranks of A and B were known, then only the code for that rank would be generated, so that the proper accessing of the structures would take place, and storage appropriate for the shape of the result would be allocated.

Type Specific Code Generation for APL

The attributes which are most important for type specific operations in APL are domain and rank. Numeric operands can often be more precisely typed, as the ratio of domains typed integer to those typed as real was approximately three to two. Thus, code can be generated for integer operations, and an internal integer format can be used for storage.

The proportion of variable ranks that can be determined (64%) enables rank specific code to be generated for APL. The semantic routines dealing with rank must be equipped for an arbitrary number of dimensions, but a small number of dimensions are used in practice, since no variables of rank greater than three were found in the database. Furthermore, since vectors account for over half of all known ranks, code can be tailored for vector operands.
As an illustration of rank specific code for APL, the rank rules for the scalar dyadic addition operation include the following cases for operands A and B:

(1) A and B both vector
(2) A vector, B scalar
(3) A scalar, B vector

for the first case, the generated code would add corresponding elements of two vectors, while in the other two cases, the object code would add a single value to each element of a vector. Thus, rather than having a generic routine to perform addition of arrays of any number of dimensions, only the code for the specific case encountered is necessary. The determination of which case occurs could be made on the basis of rank information available at compile time.

Error Detection

Type analysis can provide information about a program which may permit detection of possible sources of error. For example, determining which variables are used and which are defined in a function may reveal a variable which is used but never defined. For each defined function, compile time analysis can indicate the type of values it is expecting to receive as parameters and the type of value it returns; type conflicts between expected and actual parameters, or between expected and actual returned values, may be revealed. Also, for a defined function, those functions which it calls, as well as those functions which call it, can be deduced. This kind of information can be applied to software reliability (21) and program verification (25).
The processing of the APL data base illustrates the way in which compile time analysis can expose program errors. When the workspaces were initially compiled, calls to functions not present in a workspace were not detected, since the statement containing such a call would parse correctly. However, when the workspace was submitted to the ambiguity resolution procedure, in which a list of all defined functions within a workspace was maintained, the absence of the called function was revealed. [The action taken in this case was to insert a function definition (with no executable statements) for the called function in the workspace.]
CHAPTER VI. CONCLUSION

Summary

The primary concern of this investigation was the evaluation of type determination as an optimization method for APL. Toward this end, a data base of APL programs was acquired to direct the development of techniques for inferring the type attributes of APL variables. These techniques were then incorporated in the implementation of a type analysis phase for the ISUAPL translator.

Measurement of the extent to which type attributes could be deduced was obtained from the data base at three program levels: single line, defined function, and workspace. The data generated established that an appreciable quantity of type information could be determined at the workspace level. The application of this attribute information to code improvement for type checking and storage management, and its utilization for type specific code generation and error detection, was detailed.

In summation, the results of this study attested to the feasibility of type determination as an approach in the design of an optimizing compiler for APL.

Future Work

An extension of the work reported herein would be measurement of the run time efficiency enhancement derivable from the availability of type information at compile time. This would involve the implementation of a code optimization phase in the ISUAPL translator, based on
attribute information obtained from the type analysis phase. Such a study would need to consider the design of a pseudo-code format that allowed for code improvement, and a means for measuring the gain in execution speed attributable to the code improvement, as has been approached by Jenkins (30).

Several questions of a general nature may be posed regarding the results of this investigation:

(1) Are the techniques employed to infer type attributes applicable to any typeless language, or does APL possess characteristics which make it more (or less) amenable to type analysis? Could a comparable amount of type information be determined for any typeless language?

(2) The special treatment received by flow of control supports the contention that APL is deficient in control structures, as suggested in (2, 14, 22). Would more manageable control constructs be favorable to type analysis?

(3) The areas of run time activity most affected by the presence of type information are type checking, generic operations, and storage management. Is the cost of dynamically performing each of these of comparable magnitude, or does one account for a disproportionate share of overhead during execution?

(4) Should efforts to optimize APL, or any typeless language, concentrate on static typing as an optimization method, or are other techniques, such as delayed evaluation (27), more worthwhile?
Can such techniques be combined, at an acceptable cost, in an optimizing compiler?

(5) Can typeless languages be optimized sufficiently to compensate for their execution time expense, or must very high level language design require type declarations for efficiency purposes?

In conclusion, there exist numerous topics related to this research which await investigation.
BIBLIOGRAPHY


