Compiling APL: The Yorktown APL Translator

by Graham C. Driscoll, Jr.
Donald L. Orth

The Yorktown APL Translator (YAT) permits functions written in APL to be compiled using an existing compiler for part of the process. It also creates tables that allow the APL2 Release 2 interpreter to call the compiled code. The code can also be called from a Fortran main routine. This paper outlines the history of APL compilation, the motivation for producing YAT, the design choices that were made, and the manner of implementation. Sample APL functions and their translations are shown, and the time required to interpret these functions is compared with the time required to execute the compiled code. Possible further work is discussed.

Introduction

We undertook the work described here to investigate the extent to which compilation could accelerate the execution of APL applications (*vis-à-vis* current interpreters). Our first step, now successfully completed, was to design and build a prototype translator that produces source code for the VS Fortran compiler. We are now embarking on extended studies of its performance and of the effects of various modifications to it, e.g., idiom recognition. The translator itself, being written in APL, will be one of our test applications.

Throughout this paper, the words "translate," "translation," and so on refer to translation from APL to Fortran or another high-level language. The words "compile," "compilation," etc. refer variously, as the context makes clear, to the transformation of Fortran or another high-level language into Assembler, the direct transformation of APL into Assembler, or the entire process of translating APL to Fortran, say, and then transforming the Fortran into Assembler.

This paper discusses our translator design and the reasons we chose it, the way in which we expect the translator to be used, the code it produces, computational accelerations that we have measured, and further accelerations that we believe are possible.

The following section summarizes the characteristics of the APL language and the reasons for believing that a compiler would be a valuable supplement to APL interpreters. Next, we outline the history of previous efforts to compile APL. In the section *Design of the translator* we discuss the overall form of our translator and how we decided upon it. Details of our translation methods and results and of how the translator is used, some sample translations, and some accelerations we have measured are to be found beginning with the section *Some details of translation.*

Motivation

The fundamental unit of data in APL is the array, a rectangular structure of essentially unlimited rank containing numeric or character scalar values. Most APL interpreters provide three or four storage types for numeric data—Boolean, integral, real, and (possibly) complex—but conversions are performed as necessary, and the storage type of a numeric array is visible to a programmer only indirectly, by the amount of storage that the array occupies. No explicit statement binds an array permanently to a particular rank, shape, or type. The same name may be given, for example, to a vector and then to a matrix which has no necessary relationship to that vector.

APL has a large collection of primitive (built-in) functions, mathematical, relational, and structural. Each function accepts as arguments any arrays that are meaningful for it, not just scalars; the APL processor handles any indexing that is implicit in the definition of the function. There are also primitive operators, which have higher precedence than functions. Operators take functions (and arrays) as operands and produce functions such as Inner Product, Outer Product, and Reduce with Axis. The functions produced by operators are called "derived" functions. The arguments of a derived function are sometimes loosely said to be the arguments of the operator by which it is produced.

In this paper, we use the term "operation" to mean either a primitive function or a primitive operator, and we sometimes say "primitive operation" for emphasis.

APL has relatively few syntactical rules. There is, for instance, no hierarchy of primitive functions: The order of function execution is determined only by position, parentheses, and brackets. Contributing to APL's simplicity (from the programmer's point of view) are the systematic provision for empty arguments in function definitions and the extension of scalars and other unit arrays to conform to non-unit arguments where appropriate: The same statement form is used to add two matrices and to add a scalar to a matrix; indeed, the very same statement may be executed sometimes with two matrices as arguments and at other times with a matrix and a scalar as arguments.

Having arrays as fundamental data types, no declarations, a rich assortment of primitive functions, simple rules, and automatic handling of what would be exceptions in many other languages has made APL a language in which it is comparatively easy to write programs for a substantial range of applications.

Implicit indexing of arrays, inclusion of "exceptional" cases, and the presence of a wide variety of operations often allow the specification of a large amount of computation by a very small amount of program text. In such a situation, interpretive execution of programs can be quite efficient: Most of the execution time is spent in actual data manipulation and very little in interpreting the program text.

There are other situations, however, in which the interpretive overhead looms much larger: where the arrays are small or sparse or where there are many cases to be considered, necessitating explicit branching and handling of scalar elements of the arrays. Furthermore, the freedom to change not only the value and shape but also the type of an array requires the interpreter to check the characteristics of the arguments of each primitive operation each time it is executed, and this overhead can be significant for small arrays. (Some of this checking could be done when a programmer finishes editing a defined function, but much of it must be done at run time, and in practice interpreters almost always do it at run time.)

Consequently, APL is not used for some applications to which it seems otherwise well suited, because its execution speed is too slow for them. Indeed, sometimes APL is not used because the potential user does not know whether or not its speed would be adequate, and he believes that if the application were programmed in APL and turned out to run too slowly, then it would have to be reprogrammed in another language.

These considerations made the desirability of an APL compiler clear long ago; there are, however, good reasons why there is only one commercially available APL compiler [1]. On the one hand, the pressure for a compiler was reduced by several factors, among them the aptness of the language for efficient translation, the excellent quality of the translators that were written for it, and the fact that the early APL systems were essentially stand-alone systems and were well tuned for APL. On the other hand, the freedom that APL gives the programmer—including explicit and implicit reuse of names, dynamic establishment of defined functions, and the execution of variable character vectors as APL code—has seemed to make the job of compiling APL quite difficult indeed.

History of APL compilation

The difficulties inherent in compiling APL and especially the existence of noncompilable expressions in the language have led to the construction of APL language processors that cover the spectrum from pure interpreters to pure compilers (the latter applying restrictions to the statements that are acceptable for compilation). Hence it is often difficult to decide whether to call a given processor an interpreter or a compiler.

The Burroughs APL-700 system could perhaps be called the first APL compiler, because it was the first interpreter to keep parse trees during execution and regenerate them only when necessary. Hewlett-Packard's APL-3000 [2, 3], however, is usually credited with being the first APL compiler. This remarkable implementation went much further in run-time binding than did APL-700: It compiled code at run time based on the current storage types and ranks and even shapes of variables. Unfortunately, the performance of this system did not meet expectations, in part because it was not possible to dynamically create and run HP-3000 machine code, and therefore the target of the compiler was a relatively high-level intermediate language. Miller's design [4, 5] goes beyond APL-3000, using type inference to support code generation over a span of more than one APL statement and to reduce the number of checks required to validate the dynamically generated code on each occasion when it might be reused. Guibas and Wyatt [6], followed by Budd [7] and Treat and Budd [8], have continued to refine these code generation methods.

All these compilers generate code at run time, based largely on dynamic information, and aim for code that is

quite specific to the instance. Clearly, it can be costly to regenerate the compiled code for a statement when, on a subsequent execution, the code previously compiled for it is discovered to be no longer valid. More extensive inferring of type [9, 10] can reduce the frequency with which such situations occur, as can the generation of code that is more general than is immediately required; typically these methods generate code that is specific for storage type, but not rank or shape. Another disadvantage of run-time code generation is that run-time static analysis is necessarily limited in scope, usually to a single statement or at most a few neighboring statements, whereas a thorough analysis of APL applications should not only encompass all of each procedure (defined function) but also be interprocedural. Although at run time information may be available that cannot be deduced during static analysis but that could have a substantial positive effect on the generated code, such as whether or not a particular instance of Take is an overtake, the very fact that it cannot be known statically implies that it may vary from one execution to the next and thereby cause excessive code regeneration.

The code generation schemes in these compilers are all based on the work of P. Abrams [11]. Abrams suggested two fundamental code improvements ("optimizations") for APL code. The first, which he called beating, refers to a method of generating code for sequences of APL selection operations as if the sequences themselves were individual selection primitive operations. The second, called drag-along, involves practices now commonly known as loop jamming (or loop merging) and lazy evaluation. (Lazy, or demand-driven, evaluation computes values when they are needed insofar as possible, rather than when their computation appears in the program, so that if a selection operation picks elements from a computed temporary variable, only the selected elements of that variable will be computed.)

There is a second line of inquiry into APL compilation, based on the translation of APL to other high-level languages [12–15]. All these studies are based on the one-for-one replacement of APL primitives with code in the target language, and none use type inference as an aid to code generation. Even so, they all demonstrate the important fact that even the most straightforward approaches to APL compilation can yield dramatic improvements in many cases.

The only commercial APL compiler available today is offered by STSC, Inc. [1]. This compiler generates code for a restricted set of cases of the APL primitives, and concentrates on improving scalar code. For example, at the time Weigang's paper appeared, code for Take was generated only in the case of vector right arguments. Finely tuned code for $I \leftarrow I+1$ for scalar I and $\rightarrow LABEL$ is generated. The code is produced at compile time, not at run time, and is permanently merged with code to be interpreted. Code improvements are avoided which might produce run-time error reports differing from those that would be given were the code interpreted. This compiler performs code improvement on the scalar code it generates, as opposed to transforming the APL source before code generation, and consequently the improvements it makes resemble more those of compilers for other languages, and are less like those based on Abrams' work. A compiler that is similar in spirit to the one at STSC has been built at Cornell [16], but we do not know whether or not it has been developed to the point where it is in production use.

Our compiler differs from each of its predecessors by virtue of at least one of these characteristics: The defined function is the unit of compilation; thorough static interprocedural analysis is carried out; code improvement is done at the source level (as well as on the generated code, and in addition to the code improvements made by the Fortran compiler); all cases are compiled for each compilable primitive operation; intermediate code in a high-level language is available for the programmer's inspection; machine code is generated; and applications may consist of a mixture of compiled functions and interpreted functions.

The three things that we believe most distinguish our compiler are that we require variables to be of known fixed rank; we do a thorough analysis of the shapes of variables (including in the analysis the shapes and upper and lower bounds on shapes that the programmer has put in declaratory comments); and we use this shape information to tailor the generated code to many special cases.

There is a series of excellent papers discussing various aspects of APL compilation [17–20]. Wai-Mee Ching introduced us to type inference schemes, and his own work on APL compilation is based on a definition of compilable APL that is similar to ours. He has worked on the problem of generating code for parallel machines [21], and is currently working on an APL compiler that generates System/370 code directly [22].

Design of the translator

Writing a compiler to translate APL to Assembler would have been a very substantial job for us, and much of the effort would have duplicated work that has already gone into other compilers. So, to make our task feasible and to allow us to concentrate on the problems unique to APL, we decided to translate APL to a high-level language already having a compiler. We chose Fortran (in many ways the lowest of the high) as the target language (or first target language), since it has compilers that produce efficient code and therefore can provide a reasonable test of the viability of our approach.

Since we wrote our translator in APL and we translate APL to a language for which compilers are generally available, the entire compilation process is portable to a wide range of systems, from large mainframes to personal computers. In fact, we have already run compiled programs

on the IBM 3090 system, making use of the Vector Facility, as well as on the IBM 308X systems that were our original targets.

We soon saw that the use of a high-level intermediate language could be beneficial not only to us, by reducing the amount of work we had to do, but also to some users of the compiler. All that was needed was to make the intermediate code as readable as possible and leave it exposed to the user, rather than make one seamless package of our translator and the Fortran compiler. The translated code need never be looked at, but a programmer who is familiar with Fortran can examine it and perhaps change it or—much better—induce changes by changing the original APL code.

To make the translated code as readable as possible we eliminate some needless labels and CONTINUE statements, consolidate some sequential GO TO statements, evaluate some arithmetic expressions involving constants, and so on, in spite of the fact that these changes will make no difference in the compiled code, and we indent the bodies of DO loops and block IFs. To make correlating the original and translated code easy, we include Fortran statement numbers (ISNs) in the sequence field of the Fortran code, marked to indicate where translation of a set of APL simple expressions begins and ends, and provide an APL listing. (A block of several Fortran statements may collectively translate each of several APL simple expressions.) A sample listing is shown in the section *Examples of code generation*. Such a listing displays the original APL statements, each followed by the expressions into which it has been analyzed. Each expression is tagged by ISNs, to show where its translation can be found.

Having decided to make use of an existing compiler, we next had to consider how we wanted to restrict the APL code that we would translate, in order to ensure significant acceleration of execution. Only when the rank and type of each variable are fixed and known can real opportunities for performance improvement be found and burdensome run-time computations be avoided. Therefore, our primary requirement was that these attributes be constant and be available to the compiler. A localized name can be reused for a variable of a different rank or type; we just treat such reuse as if a different name had been used. If, however, such reuse were to occur for a global name or if a given instance of any name were to refer at various times to arrays of various ranks and types, or even just to an array of unknown rank or type, there would be little hope of producing Assembler code for the statement in which it appeared whose execution would have much advantage over interpretation of that statement.

We also had to consider how to handle those APL expressions that cannot really be compiled, such as Evaluated (Quad) Input and Execute with an argument whose value is unknown. We did not want applications for which the compiler could be used to be restricted to a subset of the APL language, so we had to accept the existence of applications for which some code would be compiled and some would continue to be interpreted. We saw two principal ways of allowing for interpretation.

We could package a rudimentary interpreter with the compiled code and maintain, during execution, the tables that would allow interpretation of statements that could not be compiled. Incorporating an interpreter in our run-time support, however, would be too large a job for a small group and is unnecessary for a prototype.

The alternative we chose was to use the VSAPL or APL2 interpreter for code that must be interpreted and to call the compiled code from it. This choice involved little work for us because we were fortunate enough to have available both an experimental auxiliary processor that allows VSAPL to call programs written in Fortran, PL/I, or Assembler (produced at the IBM Heidelberg Scientific Center) and an interface in the APL2 program product (Release 2) [23] to Fortran, Assembler, and REXX.

We chose the defined function as the unit of compilation: A function must be compiled in its entirety, together with all the functions it calls, or not compiled at all. Furthermore, the functions to be compiled would, like locked functions, not be susceptible to suspension.

There were three major reasons for this choice:

- Allowing interpreted and compiled code to be interspersed in a defined function precludes global flow analysis and certain code improvements.
- Frequent crossing of the interface between compiled and interpreted code creates unwanted overhead.
- The separation between interpreted and compiled code occurs at points that are meaningful to the programmer, and each translated procedure corresponds to one of his functions.

What constructions would we interpret rather than compile? That is, what constructions would render a function containing them unacceptable to our translator? In addition to expressions that are essentially not amenable to compilation, there are expressions that severely inhibit the analysis of a function containing them, such as a branch to a computed value that could be the number of any line in the function, and we wanted to place them outside the pale also.

In formulating the rules for the subset of APL we would compile, we were guided by six main considerations:

- Whether an expression can be executed with minimal run-time support.
- Whether its execution can actually be accelerated by compilation.
- Whether its presence would severely degrade analysis and code improvement.
- Whether it commonly appears in APL applications.

- Whether it involves really bad programming practices (which we see no need to support), e.g., branches to absolute line numbers other than 0 and 1.
- What rules will be easy to learn and remember.

In keeping with these considerations, we have imposed several requirements for a function to be accepted by our translator. The principal ones are the following:

- There must be no need for run-time syntax analysis; hence, the argument of any Execute must have known values and there must be no Quad Input, dynamic establishment of function definitions, or the like.
- To facilitate analysis, all branch targets must be labels, zero, one, or empty (fall-through), and they must be visible and manipulated in certain ways; i.e., branch statements must be of certain forms. Most of the commonly used branch expressions satisfy this requirement.
- The storage type and rank of each variable in the function must be known. (Knowledge of the shape, or at least bounds on the dimensions, is helpful, but not essential.) Usually when these attributes are given for a relatively few variables in a function they can be inferred for the remaining variables and expressions.
- Distinguished axes must be known. If they are indicated by constants or variables (of known value), then the index origin must be known also.
- Scalar extensions must be deducible or explicitly indicated.

Note that we compile all VSAPL operations except system functions and functions that involve the execution of code that is not visible to the compiler. We do not yet, however, handle recursion, shared variables, or the new APL2 constructs. The restriction on recursion could be lifted by retargeting the compiler to another compilable high-level language.

The section *How the translator is used* tells how the programmer may explicitly give the translator information that it cannot infer.

Clearly, the increased speed offered by our compiler is not free. In order to take advantage of it the programmer must understand the requirements of the compiler, be or become familiar with the application he wants to compile, determine what functions should be replaced by compiled versions, think carefully about the array attributes to be used, and make any required modifications.

If an APL compiler such as ours becomes generally available, programmers will presumably keep its requirements in mind when creating applications which might someday be compiled. In particular, they should isolate the computationally intensive parts of the application in defined functions that conform to the requirements of the compiler; include, in comments, all nonderivable information about types and ranks and, if feasible, shapes or upper and lower bounds for shapes; and, if significant attributes are unknown, instrument the application to accumulate data regarding them.

Some details of translation

The two main steps of translation are analysis and code generation.

Analysis provides information that is required for code generation but that does not appear explicitly in APL programs, such as ranks and storage types of arrays, parameters for dimension statements, and expressions for shape computations. In addition, code improvements that would generally be missed by Fortran compilers are performed, such as eliminating common APL expressions and statically evaluating expressions of the type that often appear as left arguments to Take and Reshape.

Usually the attributes of variables can be inferred from the APL code, and we go to considerable trouble to do so, although the programmer may sometimes be asked to help (see *How the translator is used*). Indexing betrays rank, Boolean and arithmetic functions give type information, explicit constants often appear in dimensions and may imply nonemptiness, and relations between the shapes of variables are often deducible (our restriction on scalar extension is intended not only to simplify the generated code but also to allow a more thorough analysis). For example, clues such as $W\leftarrow(2+\rho V)\uparrow V$ or $Y\leftarrow(\rho X)\rho V$ are used to minimize the information required of the user and to simplify the code that is generated. (The first clue tells us that ρV is less than ρW and that no check need be made to see whether W is empty, the second that ρX and ρY are equal.)

The analysis produces, among other things, a list of simple APL expressions to be executed, in the order in which they appear (or are implied) in the APL function. Frequently, we combine operations in the generated code, to avoid executing similar nests of DOs several times, or to avoid the allocation of storage for more than one element of the temporary result of an expression that, if operations are combined, can have each of its elements used immediately. Because the generated code often depends upon the ranks, shapes, and storage types of the variables involved, it is a somewhat ticklish business to determine whether it is possible or advantageous to combine the translations of a set of operations.

Three goals of the prototype translation process itself are clarity of the methods of translation, ease of modification of the translator, and ease of retargeting to other languages having compilers. For code generation, we have found it convenient to use an archetype (an object similar to a macro) for each APL primitive operation. The archetypes clearly display the translation, they can be edited like any other program, and their further translation can depend upon the target language, although of course they must contain some text specific to the currently intended target

language. The language of the archetypes contains rank-independent abbreviations for frequently occurring text, such as DO nests and branches based on whether an array is empty. There is syntax for selecting portions of archetypes, since a large part of the accelerations we have achieved comes from using code tailored to the special cases uncovered by our analysis of the functions being compiled.

We assume that applications that are being compiled will have been developed and debugged using interpretive execution. We do not provide such facilities as tracing and suspending of execution, and we make no general provisions for handling unanticipated errors that occur when the compiled code is executed. One source of acceleration is the elimination of most domain checking: It is the programmer's responsibility to incorporate checks wherever necessary in a function that is to be compiled. This requirement, we might observe, is only what is expected of fully interpreted code that is to be in general use, where the users must be protected from APL error messages.

We do provide for run-time checks for bounds on array shapes and for equality of array shapes (absence of scalar extension); these checks are incorporated or omitted from the generated Fortran text at the programmer's option.

The organization of compiled applications

APL applications are arranged in workspaces, which are collections of functions and data, and this arrangement persists after compilation, with the difference that some of the functions that are compiled are (automatically) replaced by functions that refer to Fortran subroutines. Each replacement APL function has the same name and syntax as the one it replaces, but consists primarily of a call to a Fortran subroutine through a suitable interface provided by the host APL system, such as the name association mechanism in APL2 [23]. From a user's viewpoint, there is no difference between the way an ordinary APL application is used with pure interpretation and the way it would be used after compilation. On the other hand, it is not necessary to maintain a workspace arrangement after compilation (and thereby dependency on an APL host system) if all functions in an application have been compiled and appropriate Fortran main programs are written to call them.

A set of functions may be compiled piecemeal: It is not necessary to compile simultaneously all the functions in a workspace that are eventually to be compiled. Whenever a function name is explicitly presented to the compiler, however, with specific types and ranks for its arguments, that function and all functions it calls are compiled (except for called functions that have been compiled previously and not altered since). A directory is maintained that contains APL function names, associated Fortran subroutine names, and descriptions of parameter lists; it is used to avoid recompilation whenever possible. This information is also used to construct whatever is needed by the interface through which the compiled Fortran subroutines are called. In addition, the directory contains the text for run-time error messages. Its name is based on a three-character sequence that is given at the time of translation, unless the default sequence YAT is to be used. The chosen sequence is also used as a prefix for the names of the Fortran subroutines produced during a compilation, and is therefore similar in function to a workspace name.

When a function is compiled because it is called by another function, and not because its name has been given explicitly to the translator, that function will not necessarily be replaced in the workspace. APL applications often contain functions whose parameters can meaningfully vary in rank and storage type. These functions serve as additional primitive operations, and the translator treats them in much the same way as it treats true primitive operations. All APL operations apply to arrays of various ranks and types; a requirement for compilation is that the parameters of every operation at every distinct codepoint must be of fixed rank and type. This same principle is applied to called functions; the parameters of a called function may be of different type and rank for different calls at different codepoints, but not at the same call. Whenever such a function occurs in a compilation, it is compiled separately for the different cases, and therefore could not be replaced by a call to a single Fortran subroutine.

The separate compilation of APL functions under a variety of type and rank circumstances presents real opportunities for specializing the resulting code in the various cases. For example, it is not uncommon in these functions that the parameters are regularized in rank early on, so that the principal computation applies to objects of fixed rank. An example of an expression that regularizes rank is

$$X\leftarrow(^{-}2\uparrow 1\ 1,\rho A)\rho A$$

If A is a scalar, then X is a one-by-one matrix; if A is an N-element vector, then X is a one-by-N matrix, and if A is a matrix, then X is identical to A. By statically "evaluating" the expression

$$^{-}2\uparrow 1\ 1,\rho A$$

the translator is able to recognize these different cases, generate simple code for that statement, and, in the first two cases, generate specialized code whenever the shape of X is involved.

How the translator is used

In the simplest possible case, the programmer loads the translator, copies the functions to be compiled, and gives the names of the entry points—the functions that are called by users of the workspace. The functions whose names are given are compiled, together with any functions they call.

The programmer copies into his workspace the file containing both the newly created versions of his functions and the interface data for the Fortran routines. The application runs faster than the old one, but otherwise acts just the same, in the absence of erroneous data for which the programmer has failed to put checks in the APL that was compiled.

Furthermore, instead of running the compiled functions from APL, the programmer can call them from a Fortran main routine.

Other steps may be necessary or desirable, however.

The application may contain forbidden expressions. Then, for each function containing such an expression, the programmer must either change the function or not include in the argument to the translator its name or the name of any function calling it. To help him find illicit material, the translator has a checking facility, which produces a report listing the expressions that cannot be compiled but whose inclusion is implied by the given set of entry points. If he is unfamiliar with the application programs or their execution, he can determine in which functions most of the execution time is spent using a timing tool such as [24], perhaps together with a workspace analysis tool such as [25]. He can then analyze the cost (if any) of modifying the offending functions to allow them to be compiled and the expected benefits from doing so. We believe that in most applications a relatively few functions consume most of the CPU time, and no more than those few functions would need to be modified.

The application may contain variables whose attributes (e.g., domains) cannot be inferred from the APL code. To allow the programmer to keep these attributes with the function to which they pertain, and not in some separate table that gives rise to logistical or documentation problems, and to avoid introducing any new syntax into APL—both for the sake of avoidance itself and so that the language for the interpreter and the language for the compiler remain identical—we allow declaratory comments. They begin ⍝*DCL*⍝, contain keywords such as *SHAPE* and *DOMAIN*, keywords referring to upper and lower bounds for dimensions along various axes, and so on, and may use expressions that refer to other variables in the function or to global variables present in the workspace at the time of compilation.

We emphasize again that we make every effort to determine attributes from the APL code, and thus to minimize and perhaps eliminate the need for the programmer to insert declaratory comments.

The programmer can examine the translated code and perhaps make changes in it. He can, for example, specify that the storage type for a given variable is two-byte integer rather than four-byte. He can perhaps make changes to increase execution speed; if feasible, the better way for him to do this is to revise his APL functions to induce the changes he wants.

Examples of code generation

To give some idea of the code we generate, we discuss a few instances.

A rather complicated APL primitive function in terms of code generation is Take, which is so frequently used that the code for it must be quite efficient. Its right argument can be any array. Its left argument is a vector of integers, one for each dimension of the right argument. The rank of the result is the same as that of the right argument, and its shape is the absolute value of the left argument. Each element of the left argument indicates how many of the corresponding subarrays of the right argument—rows, columns, or whatever—are to appear in the result, and its sign indicates where they are to come from; e.g., 5 indicates the first five subarrays and −5 the last five subarrays. If the absolute value of an element exceeds the number of subarrays along the corresponding axis, then fill elements (zeros or blanks) are inserted in the result to make up the deficit. If the element is positive, the fill elements follow the selected subarrays; otherwise they precede them.

The simplest case occurs when the left argument is known to be positive, fill is known to be unneeded, and the result can be given the same Fortran name as the right argument. For this case we need only set the shape of the result; no other code is required. (For an array of variable shape, we distinguish between its current shape and its maximum shape as determined by the maximum value of each of its dimensions, which determines the storage area allotted to it and is presented to Fortran simply as its dimension.) Thus $X \leftarrow 5 \uparrow X$ produces, when the length of X is known to be 5 or greater, only

```
J1X = 5
```

where J1X is the length of JX, the Fortran variable corresponding to X. (Our standard means of obtaining a Fortran name for a variable is to prefix its APL name with a J, V, A, or E, depending upon whether its storage type is integral, real, character, or Boolean. The shape variables for a variable whose APL name is *NAME* are called J1NAME, J2NAME, etc. Names of implicit subscripts begin with I.)

In the extreme case for Take, it is necessary to compute at run time the fill to be inserted along each axis (perhaps none), at which end of the axis that fill is to go, where the elements from the right argument are to be placed in the result, and from what part of the right argument they are to come. If the right argument to Take is a singleton array, then the result can be constructed by creating it completely with fill elements and then placing the single element of the right argument in the appropriate position. If fill is to be added along several axes of a non-singleton array, on the other hand, then several DO nests are required to insert the fill efficiently.

Although Catenation is not so complicated as Take, it has similar simplifications in certain cases. $A \leftarrow A , B$ yields,

under suitable conditions, only

```
      DO  240  I1 = 1,J1B
240         JA(J1A+I1) = JB(I1)
      J1A = J1A + J1B
```

where the DO loop appends the elements of B to those of A and the following line sets J1A to the new length of A. Currently, a variable whose length may possibly vary is kept left-adjusted in the storage allotted to it, and a pointer indicates the end of the variable, as shown in the example. We intend also to maintain a pointer indicating the beginning of the variable in its allotted storage and to place the variable wherever it seems best in that storage, so that $A \leftarrow B , A$ can likewise be handled without moving any of the original elements of A.

As an example of our use of the same DO nest for several APL operations, consider the code generated for the sequence

$A \leftarrow B + C - D$

$M \leftarrow N \div P$

where M and A are known to be two-dimensional arrays of the same shape. The loops for Subtract and Add can be combined and their computations merged in one statement because the temporary $C - D$ is used in the addition and nowhere else and need never be stored. The division can also be included in this same nest because of the similarity of shapes. (Since we handle only certain errors, the question of preserving order so as to get the right error message—or indeed, in the case of redundant computation, any error message—does not even arise.) From these considerations we get, say,

```
      DO  150  I2 = 1,100
        DO  150  I1 = 1,J1A
          JA(I1,I2) = JB(I1,I2) + (JC(I1,I2) − JD(I1,I2))
150       VM(I1,I2) = VN(I1,I2)/VP(I1,I2)
```

Redundant parentheses are often retained in Fortran expressions, in order both to make them correct when read with either Fortran's or APL's order of execution in mind, thus avoiding any confusion, and—where the two orders yield the same result—to omit nontrivial calculations, involving the hierarchy of Fortran operations, that would enable at most a trifling improvement in our output.

When it is possible and there are no countervailing considerations, loops are put in the order just shown, so as to run through contiguous storage locations, thereby minimizing cache faults, translation-lookaside-buffer misses, and page faults, and also to ensure that the benefits of interleaved storage are obtained. To be sure, the amount of overhead for testing and resetting subscripts can be minimized for a DO nest by putting the loops in order of increasing number of traversals. When we tested the effect of various nesting orders on execution time for the IBM 308X and IBM 3090 (scalar hardware), however, storage contiguity was almost always the dominant factor; in the cases where it was not, the 308X was slightly faster for the order based on number of traversals, but the 3090 was faster, by a larger margin, for the order based on contiguity.

The balance shifts when we consider not just reordering but removing DOs. We ran some tests in which we unrolled short loops that were traversed only two or three times. The overhead that was thus avoided more than offset any resulting disruptions in storage contiguity: For both the 308X and the scalar 3090, execution was accelerated even when the loop that was unrolled was the outermost one. During our postprocessing, therefore, this code:

```
      DO  350  I3 = 1,2
        DO  350  I2 = 1,J2A
          DO  350  I1 = 1,2
350         JA(I1,I2,I3) = JB(I1,I2,I3) * JC(I1,I2,I3)
```

would be changed to this:

```
      DO  350  I2 = 1,J2A
          JA(1,I2,1) = JB(1,I2,1) * JC(1,I2,1)
          JA(2,I2,1) = JB(2,I2,1) * JC(2,I2,1)
          JA(1,I2,2) = JB(1,I2,2) * JC(1,I2,2)
350       JA(2,I2,2) = JB(2,I2,2) * JC(2,I2,2)
```

Our aim is to produce code that will generally give the best results. A programmer may find, for some section of his program, that our nesting order or unrolling is suboptimal for execution on a particular target computer, say, or as input to a particular vectorizing compiler. If that section is critically important, he can edit the Fortran program to obtain the code that he has found to be faster.

In the previous example, merging code for operations saved loop overhead and storage space. It can also save computation. Consider the setting of a Boolean variable to indicate whether a vector X is equal to one of the rows of a matrix—e.g., whether a name appears in a list: $Z \leftarrow \vee / Y \wedge . = X$. Both the Or and the And offer the opportunity for quitting early, but nevertheless the entire inner product would have to be formed if the code generating it were not merged with the code for Reduce. (In fact, Inner Product itself is handled by merging code for "Midproduct" into code for Reduce. We define Midproduct as a derived function that applies the right operand of Inner Product to its arguments, producing an array whose rank is one less than the sum of the ranks of the arguments; Inner Product is then completed by reducing this array using the left operand. Midproduct might someday be made available directly to users of the compiler.) Merging Inner Product into Reduce exploits "lazy evaluation" (cf. the section *History of APL compilation*) to the utmost in the generated Fortran code:

```
      DO  1020  I1 = 1,J1Y
          DO  1000  I2 = 1,J1X
              IF  (.NOT.(AY(I1,I2).EQ.AX(I2)))  THEN
                  F26 = .FALSE.
                  GO  TO  1010
              END  IF
1000          CONTINUE
          F26 = .TRUE.
1010      IF  (F26)  THEN
              EZ = .TRUE.
              GO  TO  1030
          END  IF
1020      CONTINUE
      EZ = .FALSE.
1030  CONTINUE
```

A similar example is provided by this APL fragment, which produces the index of the row of $LIST$ in which $WORD$ appears:

```
⍝DCL⍝ CHARACTER DOMAIN LIST,WORD
⍝DCL⍝ 5000 18 SHAPE LIST
⍝DCL⍝ 2 SHAPELBS WORD
⍝DCL⍝ 18 SHAPEUBS WORD
⎕IO←1
I←(LIST[;⍳⍴WORD]∧.=WORD)⍳1
```

A lower bound ($SHAPELBS$) for the shape of $WORD$ is given in order to avoid additional code for a special case. The index origin is explicitly given to simplify the code, since the index I is dependent upon it. For simplicity, $WORD$ is presumed to include a final delimiter if necessary. The declaratory comments and specification of the index origin are included to make this example complete in itself; in a larger context the information in some or all of these statements might be derivable and those ones might not appear. This fragment is translated to

```
      DO  1020  I1 = 1,5000
        DO  1000  I2 = 1,J1WORD
          IF (.NOT.(ALIST(I1,I2).EQ.AWORD(I2))) THEN
              F37 = .FALSE.
              GO  TO  1010
          END  IF
1000      CONTINUE
        F37 = .TRUE.
1010    IF  (F37)  GO  TO  1030
1020    CONTINUE
1030  JI = I1
```

Since all operations have been merged, not only may execution end early for both the Inner Product and the Dyadic Iota, but also the Monadic Iota and the use of its result to index $LIST$ appear to vanish—they have no computational cost. Notice that, although $LIST$ is arranged optimally for the APL storage order, in Fortran the performance of this code would be improved if the words were placed in the columns of $LIST$ rather than in the rows (and the arguments of the Inner Product correspondingly altered).

```
      Z←Y F X
[1]   ⍝DCL⍝ INTEGER DOMAIN Y,X
[2]   ⍝DCL⍝ 3 5 SHAPE Y
[3]   ⍝DCL⍝ 3 5 3 5 SHAPE X
[4]   Z←(Y*2)∘.+Y*2
[5]   Z←Z+×X
```

Figure 1

An example of a complete APL function.

```
      SUBROUTINE YAT040(VZ,M1Z,M2Z,M3Z,M4Z,J1Z,J2Z,J3Z,J4Z,JY,JX,JYATOE, (1)
     cJYATOM)
      IMPLICIT CHARACTER(A-D), LOGICAL*1(E-H), INTEGER(I-P), REAL*8(Q-Z) (2)
      DIMENSION VZ(M1Z,M2Z,M3Z,M4Z),JY(3,5),JX(3,5,3,5),V29(3,5)        (3)
C     Translated to VS Fortran at 4:23 p.m., May. 7, 1986 by BLVersion
C     3 & CGVersion 6, with 7 6 5 as global value of COMBINE.
      JYATOE = 0                                                       (4)
      JYATOM = 0                                                       (5)
      DO 1000 I2 = 1,5                                                 (6
        V29(1,I2) = DBLE(JY(1,I2)) ** 2                                 7
        V29(2,I2) = DBLE(JY(2,I2)) ** 2                                 8
1000    V29(3,I2) = DBLE(JY(3,I2)) ** 2                                 9)
      DO 1010 I4 = 1,5                                                 (10
        DO 1010 I3 = 1,3                                               11
          DO 1010 I2 = 1,5                                             12
            DO 1010 I1 = 1,3                                           13
              IF (JX(I1,I2,I3,I4).GT.0) THEN                           14
                L33 = 1                                                15
              ELSE IF (JX(I1,I2,I3,I4).LT.0) THEN                      16
                L33 = -1                                               17
              ELSE                                                     18
                L33 = 0                                                19
              END IF                                                   20
1010          VZ(I1,I2,I3,I4) = (V29(I1,I2) + V29(I3,I4)) + L33        21)
      J1Z = 3                                                          (22)
      J2Z = 5                                                          (23)
      J3Z = 3                                                          (24)
      J4Z = 5                                                          (25)
      END                                                              (26)
```

Figure 2

The Fortran translation of the APL function shown in Figure 1.

Figure 1 shows an example of an entire (small, rather trivial) function. Its Fortran translation is shown in **Figure 2** and the corresponding listing in **Figure 3.**

Several points about this example are worth noting. Fortran statement numbers 4 and 5 in Figure 2 set up JYATOE and JYATOM, the variables that are used for run-time error handling; there are no reportable error conditions in this function. The square of JY, although it is embedded in the outer product in the original APL, in Figure 1, is

```
   APL line number, then the original APL statement
+
+  Fortran ISNs; APL statement translated there
+  +
+  +
   (4)          JYAT0END←0
   (5)          JYAT0MSG←0


[0]   Z←Y F X
[1]   ∆DCL∆ INTEGER DOMAIN Y,X
[2]   ∆DCL∆ 3 5 SHAPE Y
[3]   ∆DCL∆ 3 5 3 5 SHAPE X
   (6-9)        YAT0L2 :


[4]   Z←(Y*2)∘.+Y*2
   (6-9)        V29←JY*2


[5]   Z←Z+×X
   (10-21)      VZ←(V29∘.+V29)+L33←×JX
   (22)         J1Z←3
   (23)         J2Z←5
   (24)         J3Z←3
   (25)         J4Z←5
   (26)         YAT0L0 :
```

Figure 3

The listing produced with the Fortran translation shown in Figure 2.

Table 1 Comparison of timings for defined and primitive Grade functions.

Size	*Ratio of IBM 308X execution times*	
	Interpreted/ Compiled	*Compiled/Primitive*
10	124	—
100	148	2.8
1000	151	1.8

Table 2 Comparison of timings for defined and primitive Matrix Divide functions.

Size	*Ratio of IBM 308X execution times*	
	Interpreted/ Compiled	*Compiled/Primitive*
5,5	7.6	5.0
10,10	6.5	2.7
20,20	3.9	2.1
40,40	2.5	1.9

calculated separately in the Fortran because its elements are used repeatedly in the outer product. In the listing, Figure 3, APL statement 5 appears before statement 4's expressions have been completely shown, because the Outer Product in 4 is merged with the operations of statement 5, and therefore the expression labeled (10-21) applies to both statements. Were Signum JX implemented in Fortran as a single statement (in addition to the DO nest in which it appears), the expression that defines Signum JX would appear right in the statement that sets the elements of VZ. Since in fact it requires several statements, a variable must be set; however, those statements can appear in the DO nest for the final addition and need set only a scalar, since this value is not used except in the addition. ISNs 22 through 25 set the shape variables of the result.

Sample execution times for compiled and interpreted code

It is interesting not only to compare the speed of interpreting some APL code with the speed of executing a compiled version of it, but also to compare the speed for the compiled version with that for a finely tuned Assembler version. Therefore, APL primitive functions, for which one can time the (Assembler) interpreter code, make good test cases when defined APL functions that model them are available. Of course, the comparisons are not likely to be in any sense exact; the APL models may diverge from the algorithms the interpreter employs, and techniques may be used in the Assembler coding which are not available in APL or which would be avoided there because they would be obscure. Nonetheless, the comparisons are useful, giving some notion of the accelerations one can expect for various kinds of APL programs and of possible bounds on such improvements.

We have two such examples. They are of special interest because they represent extremes of APL coding.

The first example is an APL defined function [26] in which essentially only scalars appear as arguments to operations, except for the Index function. It produces the same result as APL's primitive ordering function, which is called "Grade" and sometimes miscalled "Sort" although it yields a permutation vector and leaves its argument unchanged. The comparisons to be made, then, are between this low-level defined function as interpreted and as compiled, and between the compiled version and a simple call to the interpreter, via a single statement of the form $Z \leftarrow \grave{A} R$, almost all of whose time will be spent in the Grade code. In the defined function, interpretive overhead looms large, as can be seen in **Table 1.** The Assembler routine—the code for the primitive Grade function in VSAPL—runs somewhat faster than the compiled code, however, as seen in the right-hand column. (Grade was too fast to time meaningfully for a vector of length 10.)

In summary, our compiled version of an APL model for Grade is 124 to 151 times faster than the model as

```
   APL line number, then the original APL statement
+
+  Fortran ISNs; APL statement translated there
+  +
+  +
   (4)          JYAT0END←0
   (5)          JYAT0MSG←0


[0]   Z←Y F X
[1]   ∆DCL∆ INTEGER DOMAIN Y,X
[2]   ∆DCL∆ 3 5 SHAPE Y
[3]   ∆DCL∆ 3 5 3 5 SHAPE X
   (6-9)        YAT0L2 :


[4]   Z←(Y*2)∘.+Y*2
   (6-9)        V29←JY*2


[5]   Z←Z+×X
   (10-21)      VZ←(V29∘.+V29)+L33←×JX
   (22)         J1Z←3
   (23)         J2Z←5
   (24)         J3Z←3
   (25)         J4Z←5
   (26)         YAT0L0 :
```

Figure 3

The listing produced with the Fortran translation shown in Figure 2.

Table 1 Comparison of timings for defined and primitive Grade functions.

Size	*Ratio of IBM 308X execution times*	
	Interpreted/ Compiled	*Compiled/Primitive*
10	124	—
100	148	2.8
1000	151	1.8

Table 2 Comparison of timings for defined and primitive Matrix Divide functions.

Size	*Ratio of IBM 308X execution times*	
	Interpreted/ Compiled	*Compiled/Primitive*
5,5	7.6	5.0
10,10	6.5	2.7
20,20	3.9	2.1
40,40	2.5	1.9

calculated separately in the Fortran because its elements are used repeatedly in the outer product. In the listing, Figure 3, APL statement 5 appears before statement 4's expressions have been completely shown, because the Outer Product in 4 is merged with the operations of statement 5, and therefore the expression labeled (10-21) applies to both statements. Were Signum JX implemented in Fortran as a single statement (in addition to the DO nest in which it appears), the expression that defines Signum JX would appear right in the statement that sets the elements of VZ. Since in fact it requires several statements, a variable must be set; however, those statements can appear in the DO nest for the final addition and need set only a scalar, since this value is not used except in the addition. ISNs 22 through 25 set the shape variables of the result.

Sample execution times for compiled and interpreted code

It is interesting not only to compare the speed of interpreting some APL code with the speed of executing a compiled version of it, but also to compare the speed for the compiled version with that for a finely tuned Assembler version. Therefore, APL primitive functions, for which one can time the (Assembler) interpreter code, make good test cases when defined APL functions that model them are available. Of course, the comparisons are not likely to be in any sense exact; the APL models may diverge from the algorithms the interpreter employs, and techniques may be used in the Assembler coding which are not available in APL or which would be avoided there because they would be obscure. Nonetheless, the comparisons are useful, giving some notion of the accelerations one can expect for various kinds of APL programs and of possible bounds on such improvements.

We have two such examples. They are of special interest because they represent extremes of APL coding.

The first example is an APL defined function [26] in which essentially only scalars appear as arguments to operations, except for the Index function. It produces the same result as APL's primitive ordering function, which is called "Grade" and sometimes miscalled "Sort" although it yields a permutation vector and leaves its argument unchanged. The comparisons to be made, then, are between this low-level defined function as interpreted and as compiled, and between the compiled version and a simple call to the interpreter, via a single statement of the form $Z \leftarrow \grave{A} R$, almost all of whose time will be spent in the Grade code. In the defined function, interpretive overhead looms large, as can be seen in **Table 1.** The Assembler routine—the code for the primitive Grade function in VSAPL—runs somewhat faster than the compiled code, however, as seen in the right-hand column. (Grade was too fast to time meaningfully for a vector of length 10.)

In summary, our compiled version of an APL model for Grade is 124 to 151 times faster than the model as

interpreted, depending on the length of the vector whose grading vector is to be computed, while the actual primitive in APL is about twice as fast as our version.

The other example is a defined function that describes the VSAPL routine for the matrix-division primitive function [27]. In this defined function, many operations have array arguments. Here, the interpreter's efficient inner loops show to advantage; with a reduction in relative interpretive overhead as the arguments grow, the advantage of the compiler is less for larger arguments, as can be seen from **Table 2.**

These two examples are reasonably typical of our experience. We have compiled and timed a number of other APL programs and found that they ran from 2 to 250 times faster after compilation.

We believe that the results for these examples demonstrate that it is well worthwhile to compile APL through an intermediate language. We look forward to studying the performance of many programs compiled from APL and using the results to improve our translations.

Acknowledgments

The support over the years of A. D. Falkoff, the manager of the APL Design Group, enabled us to produce our translator. J.-J. Girardot, while at IBM Research on leave from Ecole des Mines, St. Etienne, France, made early studies which supported our approach to the compilation of APL.

References

1. J. Weigang, "An Introduction to STSC's APL Compiler," *Proceedings of the ACM Conference on APL*, 1985, pp. 231–238.
2. R. L. Johnston, "The Dynamic Incremental Compiler of APL-3000," *Proceedings of the ACM Conference on APL*, 1979, pp. 82–87.
3. E. J. Van Dyke, "A Dynamic Incremental Compiler for an Interpretative Language," *Hewlett-Packard J.* **28,** 17–23 (July 1977).
4. T. C. Miller, "Tentative Compilation: A Design for an APL Compiler," *Technical Report No. 133* (Ph.D. dissertation), Department of Computer Systems, Yale University, New Haven, CT, May 1978.
5. T. C. Miller, "Tentative Compilation: A Design for an APL Compiler," *Proceedings of the ACM Conference on APL*, 1979, pp. 88–95.
6. L. J. Guibas and D. K. Wyatt, "Compilation and Delayed Evaluation in APL," *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, 1978, pp. 1–8.
7. T. A. Budd, "An APL Compiler for the UNIX Timesharing System," *Proceedings of the ACM Conference on APL*, 1983, pp. 205–209.
8. J. M. Treat and T. A. Budd, "Extensions of Grid Selector Composition and Compilation in APL," *Info. Proc. Lett.* **19,** 117–123 (1984).
9. M. D. Kaplan and J. D. Ullman, "A General Scheme for the Automatic Inference of Variable Types," *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, 1978, pp. 60–75.
10. A. Tenenbaum, "Type Determination for Very High Level Languages," *Courant Computer Science Report No. 3*, New York University, NY, 1974.
11. P. A. Abrams, "An APL Machine," *SLAC Technical Report No. 114*, Stanford Linear Accelerator Center, Stanford University, CA, 1970.
12. A. Andronici, G. Leoni, and E. Luciani, "A Portable APL Translator," *Proceedings of the ACM Conference on APL*, 1975, pp. 20–25.
13. M. T. Compton, "APL in PL/I," *Research Report RC-4481*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1973.
14. M. A Jenkins, "Translating APL, an Empirical Study," *Proceedings of the ACM Conference on APL*, 1975, pp. 192–200.
15. V. L. Moruzzi, "APL/Fortran Translations," *Research Report RC-3644*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1971.
16. J. D. Sybalsky, "An APL Compiler for the Production Environment," *Proceedings of the APL '80 Conference*, North-Holland Publishing Co., Amsterdam, 1980, pp. 71–74.
17. E. A. Ashcroft, "Towards an APL Compiler," *Proceedings of the APL 6 Conference*, 1974, pp. 28–38.
18. A. J. Perlis, "Steps Toward an APL Compiler," *Computer Science Research Report No. 24*, Yale University, New Haven, CT, 1974.
19. H. J. Saal, "Considerations in the Design of a Compiler for APL," *APL Quote Quad* **8,** No. 4, 8–14 (1978).
20. C. Weidmann, "Steps Toward an APL Compiler," *Proceedings of the ACM Conference on APL*, 1979, pp. 321–328.
21. W.-M. Ching, "A Portable Compiler for Parallel Machines," *Proceedings of the IEEE Conference on Computer Design: VLSI in Computers*, 1984, pp. 592–596.
22. W.-M. Ching, "Program Analysis and Code Generation in an APL/370 Compiler," *IBM J. Res. Develop.* **30,** 594–602 (1986, this issue).
23. APL2, IBM Program Number 5668-899, Release 2. See *APL2 Programming: Language Reference*, Order No. SH20-9227, and *APL2 Programming: System Services Reference*, Order No. SH20-9218, available through IBM branch offices.
24. IBM IUP 5796-PPJ, APL Performance Analysis Tools, available through IBM branch offices.
25. IBM IUP 5796-PNB, APL Workspace Structure Analyzer, available through IBM branch offices.
26. L. J. Woodrum, IBM Data Systems Division, Poughkeepsie, NY, private communication.
27. M. A. Jenkins, "The Solution of Linear Systems of Equations and Linear Least Squares Problems in APL," *Technical Report No. 320-2989*, IBM New York Scientific Center, June 1970.

Received February 6, 1986; accepted for publication June 2, 1986

Graham C. Driscoll, Jr. *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.*
Dr. Driscoll is a Research staff member in the Computer Sciences Department, working on the APL compiler project. He joined IBM in 1959 at the Watson Research Center. He received a B.A. in literature from Columbia College, New York, in 1952, a certificate in mechanical engineering in 1957 and an M.S. in mathematics in 1959, both from the Rensselaer Polytechnic Institute Hartford Graduate Center, Connecticut, and a Ph.D. in mathematics in 1965 from Cornell University, New York, where he was an IBM Resident Study Fellow from 1961 to 1964.

Donald L. Orth *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Orth is a Research staff member in the Computer Sciences Department. He is currently manager of the APL compiler project. He joined IBM in 1974 at the IBM Scientific Center in Philadelphia, Pennsylvania. He received his B.S. in 1961 and his M.S. in 1963, both in mathematics, from the University of Notre Dame, Indiana. He received his Ph.D. in mathematics in 1967 from the University of California at San Diego. From 1967 to 1969, he was a National Science Foundation Fellow at Princeton University, New Jersey. Dr. Orth is the author of *Calculus in a New Key.*