Is APL a Suitable Tool for the Design of Large-System Software?

Kevin E. Jordan

Introduction

As a programming language, APL is not new. APL was first implemented in 1965, over fifteen years ago. Since its birth, a myriad of other programming languages have come into existence. The definitions of the most noteworthy of these have been strongly influenced by advances in, or at least theories from, the field of software engineering. Examples include Pascal, Euler, Alphard, CLU, and most recently, Ada.

These languages differ markedly from APL in that they demand strategic definitions of data structures while APL allows, even encourages, data structures to be dynamic. Alphard, for instance, requires a very precise description of the structure of data, as well as a precise description of the ways in which data are manipulated [9]. APL, on the other hand, provides virtually no way of explicitly declaring in a static fashion the type, shape, or method of access of data.

The software-engineering trend toward precise static descriptions of data and data access stems from the belief that programming errors should be caught before software is put into production. By requiring precise declarations, the programmer is forced to design software conscientiously, and inadvertent bugs can be caught more easily by the language translator due to an increase in redundancy of definition. It is believed by their proponents that the increased development cost entailed by languages like Alphard, Pascal, and Ada (as opposed to APL) is more than compensated for by a resultant decrease in maintenance cost.

The question is: Because APL rejects the notion of precise static declarations, is APL an inappropriate language in which to implement large software systems? This question will be explored from three different angles. First, the development cost of APL programs will be explored; second, the proof of correctness of APL programs will be looked at; finally, the ability to compile APL programs and apply data-flow analysis to discern types, shapes, and methods of access will be discussed.

Development Cost of APL Programs

APL, as it is usually implemented, provides an extremely supportive environment for software development. User-defined functions may be developed and tested one at a time without having to recompile other functions. Moreover, run-time errors do not cause cataclysmic system aborts. When an error is detected, APL reports the type of error, the name of the function in which the error was detected, the line of the function that was being executed, and a pointer to the token that was being processed at the time. At that point, APL merely suspends execution, allowing the programmer to interrogate the values of data objects, to examine the overall state of the workspace, and possibly to correct the error and resume execution at the point of the error. Thus APL provides for and encourages software walk-through. The language translator, and the language itself, are provided as sophisticated and very powerful debugging tools. In addition, most major APL implementations such as Aplum [7] offer explicit debugging aids that allow users to trace program execution, trap errors, and set break points in functions.

That APL provides wonderful diagnostics and exquisite debugging aids would be little consolation if APL programs were prone to errors. In a book entitled _The Mythical Man Month_, Frederick P. Brooks, Jr. [2] points out that experienced programmers can produce about 1200 lines of error-free code per year, regardless of the programming language used. If that statement is true, then it follows that the more concise a language is, the fewer errors will exist in an initial implementation of a design.

APL is a notably concise language. George Mayforth recently described [5] an interactive system that he implemented in APL as an interface between humans and a database-management system called Total. The interface had previously been done in Cobol. The Cobol implementation required 6950 lines of code while the equivalent APL implementation needed only 470 lines of code, a 15-to-1 ratio. In addition, the APL version required 86 work-hours to implement, while the Cobol version required 1280 work hours. The ratio in work-hours also comes out to be about 15-to-1. Thus, the use of either language yielded about 5.4 line of corrected code per hour.

Mayforth gives figures that weigh the monetary cost of executing the APL program versus that of executing the Cobol version. In a test that exercised equivalent functions of both implementations, the APL version cost 97.7 system billing units (SBU's) and the Cobol version cost 83.8 SBU's. In real time, however, the APL version required an average of 33 minutes to complete, while the Cobol version required an average of 36 minutes. Thus, although APL consumed 17% more computer resource, it executed in 6% less real time. Figuring development cost at $20 per programmer hour, the cost of the APL version was $1720, and the cost of the Cobol version was $25600. With the cost of one SBU at 35 cents, Mayforth points out that 4776 runs would be required to balance the production-cost difference between the APL and Cobol implementations.

Cobol is notably verbose, yet it is probably not more than 33% less concise than Pascal, Alphard, Ada, and others. Thus, a code-reduction factor of between 10- and 15-to-1 can probably be expected between these languages and APL. Given the code-reduction factor between APL and the software-engineering languages, and if the statement concerning lines of error-free code per year is true, then choosing APL as the language for implementing a given system should result in fewer errors than if the system were implemented in Pascal or Ada, for example. Moreover, any remaining errors will be made easier to find and correct by the APL programming environment.

Some errors, however, are subtle enough that they escape detection by the usually limited testing of the human programmer. As often as not, these errors turn out to be cataclysmic. A question remains then: Can APL programs be statically examined to detect errors? Specifically, is APL too dynamic to make analysis by automatic verification systems or automatic test-data generators infeasible? If the answer to these questions is yes, then APL is probably not a viable alternative to be used in the implementation of very large sotware systems.

Automatic Verification of APL Programs

At first glance APL seems like an impossible language for which to build an automatic program verifier. It is true that APL's lack of explicit data declarations makes verification of APL programs a challenge, but Susan Gerhart has shown [3] that given two equivalent programs written in APL and in a structured language such as Pascal, the APL program is actually less difficult to prove correct. Once again, this is the result of the conciseness of APL. APL owes its conciseness to the power of its primitive functions. APL's basic data structure is the array and its primitive functions are designed to operate on arrays. For example, in the simple APL expression:

```
Z←A+B
```

A and B can be arrays of any rank and size, and the primitive function + will add them together element by element as long as A and B are the same shape or one of them is a one-element array. To mimic the semantics of this expression in a language such as Pascal requires something like:

```
FOR I := 1 TO LENGTH_OF_A DO
  Z[I] := A[I]+B[I]
END;
```

and this code mimics the APL expression only when A and B are both one-dimensional arrays of known length. Much more Pascal code would be required to capture all the semantics of the APL expression.

In another example, consider a program to perform a table look-up. The program accepts as input a matrix to be searched and a vector to search for. The program must return as a result the number of the first row that matches the vector, or 0 if no match is found. In APL, the program can be implemented as follows:

```
∇ Z←MAT SEARCH VEC
  Z←1↑(MAT∧.=VEC)/ιρ1↑ρMAT
∇
```

In Pascal, the subprogram would look something like this:

```
PROCEDURE SEARCH
  (MATRIX: ARRAY[1..NO_OF_ROWS,
    1..NO_OF_COLS] OF CHAR;
  VECTOR:ARRAY [1..NO_OF_COLS] OF CHAR;
  NO_OF_ROWS, NO_OF_COLS: INTEGER;
  VAR RESULT:INTEGER);
VAR
  ROW, COL: INTEGER,
  MATCH: BOOLEAN;

BEGIN
RESULT:=0;  ROW:=1;
WHILE (ROW<=NO_OF_ROWS) AND (RESULT=0)
DO BEGIN MATCH:=TRUE;
  COL:=1;
  WHILE (COL <= NO_OF_COLS) AND
    (MATCH=TRUE) DO BEGIN
    IF MATRIX[ROW,COL]=VECTOR[COL]
      THEN COL:=COL+1
      ELSE MATCH:=FALSE
    END;
  IF MATCH=TRUE THEN RESULT:=ROW
    ELSE ROW:=ROW+1
END;
END;
```

The APL version is much more concise and much less prone to errors. Furthermore, the APL version can operate on characters as easily as it can operate on numeric arguments. The Pascal program, on the other hand, must be recompiled with new declarations each time the programmer wants it to operate on arguments of a different type.

Clearly, APL's conciseness stems from the fact that most of the looping operations that occur in APL programs are implicit in the primitive functions. Gerhart points out that verifying loop conditions and finding paths of execution are the most difficult parts of program verification. She shows that because most loops are implicit in APL's primitive functions, APL programs have fewer paths and are therefore easier to verify than equivalent programs in other programming languages.

The most difficult part of Gerhart's work was to pin down data types and shapes. Since APL programs do not contain explicit formal declarations of data types and shapes, this information must be determined through context analysis. Thus, the bulk of Gerhart's automatic verification system was centered around data-flow analysis. 'Knowing' the domain and range of each primitive function, Gerhart's data-flow analyzer simply formed a conjunct of the output predicates of each primitive function along a path to build the output predicate of that path. Whenever possible, the conjunct was simplified and, if a conflict occurred between the conjunct at any given point and the input predicate of the next primitive function to be applied, an error was signalled.

When given the input and output predicates of an APL program, Gerhart's system was able to verify that the input constraints did not conflict with the output constraints. More interesting, however, was the ability of the system to produce an input predicate when it was given the output predicate. Certainly, the ability of the system to produce input predicates has implications for automatic test-data generation. A test-data generator could use the input predicate to produce a set of data to feed the object program.

At this point, the cost of developing programs in APL has been shown to be less than the cost of developing equivalent programs in other languages. In addition, Gerhart has shown that proving APL programs to be correct is easier than proving equivalent programs in many other languages to be correct. These findings imply that APL programs are less costly to maintain than equivalent programs developed in other languages. Because APL is usually implemented as an interpreter, however, APL programs usually cost more to execute on a computer. This flaw could be remedied through true compilation of APL programs.

Compilation of APL Programs

In a recent paper on the compilation of APL, Clark Wiedmann [8] gives some empirical evidence of the inefficiency of a typical interpreter. He determined the number of machine instructions executed by Aplum to evaluate the simple expression:

```
    K←K+1
```

to be about 800. This number is awesome, especially when K is a scalar variable, but it is not surprising when considering the necessity of the interpreter to be general. A compiler, on the other hand, might be able to reduce this expression to two or three machine instructions--or even a single machine instruction if K is a value that can be kept in a machine register (i.e. a loop counter). Thus, a reduction factor in execution time of from 100 to 1000 is feasible for some programs if compilation is possible.

We believe [4] that an APL compiler is possible. When compared to the grammar of Pascal, Algol, or Ada, the grammar of APL is trivial. When certain conditions are met, therefore, parsing APL programs is an easy task. The only difficulty arises with the syntax of identifiers. For example, the expression:

```
    A B C
```

is totally ambiguous when examined statically. From this statement alone, it is impossible to tell:

--whether C is a variable or a niladic function,
--whether B is a monadic or a dyadic function,
--or whether A is a variable or a monadic function.

In practice, statements like this are rare. George Strawn has shown [6] that in practice, 95% of APL identifiers are not ambiguous, especially when considerd in context. This percentage can be increased to nearly 100% if APL's scope rules are changed [8].

Presently, APL allows variables to be declared local to a function if they are placed in the header of the function. All other identifiers referenced by a function are, by default, global objects. Global objects may be either variables or other functions. In addition, when one function calls another, all variables declared local in the caller are available to the callee. Thus, it is possible for the callee to cause side effects (whether intentionally or not) to so-called local variables of the caller.

If APL's scope rules were to be changed so that global variables must be declared, each function to be called must have its specific syntaxes declared (niladic, monadic, dyadic), and all other identifiers referenced within a function are by default strictly local, then all of the ambiguities vanish. Moreover, since strictly local variables are known only to the functions within which they are defined, unwanted side effects disappear as well. With this change in scope rules, APL programs may be parsed trivially by a compiler. In fact, based on these changes, an LL(1) parser for APL has been constructed [4] and operates on a grammar that requires only nineteen productions to define the APL language. (See Appendix.)

The real work involved with the compilation of APL programs grows from the need to produce efficient object code. Without the employment of data-flow analysis to perform optimization of the code produced by the parser, the object code generated by the compiler would be overly general, and its execution would probably be no more efficient than the execution of the APL interpreter. Since the types and shapes of data objects are not explicitly declared in APL, data-flow analysis must be utilized to learn this information from context.

The interaction of forward data-flow analysis with backward data-flow analysis can pin down virtually all the types and shapes. Those values whose types and shapes cannot be determined, or vary widely, will simply result in the generation of very general code unless the user supplies more specific information. In other words, in keeping with the spirit of APL, explicit data declarations will not be required, but the user should be aware that those objects whose characteristics cannot be pinned down may possibly yield overly general code.

Consider the following sequence of code:

```
[1]   A←+/B
[2]   C←B[4]
[3]   D←A+1
```

Even though no declarations are provided by the user, a compiler using data-flow analysis can determine the types and ranks (number of dimensions) of all the variables in this example. The analysis would proceed as follows:

--Beginning with forward flow analysis, all that can be determined in Line [1] is that B is numeric or empty.
--At Line 2, B is found to be a non-empty one-dimensional array, hence C must be scalar.
--A deduction backward from Line 2 would then discover that in Line 1 B must be numeric and, therefore, A must be a numeric scalar.
--Another forward deduction would then result in the knowledge that C is numeric, and in Line 3 that D must be a numeric scalar because A is.

The only characteristic that cannot be determined is the exact size of B.

Using data-flow analysis, Bauer and Saal [1] discovered that the type and shape attributes of 75% of the data objects could be pinned down in thirty test programs. This figure is extremely encouraging, considering that their basic block of analysis was a single line of code and that they allowed information found at one line to flow to the next, only if the subject program had no branches. It is likely that an approach that breaks programs up into more conventional blocks by finding branch statements and their targets, and that performs a true analysis of the flow of data between blocks along a path, can yield a figure higher than Bauer and Saal's 75%.

On a different note, it should be made clear that APL compilers will not supplant APL interpreters. It is the interpretive environment that facilitates software development. Functions will still be created and debugged using the interpretive environment; only when the programmer is reasonably certain that a function is correct should it be compiled.

<u>Limitations</u> <u>of</u> <u>APL</u>

As of this writing, APL has four major limitations. First, APL operates only on rectangular arrays--there are no list or record structures as in some other languages; second, user-defined functions may be passed a maximum of two arguments and may only return one result; third, APL does not provide for separate compilation; fourth, APL does not allow users to define their own data types. The first three limitations may soon disappear, but the fourth limitation cannot be remedied without violating the spirit of APL.

General heterogeneous arrays will soon be universally accepted into the APL community. General arrays, often called nested arrays or arrays of arrays, will allow users to build non-rectangular data-structures and will provide for mixing of data types within a single object. General arrays can be considered an analog of Pascal's "record" type.

The limitation that user-defined functions may be passed a maximum of only two arguments and may deliver only one result is a recognized problem. At this point, there is no technical difficulty in implementing remedies beyond generalizing the types of arrays allowed. Implementers, however, are correctly reluctant to implement new features unless there is agreement on the details in the APL community, and a good understanding of possible side effects.

Separate compilation is the ability to develop related modules of a software system independently. When it is time to integrate a module with others, all names not essential to its interfaces are hidden from other modules, thus avoiding collision of name spaces. To my knowledge, no currently available APL system supports such separate compilation.

Finally, user-defined data types create two problems. First, user-defined data types require precise declarations, and precise declarations are not in keeping with the spirit of APL. Second, user-defined types are usually used to define how a program intends to use the computer's memory. This often has machine-dependency ramifications. APL, however, is a machine-independent language; it attempts to insulate the user from hardware peculiarities as much as possible, putting on the implementors the burden of exploiting each feature where most appropriate.

Conclusions

The current trend in the field of software engineering toward languages that require explicit, precise declarations of data objects implies that APL, a language that rejects the idea of explicit data declarations, is unacceptable as a tool in which to implement software. It is widely believed that new languages like Pascal, Alphard, and Ada tend to reduce development and maintenance costs. Because of its conciseness, however, APL can reduce development and maintenance costs even more than these "structured" languages. APL is much more concise and, therefore, much less prone to errors. Because of this conciseness it might be said that APL is an even higher-level language than these other more recent languages. In the words of Clark Wiedmann, "an ounce of simplicity is worth a pound of redundancy." At present, APL does present a few inconveniences for the development of large software systems. These limitations are, however, in great part due to the forms of implementation, and may be overcome in the future.

Kevin E. Jordan
University Computing Center
Graduate Research Center
University of Massachusetts
Amherst, Massachusetts
USA 01003

Appendix. An LL(1) Grammar for APL

The following grammar follows APL's law of execution which states: expressions shall be executed from right to left. Thus, if the grammar's syntax seems strange, bear in mind that the lexical analyzer delivers tokens in the same order that it processes source text, i.e. right to left.

```
<function>::=<stmt_list>¦ε
<stmt_list>::=<stmt><stmt_list'>
<stmt_list'>::=<stmt_list>¦ε
<stmt>::= <stmt_body>end_of_line
<stmt_body>::=<exp_list>T-¦ε
<exp_list>::=<exp><exp_list'>¦;<exp_list>
<exp_list'>::=;<exp_list>¦ε
<exp>::=<simple_exp><sub_exp>
<simple_exp>::=<index_list><simple_exp'>¦
  <simple_exp'>
<simple_exp'>::=)<exp>(¦constant ¦variable ¦
  niladic_function
<index_list>::=]<exp_list>[
<sub_exp>::= <assignment>¦-¦<axis_op>¦
  <fn_exp>¦ε
<fn_exp>::=<dyadic_exp>¦<monadic_exp>
<dyadic_exp>::=dyadic_function<simple_exp>
  <sub_exp>
<monadic_exp>::=monadic_function<sub_exp>
<axis_op>::=right_axis_bracket <exp>
   left_axis_bracket
<assignment>::=-<target>
<target>::=<index_list><target'>¦<target'>
<target'>::=variable<sub_exp>
```

References

[1] Alan M. Bauer and Harry J. Saal. Does APL really need run-time checking?, Software--Practice and Experience 4 (1974) pp. 129-38.

[2] Frederick P. Brooks, Jr. The Mythical Man-Month, Addison-Wesley Publishing Company, Reading, Mass. (1975).

[3] Susan Lucille Gerhart. Verification of APL Programs, PhD. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Penna. (Nov. 1972).

[4] Kevin E. Jordan and Clark Wiedmann. Research project, University Computing Center, University of Massachusetts (1980).

[5] George R. Mayforth. An APL-TOTAL Interface, APL79 Conference Proceedings APL Quote Quad 9 4 Part 1 (June 1979) pp. 397-408.

[6] George O. Strawn. Does APL really need run-time parsing?, Software--Practice and Experience, Vol. 7, (1977) pp. 193-200.

[7] Clark Wiedmann. APLUM Reference Manual, Control Data Corporation, Minneapolis, Minn. (1978).

[8] Clark Wiedmann. Steps toward an APL compiler, APL79 Conference Proceedings, APL Quote Quad 9 4, Part 1 (June 1979) pp. 321-28.

[9] Wm. A. Wulf, Ralph L. London, and Mary Shaw. Abstraction and Verification in Alphard: Introduction to Language and Methodology, Carnegie-Mellon University and USC Information Sciences Institute (June 1976).

⎕