TOWARDS AN APL COMPILER

by

E.A. Ashcroft

Research Report CS-74-01

Department of Applied Analysis and
Computer Science

University of Waterloo
Waterloo, Ontario, Canada
January, 1974

TOWARDS AN APL COMPILER

E. A. Ashcroft
Computer Science
University of Waterloo
Waterloo, Ontario
Canada

Keywords: Compilers, Interactive Systems

Introduction

In this paper we show that a modification of the APL function calling mechanism allows compilation of defined function in many cases. Familiarity with the APL language is assumed; suitable references for the language are the APL/360 Users Manual (IBM, GH 20-0683-1) and APL/360 Reference Manual (S. Pakin, Science Research Associates Inc.).

There are several features of APL which make it difficult to design a compiler for the language. Among these are:

i) Identifiers are typeless; not only the values but also the types of values associated with an identifier can change during execution.

ii) Arrays can be created, at any time during execution, whose numbers of dimensions depend on run-time values.

iii) The language is interactive; there is no such self-contained thing as a program.

We will quickly consider the effect of (iii). Because there are no programs to compile, we must limit ourselves to compiling user-defined functions. However not all user-defined functions should have to be compiled. Typically, the user takes advantage of the interactive nature of APL and fuilds up his functions by stages. Each function will go through successive modifications until it reaches a stable state of, hopefully, correctness. Only then would it be worthwhile compiling it.

We can thus envisage an APL compiler which is invoked by the user (with a built-in APL function or by a system command) to compile specified defined functions. A typical piece of computation will then consist of alternately interpreting some APL statements and executing some machine code.

Since the compiler is going to be applied selectively, it is not unreasonable to impose restrictions on the sorts of defined functions to which it can be applied. Of course we shall endeavour to make the set of 'compilable' functions as large as possible, but will not be overly concerned if some functions are not compilable (especially if, in some cases, this is a result of poor programming style).

We shall now consider the more substantial problems resulting from properties (i) and (ii) above. We shall find that many of the difficulties are resolved by a modification of the APL function-calling mechanism. Not only does this make compilation feasible in many cases, but it also significantly improves the power and expressibility of the language.

Type-determination at compile-time

The major difficulty resulting from points (i) and (ii) above is that it is not possible in general to determine, at 'compile-time', the types of values that identifiers will possess at run-time. In this paper we will take the view that once this difficulty is removed, defined functions can be compiled relatively straightforwardly : type-determination is the main problem we will consider. Moreover, we shall take 'type' to be synonomous with 'rank', i.e. 'number of dimensions': a scalar value is of type 0 , a vector is of type 1, and an n-dimensional array is of type n .

It would clearly be advantageous to a compiler to be able also to determine the shape of the values to be possessed by identifiers, i.e. the lengths of the dimensions as well as the number of dimensions. This would require even more restrictions to obtain compilable functions than we give here, so we will not consider it further. Here, a function is 'compilable' if and only if it is possible to determine the rank or type of the values associated with each identifier-occurrence in the function. (We will allow the same identifier to have different types of values at different points within the function-definition. Provided each occurrence of an identifier, at a particular syntactic position in the function-definition, is associated with values of a single type, then we will say that we can determine types.)

To determine types we must associate a unique type with every expression in the function definition. We ensure this by requiring two properties. Firstly, the expression must be type-transparent, i.e. the

type of its result must depend only on the types of its arguments; given the types of the values for the identifiers in the expression we must be able to specify the type of the result of the expression (if any). Secondly, the types of values that these identifier-occurrences can possess must be fixed, to ensure that the result is of fixed type. This argument may sound circular: for type determination we need type determination. In fact, it is just a sort of consistency, which we will call type-consistency.

Type-transparency

Type-transparency of expressions is ensured if all the basic operations and defined functions are type-transparent.

The problem of making defined functions type-transparent will be considered later.

Most APL basic operations are type-transparent, but there are two exceptions.

1. Dyadic-ρ: A ρ B is an array with shape A , and rank ρA . The type of the result depends on the length of vector A .

2. **Dyadic transpose ⌽ : the rank of A ⌽ B depends on the number of** different elements in the vector A .

To avoid these cases we must impose our first restriction on 'compilable' functions:

Restriction 1.

In a compilable function, every occurrence of dyadic-ρ or dyadic-⍵ must have a left argument consisting of a constant expression, i.e. an integer or a sequence of integers.

The effect of this restriction can be minimised by limiting other uses of dyadic-ρ and dyadic-⍵ to relatively small functions, possibly calling other, larger, compilable functions. Hopefully, in this way we can keep the interpretive phase of execution as short as possible (as noted previously, we expect to have mixtures of compiled and non-compiled functions anyway).

Type-consistency

Once all expressions in a defined function are type-transparent, the results of these expressions will have fixed types if the identifiers in the expressions have fixed types. If the types of the parameters of the function, and the types of the global variables, are fixed, then it is straightforward to derive the types of identifier-occurrences at subsequent positions in the definition text, by essentially following all possible execution-paths, up to repetitions of statements, keeping track of the types possessed by identifiers. When statements are repeated, if the types of any (subsequently used) identifiers have changed, then we have type-inconsistency. Otherwise the definition will be type consistent, for the given initial typing of parameters and

globals, i.e. for parameters and globals of these types, the type of every expression is fixed for all computations. This will be one of our requirements for compilability:

Restriction 2.

In a compilable function, we must have type consistency for some initial typing of parameters and globals. (We can assume type transparency from the rest of the Restrictions, so type-consistency is **checkable.) (The initial typing is given by other restrictions.)**

This is not a drastic restriction. Type inconsistency usually results from poor programming style - using the same variable for two purposes simply because it is possible to write one piece of code to take care of both cases.

All that remains for type-determination in a defined function is to satisfy the following requirements:

a) ensure that all defined functions used within the function are type-transparent.

b) ensure that the type of the function's parameters and globals are fixed.

One way to do both (a) and (b) is to allow type declarations in function headings, as in Algol, specifying the types of the parameters

and the type of the result of the function. A function called with the wrong types of arguments would produce an error.

This solution is unnecessarily restrictive. A better solution will be given after we look at the current function-calling mechanism in APL, and one of its desirable side effects.

Typeless-parameter-passing

Since the formal parameters of defined functions are typeless, it is possible to call the same function for arguments of arbitrary types. This is a very useful feature, and one of the distinguishing features of APL.

There seem to be three main ways in which this feature is used:

A. The function may decide what type of argument it has been given, and do completely different calculations depending on the decision. The function is essentially the union of a finite number of other functions.

B. The function may have been designed to work on some simple types of arguments, but continues to work for higher types because the basic APL operations often work for higher types. The result of the function will usually be some array containing the results that would have been obtained if the function had been applied to the simpler-type components of the arguments. This use will be discussed further below.

C. The function may have been designed for all types of arguments, but the result of the function is not simply an array of the results for the simpler subarrays of the arguments; i.e. not case (B). (For example, the following function REV , for array argument A , returns an array B identical to A except that each dimension is reversed:

```
     ∇B ← REV A; N
[1]    N ← 0
[2]    B ← A
[3]    → ((⍴⍴A) < N ← N+1)/0
[4]    B ← ⌽[N]B
[5]    → 3
[6]    ∇
```

Of these three uses, case A seems least essential; we will assume that such functions are replaced by their component functions. (If a function containing such a case A function is to be compilable then the types of the arguments to the case A function will be determined, and we can call the appropriate component function instead.) Case C is trickier. We will not be able to compile case C functions, but they may be found to be type-transparent, and thus may be used within other compilable functions. This point will be returned to later; for the moment we will forget case C functions.

This leaves us with the case B functions. They constitute a feature of APL which is particularly appealing. Suppose we write the following function AT to give the percentage increase on a sum invested for N years at I percent:

```
     ∇INC ← N  AT  I; I1
[1]    INC ← M ← 0
[2]    I1 ← 1 + I ÷ 100
[3]    → (N < M ← M+1)/0
[4]    INC ← I + I1 × INC
[5]    → 3
[6]    ∇
```

It is very gratifying to find that, if it is given a vector I of interest rates it will return the vector of corresponding percentages increases. On the other hand, it is annoying when one then tries supplying a vector N of years instead and gets the result corresponding to the smallest element of N . The reason why the function 'extends' for one argument and not the other can be found by looking at the code of the defined function. There is a logical reason why it does not extend for N - the number of iterations of the loop gives the percentage increase for that number of years, and the total number of iterations is a single number. On the other hand, argument I does extend, not for logical reasons, but because the basic APL operations + and × , that are applied to I , happen to work for both vectors and scalars. The final result of the function comes from these operations on vector I , giving a vector of values. In fact, our function effectively behaves like a primitive operation as far as I is concerned - it extends in the same way. This effect can be achieved deliberately, but it is remarkable how often it occurs unintentionally; it might be termed the Serendipity Effect.

We shall show that by modifying the function-calling mechanism, we can make the uniform extension of functions a part of the language, and at the same time satisfy our type-determination requirement (a) and (b)

Typed-parameter-passing

The idea is very simple. Consider the function AT . We define the result of supplying a vector of interest rates I to be the vector of results for the corresponding scalar components of I . We can similarly define the result for a vector of years N . We can realise this defined behaviour by a more complicated function-calling mechanism. Instead of simply passing over the vector actual parameter N say, it passes over the scalar components of N , and the function is re-evaluated each time. The calling mechanism then assembles these results into a vector - the result of calling the function. All this could be achieved by the generation, by the calling mechanism at run-time, of a simple loop within which the actual call of the function is embedded. It can also be made to work for arguments of higher types, by the generation of several nested loops. Thus we can make the function extend uniformly.

Of course, the calling mechanism must be told that the function requires scalar arguments and returns a scalar result. This can be achieved by declarations in the function heading: a function with such declarations we shall say is 'typed'. We suggested parameter declarations earlier, as a way of satisfying requirements (a) and (b). Note, however, the crucial difference in the present case - if called with parameters of higher type than those declared, the function will still work.

Nevertheless, note that the function is only actually executed for scalar parameters. This satisfies half of requirement (b) for type-determination. The second half can be satisfied by declaring or prohibiting globals in compilable functions:

Restriction 3.

A compilable function must be 'typed', by declarations of its parameter and result types, and must also have declarations for the types of all its global variables. The function may not change the type of any global, and the result type must be correct (both can be checked if the function is type consistent for the declared initial types).

All that remains for type-determination of a function is to satisfy requirement (a) - type-transparency of the functions used within the given function. If these functions are typed, then for any ranks of arguments we should be able to specify the ranks of the result from the conventions we use in extending functions. There must be general rules for defined functions of all types, and the next section suggests suitable notation and conventions. This notation was settled on after many discussions with D. M. Jackson, to whom the author is indebted.

The extension of typed functions

For typed function F , with heading ∇C ←A F B, the 'type' of the function will be (i, j, k) if A is declared as type i , B as type j and C as type k . Similarly for monadic functions.

In the simplest cases, type (0, 0, 0) , we should expect functions to extend in the same way as scalar operations like addition. In A+B , if A and B are both scalars we get a scalar result: the sum. If A is scalar but B is an array, we get an array as result: the result of adding A to every element of B . If both A and B are arrays of the same type, and corresponding dimensions are of the same size, then the result is an array of the same type and size, in which each element is the sum of the corresponding elements of A and B . (If A and B do not satisfy the above conditions, the result is undefined.)

There are really two extension mechanisms at work here. If the first mechanism were used in the second case, we would expect a result of higher type, containing the sum of all pairs of elements from A and B ,

e.g.

$$(1\ 3\ 4\ 7) + (9\ 1\ 6\ 2) = \begin{pmatrix} 10 & 2 & 7 & 3 \\ 12 & 4 & 10 & 5 \\ 13 & 5 & 10 & 8 \\ 16 & 8 & 13 & 9 \end{pmatrix}$$

On the other hand, if the second mechanism were used in the first case, the result would be undefined.

The first mechanism we shall call a 'free association', and the second we shall call a 'paired association'. The example above is the 'free' addition of (1 3 4 7) and (9 1 6 2) . Note that the type of (the result of) a 'free' addition is equal to the sum of the types of the arguments, whereas the type of a paired addition is equal to the (identical) types of the arguments.

We introduce a notation to indicate the results of typed functions (of all types), which agrees with the basic operator extension mechanisms for type (0, 0, 0) functions. Even for (0, 0, 0) functions, we introduce extra generality, for example allowing free association for array arguments. (It is natural, in the percentage-increase example, to give a vector of years N and a vector of interest rates I , and expect to get back a TABLE of results.)

In the following, F is a function of type (i, j, k) , A is of type n , B is of type m , U and V are vectors of positive integers, permutations of subsets of the dimensions of A and B respectively, i.e. $\rho U \leq n$ and there are no repeated elements, and similarly for V . We wish to specify the result X of $(\neq U \neq A)\ F\ \neq V \neq B$. The definition of $F\ \neq V \neq B$, for monadic function F of type (j, k) is similar; simply ignore all references to A, U, n and i . If U or V are empty, it is equivalent to omitting the brackets, so A F B is a special case of this notation. The dimensions in brackets are always left 'free'. Let $p = n - (\rho U + i)$ and $q = m - (\rho V + j)$.

i) If $p < 0$ or $q < 0$ then X is undefined, otherwise X is of rank $\rho U + \rho V + k + (p \lceil q)$.

ii) If U is non-empty, say $U = \alpha U'$ for some $\alpha \leq n$, then

$$X[i_1;;\ldots;] = (\neq U' \neq A[;;;\overset{\leftarrow \alpha \rightarrow}{i_1};;;])\ F\ \neq V \neq B$$

for all appropriate indices i_1 .

iii) If U is empty and p > q then

$$X[i_1;;\ldots;] = A[i_1;;\ldots] \; F \; \dagger V \dagger \; B$$

for all appropriate indices i_1 .

iv) If U is empty and $p \leq q$ then

a) if V is non-empty, say $V = \beta V'$ for some $\beta \leq m$, then

$$X[i_1;\ldots;] = A \; F \; \dagger V' \dagger \; B[;;\ldots;\underset{\leftarrow\ \beta\ \rightarrow}{i_1};;;]$$

for all appropriate indices i_1 .

b) if V is empty and $p < q$ then

$$X[i_1;;\ldots;] = A \; F \; B[i_1;\ldots;]$$

for all appropriate indices i_1 .

v) If V is empty and p = q then

$$X[i_1;\underset{\leftarrow k\rightarrow}{i_2\ldots i_p};;;;] = A[i_1;\underset{\leftarrow i\rightarrow}{i_2;\ldots;i_p};;;;] \; F \; B[i_1;i_2;\ldots;\underset{\leftarrow\ j\ \rightarrow}{i_p};;;;\ldots;]$$

for all appropriate $i_1, i_2, i_3, \ldots, i_p$. To be defined in this case, the first p dimensions of A and B must match in length, and all the rank-k results of F (as expressed by the right-hand-side of the above identity) must have the same shape.

This formal definition may be a little hard to follow. A few examples should help. Consider scalar functions (i.e. type (0, 0, 0)) such as addition. If arrays A and B are the same shape, then A+B is defined by case (ivc) , which is exactly the usual definition of array

addition. If A is a vector and B is a scalar, then A+B is defined by case (iii), which again is the usual definition of addition. On the other hand, if A is a two dimensional array and B is a vector of the same length as the last dimension of A , then A+B is defined by adding B to each row of A (case (iii) followed by case (ivc)).

e.g.
$$\begin{matrix} 7 & 1 & 6 & 2 \\ 9 & 3 & 1 & 4 \\ 5 & 9 & 9 & 6 \end{matrix} + \begin{matrix} 6 & 3 & 1 & 5 \end{matrix} = \begin{matrix} 13 & 4 & 7 & 7 \\ 15 & 6 & 2 & 9 \\ 11 & 12 & 10 & 11 \end{matrix} .$$

If we want to add B to each column of A , we simply leave free the second dimension of A : (⍤2⍤A) + B

e.g.
$$\left(\begin{matrix} ⍤2⍤ & 7 & 1 & 6 & 2 \\ & 9 & 3 & 1 & 4 \\ & 5 & 9 & 9 & 6 \end{matrix}\right) + \begin{matrix} 6 & 3 & 1 \end{matrix} = \begin{matrix} 13 & 12 & 6 \\ 7 & 6 & 10 \\ 12 & 4 & 10 \\ 8 & 7 & 7 \end{matrix}$$

The result is transposed because the free dimensions come before the paired dimensions.

In general, for (0, 0, 0) functions, the dimensions U of A and V of B are left free; as many dimensions as possible from the right are paired (of those not in U and V); and the remaining dimensions, either all in A or all in B , are free. In the result, the free dimensions of A come first (U preceding the rest), then come the free dimensions of B (V preceding the rest) and then we have the paired dimensions. For (i,j,k) functions, the last i and j dimensions of the two arguments are the ones to which the functions are applied; the rest are treated as above.

Using these rules, it can be seen that inner product A f.g B can be represented by f/A g⍤1↓⍳⍴⍴ B⍤B (matrix product is +/A×⍤2⍤B) , and outer product A∘.fB is (⍤⍴⍴A⍤A)fB (or equivalently A f⍤⍴⍴B⍤B) . The free addition of two vectors A and B is simply ⍤1⍤A + ⍤1⍤B or ⍤1⍤A + B or A + ⍤1⍤B (a special case of outer product).

Catenation is an example of a type (1, 1, 1) basic operation, and the way it normally extends to arrays satisfies the rules given above. The rules also allow the catenation of, for example, a rank 2 array and a vector:

$$\begin{pmatrix}1 & 2 & 5 & 4\\ 9 & 1 & 6 & 7\end{pmatrix}, (2 \quad 1 \quad 6) = \begin{pmatrix}1 & 2 & 5 & 4 & 2 & 1 & 6\\ 9 & 1 & 6 & 7 & 2 & 1 & 6\end{pmatrix}.$$

Catenation is one of a number of basic vector operations (catenation, reduction, reversal, rotation, lamination, compression and expansion) which allow a square bracket notation to indicate which dimension is to be the one to which the operation is to be applied. This feature could be combined with the ⍤ ⍤ notation, but we will not pursue this here. The ⍤ ⍤ notation was chosen because it looks similar to the square bracket notation, but it must be emphasised that the two notations pick out dimensions for completely different purposes.

The notation introduced here seems flexible, powerful and useable. For the purposes of this paper, its important feature is that it makes it possible to determine the type of the result of a function call in terms of the types of the arguments.

Type-transparency of defined functions

If a function F is typed, of type (i,j,k) say, then the type of (⍞U⍞ A)F ⍞V⍞ B is ρU + ρV + k + (p⌈q) , where p = n - (ρU + i) and q = m - (ρV + j) . Provided the lengths of U and V are fixed, the expression is then type-transparent.

Restriction 4.

In a compilable function, all calls of typed functions ⍞U⍞ A F ⍞V⍞ B must be such that U and V are constant expressions, i.e. literally integers or sequences of integers.

We could now ensure type transparency of all function calls within a given function by requiring that all such called functions be typed. In fact, we can be a little more general than this. It is possible to have functions, particularly case C functions, which can not be typed in the sense we have been considering but which are verifiably type-transparent. Such functions will not be compilable, because of Restriction 3, but could still be called from within other compilable functions.

For such functions, we would require declarations which express the type of the result as, say, an arithmetic expression in terms of the types of the arguments and globals. The types of the arguments and globals would not be fixed, but would be denoted by some conventional symbols, say 'N', 'M' etc., meaning arbitrary types. The type of the (monadic) case C function REV given earlier would be (N, N) for example; for any type of

argument, it returns an array of the same type. Functions with such declarations we can say are p-typed, meaning the typing is parameterized.

Of course, it is not sufficient to simply give the p-typing, we must also check that the p-typing is correct. To do this we need a sort of parameterized type-consistency check; starting with the initial typings N , M etc. we follow all executions, keeping track of the types of identifiers as arithmetic expressions in N , M etc. (To do this we need type-transparency of expressions.) If we repeat a statement during the checking, with different parameterized typings of some (subsequently used) identifier, or if we change the type of a global, then the type consistency check fails. (Some limited deductive power will probably have to be built into the checker to decide whether two expressions denote the same types or not.) If the check does not fail, and the result type agrees with the one declared, then the p-typing is correct. Thus for p-typing we need type transparency and a sort of type-consistency. The p-typed functions therefore must satisfy all the restrictions for compilable functions, except Restriction 3: we do not determine types. Of course, if we call a p-typed function with fixed types of parameters and globals we get a determined type of result, i.e. it is type-transparent.

The final restriction on compilable functions is then

Restriction 5.

In a compilable function, all function calls must be of typed or p-typed functions.

Summary

We can now see the overall picture. Defined functions will be of three sorts, typed, p-typed and untyped. All typed or p-typed functions can call only typed or p-typed functions,and to be compilable a function must be typed. We can therefore ignore untyped functions.

To be compilable, a typed function must have a unique type associated with every expression in its definition (and in particular, any expressions that can denote the eventual result of the function must have the type that was declared). To ensure this we must disallow certain basic operations and also make a syntactic check that the types resulting from the original parameter (and global) types can never conflict. In doing this we use the typing or p-typing of the functions used in the definition.

For a p-typed function (or a non-compiled typed function), we still have to check that the declarations are correct. This involves a type-consistency check that is a parameterized version of the one used in type-determination, and once again we use the typing or p-typing of the functions used in the definition.

Comments

1. It is difficult to estimate how much execution time would be saved by the compilation of defined functions in the way suggested. Each function call will, in general, result in several executions of the function. This is offset by the fact that the arguments for which

the function is actually executed will be simpler, and presumably the basic operations will be correspondingly faster. More importantly, since types will be determined, a great deal of 'overhead' can be eliminated. By compiling the function we expect a net time saving, but simulation studies should be performed to confirm this.

2. The modification to APL, itself, namely the function extension mechanism, seems to have many advantages. As it stands, it is an appealing extension to the language. It also opens up the possibility of further improvements, in particular, it appears possible to remove the distinction between basic operations and typed functions, so that typed functions could be used in reduction, for example. It would be worth having simulation studies of the function-extension feature just to see if the notation and conventions are well-designed.

Acknowledgement

I am indebted to D. M. Jackson, whose indignation at the fickleness of the Serendipity Principle initiated many discussions on function extension mechanisms and notation.