DECISIONS FOR "TYPE" IN APL*

W. E. Gull
M. A. Jenkins
Dept. of Computing and Information Science
Queen's University
Kingston, Canada

Abstract

The meaning of "type" in an APL extended to contain nested arrays is discussed. It is shown that "type" is closely related to the variety of empty arrays of the same shape and to the possible fill values needed in the "expand" and "take" functions. Choices for fill functions are systematically presented. They are classified according to the possibility of maintaining important identities involving level-manipulating functions in the case of empty arguments, to their effect on other design choices still to be made (the restriction to homogeneous arrays and the definition of the nature of basic data), and to their ability to express "type" in a natural way.

1. Introduction

The APL community has been grappling with the design of a recursive data structure extension to the language for several years. This paper is an attempt to summarize the choices that must be made in completing the design of such an extension. The present controversy is mainly concerned with three issues:

(i) Should one allow arrays whose items are of different "type", i.e. "heterogeneous" arrays, or should one restrict the extension to "homogeneous" arrays?

(ii) Should the items of an array always be arrays - which leads to a "floating" system, where the basic scalars contain themselves as items -, or should one have two sorts of objects, namely arrays and basic data - which leads to a "grounded" system?

(iii) Should an empty array be completely defined by its shape, or should there be more than one - possibly many - different empty arrays of the same shape?

* The work in this paper was supported in part by the National Research Council of Canada, Grant A7892.

The first two issues are discussed extensively in [2], where it is shown that the universe of grounded heterogeneous arrays (system-1-arrays) "contains" the universe of floating arrays (system-0-arrays) as a proper subset. A natural mapping is constructed from the grounded to the floating universe, and it is shown to be onto but not one-to-one. Arrays in the grounded universe that contain a basic scalar in a sequence of nested rank-0 arrays and that differ only by the number of such nestings coincide in the floating universe; this effect is known as the <u>telescoping scalar phenomenon</u>.

In [2] it is argued that a homogeneous subset of the grounded universe provides all the capability required of a data structure extension to APL and that this subcase has the interesting property of using the arrays of current APL as the building blocks in creating multi-level structures. A consensus on issues (i) and (ii) has not yet formed within the APL community; we leave their resolution to other forums. Here, we concentrate on (iii), an issue left unresolved in [2]. Our purpose is to explore the possible decisions on (iii) and indicate the interdependence of this issue with the decisions on issues (i) and (ii).

2. The concept of "type" in APL

The word "type" is used to mean many different things in the programming language literature. In Fortran and Algol declarations bind identifiers to locations or arrays of locations. "Type" is used in a declaration to indicate the contents that the location may hold. This may be viewed as providing information about the representation of data. In [1] Hoare gives a much more general concept of type, which Wirth has adopted in Pascal [9]. While a "simple type" in Pascal is essentially the same concept as Algol "type", Pascal allows the construction of new types from existing ones using the structuring mechanisms of powerset, record and fixed-sized arrays. Thus, "type" in the Hoare sense provides information not only about content but also about structure. In recent work (Liskov [8]) the concept of type has been further generalized to "abstract data type", in which a data structure and operations on it are bound into a module with access to the contents of the data structure limited to that provided by the operations defined as part of the "data type". Simula classes are an example of such abstract data types.

All of the languages discussed above are "typed" languages, i.e. identifiers are bound to objects of specific "type" by declarations. "Type" information is used at compile time to verify that the objects are properly typed and to generate code that is specific to the "type" of the objects being manipulated. Hence in a typed language little information is required at run time. APL, on the other hand, is a "typeless" language, i.e. there are no fixed bindings between identifiers and objects they can represent. That is not to say that APL objects do not have "type" attributes; each APL data object is either an array of numbers or an array of characters. Moreover, the information concerning the shape of the array and the "type" of its items must be carried with the object at run time.

"Type" manifests itself in APL mainly in the definition of the primitive functions that have need of a fill item: "expand", "take", and, in some APL implementations, "reshape". It also plays an implicit role in the domain constraints of many of the primitive functions. The "type" of a non-empty array may be viewed either as describing attributes of the entire array or as describing attributes of some or all of the items of the array. Either view is compatible with the non-empty arrays of current APL, since all arrays are homogeneous and only one-level arrays exist. In [4] the two views of "type" are examined in the context of nested arrays, and it is shown that, if "type" is a property of items, then there can be only a single empty vector, but if it is a property of the entire array, then there can be an empty vector corresponding to each "type". (The same rule applies for empty arrays of other shapes.) In [2] only a single empty vector is proposed, whereas in APL there are two empty vectors, ⍳0 (numeric) and '' (character); hence, the proposal of [2] is not a pure extension of current APL. In the rest of the paper we are concerned with finding an extension that preserves the distinction between ⍳0 and '', on the assumption that only a pure extension of APL will be acceptable to the APL community. Thus, we explore the second choice for issue (iii), namely that there will be more than one empty array of each shape.

How many should there be? We are guided in answering this question by the following assumption, which we take as being intuitively desirable: For every distinct empty vector there exists a distinct fill item.

We postulate the existence of two functions defined for all arrays:

vacate A, denoted $\backslash A$, whose value is the empty vector with the same fill as A,

fill A, denoted αA, whose value is the 0-rank array whose item is used to fill from A.

Then the above assumption may be stated more precisely by the identities

(1) $\backslash\backslash A \leftrightarrow \backslash A$
(2) $\alpha\backslash A \leftrightarrow \alpha A$
(3) $\backslash\alpha A \leftrightarrow \backslash A$

which state

(1) vacating is idempotent,

(2) the fill of an array is the same as fill of its vacate,

(3) the vacate of an array is the same as the vacate of its fill.

From these identities we have

$$\begin{aligned} \alpha\alpha A &\leftrightarrow \alpha\backslash\alpha\backslash A && \text{by (2)} \\ &\leftrightarrow \alpha\backslash\backslash A && \text{by (3)} \\ &\leftrightarrow \alpha\backslash A && \text{by (1)} \\ &\leftrightarrow \alpha A && \text{by (2),} \end{aligned}$$

which shows that fill is also idempotent.

The fill function provides a manifestation of the underlying attributes of the array that are preserved into emptiness. Thus, if one views this manifestation as one of "type", then the meaning of "type" in an extended APL will be determined by the choice of the set of values that fill may take on.

3. The choices for fill in extended APL

The universe of nested arrays given in [2] has to be redefined in order to provide the larger set of empty arrays that is required by the corresponding values of the chosen fill function. First we purge from the universe all arrays that are empty or contain empty arrays at some nesting level. Then we add new empty arrays corresponding to the fill values of the non-empty arrays, building all arrays that contain the new arrays as items, extending the domain of the fill function to them, and repeating that process as long as new fill values occur. The resulting extended universe can be described as the smallest set $U(\alpha)$ that

- contains each non-empty array of the original universe,
- is closed under non-empty array construction by the structure and level increasing primitive functions of extended APL,
- contains a distinct empty array for each possible value of the fill function and each shape with at least one zero.

Characterization of Homogeneity and Heterogeneity

Let us begin by trying to use fill to characterize "homogeneity". For a given fill function α we define an array to be α-uniform if

$$\alpha A \leftrightarrow \alpha \text{ f } A \ ,$$

for every selection function f.

Let α_0 denote the fill function of current APL. Then all arrays of current APL are α_0-uniform.

Let us divide the set of all non-empty arrays with non-empty subarrays into nine disjoint classes according to Figure 1.

basic data are / items are	only numbers	only characters	numbers and characters
all basic data	R	S	T
all not basic data	U	V	W
both, basic data and not basic data	X	Y	Z

Figure 1. A Classification of non-empty arrays

In a grounded system the second row "items are all not basic data" could synonymously be labelled "items are all arrays", while in a floating system this description qualifies every array. Thus, in a floating system the vertical distinction is of much lesser importance than in a grounded system.

The non-empty arrays of the Gull-Jenkins proposal ([2], section 4) include the arrays R∪S∪U∪V∪W of figure 1. The fill function implicit in [2] is defined as follows:

$$\alpha_1(B)\ A = \begin{cases} 0 & \text{if } A \in \text{R} \\ \text{' '} & \text{if } A \in \text{S} \\ B & \text{if } A \in \text{U}\cup\text{V}\cup\text{W} \ , \end{cases}$$

where B is $<\theta$, the 0-rank array that contains the (unique) empty vector θ of the proposal. $\alpha_1(B)$ is undefined on empty arrays.

The non-empty arrays of section 4 in [2] are $\alpha_1(<\theta)$-uniform and are called homogeneous. The arrays in T∪X∪Y∪Z are not $\alpha_1(<\theta)$-uniform and are called heterogeneous.

All arrays in R∪S∪U∪V∪W are $\alpha_1(B)$-uniform for any choice of a non-basic 0-rank array B.

The development of section 2 above suggests that we should postulate the existence of three empty arrays of each shape. While this choice is intuitively appealing if the universe consists solely of homogeneous arrays, it does not generalize easily to a universe that includes non-homogeneous arrays.

The fills for the homogeneous grounded proposal classify R, S, and U∪V∪W of figure 1, respectively. If we wish to maintain compatibility with current APL and have a homogeneous proposal a "natural" subcase of a heterogeneous one, we must add either two new fills that classify T and X∪Y∪Z, respectively, or add just one fill characterizing both kinds of heterogeneity at once. Thus, either five or four fills must be provided for such a proposal. Neither choice is particularly appealing. They both involve adding more "artificial" empty arrays to the universe, a point we discuss below.

In a floating system the classification of arrays according to figure 1 does not work well. Homogeneity is not easily defined or preserved in a floating system, and the purpose of our classes is to make the distinction between homogeneous and heterogeneous arrays clear. The telescoping scalar phenomenon implies that the number of levels in an array is not a good criterion for classifying arrays, and hence any attempt to impose a definition of type based on the nine classes above is bound to be artificial.

Other Choices

Let us explore some of the remaining possibilities for defining fill. The fill function α_0 of present APL is such that all arrays are α_0-uniform. It may be asked, whether a fill function can be found with respect to which all arrays of a heterogeneous universe would be uniform, too. In order to achieve this goal one has to sacrifice a very fundamental principle, namely that two arrays are equal if they have the same shape and the same items at all corresponding positions (principle of extensionality for non-empty arrays, More, [6]). Giving up extensionality for non-empty arrays is such a fundamental deviation from present APL that we will not investigate such constructions any further.

Drawing upon the use of "type" in other languages, we see that it makes sense either to have the fill item typify solely the content of a non-empty array or, which is the more modern approach, to typify both the content and the structure of the array. Moreover, the typification could be either local or global, i.e. the fill could typify a single item of the array, or it could typify all the items. We will first concentrate on local fill-definitions and present two extremes with regard to the structure information provided.

In More's array theory (see [7]) the prototype concept is used to define a fill which typifies both the structure and content of the first item. Thus, it is a local structured fill. We can adapt the prototype concept to APL as follows: Let τ be the function that maps a number to zero and a character to blank. (If other basic data types are added, then typical items must be designated.) Extend τ to all arrays by applying τ to the basic data held in an array. Thus, τA is an array with the same structure as A but with the basic data replaced by the corresponding typical items. Then we define for a non-empty array A

$$\alpha_2 A \leftrightarrow \tau(,A)[1] \quad .$$

Note that this implies

$$\tau A \leftrightarrow {>}\alpha_2{<}A \quad .$$

This definition of a local structured fill α_2 is independent of the decision on issue (ii), and α_2-uniformity implies homogeneity (but not vice-versa).

α_2 produces a large collection of empty arrays. At the other end of the spectrum we can define a fill to simply characterize the content of arrays. We define a local unstructured fill α_3 recursively to be 0 if its first item is a number, to be ' ' if it is a character, and to be the fill of its first item otherwise.

There are many choices for a local fill function that lie between the unstructured α_3 and the fully structured α_2; however, none seems a more suitable choice since the amount of structure information preserved is arbitrarily determined.

We now turn to global fill functions. The function τ above typifies the content and structure of an array globally; however, it cannot be used as a fill function since it is not an extension of α_0. Both extensions of $\alpha_1(B)$ described above are global fill functions. The one with five values also provides one-level structure information, since it distinguishes basic arrays from others. The one with four values does not have this property, because arrays in T and in $X \cup Y \cup Z$ have the same fill.

A definition for a global unstructured fill can be made by adding an artificial 0-rank array to the universe. Then α_4 of a non-empty array is defined recursively to be 0 if all its items are numbers or arrays with fill 0, to be ' ' if all its items are characters or are arrays with fill ' ', and to be ⍰ otherwise. Note that the idempotency restriction of fill requires that ⍰ must be an "alien" array, i.e. one that cannot be constructed from the universe of recursive arrays with numbers and characters as basic data. The additional one or two fill arrays required for the extensions of α_1 must also be "alien" in this sense. The properties of such alien arrays are not completely determined by the fill function. For example, the item of ⍰ may be considered a basic datum. Then in a floating system ⍰ $\leftrightarrow$ <⍰, whereas in a grounded system ⎕ $\not\leftrightarrow$ $<X$ for any array X. Another choice in a grounded system is to define ⍰ $\leftrightarrow$ <\⍰.

Haegi [3] defines a universe similar to that of the homogeneous proposal of [2], where the (implicit) fill function is an α_1(⌼), introducing an alien array ⌼ that has the property

$$⌼ \leftrightarrow <\backslash ⌼ \; .$$

⌼ behaves like the NIL of Lisp.

In this section we have investigated only a few of the many possible fill-definitions, but we feel that the omitted ones are even less appealing than some of the ones we have presented.

4. Evaluation of the Choices

On what basis should we choose among the fill functions defined in the previous section? One way is to examine important identities that hold for non-empty arrays and see if they extend to the empty arrays associated with each choice. A second approach is to evaluate how the choice of each fill function affects or is affected by the decisions made on issues (i) and (ii). Finally, we may compare the fill functions with respect to their ability of expressing "type" in a natural way.

Identities

We define a monadic function f to be scalar if indexing commutes with f, i.e.

$$(\text{f } A)[I1;\ldots;IN] \leftrightarrow \text{f } A[I1;\ldots;IN]$$

for all suitable indices $I1;\ldots;IN$.

If f is a scalar monadic function, then for any array A and any fill function α the identity

$$\mathrm{f}\ K\uparrow\alpha A \leftrightarrow K\uparrow\mathrm{f}\alpha A$$

holds for any positive integer K. Should this identity hold for $K \leftrightarrow 0$? Assuming that $K\uparrow A \leftrightarrow \backslash A$ when $K \leftrightarrow 0$ the identity reduces to

$$\mathrm{f}\backslash\alpha A \leftrightarrow \backslash\mathrm{f}\alpha A \quad ,$$

and using identities (2) and (3) of section 2 we have

$$\mathrm{f}\backslash A \leftrightarrow \backslash\mathrm{f}\alpha\backslash A \quad .$$

We can express this in words as

<u>Postulate 1</u>: The result of the application of a scalar function f to any empty array is the same as the empty array obtained by vacating the result of applying f to the fill of the empty array.

If we accept this principle, it leads to an interesting identity for the scalar function "raise" $\underline{\uparrow}$ (defined in [2]), which is the result of applying the "by-scalar" operator $\ddot{|}$ of [5] to the seal function $<$, i.e.

$$\underline{\uparrow}A \leftrightarrow \ddot{|}{<}A \quad .$$

"Raise" is the scalar function that coincides with the "seal" function on 0-rank arrays, e.g.

$$\underline{\uparrow}1\ 2\ 3 \leftrightarrow ({<}1),({<}2),{<}3 \quad .$$

Postulate 1 then implies that

$$\begin{aligned}\underline{\uparrow}\backslash A &\leftrightarrow \backslash\underline{\uparrow}\alpha\backslash A\\ &\leftrightarrow \backslash{<}\alpha\backslash A \quad ,\end{aligned}$$

since the result of α is always a 0-rank array, and hence

$$\underline{\uparrow}\iota 0 \leftrightarrow \backslash{<}0 \quad ,$$

and

$$\underline{\uparrow}\text{''} \leftrightarrow \backslash{<}\text{' '} \quad .$$

Thus, postulate 1 implies that

$$\underline{\uparrow}\iota 0 \not\leftrightarrow \underline{\uparrow}\text{''}$$

unless

$$\backslash{<}0 \leftrightarrow \backslash{<}\text{' '} \quad .$$

In a grounded system seal and raise are the fundamental level increasing functions, and unseal $>$ and lower $\underline{\downarrow}$ are the corresponding level decreasing functions. The identity

$$>{<}A \leftrightarrow A$$

holds for all A, and

$$\underline{\downarrow}\underline{\uparrow}A \leftrightarrow A$$

holds for all non-empty A.

Should the latter identity also hold for empty arrays? If so, then $\underline{\uparrow}$ must be one-to-one and hence

$$\underline{\uparrow}\iota 0 \not\leftrightarrow \underline{\uparrow}\text{''} \quad ,$$

which implies

$$\backslash{<}0 \not\leftrightarrow \backslash{<}\text{' '}$$

and hence

$$\alpha{<}0 \not\leftrightarrow \alpha{<}\text{' '} \quad .$$

Thus, we might consider

<u>Postulate 2</u>: $\underline{\downarrow}\underline{\uparrow}A \leftrightarrow A$ for every array A.

Figure 2 gives the values of the fill of <0 and $<$' ' for the four fill functions we have defined.

	$\alpha{<}0$	$\alpha{<}$' '
$\alpha_1(B)$	B	B
α_2	<0	$<$' '
α_3	0	' '
α_4	0	' '

Figure 2 Table of fill values

The table illustrates that postulate 2 fails for $\alpha_1(B)$ (for any B) but holds for α_2, α_3 and α_4.

The raise and lower functions play a fundamental role in the grounded system, being used to perform data structure transformations that add or remove a level to the structure of the array but do not alter the contents. The functions can be generalized in a number of ways. In [5] slice-raise, denoted $I\underline{\uparrow}A$, is defined to yield an array whose items are slices of A. I is used to denote the axes of A along which slices are to be taken. Thus, each item of $I\underline{\uparrow}A$ has shape $(\rho A)[I]$ and the shape of $I\underline{\uparrow}A$ is determined by the remaining axes. For example, if A is $2\ 3\ 4\rho\iota 24$, then $1\ 3\underline{\uparrow}A$ is a vector of 3 items, each with shape $2\ 4$. The slice-lower function is defined by

$I\underline{\downarrow}A \leftrightarrow B$
if and only if there is a unique B such that $A \leftrightarrow I\underline{\uparrow}B$.

This implies $I\underline{\downarrow}I\underline{\uparrow}A \leftrightarrow A$ for every non-empty array *A*. If we extend the latter identity to empty arrays, we have the following generalization of postulate 2:

Postulate 3: *I*↑̲*I*↓̲*A* ↔ *A* for every array *A* and *I* satisfying ∧/*I*∊⍳⍴⍴*A*.

The importance of postulate 3 depends on how one views the use of multi-level data structures in extended APL. If transformations of the form accomplished by slice-raise are used as a programming device to ease the expression of algorithms applied to slices of an array, then it may be important that such transformations are one-to-one. [5] shows that the present axis operator can be defined in terms of such transformations and this may suggest that postulate 3 is more fundamental than one would at first suspect. Of the fill functions we have defined only α_2 preserves enough information to permit slice-raise to be defined to be one-to-one.

The development of the identities in postulates 2 and 3 has been done in the grounded system. However, similar level-manipulating functions exist in a floating system, and the same considerations apply.

Interdependencies

The second criterion for evaluation of the choices for a fill function is the relationship of each choice for issues (i) and (ii). Let us begin with (i). If the arrays of APL form a heterogeneous universe U̲, then we can ask whether there exists a homogeneous subset H̲ that has a natural embedding in U̲. (This is primarily of interest in a grounded system, since it is difficult to preserve homogeneity in a floating system.) In particular we would expect for a vector *V* ∊ H̲

(1+⍴*V*)↑*V* ∊ H̲ ,

i.e. overtaking preserves homogeneity. The extensions to α_1 can be defined to have this property, but α_2 has it too. Neither α_3 nor α_4 have this property, since with α_3

3↑(<1 2),<'*ABC*' ↔ (<1 2),(<1 2 3),0
(which is in set X of Figure 1).

and with α_4

3↑(<1 2),<'*ABC*' ↔ (<1 2),(<'*ABC*'),⍰ .

This implies that if the arrays of APL are chosen to be homogeneous (in the sense of being α_1-uniform), then only α_2 may be considered as an alternative choice for the fill function, since overtake would be undefined for α_3 and α_4.

The evaluation of the fill functions relative to decision (ii) is not independent of decision (i), as the above discussion indicates. We can note nevertheless that α_2, α_3 and α_4 are suitable for either a floating or grounded system if homogeneity is not an issue.

Typifying global attributes

A third way to measure the usefulness of a fill function is to determine how well it expresses "type" in the sense of a global attribute describing both, structure and content of an array. α_1 and its five-valued extension either give content information (for arrays in R∪S∪T) or give structure information (for the other arrays) but not both. α_3 and α_4 lose any structure information. However, the remaining fill function, α_2, although defined locally, can be used to yield global content and structure information. We noted in section 3 that

τ*A* ↔ >α_2<*A* .

More [7] calls τ the type function, and we see that the choice of α_2 as the fill function allows us to construct τ.

5. Conclusions

We have seen the following:

1. The distinction between homogeneous and heterogeneous arrays is interesting only in the context of a grounded system. In this context α_2 and the extensions of α_1(*B*) to U̲ introduce "artificial" arrays; hence α_2 seems preferable if the choice for (i) is heterogeneous. However, if the choice for (i) is homogeneous, the decision between an α_1(*B*) and α_2 depends on the importance one attaches to uniformity and simplicity relative to preserving postulates 1 to 3. The NIL-like properties of ⌻ make α_1(⌻) an interesting choice in this restricted form of an APL extension.

2. The resolution of (iii) using α_2 permits the extension of important identities to empty arrays and is independent of the choices for issues (i) and (ii). Neither α_3, α_4, nor the extensions of α_1 can preserve postulate 3, although α_3 and α_4 preserve postulate 2.

3. α_3 and α_4 can be used if structural properties are deemed to be less important. α_3 is preferred, since it introduces no "artificial" arrays.

Figure 3 summarizes these conclusions.

Issue (ii) / Issue (i)	FLOATING	GROUNDED
HOMOGENEOUS		1st α_1(⌼) 2nd α_2
HETEROGENEOUS	1st α_2 2nd α_3 3rd α_4	1st α_2 2nd α_1(*B*) extended 3rd α_3 4th α_4

Figure 3. Preferences for Fill Function

In conclusion we note that α_2, the fill function that corresponds to More's prototype function in [7], appears the most suitable candidate for defining the empty arrays of extended APL, unless the extension is restricted to homogeneous grounded arrays.

6. References

[1] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R. Structured Programming. Academic Press, London and New York (1972).

[2] Gull, W.E. and Jenkins, M.A. Recursive data structures in APL. To appear in Comm. A.C.M. (1978). Also available as Technical Report 77-58, Computing and Information Science, Queen's University, Kingston, Canada.

[3] Haegi, H.R. The extension of APL to treelike data structures. APL Quote Quad 7, 2, 8-18 (1976).

[4] Jenkins, M.A. and Michel J. On types in recursive data structures: A study from the APL literature. Proceedings of the Third Jerusalem Conference on Information Technology, August 1978. Also available as Technical Report 77-59, Computing and Information Science, Queen's University, Kingston, Canada (1977).

[5] Jenkins M.A. and Michel J. Operators in an APL including nested arrays. Submitted to APL Quote Quad. Available as Technical Report 78-60, Computing and Information Science, Queen's University, Kingston, Canada (1978).

[6] More, T. Jr. Axioms and theorems for a theory of arrays. IBM Journal of Research and Development 17, 135-175 (1973).

[7] More, T. Jr. Types and prototypes in a theory of arrays. Technical Report G320-2112, IBM Cambridge Scientific Center (May 1976).

[8] Liskov, B. and Zilles, S. Programming with abstract data types. SIGPLAN Notices 9, 4, 50-59 (April 1974).

[9] Wirth, N. The programming language Pascal. Acta Informatica, 1, 35-63 (1971).