An Array-Oriented (APL) Wish List
Ideas I Think May be Useful

Jim Lucas
jel@danbbs.dk

Abstract

The fundamental data structure in APL is the array. Arrays are, in most APL dialects, the only data structure, from scalars as arrays with no dimensions to the complex structures of nested arrays. But there are other ways in which array concepts could be applied, yet so far they have not. These are the top items on my "Array-Oriented Wish List".

Less fundamental—but far easier to implement—should be certain operators and functions I propose, which I think would enhance the power of existing array operations. Some of these already exist in one or more dialects of APL, and I think all would benefit if they were universally adopted.

In this suite of proposals, some are general, others more specific, and some could interact with each other in positive synergy, but none of them requires any of the others to be useful. In some cases, I suggest potential variants, each reasonable in its own right.

The purpose of this paper is to present the concepts. In most cases, I don't attempt to go into details of implementation. That would require a much more extensive treatment of each concept. I hope that others—especially implementers—might take up the challenge.

Arrays in Current APL

From its inception, the fundamental data structure in APL has been the array. Scalars were not treated as some sort of "more fundamental" data object ("atoms"), which were assembled to form arrays as structure of secondary consequence. Instead, scalars were considered to be simply a limiting case of arrays in general. Any special characteristics they possessed derived from the constraints which separated them from the larger, unrestricted set of arrays, just as zero, one, and two are integers which, defined by one constraint, are discovered to have additional "special" properties.[1]

APL01, 06/01, New Haven, CT USA

Data arrays

Simple arrays

For a long time the only arrays in APL were so-called "simple" arrays, i.e., arrays consisting of a single data type. Still, this was not as restrictive as in other programming languages, since APL only distinguished between "character" and "numeric" arrays, but not between logical and various numeric "types"—Boolean, integer, double precision, etc.—at least not at the programming level. APL arrays had no theoretical restrictions on number of dimensions or size, and these could be changed at will. Even their type could be changed, but only for the array as a whole; character and numeric elements could not be mixed in the same array.[2]

Heterogeneous arrays

An obvious next step was to allow arrays of mixed type. However, this presented serious design problems, including efficient implementation, a major concern of APL implementers. In fact, implementation of heterogeneous arrays, as they are called, waited on implementation of the next logical extension: "nested" arrays. Heterogeneous arrays have been implemented as if they were nested arrays, though they appear different to the user or programmer.

Nested arrays

Nested arrays are arrays in which individual elements can themselves be arrays other than simple scalars. Such arrays are considered to be "enclosed"; they are scalars, but with internal structure. In one sense, nested arrays allow one to

[1] In fact, deciding which is the defining property and which are the derived ones is somewhat arbitrary.
[2] Empty arrays of one type *could* be catenated to arrays of the other type, but the result — even if empty — was of one type or the other, not mixed.

avoid the question of heterogeneous arrays, since each enclosed element of a nested array can be either character, numeric, or nested (a new type). Still, true heterogeneous arrays have shown themselves to be preferable in various situations.

Nested arrays also allow one to avoid direct implementation of a second potentially problematic extension, non-rectangular arrays, or arrays in which different subarrays could be of different size. E.g., one could not have a matrix with rows of different lengths, but one could put the same data into a nested structure as a vector of vectors, with each enclosed vector of a different length.

Two systems

In fact, two distinct sytems of nested arrays[1] have been developed. In the one (common to Sharp APL—including SAX—and J), enclosing a simple scalar turns it into something different, while in the other (APL2, and most others, which are following APL2's lead) a simple scalar is identical to its enclose.

In the first, allowing heterogeneous arrays would be an independent question. However, the latter system requires heterogeneous arrays, since a "nested" array consisting of an enclosed character scalar and an enclosed numeric scalar *is* a heterogeneous array.

Rank vs. Depth

Rank—the number of dimensions—and Depth—the number of levels of nesting—of arrays are complementary concepts. A nested array where all elements at each level are identical in shape can be reversibly converted into a non-nested array, in which the levels of depth are transformed into additional dimensions. The same amount of information is contained in both representations, though the quality of the information representation differs.

In theory, non-uniform nested arrays could be converted to "ragged" arrays without depth, i.e., arrays composed of subarrays of different sizes. So far, no APL has implemented ragged arrays.

[1] In the one system—that in which simple scalars are *not* identical to their enclosures,—nested arrays are now referred to as "boxed", to emphasize that they are in some significant way different from "enclosed" arrays. In fact, they are different developments of the same fundamental concept of enclosure, each internally self-consistent. Other, independent differences—including prototypes and strand notation—won't be dealt with here. In this paper I will use the term "nested" to refer to both "boxed" and "enclosed" arrays.

"Generalizing" arrays

Some people refer to nested arrays as "generalized" arrays, but nesting is only one way in which arrays can be generalized. E.g., there have been various proposals over the years for implementing "function arrays", so that different functions could be applied to different elements or element-pairs in their array arguments.

Event Arrays and Distinguished Values

In APL, an expression like `÷A` could result in a *DOMAIN ERROR* if some elements of *A* are zero. Yet taken individually, only some of the elements of *A* would generate errors, and frequently one would only want to identify those elements, but still get the results from the others. This is the premise of a paper I presented at APL85, which I updated as an article [5] in the FinnAPL journal last year.

Events as values

The basic proposal is that events—errors, interrupts, and possibly other events—should constitute a new datatype. With heterogeneous arrays, event values could be elements on equal footing with characters, numbers, and enclosed arrays. Different types of events would have different "event values". Of course, it would be necessary to define what results the various primitive functions would return if one or both arguments are events, and if the two arguments of a dyadic function are two different events.

The 1985 paper undertakes a detailed consideration of these issues, and the 2000 article takes an updated look at them. Some details could reasonably be decided in more than one way, but one important consideration is that it should be possible to choose at any time whether event values are passed as results or whether execution should be interrupted, as it is currently. This choice of action should, in fact, be specifiable separately for different types of events and even in different functions. I think it is reasonable to do this by extending the `⎕trap` facility that is currently used by both Sharp and Dyalog APLs.

More "data type" extensions

But if one new data type can be added, why not others? J has introduced extended precision and rational datatypes, but my thoughts are in different directions.

Distinguished types with "ordinary" values

Some applications go to a great deal of trouble to keep track of qualitative differences in their data, e.g., whether a bond price came from an actual trade, an unmet bid or offer, a value computed by a model, or a trader's mental estimate. But why not give those values a second attribute?

In addition to their value, each class of value could have a "type". The types could be ordered, with a precedence such that combining two different types would always give the result value the "lower" type. The meanings given to the different types should not be built into the interpreter, though, but assignable by the programmer. E.g., instead of the above interpretation, the types could indicate relative degrees of certainty in measurement:: "precise", "slightly uncertain", "very unsure", "value unknown", "value suspect", etc.

Enumerated classes

Another possibility is "types" consisting of finite sets of values. This is particularly a candidate for allowing users/programmers to define their own "classes". E.g., NaN (not a number) might seem a good way to indicate missing data in a database, but with enumerated types there could be different values to indicate *why* data is missing from a database: "not applicable", "not available", "pending input", "pending validation", etc.

Why not just simulate them?

Both "distinguished types" and "enumerated classes" can readily be simulated in APL (the former, e.g., by pairs of values); in fact, I have seen both. But they require a great deal of additional code, since every primitive operation (plus, rotate, shape,...) has to be replaced by a complex function that handles the "calculus" of such values in a reasonable and consistent way. How much simpler—and more efficient—to have a general facility with a consistent calculus built into the interpreter.

I think complex numbers provide a case in point. Many are the people who wrote suites of functions to deal with complex numbers as ordered pairs ($N \times 2$ arrays). Is there a single one of them who has encountered the primitive implementation of complex numbers in APL2, Sharp APL (including SAX) or J, who would prefer to use—and extend—his old function suite? I doubt it. The advantages of the primitive implementations are too great.

Operators

In proposing new operators, I'll start with a simple pair, what I'll call "Pad" and "Trim".

Pad & Trim

Every place I've ever programmed has had a function to catenate two arrays and guarantee that the result was two dimensional, with the one argument above the other and the "smaller" one padded with fill elements to match the width of the "larger", and treating both vectors and scalars as one-row matrices[1].

I've seen more complex utilities to handle a similar operation on arrays of arbitrary rank. Less frequent have been the "opposite" utilities, which trimmed the larger array to match the shape of the smaller one (on all dimensions but one). I've also seen code to perform similar "justification" of two arguments before addition, multiplication, etc. I've long thought that a monadic operator—actually a complementary pair—would be a more sensible way to handle such enforcement of conformability in a general way. In A (predecessor to A+), I even wrote my own operators to do just that.

Is it worth it?

Well, I wouldn't be proposing such operators if I didn't think so. I've already noted that while the most common use of such an operator would seem to be with catenation and to pad the smaller argument to be conformable with the larger, other potential uses are not unknown. Another possibility would be with the monadic function known as "mix" (↑) in Dyalog APL and as "open" (>) in J. Currently Dyalog's "mix" automatically pads lengths but not ranks, while J's "open" pads both. But what if the ragged lengths (or ranks) were a mistake? I think it would be better if the default were to signal length and rank errors, but with the possibility of overriding that behavior with the Pad (or Trim) operator. Will the repertoire of uses expand if Pad and Trim are implemented as operators? Perhaps not, but is lack of ability to predict extended generalization really a good argument against implementing something useful?

That's a rhetorical question, not because I think it has only one answer, but because I'm sure different people would answer it differently. Sharp

[1] Depending on the particular implementation of this utility, arguments of rank greater than two were often either reshaped to two-dimensional or truncated, so that only one plane was included in the result.

APL and J have implemented determinant as monadic versions of inner product with - and × (* in J). While any other functions can be used in the place of - and ×, I'm only aware of one other pair (+ and ×) that are considered to have a meaningful interpretation, and I'm not even sure how that is used. Is it then wrong that they implemented this facility? You may disagree, but in my opinion, absolutely not!

Perhaps a more relevant example would be ⍷, which is essentially a souped-up string search function. The fact is that string search utilities were so universal that it didn't make sense *not* to have it as a primitive, for both increased efficiency and simplification of code. I think the same is true of Pad, and for the sake of symmetry and generality it makes sense to include its complement, Trim, as well.

Not as simple as it sounds

Another reason for wanting these operators as primitives is that implementation is anything but simple. Exactly how the shapes should be adjusted depends on the function. With scalar functions, it's easy. Pad, e.g., should just insure that both arguments are the same rank as the greater of the two and that the length of each axis is also the greater of the two. Well, it's not quite that easy, since one might argue that scalar (or singleton) extension should still hold, though one might equally well argue that it shouldn't. And what should one do if the ranks of the two arguments differ by more than one? Should the leading or trailing axes be matched, or should axes of the same length and ordering be considered equivalent? Hard decisions.

With non-scalar functions, things get even more difficult. With catenation, examining existing utilities should suggest which conventions should be adopted, *if* there seems to be a common standard. Otherwise, decisions would have to be made, as has been done in the past with such cases as `0÷0`. Similar, but not necessarily identical, decisions would have to be made in connection with the various other non-scalar functions. Then there's the question of what to do about derived and user-defined functions.

The Rank operator

While I happen to think that the *rank* operator, as implemented in Sharp APL and J, is useful in its own right, it might also provide a simple solution to some of the above difficulties. In these dialects of APL, the concept of function rank defines in a consistent way how the application of functions, even derived functions, extends to arrays of any rank. This same principle should hold with functions derived by these new operators. And while it is not in general possible to *derive* the rank of a user-defined function, application of the rank operator to such a function will define its behavior.

I won't go into detail here about the use and function of the rank operator, since they are well documented elsewhere. But I will say that I think its usefulness—independent of Pad and Trim—is such that it should be an element of every APL implementation, even those that differ from Sharp APL and J in other fundamental respects. And while it is easy enough to simulate using nested arrays as an intermediate, that approach has serious inefficiencies as compared to the rank approach.

Fill

Let's assume we've implemented Pad and Trim, and that I want to do something like `A +Pad B`. With any new elements as zero, the corresponding elements in the result will simply be the elements that existed wherever there was an unmatched element in the argument. That seems fine enough, but what if I want `A ×Pad B`? In that case, all the corresponding result elements will be zero. In order to have the unmatched values passed unaltered to the result, the fill elements should be 1, not 0.

To me, this is nothing new. Many is the time I've wanted to be able to specify a fill element other than the default. Instead, I have to write contorted code to substitute some other value for the 0's (usually, though sometimes for blanks or enclosed elements) before I can proceed with my computations. An obvious answer is to have an operator which allows one to specify the fill element(s). I'll call that operator Fill.

In its simplest form, the Fill operator would take a scalar value for its second operand, e.g., 1 instead of 0 for filling numeric arrays. But with nested arrays, one should really be able to specify separate fill elements for each element of the nested structure, with pervasive application where the structure of the operand is not as deep as that of the argument array(s).

Another possibility would be to allow specification of a vector of different fill elements for filling the new elements in a vector with expand, overtake, or Pad. But if we go that far, why not allow specification of different fill elements even when argument(s) and result are not vectors? The difficulty is, of course, that the

region to be filled will probably not be rectangular. I propose that in this case (and perhaps even in the rectangular case), the fill operand should simply be a vector, with the only conformability criterion being that the total number of elements should be the same as the number of new elements created.

Rest

However, there could be some difficulty in determining the number of fill elements needed by the Fill operator. Well, how about another operator? I'll call it Rest. Used in conjunction with a selection specification (e.g., take), the result would be the complement of the selection. If that complement is rectangular, it should retain its shape (since one can always explicitly ravel it), but otherwise it will be a vector, with the elements in the same order as they were in the original argument. E.g.,

```
      1 3↑Rest 3 3⍴⍳9
4 5 6
7 8 9

      2 2↑Rest 3 3⍴⍳9
3 6 7 8 9
```

One could thus construct an array and use Rest to extract from it precisely the elements which would be needed by Fill. The simpler, scalar specification of non-default fill elements could then be considered an example of scalar extension.

Control Structures

It's generally important in APL to avoid loops where possible, and a standard way of doing this is to substitute an "if" construct with a "where" construct. E.g., to add one to those elements of an array *A* which are greater than zero,

```
A←A+A>0
```

This actually performs addition on every element of A, but by adding zero to some elements, the result is the same as if the addition were performed only *where* the condition is true.

Now that some APLs have implemented control structures, there is some tendency to use an :If structure in a :For loop to accomplish the same purpose.[1] Because of the repeated interpretation of the code in the :For loop, this process is much less efficient than the above "APL-style" approach[2].

The problem with control structures is that they simply aren't array oriented. While an expression like the above could be contained within an :If clause or a :For loop, the decision process is all or nothing. The only way to apply an :If condition separately to each element of an array is to introduce a looping structure, which is inherently inefficient. Instead, I propose a new operator.

Where

This operator would apply its function operand eachwise only on those cells of the derived function's argument(s) corresponding to 1's in the Boolean operand, which must be conformable with the argument(s). The elements corresponding to zeros would be passed unchanged to the result. Thus, the above APL expression could be rendered as

```
A←1∘+ Where (A>0) A  ⍝ the ∘ is composition, in Dyalog APL
```

In this example I've deliberately used composition to turn the 1+ into a monadic function and avoid the problem of deciding *which* argument should have its elements passed to the result in the dyadic case. Obviously, either the left or right argument will have to be selected as the one that always gets used. I suggest that it should be the right argument, since that's the only possibility in the monadic case.

More complex examples

Plus is much too simple a function to demonstrate the real usefulness of Where. It could also be used with derived functions, and in combination with the Rank operator, it should be possible to use it reasonably in conjunction with non-scalar functions and even user-defined functions.

A question of efficiency

With primitive and even derived functions, the interpreter should be able to skip processing of those elements which are not selected by the operand. But what about user-defined functions? Surely, the operator can't be expected to trace complex internal logic and execute only those parts that are relevant to the selected elements. No, but it can execute the function with the full array argument(s), then replace those elements of

[1] This seems to be particularly prevalent among those for whom APL was not their first programming language.

[2] [Editor's note: Some implementations do not re-interpret loop bodies in such cases]

the result not selected by the operand with the original argument values.

This replaces the inefficiency of re-interpreting the function for each element of the argument(s) with the lesser inefficiency of generating "unnecessary" result values. Of course, one must beware of functions with side effects. The each-selected-element approach could still be forced by using the Rank operator.

Function arrays and Which

If one had function arrays, then one could use a vector of functions instead of just a single function as an operand to Where. The array operand could then be composed of indices into that vector, indicating which functions should be applied to which elements of the argument(s). Extended in this way, the operator should probably be called Which, rather than Where. I suggest that origin-1 indexing be implied, with a zero operand value indicating that the argument should be passed unchanged, as with the Boolean condition for Where.

"Undefined" as a "Value"

The rationale

It is useful and even important to have a means of specifying nothing, i.e., the lack of *something*. Zero measures the lack of quantity. Empty arrays are different; they represent objects that have form (or structure), even though they have no content. But so far in APL we have no way of representing the lack of a value or an object, except by the lack of representation.

We can use `⎕NC` on a specific name to determine whether or not it has a value. However, there are circumstances in which we not only want to know if a name is defined, but we want to use that information to control whether we supply a value in another context.

In my experience, the most common such use is where a subfunction is called monadically or dyadically depending on whether the function calling it was itself called monadically or dyadically. Wouldn't it be simpler if instead of reporting a `VALUE ERROR`, we could just call the subfunction dyadically at all times, but have it interpreted monadically if the name of its left argument is undefined?

There are, however, circumstances under which we would want that `VALUE ERROR`, even if it's just an event value in a heterogeneous array. It would therefor be helpful to have "undefined" itself as a "value".

The details

The first aspect of implementing "undefined" as an attribute or "value" is that it can be associated with a name. One way this can be done is by simply giving all localized names an initial value of UNDEFINED, unless or until another value is assigned to them (or passed, if they are internal argument names). This convention would not need a separate symbol or notation for the UNDEFINED "value". In some circumstances (e.g., being used as a left argument to an ambivalent function) the use of such names would not generate an error, but in others cases (e.g., being used as a right argument to a primitive function) it could still generate a `VALUE ERROR`.

By using such explicitly-undefined named objects, one could build compound objects (arrays) in which some items are defined and others are not. But if we want to be able to do that, we will certainly want to be able to create such arrays even in situations where an UNDEFINED named object doesn't exist. For this, we would need a notation, a symbol, to represent that "value" as a primitive constant. I would like to propose the use of `∘` ("jot"), partly for its graphic simplicity and partly because of its current use in outer product. Unfortunately, at least one APL has already assigned it another, potentially incompatible meaning.[1]

Additional uses

Missing data

One obvious use of UNDEFINED as a value is to indicate lack of data. Actually, this is not as straightforward as might first appear, because I'm proposing that an UNDEFINED value as a left argument should invoke the function's monadic case. E.g., `UNDEFINED÷2` should give `0.5`, and not `UNDEFINED`. (For the latter sort of behavior, see my above proposal for an event data type.) Nevertheless, there are undoubtedly contexts in which it would make sense to use UNDEFINED to represent missing data.

Thorn (format)

Thorn (`⍕`) can be used dyadically for formatting numbers, but not for formatting characters. This presents a problem if one wants to use dyadic thorn on some (numeric) elements of a nested array, but monadic thorn on some (character, but possibly also numeric) others. Being able to supply a value of UNDEFINED as

[1] In Dyalog APL, the jot is now used as a composition operator

a left argument for those elements where monadic use is desired seems reasonable.

Grade

Similarly, grade on numerics can't take a left argument, but on characters it must. To specify a mix of monadic and dyadic use, left argument values of UNDEFINED would force monadic use.

From (indexing)

Aside from distinguishing between monadic and dyadic use of functions, there is one other very important APL context where the simple presence or absence of a value is significant: indexing. One of the difficulties in replacing bracket-semicolon indexing with an indexing function is finding a way to represent the case of an elided axis, i.e., a simple way to specify "all indices" along an axis without explicitly enumerating them. Yet what could be simpler than specifying a value of UNDEFINED, an explicit equivalent of elision?

Ambivalent operators

Unlike functions, operators in APL are either monadic or dyadic, but not both. The reason for this is notational, not mathematical. Allowing both operators — which have long left scope — and functions — which have long right scope — to be ambivalent would require more complex syntactic rules to insure unambiguous parsing. However, an operand with an UNDEFINED value opens up the possibility of essentially monadic use of otherwise-dyadic operators.

I won't propose here monadic variants for existing primitive dyadic operators, but I will suggest that there is interesting potential in those APLs in which user-defined operators are possible. And I will point out that APL's outer product appears to be just such a construct, with ∘ ("jot") as the symbol for UNDEFINED. In fact, I think jot would be an ideal symbol, partly because of this long-standard use.

Comments on Some Functions

Nub and Nubsieve

Dyalog APL has implemented the monadic function Unique, which returns the unique elements of its argument. It only works on vectors. A user-defined utility is still necessary to eliminate duplicate rows from a matrix. On the other hand, J's Nub primitive returns unique subarrays of rank one less than that of its argument, e.g., rows of a matrix, planes of a 3-D array, as well as scalar elements of a vector. I find this extension to be invaluable, and I think every APL should implement it.

For greater versatility, they should also implement Nubsieve, as found in J and Sharp APL (and which is similarly extended to arrays of rank >1). Nubsieve returns a Boolean vector result, which will select out the nub when used as left argument to compression along the first dimension. Nubsieve is useful for applying the nub-generating selection to additional data which may parallel that data used in defining the nub.

Without

I believe that all modern APLs have now implemented this primitive, which removes from its left argument all occurrences of the elements in its right argument. As with Nub and Nubsieve, J and Sharp APL have extended this function to arrays of higher rank, and I think this extension would be a valuable addition to any APL.

I also believe this primitive should have a boolean sieve-conterpart—like Nubsieve for Nub—for greater versatility. Even Sharp APL and J don't currently include this extension, but I think it's even more important than Nubsieve. If one wants to eliminate certain elements or subarrays from one variable or dataset, then it makes sense that one would want to be able to select corresponding data—or make a corresponding exclusion—from parallel data.

In Conclusion

I have proposed here a motley, but mostly mutually independent, group of enhancements to APL as it currently exists. With one exception, I have not proposed particular symbols for them. Though that is an important topic, its discussion would distract from the real purpose of this discussion, which is the proposed functionality.

Space and time have not permitted me to present the full detail of my thoughts and analyses regarding these proposals, but I hope that they stimulate a lively discussion. I also hope that before too long I may see some of them appear in APL.

References:

[1] R. Bernecky; APL84 Conference Proceedings (*APL Quote Quad*, vol. 14, no. 4), p.53

[2] R. Bernecky & R. Hui; APL91 Conference Proceedings (*APL Quote Quad*, vol. 21, no. 4) p.39

[3] J. Brown; APL84 Conference Proceedings (*APL Quote Quad*, vol. 14, no. 4), p. 81

[4] Lucas, Jim, "Array Oriented Exception Handling", APL85 Conference Proceedings (*APL Quote Quad*, vol. 15, no. 4), pp. 1-4.

[5] Lucas, Jim, "When Standard 'Datatypes' Aren't Enough", *APL-uutiset 2/2000* (*APL News 2/2000*, publication of FinnAPL, Helsinki, Finland).