NESTED ARRAYS, OPERATORS, AND FUNCTIONS

Bob Smith
Manager, Special Research
Projects Department
STSC, Inc.
7316 Wisconsin Ave.
Bethesda, MD, USA 20014
(301) 657-8220

Abstract

Although the original ideas of nested arrays go back over ten years, it has been only within the past few years that these ideas have matured. During the same period, several new operators and functions have been proposed. This paper describes an experimental research implementation of these proposals. Details of some of the proposals are also described.

In March of 1981, STSC released to its customers an experimental research implementation of nested arrays, an enhancement considered by many to be the next major evolutionary step for *APL*. Our interest in this topic is not as a language enhancement, but first and foremost as a high gain productivity tool for programmers. Although *APL* is already considered a highly productive language, we believe nested arrays can make it far more so.

As an experimental research system, NARS, as it is known, represents a radical departure from our past methods of designing, implementing, and testing system enhancements.

Previously, enhancements would be designed on paper, and then reviewed for completeness, compatibility, consistency, and desirability. Assuming an enhancement passed this stage, it would be implemented in 370 Assembler language, tested, and released.

However, because of the expected impact of nested arrays, the above approach seemed infeasible. We re-examined our previous methods and discovered the hidden assumption that once an enhancement passed the design review, it was virtually assured of becoming part of our remote computing service. This meant that the design reviewers had to attempt to peruse every facet of an enhancement without ever actually using it. This method has proved practical for previous enhancements. But we wondered if it was enough for an enhancement on the scale of NARS.

Moreover, the validity of our previous methods is reinforced as a consequence of the usual high cost of implementing and maintaining enhancements. The cost of correcting a design flaw is significantly less if found prior to implementation. Of course, it appeared that NARS would require substantial programming resources.

So, we were confronted with the two horns of a dilemma: how to manage a large design which was impractical to review without a working implementation; and how to produce a working implementation which seemed uneconomical because of the high cost of making the changes which would result from a design review.

The dilemma was resolved in two ways. First, we decided to implement NARS in an intermediate form which provided all the features, but for which there was no commitment to apply these exact enhancements to our remote computing service or system software product. This way reviewers could test a working system according to the above criteria without having to resort to extended thought experiments over a paper design. Moreover, NARS was designed with change in mind, so our reviewers wouldn't be hampered by such cost considerations.

We call this form an experimental research system. It provides a testing ground not only for nested arrays, but also for many other new ideas which would be better evaluated through a primitive level implementation.

Given that we remove the long-term commitment to support these enhancements, we can then afford to implement for function

rather than speed. The second problem was then resolved by implementing NARS primarily using the internal-*APL*-function facility (also known as "magic" functions) provided by IBM's VSAPL product. The "magic" in these functions is that they are written in *APL* by a systems programmer, and then may be invoked by the system on behalf of a user when a primitive function or operator is used. The ability to use *APL* as the implementation language for primitives provided sufficient leverage to make the project feasible. The entire implementation took seven elapsed months.

Of course, an implementation of a primitive in *APL* is expected to run more slowly than one in 370 Assembler, but it seemed inappropriate to optimize an implementation without knowing how it is to be used. Given that some of the features in NARS might not make it to our production system, the choice was clear -- implement in *APL*. Later, when it becomes clearer where the system needs to be sped up, we can do so based upon experience rather than prediction. Meanwhile, we have the benefit of using the features as primitives in order to gain experience.

So far, from our limited experience gained prior to releasing NARS, it has more than met our expectations of improving productivity. Subjective estimates have ranged from a 25% to 60% reduction in programming time.

Another novel aspect of NARS is that although it represents only preliminary ideas, it is available to our customer base. We reasoned that with a bigger community of users, these ideas would be explored more thoroughly. We can learn much from the experience of our customers, who after all are the ultimate consumers of our work. Also, if our expectations about NARS improving productivity prove correct we want our customers to be able to take advantage of it immediately.

At the same time we were considering implementing nested arrays, there appeared several proposals for new operators and functions, notably from K. E. Iverson [1]. Since we had no long-term commitment to release the features of NARS, we decided to incorporate these new operators and functions. Again, we were using NARS as a testing ground for new ideas.

System Features

NARS contains a plethora of ideas, as is appropriate for an experimental research system. The following list is a selection of just some of the features of NARS (see reference [2] and its supplements for a complete list and description):

- Nested arrays data structure
- Over eight new operators producing over twenty new functions derived from these operators
- Fifteen new primitive functions
- Extensions to existing functions
- Strand notation

The following sections expand upon these topics.

Nested Arrays

One reason *APL* is so successful is its ability to represent in a simple manner a wide variety of algorithms and data structures. The data structures of *APL* are rectangular, multi-dimensional, and flat -- the latter term meaning that all items of arrays are simple scalars. Going hand-in-hand with these data structures is a rich set of primitive functions and operators with which a programmer can design algorithms. Looking at this combination, we see another hallmark of *APL*: the data structures and primitives were designed with each other in mind. A system rich in one aspect and poor in the other would be frustrating if not useless.

However, many programming tasks can best be described using data structures in which the items are not simple scalars. For example, consider representations of the names of the months of the year. Representations in current *APL* data structures require a certain degree of artificiality. One must use either pad characters in a matrix representation, or delimiters in a vector representation. We would represent the number of days in each month in a 12-item vector, so why not the names, too? In a nested arrays system, that is exactly what one does.

In general, a system with nested arrays extends the power of *APL* by being able to represent data of non-zero depth. Also, in keeping with the past, the system includes a set of primitive functions and operators, tailor-made for manipulating these new data structures.

Continuing the above example, the names of the months of the year are best represented in a 12-item vector whose items are each character vectors. This structure has a depth of one. Note that because the individual names are in separate items, we need no longer resort to artifices like pad characters. Moreover, explicit delimiters are not needed as the separation between items is represented through structure rather than data. This particular representation is called a vector of vectors, and can be created as follows:

```
      MONTHS←('JANUARY') ('FEBRUARY') ...
```

The above line also illustrates strand notation, used to enter a nested vector in a simple and convenient manner. It is described in more detail later.

New Operators and Derived Functions

Of the several new operators, the only one specific to nested arrays is the each operator (symbol ¨), which is monadic as an operator, and produces an ambivalent derived function. It is used to apply the function which is its argument to the items of an array to produce corresponding items in the result. For example, to determine the length of the names of the months in the above example, use

```
      ⍴¨MONTHS
(7) (8) (5) (5) (3) (4) (4) (6) (9) (7) (8)
      (8)
```

Since monadic rho returns a vector, each item of the above result is a vector (specifically in these cases a one-item vector). The parentheses in the display indicate that the item is not a simple scalar.

As an example of the ambivalence of the derived function, we can reshape the month's names to their three-letter abbreviations as follows:

```
      3⍴¨MONTHS
(JAN) (FEB) (MAR) (APR) (MAY) (JUN) (JUL)
      (AUG) (SEP) (OCT) (NOV) (DEC)
```

Another example of the each operator comes from partition functions [3]. Given two matching vectors, one with account codes in ascending order (some appearing more than once), and one with corresponding amounts to be charged to these codes, for each distinct account code, how much is to be charged to it? From the following variables

```
   AMT← 3.11 ¯6.20  9.12 ¯4.50  6.41  7.93
  ACCT←   83    83    83   101   101   201
```

we can create a partition vector of the breakpoints in *ACCT* with

```
      PV←ACCT≠¯1⌽ACCT
      PV[1]←1
```

The amounts corresponding to the same account code can be collected into individual items by the function partitioned enclose, as follows:

```
      PV⊂AMT
(3.11 ¯6.2 9.12) (¯4.5 6.41) (7.93)
```

and individually summed by applying +/ to each item via the each operator

```
      +/¨PV⊂AMT
6.03 1.91 7.93
```

Other operators that have been implemented include

- Power operator: applies a given monadic function a specified number of times. Essentially, this operator is a loop statement for a single function, although that function can be arbitrarily complex.
- Power limit operator: applies a given monadic function until the result is identical twice in a row. One use is in root finding using a Newton-Raphson technique.
- Power series operator: applies an inner product between multiple copies of a matrix until successive accumulations are identical. This versatile operator applies mathematical power series to solve a surprising variety of (often difficult) problems: paragraph formation, non-overlapping substring searches, mini-max problems, permutation cycle structure, and, in general, various properties of directed graphs like transitive closure, least cost transversal, etc.
- Composition operator: glues two functions together to form a single derived function, or binds a data argument to a dyadic function to form a monadic function.
- Dual operator: applies one function to an array between the application of another function and that other function's inverse. An example in a physical context is the effect of opening a drawer, modifying the contents, and then closing the drawer. The dual operator provides not so much a class of new functions, but a clearer understanding of algorithms, and in particular, the role of inverse functions.
- Commutation operator: switches the arguments of a dyadic function.
- Convolution operator: applies a moving window inner product between two vectors. Applications include polynomial multiplication, substring searching, and weighted moving averages.
- Direct definition operator: allows a user to create a derived function whose text uses ⍺ and ⍵ for the left and right arguments, and underbar (_) for the name of the derived function.

New Functions

Of those that pertain just to nested arrays, there are seven new functions:

- Enclose: follows the lead of More, Brown, and others, in that the argument is returned as the (only) item of the scalar result. Since, in this theory, the item which a simple scalar holds is itself, the enclose of a simple scalar is idempotent. This theory goes under various names (axiom system 0, floating system) none of which are mnemonic.
- Disclose: left inverse of enclose. Disclose is extended to return the first item of a non-empty array or the prototype of an empty array.
- Pick: selects an item of its right argument perhaps down through several nesting levels, as controlled by its left argument.
- Partitioned enclose: encloses consecutive items of its right argument into a vector of arrays, as controlled by its left argument.
- Simple: determines whether or not an array is simple.
- Split: separates the outer axes of an array into two levels. For example, the split along the last coordinate of a 3 by 4 matrix is a three-item vector, in which each item is a four-item vector.
- Mix: left inverse of split when split is used on consecutive axes.

Additional functions include operations on sets, equivalence of arrays, and type (converts numbers to zeros and characters to blanks).

Extensions to Existing Functions

Of the several extensions implemented, two of the more interesting are extensions to monadic iota and to indexed reference and assignment.

Over the years there have been a variety of proposals to extend monadic iota to vectors. Unfortunately, most of these extensions have been incompatible with the existing definition, which treats scalar and one-item vector arguments alike. The extension used in NARS has ιR defined on vectors as

$$\supset\circ.,/\iota"R$$

That is, first apply monadic iota recursively to the items of the argument; the recursion terminates upon encountering a simple scalar. Next, reduce this vector by an outer-product-catenate which returns a scalar since its argument is a vector. Finally, disclose the item in that scalar. For example,

```
      ⍳ 2 3
(1 1) (1 2) (1 3)
(2 1) (2 2) (2 3)
```

Note how this definition treats scalar and one-item vector arguments alike. The utility of this extension becomes evident in conjunction with the extension to indexing.

Indexing is extended such that an array of any rank may be indexed without using semicolons. For example,

```
      A← 2 3 ⍴⍳6
      A[(1 3) (2 1) (1 1)]
3 4 1
```

With the above extensions, identities like

$$A \leftrightarrow A[\iota\rho A]$$

hold for all arrays, even scalars. Together, these two extensions address the need to perform more complicated selections of indices in indexed assignment. For example, to set to zero the items on the main diagonal of a matrix, use

$$M[1\ 1\ \lozenge\iota\rho M]\leftarrow 0\ .$$

Earlier proposals for solving this problem have resorted to more complicated syntax like enclosing a selection expression in parentheses to the left of an assignment arrow.

Strand Notation

This notation (originated by Trenchard More) is an extension of the familar means of entering numeric constants. The rule for numeric constants is that whenever two or more numeric scalars appear adjacently, they form a vector. Strand notation extends this rule to named variables as well as expressions. With nested arrays, we can now represent the result as a nested vector of arrays. For example,

```
      A← 1 2 3
      A (A×2)
(1 2 3) (2 4 6)
      A A×2
(2 4 6) (2 4 6)
```

This notational convenience has many uses, not the least of which is in passing multiple parameters to a function. If *INPUT* is a user-defined function which requires three arguments, *X*, *Y*, and *Z*, they may be passed as a single argument by

```
      INPUT X Y Z
```

On the inside of *INPUT*, the argument is a single vector whose three items are the values of *X*, *Y*, and *Z*.

A further extension is made to assignment to allow multiple names to its left. In particular, to split apart the argument to *INPUT*, use

```
    ∇ INPUT R;A;B;C
[1]   A B C←R
```

The identifiers *A*, *B*, and *C* now contain the same values as do *X*, *Y*, and *Z*.

Miscellaneous Extensions

This category includes

- Ambivalent user-defined functions: the function header syntax is extended to require braces around the optional left argument.
- Operators accept user-defined functions as arguments.
- Heterogeneous data: numbers and characters may be mixed at will in any array.

Conclusions

Although we've had but a short time in which to use this system, the results have been very positive. The decision to implement for function rather than speed has greatly improved our ability to respond to design changes. Using *APL* as an intermediate implementation language was key to the whole project, enabling us to produce a complete working implementation of a large design in a very short time. Only after we gain sufficient experience with the system and settle upon the design need we re-implement for speed.

References

[1] Iverson, K. E., Operators and Functions, IBM Research Division Report RC7091, 1978.

[2] Cheney, Carl M., Nested Arrays Reference Manual, STSC, Inc., 1981.

[3] Smith, Bob, A Programming Technique for Non-Rectangular Data, APL Quote-Quad, Vol. 9, No. 4 -- Part 1 (APL 79 Conference Proceedings), 1979.