50 Years of APL Datatypes

Bob Smith
Sudley Place Software
Originally Written
29 Sep 2016
Updated
4 Nov 2016

Introduction

The language APL has progressed in fits and starts. Sometimes it moves because someone has a great idea for a new primitive, but the major moves have occurred because someone introduces a new datatype. New datatypes have such a profound effect on the language precisely because they apply to all primitives, and so each primitive is re-evaluated in the light of the new datatype. In fact, that is a hallmark of these datatypes in that they made us re-think all of the existing primitives as well as consider defining new ones, and as a result opened up new **Application Domains**.

What's A Datatype?

For the purposes of this paper, I limit the discussion to **Built-in Data Types** defined[13] as "a data type for which the programming language provides built-in support".

It must be something that can be stored in a variable, but that's not necessarily precise enough as it leaves Shared Variables, Name Associations, etc. in limbo. For my purposes, I require also that it also be an object to which almost all primitive functions and operators are sensitive (and not in the sense that the test is solely to decide whether or not to generate an error).

Here's my list of the major datatype milestones in APL.

Booleans

Booleans were the first datatype that caught my attention. As Philip Abrams[14] remembers "Booleans have existed from the beginning. Iverson's *A Programming Language* (1962) had logical (aka boolean) values. They were used, among other things, to control compression, mask and mesh." Bernecky credits Larry M. Breed with making a "fundamental, far-reaching, design decision"[9] to represent Boolean data in APL\360 in the form we now have, unchanged in its format from the original implementation in 1966.

They are ideally suited to construct loopless algorithms, a process Bernecky aptly calls "the replacement of control flow by data flow, frequently improving performance as a result of eliminating conditionals"[9]. That's a keen observation as one can think of the `1`s and `0`s in a Boolean vector as saying do this, or do that – that is **Control Flow** converted to **Data Flow** by virtue of using a Boolean vector, or equivalently, a **Control Structure** replaced by a **Data Structure**.

I found many uses for them, especially in what I called partition functions[6], the topic of a paper at APL79. According to Wikipedia[1], APL is one of the few (if not the only) programming language to take full advantage of that datatype, not only storing the values one per bit, but accessing them in a manner consistent with other datatypes (e.g., `V[3]` is the third element in `V` regardless of whether it's an integer, float, or Boolean). In addition, nearly every primitive treats them specially often using a special higher performance algorithm such as summing the bits in a Boolean vector by using a table of bytes. My early papers on Booleans are collected into one file[10].

Booleans are ideal for use in structural primitives such as Compression and Expansion, both designed specifically for a Boolean left argument,

and they are returned by many primitives such as the fourteen dyadic Boolean functions (=≠≤<>≥≡≢∨∧⍱⍲∈⍷) along with the monadic Not function (~). These functions are also significant when their arguments themselves are Boolean. Reduction and scan on Booleans are handled specially for many of these same dyadic Boolean functions and especially Plus reduction.

To illustrate this with an anecdote, some years ago I wrote a function called `RELABEL` which relabels a user-defined function to use the set of labels `L1`, `L2`, etc. The first item of business is to find all of the labels and their references. The latter is tricky because one must skip over the text that looks like a label but appears in a comment or in a character constant, and at the same time avoid being fooled by a comment symbol in a character constant or an unmatched quote mark in a comment. The `RELABEL` function did all of this without a single loop thanks to the use of Booleans.

The breadth and depth of the combinations of these functions operating on Booleans and returning Booleans opened up for me a wealth of algorithms as well as a way of thinking about data and algorithms (Control vs. Data Flow). One of the most important reasons for this was that, in a limited, but very useful sense, Expansion is an inverse of Compression. This powerful relationship was exploited time after time in my own code. One example of this is the algorithm for Partitioned Plus Scan, which in modern notation looks like this

```
      P←1 0 0 1 0 1 0 0 0
      R←⍳⍴P
      P ⋄ R ⋄ +\R-P\⍪2-\P/+\⍪1↓0,R
1 0 0 1 0 1 0 0 0
1 2 3 4 5 6 7 8 9
1 3 6 4 9 6 13 21 30
```

where `P` is the Boolean partition vector (always with a leading `1`). The nested arrays solution to this is

```
      (+\P)⊂R
 1  2  3  4  5  6  7  8  9
      +\¨(+\P)⊂R
 1  3  6  4  9  6 13 21 30
      ∈+\¨(+\P)⊂R
1  3  6  4  9  6 13 21 30
```

Another very interesting observation on this one-liner is to note the two interleaved pairs of inverses: `P\` and `P/` as one and `¯2-\` and `+\` as the other.

This interleaving of inverses forms an operator template called the Commutator operator (deserving of its own symbol!) coming from mathematics (in particular, Group Theory from the 1830s), the template of which is

$f^{-1}g^{-1}fg$

or, in APL,

```
(f⍣¯1)(g⍣¯1)f g
```

Related to this is the Dual Operator whose template is very similar to that of the Commutator Operator:

```
Commutator:  (f⍣¯1)(g⍣¯1)f g
Dual:                (g⍣¯1)f g
```

The Dual Operator is found everywhere, even outside programming languages, such as Open a drawer, Change its contents, Close the drawer.

These are yet other examples of what makes APL so interesting – discovering templates such as this and the incorporating them into

your toolbox. Mathematics teaches us that, among other things, the inverse is an important property of a function; APL makes it practical. I'll wager to say that there is no other programming language for which such observations are so common.

The importance of this datatype is indicated with its introduction in APL\360, the very first implementation of APL from IBM, and continuing with its presence in nearly every implementation of an APL-like interpreter since then.

The significance of Booleans is that they are the most compact storage of data, their presence brings forth a host of fast special purpose algorithms, and they occur often enough in normal computation that this special treatment is very worthwhile.

Characters

Initially, APL systems were limited to uppercase italic and uppercase underlined italic characters along with the special characters of APL and a few others. As the language became more widely used, more characters were added. IBM's APL2 pioneered the way with its National Language support based upon code pages[11] and support for both one- and four-byte characters.

After the Graphical User Interface (GUI) was introduced, APL vendors scrambled to adapt to it. One outcome was the proliferation and incompatibility of many different APL fonts which hindered transfer of functions and data across APL implementations. The Unicode standard largely did away with those incompatibilities as all APL characters are present in Unicode at fixed agreed upon codepoints, and could be displayed on any device that supports Unicode. As hardware and software improved, it became easier and easier to display APL characters anywhere you like.

Its significance is that, as standards are supposed to do, it unified the

competing and conflicting local specifications across APL vendors into a cohesive whole with value well above and beyond just APL characters.

Also, in the past APL has been criticized for “all those funny symbols”. Now with introduction of Unicode, we can say “You think APL has lots of funny symbols, let me show you Unicode”.

Nested Arrays

Nested arrays are near and dear to my heart as they represent my first foray into APL systems development. It started with reading voraciously Jim Brown’s doctoral thesis on “A Generalization of APL”[5] sometime after I joined STSC in 1971. After moving to California in 1978 to work for Roy Sykes, I was called back to company headquarters in Bethesda MD to design and implement Nested Arrays. Actually, I was tasked only with designing the feature and when the question of who would implement it came up, I just raised my hand, which is how the first Nested Arrays Research System (NARS)[7] was born. My first task was to learn how to program in IBM Assembler Language. They say that in order to learn a new programming language well you need a good starter project to motivate you. Mine was Nested Arrays.

The significance of nested arrays is that they removed the requirement of shallow rectangularity in APL. In effect, we could now represent a tree of data in a natural way. Before this, various techniques were used such as padding out short character names (e.g., calendar month names) with blanks so that a collection of such names fit into a rectangular array, or my favorite was to use a Boolean vector to partition data into different size pieces and then allow a function to be applied separately to each piece as shown above. Nested Arrays made all that so much easier.

We all remember those days of trying to fit non-rectangular data into a

rectangular array. Nested arrays led us into the woods so we could see trees.

Our terminology changed, too. The arrays we were so use to were now called “simple” arrays, the new ones “nested”. Heterogeneous arrays were now possible, made by catenating scalar characters and numbers. Enclose, Disclose, Partitioned Enclose, Pick, Split and Mix (to use the old names), Depth, and Each (the only new operator) were new primitives, and all of the old primitives were changed to accommodate Nested Arrays. For example, monadic Iota and Indexing were changed so that in combination, the identity `A≡A[⍳⍴A]` which used to be true for vectors only now is true for all rank arrays, even scalars.

Overall, of all the extensions made to APL over all the years, I consider Nested Arrays to be the most pervasive as well as the only structural datatype, and consequently the most significant.

Nested Arrays weren’t without controversy as there were two competing designs called Floating and Grounded which differed in many ways, but fundamentally on whether the enclose of a simple scalar produced a new array or the same array. I was in the latter camp which I think was Floating, but I’m not certain as those names never really meant anything to me. Anyway, all of the major APL implementations now use the same design, which, I think, is Floating. You shouldn’t read into this that Floating is better than Grounded, because many other factors (economic, etc.) affected the outcome.

Arithmetic Progression Arrays

This datatype is an outgrowth of the VSAPL datatype of **Arithmetic Progression Vectors**. The extension from vectors to any rank/shape array is straightforward. The data portion of the storage of an APA is of fixed size; beyond the standard array header, it requires only two additional signed integers of offset and multiplier. This makes it

possible to represent a huge array in a tiny amount of storage. This datatype can also be used to represent the arbitrary reshape of an integer singleton, in which case the offset is the value of the singleton and the multiplier is zero, so it's not only monadic iota that generates an APV.

APVs can also be used by APL system programmers in indexing to substitute for an elided array coordinate, so as not to have to handle elided coordinates specially all through the indexing code.

Its significance is to provide a shorthand for a common structure saving both storage space as well as execution time because this datatype can be processed more efficiently by various primitives.

This datatype is an instance of an excellent idea of Bernecky's called **Array Predicates**[8]. In particular, APVs with a multiplier of `1` and an offset of `0` or `1` are an example of the array predicate **Permutation Vectors**. This predicate allows the system developer to handle such arrays specially (and much faster) when various primitives encounter them. For example, the grade of a PV can be done in linear time. These array predicates cost essentially nothing in execution time and are very simple to implement. Another array predicate that is of use (and also costs next to nothing to assign and test for) is **All2s** used by the Encode primitive as its left argument to return a Boolean result from integer right arguments.

Complex Numbers

This datatype moved APL into the world of two-dimensional numbers. They are the first step into Hypercomplex numbers. The term dimension has two meanings here. One is the usual APL term of an array coordinate, first dimension, last dimension, etc. The other meaning is a mathematical one where it refers to the number of coefficients of a number, where Real numbers have one coefficient, Complex numbers have two, etc. Complex numbers are an optional

feature of the Extended APL Standard and are part of almost all major APL implementations.

As my implementation of APL is an Experimental system, I decided to extend Complex numbers beyond the usual floating point number coefficients, and to allow the coefficients all to be one of 64-bit integers, 64-bit floats, multiple-precision integers/rationals, or multiple-precision floats.

Its significance is to address problems in mathematics, physics, chemistry, biology, economics, electrical engineering, etc. where previously we had to make do with user-defined functions that simulated a Complex number. You may remember Paul Penfields's electric circuit analysis package Martha[12]. As with other datatypes, Complex numbers opened up whole new application domains for APL to address.

Hypercomplex Numbers

These numbers extend the mix of numeric datatypes from Real (one-) and Complex (two-) numbers to Quaternions (four-) and Octonions (eight-)dimensional numbers. Mathematicians have defined multi-dimensional numbers beyond eight, however they have a fatal flaw (called **Zero Divisors**) which makes them much less interesting.

Quaternions are used much more than you might think. Video games often use them as they are convenient for rotating, scaling, and translating figures on the screen all at the same time by multiplying by one number. Also NASA uses them in commands for attitude control systems sent to spaceships, etc[2].

Octonions are the crazy uncle of datatypes and have not found many practical uses as yet, although they are used in abstract mathematics (exceptional Lie groups, string theory, special relativity, and quantum logic)[3].

I've found them fascinating and have written numerous papers on Hypercomplex Numbers which can be found on my website[4] on the topics of overview, notation, implementation, quotients, and non-commutativity. That is, Quaternions and Octonions are both non-commutative; that is, `a×b ←/→ b×a`.

The significance of Quaternions and Octonions to the APL user is unclear. More time is needed to apply them to various problems, perhaps starting with spatial rotations.

Multiple Precision Numbers

These numbers allow APL to handle two new classes of problems: ones that require an exact (as in infinitely precise) solution and ones that require a highly precise but still inexact solution. MP numbers come in two forms: integer/rational and floating point. That is, the extend the fixed-precision datatypes of Integer and Floating Point to the corresponding multiple-precision datatype.

MP Integer/Rational

This form is used when an exact solution is needed and no floating point functions are used. For example,

```
      ⎕PP←5
      2*100×⍳2
1.2677E30 1.6069E60
      2*100×⍳2x
1267650600228229401496703205376
160693804425899027554196209234116260252220299378279
2835301376
```

showing how MP numbers calculate the full result. They also return exact results in places you might not have expected:

```
      ⌹a←3 3⍴⍳7
¯0.19048  0.047619  0.14286
¯1.619    0.90476  ¯0.28571
 1.4762  ¯0.61905   0.14286
      ⌹b←3 3⍴⍳7x
 ¯4r21   1r21  1r7
¯34r21  19r21 ¯2r7
 31r21 ¯13r21  1r7
      (⌹a)+.×a
 1           5.5511E¯17 ¯1.1102E¯16
¯3.9968E¯15 1          ¯2.6645E¯15
 3.7748E¯15 1.7764E¯15  1
      (⌹b)+.×b
1 0 0
0 1 0
0 0 1
```

My favorite example comes from Project Euler where they ask for the low-order ten digits of the sum of the first `1000` powers of `N*N`. The obvious solution is to write `¯10↑⍕+/*⍨⍳1000` but of course that expression quickly runs out of precision and yields an answer of infinity (∞) because the integer exponents overflow the range of floating point numbers. A trivial change to this failing solution converts the number `1000` from an integer to a multiple-precision integer and by appending one character transforms it into a working solution (`¯10↑⍕+/*⍨⍳1000x`) to yield the correct answer of `9110846700`. Note how the initial MP number propagates through the expression returning an MP number at each stage. Key to this is that the system doesn't type demote MP numbers, even though they could be represented in a smaller storage type. BTW, encountering this exact problem is what convinced me to stop what I was doing and implement multiple-precision numbers in NARS2000.

MP Floating Point

Multiple-precision floating point numbers on the other hand might not

be so obvious as to where you might use them. Most of us barely understand fixed-precision FP much less multiple-precision. Do any of us really know how much precision we need in our floating point calculations? Not really (and I'm no exception), and that's why we need multiple-precision floating point numbers. In order to determine whether or not your floating point calculations are working just fine or are being strained to beyond their precision limit is to convert your code to MP FP, run it, and compare the multiple- and fixed-precision results. It turns out that it's trivial to convert an APL program to its multiple-precision version at which point you can easily experiment with various levels of precision (including 64-bit FP this time as MP) and see whether 64-bit FP numbers are working for you. If the results are different (and they can be markedly different), then you need to re-examine your algorithm's use of 64-bit floats or just switch over to using MP floats.

To illustrate this point, here's a problem I found online with a simple answer: find the limit as `x` approaches ∞ of $x-\sqrt[3]{x^3-x^2}$. As an anonymous function, this looks like `f←{ω-3√-/ω*3 2}`. Trying a large value shows

```
      ⎕PP←40 ⋄ ⎕FPC←512
      f 1e6 ⋄ f 1e6x
0.3333334452472627
0.3333334444445061728806584663923417162208
```

Note that the fixed precision floating point result has **eight** (out of `17`) fewer correct significant digits than the MP result. In other words, for this admittedly special calculation, fixed precision arithmetic for this particular argument (`1e6`) generates an answer with almost half of its significant digits wrong. Moreover, roughly speaking, as the argument to the above function increases by a factor of ten, the fixed precision result **loses** a significant digit, and the multi-precision result **gains** a significant digit.

The reason for the loss of significant digits with each increasing power of ten is straightforward. As the answer to the limit problem is `÷3`, the cube root is calculating a number ever closer to `⍵-÷3` so that the difference between `⍵` and the cube root approaches `÷3`. However, as the cube root gains significant digits to the left of the decimal point (due to the rising power of ten) it correspondingly loses significant digits to the right of the decimal point (due to the fixed precision), and so the difference between `⍵` and the cube root becomes less and less precise to the point of being meaningless. The MP calculation has the same problem, but its precision can be set to an arbitrarily large value so the problem can be pushed back as far as you like.

While your fixed precision floating point calculations might not be this extreme, it does illustrate how easy it is for a calculation to go awry and produce many fewer significant digits than you might expect. The moral of this story is don't program in fixed precision floating point arithmetic on a wing and a prayer. Use MP arithmetic to validate your fixed precision floating point calculations.

As mentioned above, another key to the utility of MP numbers is the ease of conversion between fixed- and multiple-precision arithmetic – simply append to all numeric constants an `x` (for Extended-precision) and you're done. There's no need to rewrite your code and then retest it to make sure you haven't introduced some bugs. Moreover, because the system automatically promotes from MP Integer/Rational to MP Floating Point, start by marking all constants as MP Integer and let the system promote to MP Floating Point as needed – it all happens under the hood, you don't need to do anything more than run it. When I say all numeric constants, I mean all, because every decimal number in APL is expressible as a fraction, e.g., appending an `x` to (say) `1.1E¯2` yields the MP Rational (not Floating Point) number `11r1000`. If the constant were `1.2E¯2` , then appending an `x` yields `3r250` as the system automatically removes common factors from the numerator and denominator.

Implementing MP arithmetic is a big job as it touches every primitive, and most particularly, every primitive that does numeric calculations must duplicate that code in MP arithmetic. At the same time, that also attests to its utility – every primitive can benefit from its presence. You can find online excellent libraries of MP routines[15,16]. No matter how much work it is to implement, MP really belongs in the language.

The significance of MP numbers is that they allow the user to address new classes of problems that require more than the fixed precision of integers and floats by making trivial changes to the existing fixed precision code.

I have had both forms of MP numbers in my implementation of APL for several years and, by example, have tried to convince Dyalog to put them into their implementation. Now that you are all convinced that you need MP numbers, I want you to turn to the nearest Dyalog representative and say to them “We need multiple-precision numbers!”.

Conclusions

As you can infer from the above, I believe datatype is king. There have been many individual functions and operators introduced over the last 50 years, but I see datatypes as having driven the most significant changes in the language as well as making it more and more relevant to modern problems.

I have been honored to have been present at the start of so many of these events.

The last 47 years from when I was first introduced to APL in 1969 have been an exciting adventure as I’ve been introduced to a way of thinking about programming unlike any other. Although I’m programmed in many other languages since 1969, APL is the one language I keep coming back to – it just makes sense to me. Even after I left STSC in 1983 and my wife and I started our own software company, I was still

going to APL standards meetings and conferences. I used APL in our software business for our flagship product 386MAX to create an optimization algorithm. Once you've learned to think in APL, computer programming and thinking logically in general becomes so much easier.

Acknowledgments

I'd like to thank Philip Abrams, Bob Bernecky, David Liebtag, and Roger Moore for their helpful advice.

Online Version

This paper is an ongoing effort and can be out-of-date the next day. To find the most recent version, go to http://sudleyplace.com/APL/ and look for the title of this paper on that page.

References

1. Wikipedia, "Bit Array", https://en.wikipedia.org/wiki/Bit_array#Language_support
2. Wikipedia, "Quaternion", https://en.wikipedia.org/wiki/Quaternion
3. Wikipedia, "Octonion", https://en.wikipedia.org/wiki/Octonion
4. SudleyPlace.com, http://sudleyplace.com/APL
5. Brown, James A., "A Generalization of APL", Ph. D. dissertation, Syracuse University, 1971, www.softwarepreservation.org/projects/apl/Books/AGENERALIZATIONOFAPL
6. Bob Smith, 1979, "A programming technique for non-rectangular data", In *Proceedings of the international conference on APL: part 1* (APL '79), ACM, New York, NY, USA, pp. 362-369, http://dl.acm.org/citation.cfm?id=804488
7. Bob Smith, 1981, "Nested arrays, operators, and functions", In *Proceedings of the international conference on APL* (APL '81), ACM, New York, NY, USA, pp. 286-290, http://dl.acm.org/citation.cfm?id=805376

8. Robert Bernecky, 1998, “Reducing computational complexity with array predicates”, In *Proceedings of the APL98 conference on Array Processing Languages*, pp. 39-43, http://dl.acm.org/citation.cfm?id=327614
9. Robert Bernecky, 2016, “A Compendium of SIMD Boolean Array Algorithms in APL”, preliminary draft.
10. Bob Smith, 1982, “Boolean Functions, 2nd Ed.”, STSC, http://www.sudleyplace.com/APL/boolean.pdf
11. Wikipedia, “Code Pages”, https://en.wikipedia.org/wiki/Code_page
12. MARTHA, http://marthallama.org/
13. Wikipedia, “Primitive data type”, https://en.wikipedia.org/wiki/Primitive_data_type
14. Philip Abrams, 2016/10/02, personal communication.
15. Multiple-Precision Integer/Rational, http://mpir.org/
16. Multiple-Precision Floating Point, http://mpfr.org/