Pure Functions in APL and J

Edward Cherlin
APL News
6611 Linville Drive
Weed, California
USA
Telephone: 916 938 4684
Reuters Mailbox: edmc

Abstract

Any expression in combinatory logic made up of combinators and variables can be abstracted into a pure combinator expression applied to a sequence of variables. Because there are great similarities between combinators and certain APL operators, a similar result obtains in many APL dialects. However, rewriting arbitrary APL expressions as pure functions requires new operators, not provided as primitives by any dialect. This paper defines functional completeness, gives a construction for achieving it, proves a conjecture of Kenneth Iverson that J is functionally complete, and shows how closely the major APL dialects have approached these conditions.

Keywords: APL, combinatory logic, completeness, functional programming, abstraction

Introduction

Over a period of years Kenneth Iverson has added functional programming features to dialects of APL that he has designed [9 13 14]. He conjectured [15] that the set defined for the J language was sufficient to allow any expression to be rewritten as a pure function, in the manner of combinatory logic. A constructive proof of this conjecture is given below.

In Combinatory Logic, founded by Schönfinkel [22], Curry, and Feys [5], a very small set of primitives (Table 1) with nearly trivial syntax, including parentheses for grouping, is shown to be an adequate foundation for mathematics. This set can be used to create a model of the natural numbers and to express all computable functions of natuural numbers in the sense of Church [4], Gödel [6], Markov [16], Post [20], or Turing [24], and thereby, via Gödel numbering, all proofs from a recursively defined set of axioms.

TABLE 1

Primitive combinators

Sign	Definition	Name	APL analog
I	**Ia=a**	identity	⊢ω ←→ω
K	**Kab=a**	left	α⊣ω ←→α
C	**Cabc=acb**	swap	α f⊂ ω←→ω f α
B	**Babc=a(bc)**	composition	f⍥g ω ←→f g ω
W	**Wab=abb**	duplicate	f⊂ω ←→ω f ω

The **T** combinator, defined by the expression **T=CI** or the relation **Tab=ba**, has no analog in APL, since **b** would have to be a function, and no operator can produce an operator.

All combinators can be written as combinations made from just two of the combinators defined here (**K** and **S**), or even only one combinator defined specially for the purpose. The most common primitive combinators are listed in Table 1, with Dictionary [13] or SAX APL [21] analogues that were described in [3] as trivial functions or operators. Each primitive combinator rearranges its arguments, or selects one, but does no other processing. Iverson has adopted two other combinators defined by Curry as Phrasal Forms in the J language [9]. They are displayed in Table 2 along with the analogous dyadic forms.

Any combinator can be applied to any combinator expression. For example, **K(a)** is a legitimate expression for any **a**. **K** by itself represents the Left function which returns its left argument, and **K(a)** is a constant function which returns the value of **a** when applied to any argument. Creating derived functions in this manner by supplying one argument is called currying in honor of Haskell Curry. It is analogous to the With operator in Iverson's recent dialects, written as ⍤⍤ in Dictionary APL [13] and as `&` in J [9]. For example `1⍤⍤○` in Dictionary APL is the Sine function. It is also possible to curry the right argument, as in **(CWa)b → Wba → baa** or in J, `Decrement =. -&1`.

Table 2

Phrasal Form Combinators

Sign	Definition	Name	J Analogue
S	**Sfgx=fx(gx)**	hook	`(f g)y↔y f g y` `x (f g) y↔x f g y`

Φ	**Φfghx=f(gx)(hx)**	fork	`(f g h) y↔`
			`(f y) g h y`
			`x(f g h)y↔`
			`(x f y)g x h y`

Combinators are generally considered to be too powerful to add to APL, since they apply to arbitrary combinator expressions, whereas APL functions apply only to array values, and operators apply only to arrays and functions. Nothing applies to operators or produces operators in any APL program product, except operator definition (:: in J). Suggestions for hyperoperators, able to apply to all lower types and to each other, have been made and rejected for APL many times. These combinator analogues that observe the APL type restrictions are less offensive but still controversial.

Definitions

We consider APL-like **languages** containing variables, functions and operators, with type restrictions as in APL. Within those languages, we consider the class of **simple expressions** containing only functions (primitive or derived), literal data and data variables, with no assignments and no operand variables, and the class of **pure function expressions,** containing literal data, functions and operators, applied to a nested vector of arguments, where no function or operator definition includes any reference to array variables. Such a language will be called **functionally complete** if any simple expression in the language can be rewritten as a pure functional expression.

This property is related to combinatory completeness and is attainable for APL, although no dialect has yet implemented the precise set of primitive operators for doing so. Functional completeness is obviously impossible for "Classic" APL [10], which has no compound operator expressions, but can be done with a small set of operators added to several varieties of nested array APL, in which derived functions produced by operators can be operands to other operators.

The process of translating an arbitrary expression into a function is called **abstraction**, and the function that results is called the **abstract** of the expression. Abstracts are difficult to read at first sight, looking quite abstract and even abstruse, but like other expressive notations, they grow on users, and soon seem entirely natural.

An Example

Consider the idiom `(∨\S≠' ')/S` for removing *leading* blanks from a character vector, and its translation into J as `(+./\S~:' ')#S`. We can transform it step by step into an abstract as follows:

`(+./\S~:' ')#S`	
`((+./\(~:&' ')S)#S`	Curry
`(((+./\&(~:&' '))"1 S)#S`	Compose, rank
`(((+./\&(~:&' '))"1 S)#{:S`	Identity
`(((+./\&(~:&' '))"1) # {:)S`	Fork

Functional Programming

The concepts of functional or applicative programming derive from Backus's attempt to define a programming language not tied to von Neumann computer architectures, which process data one word at a time. He set forth his solution in his Turing award lecture [1]. Taking some concepts from APL, LISP and elsewhere, he defined a language called FP (Functional Programming), which includes operators and array functions but no assignments. [17] is a good example of the resulting programming style in standard APL.

Paul Hudak surveyed the field of functional programming in [8], giving credit to APL's contributions. Functional programming languages, in general, offer many of the advantages of APL notation, such as conciseness, expressive power, and leaving many of the details of execution to the interpreter, and some have ideas worth looking at for inclusion in APL. In fact Hai-Chen Tu, working with Alan Perlis, created a functional variant of APL, described in [23]. However in many of these languages, array handling is de-emphasized in favor of lists and tuples, with strong typing and other features that make them almost, but not quite, entirely unlike APL.

Functional programming languages have a variety of implementations of abstraction. Some use a special construct like Church's Lambda [4]. In such languages, which have the flavor of LISP, an expression such as (SUM(X X)) can be turned into a function in the form LAMBDA(X(SUM(X X))). Abstracts in Lambda calculus have no free variables but in general mention bound variables. An extension of this technique for defining recursive functions uses the paradoxical combinator Y. The Y combinator has the property that Ye = e(Ye) for any e, which is to say that Ye is a fixpoint of the expression e. In those languages where Y can be defined, $Y(\lambda f.(...f...))$ is a non-recursive definition of the function defined by the expression in the innermost parentheses. Again, the point is that there are no free variables, and *f* is not the name of the function.

This treatment of the function as argument is not possible in ordinary APL. The capabilities of the Y combinator can be

added to the language, as in Dictionary APL and J, as a special feature of a definition operator. Thus,

```
f=. '...$...'::'...$...'
```

defines an anonymous ambivalent recursive function and assigns it a name. This notation subsumes much of the Lambda notation by using fixed argument names (*x.* and *y.*) so that the header (λ and argument names) can be omitted, and using nested arrays to pass multiple arguments. There are major differences, however. APL uses call-by-value, and Lambda calculus uses a form of call-by-name, not evaluating arguments until they are used. In Lambda calculus there are none of the APL type distinctions among data, functions and operators. Iverson's Definition operator also allows mutual recursion between the monadic and dyadic cases, although this is not recommended practice.

In Lambda calculus $Y=\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$, and in combinators it can be defined as **Y=UU**, where **U=BW(W(B(CB)))**. The derivations are:

Ye	
$\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))\ e$	definition
$(\lambda x.e(x\ x))(\lambda x.e(x\ x))$	apply
$e(\lambda x.e(x\ x))(\lambda x.e(x\ x))$	apply
e(Y*e*)	

Ye
UUe
BW(W(B(CB)))Ue
W(W(B(CB))U)e
W(B(CB))Uee
B(CB)UUee
CB(UU)ee
Be(UU)e
e((UU)e)
e(Ye)

In the first derivation, the applications are carried out by substituting the second expression for the argument variable in the first expression, and the last line is derived from a comparison of the two before it. In the second, after substituting the definitions of Y and U, the leftmost combinator is applied at each step according to the rules given in Table 1 above.

We are looking at abstraction methods which remove even the bound variables from expressions. The combinator expression for Y is an example of such an abstract, even though the Lambda calculus form is not.

The first great advantage of abstraction is that it does not require the apparatus of function editing or □*FX* to create a function and apply it. Using an editor to create a function is inconvenient or impossible under program control, and even using the □*FX* system function to translate character data into a defined function is impossible inline. It is necessary to define the function in one place and then use its name in another. Both of these methods of definition require a function to be named, but do not return the function as a value, ready to apply.

A pure function expression abstracted from an ordinary expression is already a function without the need for any further apparatus. It can be named in J by assigning it to an identifier, or it can be applied to its arguments unnamed.

However, an even greater advantage of function abstraction is that an abstract can be the operand of any operator. Given the abstract of the idiom for removing leading blanks, one can immediately remove blanks from each vector in an array of boxed character vectors using the "Under-Open" idiom *F"*>. Similarly the Each operator can be used to apply a function abstract to each of an array of enclosed character vectors, without any need for looping.

A third benefit that exists at least in J is that the abstract of an expression, applied as a function, typically executes much faster than the expression that mentions the variables explicitly [18]. Why this is so, and whether it must be so, should be researched. It could be a useful transformation in an optimizing compiler if abstracts are generally faster than ordinary expressions. For those expressions for which it is so, the abstract should be considered the standard form for idioms in J, since readability is not the main issue in composing and using idioms.

J forces the use of abstracts when writing operator definitions with the `::` operator. Formal operands are represented by *x.* and *y.* in operator definitions, but the arguments of the derived functions cannot be mentioned in the definition.

Construction and Proof

The general technique for abstracting in J uses the following combinatory features (functions, operators and phrases):

`}:`	Identity; Left
`{`	From indexing
&	With (curry and compose)
`(f g h)`	Fork

First we need a slightly different composition operator:

```
c=.2::'(x.&y.)"_'
```

The function *x&y*, where both *x* and *y* are functions, has the rank of *y*, but we need a composition operator that applies both of *x* and *y* with their individual ranks. We do this by giving the derived function *x c y* infinite rank.

The next step is to pull all variables out of the expression, and create a single nested array, which we will call *x.*, containing their values in any order. Then:

Case 1. Constant result of *C*.

```
(C&}:) x.
```

Case 2. Result, the *N*th argument

```
(N&{) x.
```

Case 3. *F A*

If *A* is constant

```
((F A)&}:) x.
```

If *A* has the abstract *G*

```
(F ⊂ G) x.
```

Case 4. *A F B*

If *A* and *B* are constants

```
((A F B)&}:) x.
```

If *A* is constant and *B* has abstract *G*

```
((A&F) ⊂ G) x.
```

If *A* has abstract *G* and *B* is constant

```
((F&B) ⊂ G) x.
```

If *A* and *B* have abstracts *G* and *H*

```
(G F H) x.
```

Proof. By induction. Cases 1 and 2 are the bases, and cases 3 and 4 take care of monadic and dyadic functions. Every APL expression that is allowed here is a constant, variable or function application. Any side effects of the original expression are also produced by the abstract, which applies the same functions in the same order.

Although this construction is general, there are opportunities for greater efficiency in many special cases. No use has been made of the Hook phrase or Both operator here, but they can simplify many expressions for abstracts, and in some cases it is simpler to use the commute operator to rearrange the order of arguments. In the monadic anddyadic cases (that is, where the original expression contains either one or two distinct variables) the usual argument syntax can be used with Left }: and Right {: standing in for the left and right arguments, instead of indexing into a nested argument vector.

So the primitives of J come close to providing functional completeness, and we have only to define one new operator to achieve it. What about other APL dialects? The Left and Right functions and the Both operator are in the enhanced APL standard [11], and are easy to define if they are not available. The APL2 Pick function ⊂ is close enough to From { to use in this construction. User-defined operators are also in the standard, and in most current APL products, along with nested arrays, so constructing the rest of the apparatus is easy, except for the Fork phrase. The Hook is easy, since it can be written as a dyadic operator, but the fork combines three functions.

It turns out that a fork can be simulated in several ways using two user-defined operators. The trick is to use a nested array to hold two intermediate results, so that they can be separated when they are used. This device can simulate a variety of three-argument operators, but by no means all. Define (in APL2)

```
   ∇R←x (f O h) y
[1] →(0=⎕NC'x')/MONAD
[2] R←(x f y) (x h f)
[3] →0
[4] MONAD:R←(f y) (h y)
   ∇
   ∇R←x (g P t) y
[1]  →(0=⎕NC'x')/MONAD
[2] R←⊃g/x t y
[3] →0
[4] MONAD:R←⊃g/t y
   ∇
```

Then we can write the fork form *x (F G H) y* as the APL2 expression *x G P(F O H) y*. This is a bit clumsy, but not excessively so. The conclusion is that any APL with a reasonable implementation of user-defined operators has the tools for abstracting expressions.

Implementation Alternatives

Although writing functions with no variables has an aesthetic appeal to some and is apparently more efficient than conventional notations, it is not to everyone's taste. There are numerous possible ways to write anonymous functions on the fly, providing syntax for some subset of the capabilities of Lambda and Y. Lambda itself has been implemented in Alan Graham's APL0 [7]; Iverson's : : operator covers most of Lambda and Y; and Iverson [12]. Tu [23], and Benkard [2] have all proposed to allow expressions containing the formal variables α and ω to act as functions, with varying syntax. Thus

```
2'α+ω'3 ⍝Iverson
2{α+ω}3 ⍝Tu
2(α+ω)3 ⍝Benkard
```

have all been defined to return 5. These αω-forms allow any expression in one or two variables to be written as an abstract merely by substituting the variable names. With nested arrays they are equivalent to the abstracts possible in J. That is, a form using only one or two arguments is equivalent to a form allowing an arbitrary number of arguments if the arguments can simply be stuffed into a single list.

Benkard also proposed a similar syntax for operator abstracts, using $\underline{\alpha}$ and $\underline{\omega}$ as the operands along with the formal arguments. This would allow great freedom in defining anonymous operators without the problems presented by adding

hyperoperators. Of the three notations given above, none is completely satisfactory because the quotes, braces and parentheses are all used for other purposes in various dialects. Perhaps a new pair of overstrikes would be appropriate.

It is easy to write Lambda expressions which evaluate correctly, even though an argument is undefined. Using call by value, as in APL, such an expression would be an error. There are Lambda expressions using functions as arguments that could not be written directly in APL. In both cases the behavior of Lambda expressions can be simulated in APL by passing arguments as character vectors and executing them at the appropriate times. In the end all of these systems are highly expyressive and complete in Church's sense.

Iverson's phrasal forms are quite convenient for writing pure functions, but they have evoked opposition on the ground that they change the syntax of APL. This opposition is similar to Iverson's rejection of strand notation as in APL2. Although the fork form can be simulated, the capability for abstraction is a strong argument for it or something equivalent and for the other features of J that support abstraction.

Unlike the case in combinatory logic, this construction does not extend to all expressions. In particular, there is no way in J or any other APL-like language to rewrite all operator expressions without variables for the operands. This is due to the APL type structure. We do not have functions as array elements; operators returning data (such as an explicit way to access the identity elements of primitive functions); or any primitive or user-defined facility that can accept operators as arguments and return them as values.

References

1. John Backus, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs", Turing award lecture, *CACM* **21**, No.8, pp. 613–641, (Aug. 1978).

2. J. Philip Benkard "Nonce Functions", *APL90 Proceedings, Quote-Quad* **20**, No. 4, pp. 27–39, (Aug. 1990).

3. Edward Cherlin, "APL Trivia", *APL90 Proceedings, Quote-Quad* **20**,4, pp. 71–75, (Aug. 1990).

4. Alonzo Church, *The Calculi of Lambda Conversion*, Princeton University Press, (1941).

5. Haskell B. Curry and Richard Feys, *Combinatory Logic*, North-Holland, (1958).

6. Kurt Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I", *Monatshefte für Mathematik und Physik*, Vol. 37, pp. 349-360, (1931). Translated into English as "On Formally Undecidable Propositions of the Principia Mathematica and Related Systems I", in Davis, *The Undecidable*, Raven Press, (1965).

7. Alan Graham, "APL0: A Simple Modern APL", *APL89 Conference Proceedings, Quote-Quad*, **19**, No. 4, pp. 178–183, (1989).

8. Paul Hudak, "Functional Programming Languages", *ACM Computing Surveys*, **21**, No. 3, pp. 359–411, (Sept. 1989).

9. Roger Hui, Kenneth Iverson, Eugene McDonnell, and Arthur Whitney, "APL\?", *APL90 Proceedings, Quote-Quad* **20**, No. 4, 192–200, (Aug. 1990).

10. International Standards Organisation, *Programming Language APL, ISO standard 8485*, (1989).

11. International Standards Organisation. *Programming Language APL, Extended, Draft Proposal*, Eugene McDonnell, Editor, (October 1990).

12. Kenneth Iverson, "Operators", *ACM Transactions on Programming Languages and Systems*, **1**, No. 2, pp. 161–176, (October 1979).

13. Kenneth Iverson, "A Dictionary of APL", *Quote Quad*, **18**, No. 1, pp. 5-40, (Sept. 1987).

14. Kenneth Iverson and E. E. McDonnell, "Phrasal Forms" *APL89 Conference Proceedings, Quote Quad*, **19**, No. 4, pp. 197–199, (Aug. 1989). Corrections, *APL News*, **20**, No. 3, pp. 5–6 and **20**, No. 4 p. 1, Springer-Verlag, New York, (1989).

15. Kenneth Iverson, personal communication, (1990).

16. A. A. Markov, "Teoria Algorifmov" (Theory of Algorithms), *Trudy Mat. Inst. Steklov* , **38**, pp. 176–189.

17. Eugene McDonnell, *The Four Cube Problem*, APL Press, Weed, CA, (1981).

18. Eugene McDonnell, oral presentation, APL90, (Aug. 1990).

19. S. L. Peyton-Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall International, Englewood Cliffs, NJ, (1987).

20. Emil Post, "Formal Reduction of the General Combinatorial Decision Problem", *American Journal of Mathematics*, **65**, pp. 197–215, (1943).

21. Reuter:file, Inc., *SAX Language*, Edition 1.2, (March 30, 1989).

22. M. Schönfinkel, "Über die Bausteine der mathematischen Logik", *Mathematischen Annalen*, **92**, pp. 305–584, (1924).

23. Hai-Chen Tu, *FAC: Functional Array Calculator and its Application to APL and Functional Programming*, YALEU/DCS/TR–468, Yale University, New Haven, CT, (Apr. 1986).

24. Alan Turing, "On Computable Numbers with an Application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society*, **42**, pp. 230–265, (1936).