APL function definition notation

John Bunda*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

A pure functional notation for defining APL objects is described, and contrasted with previous work in this area. The notation is extended to address both theoretical and pragmatic programming considerations. The notation is compatible with existing implementations, and is shown to straightforwardly incorporate popular extensions to the language.

Introduction

The functional programming paradigm has been studied as not only a programming methodology [Bac78,Hug84], but as the basis for a non-Von Neumann model of computation [Tur84,Hug82]. LISP is most often cited as the prototypical functional language, but APL has also played a part in the evolution of this programming paradigm. Backus [Bac78] drew considerably on his experience with APL in developing his functional notation.

Inspiration aside, APL has been all but ignored in recent functional programming research. One reason for this is that, despite the richness of APL notation, it is not possible to express many computations functionally. Pure LISP is a computationally complete functional subset of LISP. However, in APL, the imperative notions of "statement" and "goto" are sometimes *required* in order to express certain computations. There is no complete functional subset of common implementations that might be called Pure APL. From a formal perspective, APL remains as imperative as Pascal or Fortran.

Function definition notation

Alternatives to the conventional ∇-form of function definition have been proposed. Iverson's original direct definition notation [Ive76] is purely functional. However, its alternation is a simple if-then-else, and the definition form cannot be applied directly in an expression. Morrow [Mor77] defines a truly direct lambda-style notation, but does not include a conditional construct. Other proposals abandon the functional model, and reintroduce imperative features, essentially providing syntactic sugar for conventional ∇-form definitions [IW81], [Ive83], [Met80].

*Research supported by IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, New York, June-September 1985

In this paper, some of the pragmatic drawbacks of Iverson's original direct definition notation are addressed. We define a pure functional notation for defining APL objects that is compatible with existing implementations of the language (some symbols are given new meaning within our definitional form). A general alternation construct is introduced, which permits explicit restriction and partitioning of the domains of defined functions. The notation is extended with a naming and encapsulation facility that also eliminates a problem with application of mutually recursive definitions. The notation is shown to permit the graceful definition of ambivalent functions [IBM84,Ive83], and to generalize straightforwardly to defined operators [IBM84].

A notation for pure functional forms

The general form of a function definition expression in this notation is a *guarded alternation* [1]. As in other APL direct-definition forms, the names α and ω denote the values of left and right arguments respectively. The form:

$$\{guard_1 \rightarrow expr_1 \diamond guard_2 \rightarrow expr_2 \diamond \ldots \diamond guard_n \rightarrow expr_n\}$$

We use **true** to denote the APL numeric scalar value 1. When a function is applied, alternatives, separated by the symbol ◇ , are considered from left to right. If $guard_i$ has the value **true**, $expr_i$ is selected as the expression defining the function[2]. An unguarded alternative is considered to be implicitly guarded by **true**. A guard having no value or whose evaluation results in an error is considered to be the same any guard with a value other than **true**. That is, errors resulting from evaluation of guards are not signalled (though an error in a selected $expr_i$ would be signalled normally).

A definition expression enclosed in braces denotes a function, and it may be applied directly as part of another expression. For example, the following expression defines and applies the absolute value function:

```
{ω≥0 → ω ◇ ω<0 → -ω} ¯3
3
```

This definition contains two alternatives: $\omega \geq 0 \rightarrow \omega$ and $\omega < 0 \rightarrow -\omega$ When the function is applied to the argument ¯3, the guard: $\omega \geq 0$ is evaluated, and since its value is not **true**, the second alternative is considered. The value of the second guard is **true**, so $-\omega$ is selected as the function's definition; hence the value of the entire expression is 3.

In this example, the guards restrict the domain of the function to numeric scalars (or singleton vectors if coercion is admitted). If the function is applied to an array or character argument, neither guard will be **true**. If no alternative can be selected, a domain error is signalled at the application of the function.

```
      ABS ← {ω≥0 → ω ◇ ω<0 → -ω}
      ABS 1 2 3
DOMAIN ERROR
      ABS 1 2 3
          ∧
```

[1] The notation is similar to the alternation of [Dij76] or [Red84], but our semantics is deterministic.

[2] cf. [Ive76], in which a boolean expression selects one of two alternatives in an if-then-else fashion.

Associating a definition with a name via the assignment arrow (as shown with the *ABS* function above) makes it unnecessary to explicitly define a function each time it is used in an expression. A further consequence is that it is then possible for a function to refer to itself. For example, the factorial function might be defined:

$$FACT \leftarrow \{\omega=0 \rightarrow 1 \diamond \omega>0 \rightarrow \omega \times FACT\ \omega-1\}$$

This method has several disadvantages. First, there is no way to apply such a function directly; it must first be defined and associated with a name. Second, the function cannot be renamed in the global environment without also requiring a change to the definition itself.

Following [IW81], we use the symbol Δ to denote self-reference in a definition. This symbol is local name, independent of any global names associated with the definition. So factorial might also be defined:

$$FACT \leftarrow \{\omega=0 \rightarrow 1 \diamond \omega>0 \rightarrow \omega \times \Delta\ \omega-1\}$$

Since it is strictly local, Δ solves the global/local name problem mentioned earlier for simple recursions, but the problem remains with mutually recursive definitions (e.g. *F* calls *G*, which calls *F*, and so on). We will return to this problem shortly.

Symbolic substitution

We extend the notation to allow locally defined abbreviations, which we will call *symbolic substitutions*. Symbolic substitutions are delimited by semicolons, and appear following the guarded alternation *defn* within the definition braces. The general form is:

$$\{defn\,;\ name_1 \leftarrow expr_1\,;\ \ldots\,;\ name_n \leftarrow expr_n\}$$

Each $name_i$ is any APL identifier, and each $expr_i$ is an APL expression. If $name_i$ occurs in *defn*, upon evaluation, $expr_i$ is substituted. An $expr_i$ is only evaluated if an expression containing its $name_i$ is evaluated. For example, the following:

```
{⍳⍵;A←10÷0} 3
1 2 3
```

does not produce an error. On the other hand, if a substituted expression is evaluated, it is only necessary to evaluate it once [3]. These names are local, invisible in the calling environment. It is also important to distinguish the use of $\leftarrow$ in this definition form from assignment in the imperative sense. Multiple assignments to a name have no meaning here; only one subtitution may apply to a name within a given scope.

Since an $expr_i$ may be any APL expression, $name_i$ may stand for any value or function. The following identities hold:

$$\{X+\omega\,;X\leftarrow 1\} \Leftrightarrow \{1+\omega\}$$
$$\{X\circ .+X;X\leftarrow \iota\omega\} \Leftrightarrow \{(\iota\omega)\circ .+\iota\omega\}$$
$$\{F\ X;F\leftarrow\{\omega\}\,;X\leftarrow\omega\} \Leftrightarrow \{\iota\omega\}$$

The third equation shows how substitutions may be provided for an argument name. This is so often desirable that a shorthand for renaming of the symbols α, ω, and Δ is provided. A *template* may prefix the rest of the definition body, separated from it by a colon. The denotation of names in a template is similar to the conventions of the del-form function header (except that a single name defines a function name without substitutions for arguments instead of a niladic definition). This shorthand is defined by the following identities, where *defn* stands for a definition body:

$$\{FOO:\ defn\} \Leftrightarrow \{defn\,;FOO\leftarrow\Delta\}$$
$$\{MFOO\ R:\ defn\} \Leftrightarrow \{defn\,;MFOO\leftarrow\Delta\,;R\leftarrow\omega\}$$
$$\{L\ DFOO\ R:\ defn\} \Leftrightarrow \{defn\,;DFOO\leftarrow\Delta\,;L\leftarrow\alpha;R\leftarrow\omega\}$$

Some sample definitions:

```
{FACT N: N=0 → 1 ◇ N>0 → N × FACT N-1}

{SQRT N: N≥0 → N TEST APPROX N;
         APPROX←{(ω+N÷ω)÷2};
         TEST←{EPS≤|α-ω → ω ◇ ω TEST APPROX ω}
         EPS←1E¯10}
```

The first example is a refinement of the factorial function defined earlier. The second is a version of Newton's method for approximating square roots. Note that definitions are allowed to span multiple lines.

The template is just a shorthand for local substitution; it *does not* define any global referents. Conversely, as before, a global name need not be the same as the local name, for example:

```
FOO←{FACT N: N=0 → 1 ◇ N>0 → N × FACT N-1}
FOO 3
6
```

Here, *FACT* is the local name the definition, and its global name is *FOO*.

Local names may also *occur free*, i.e. be referenced globally, in local function definitions. This provides a solution to the mutual recursion problem mentioned earlier. Observe that the definition:

$$\{FOO\ A:\ A\leq 0 \rightarrow 0 \diamond A>0 \rightarrow BAR\ A\,;BAR\leftarrow\{FOO\ \omega-2\}\}$$

does not require any global names; it may be applied directly in an expression. Circular references can lead to non-terminating substitution, as in:

$$\{FOO\ \omega\,;FOO\leftarrow BAR;\ BAR\leftarrow FOO\}$$

Function ambivalence

Some implementations of APL allow defined functions to be applied monadically or dyadically, allowing overloading similar to that of APL primitives. A problem then arises in the definitions of these functions; namely, determining the definedness of the left argument. In our notation, the nonstrictness of alternation provides a simple solution. The expression: $\alpha\equiv\alpha$ is true whenever α is defined (and a value error otherwise). Therefore, an ambivalent function can be defined in the form:

$$\{AMBI:\ \alpha\equiv\alpha \rightarrow dyadic\text{-}defn \diamond monadic\text{-}defn\}$$

where *dyadic-defn* and *monadic-defn* are expressions giving the dyadic and monadic definitions of *AMBI*, respectively.

Defined operators

APL2 [IBM84] allows the definition of APL operators, objects similar to the primitives *reduce* and *product*. A defined operator is identical to a defined function, except that in addition to right and left arguments (which must be values), it may also accept one or two *operands*, which may be functions. These objects behave syntactically like APL primitive operators.

In APL2, an operator definition's header contains additional names to stand for the operands. By denoting the left and right operands $\underline{\alpha}$ and $\underline{\omega}$ respectively, our notation may be extended directly to these higher-order objects. The template prefix may also be extended in the APL2 manner. For example, here is a definition of a simple left-associative reduction operator for vectors:

```
{(F LREED)R: SINGLE← RS
          ◇ VECTOR← ((F LREED)¯1↓R)F RS;
                    RS←(⍳0)⍴⊂R;
                    SINGLE←1=⍴,R;
                    VECTOR←1=⍴⍴R}
```

Operators must be of fixed operand valence; the valence of a defined operator is monadic if it contains no reference to the right operand $\underline{\omega}$.

Research directions

Since APL has been traditionally used in an imperative style, it should not be surprising if use of this notation sometimes seems awkward or reveals "shortcomings" in the language. For example, in defining a general inner-product style operator, one is struck by the lack of a first-row or first-column primitive, though these are easily (if cumbersomely) defined using take. The rank operator [Ive83] and APL2's take with axis [IBM84] are better, but not altogether satisfying; rank requires an

[3] This is permitted in functional programs due to the property of *referential transparency*, i.e. expressions have no side effects. See [HM76,FW76].

explicit transpose (which might need to be computed), while take with axis would require similar computation, and is also origin-dependent. Research in functional APL programming is likely to lead to proposals for new or alternative language primitives.

The constructive definition of lists often leads to natural recursive decompositions in LISP. This seems to suggest that nested arrays, which can be defined in a similar inductive manner might lead to more elegant or natural solutions to some problems. On the other hand, this generality is occasionally undesirable, since it can lead to inefficiency when complete generality is not required. This has led some researchers to study inclusion of arrays in functional languages. "Pure APL" as defined here might provide a viable starting point for research in functional array languages.

Formally, functional programs have many attractions. The semantics of a functional language is usually simpler than that of an imperative one, since the notion of "machine state" has been abstracted; a program is given by an expression rather than a sequence of state transitions [Tur81]. Correctness arguments are usually simpler, because they may be stated as equations in the language itself; it is not necessary to reason about state sequences. Useful algebraic laws and formal relationships among programs are more common and easier to exploit in pure functional forms than in equivalent imperative programs. It would be interesting to explore the application of functional programming implementation techniques [Tur79,Hug82] to "Pure APL". Optimization strategies such as [HM76,FW76] might also prove useful.

Summary

A notation for defining pure functional forms of APL computations is defined. The notation is extended to address both theoretical and pragmatic considerations. The notation is shown to support some popular language extensions in a clean, straightforward manner.

The theoretical and practical advantages of pure functional forms are beyond the scope of this paper. However, we do note that the notion of "Pure APL" could lead to some interesting and significant results in language design and implementation.

Acknowledgements

The central ideas of this paper were set down during the summer of 1985 at Yorktown. Thanks to Fritz Rühr and Don Orth, who both provided valuable insights and criticism from the beginning. I would also like to thank Christian Lengauer, my advisor, whose ideas on programming notation and formal semantics were a major influence. Special thanks to my friend and colleague John Gerth; without John's efforts this paper would not appear here.

References

[Bac78] J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):147–174, August 1978.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentiss-Hall, Englewood Cliffs, NJ, 1976.

[FW76] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In *Proceedings Third International Colloquium on Automata Languages and Programming*, 1976. Edinburgh.

[HM76] Peter Henderson and James Morris. A lazy evaluator. In *Proceedings Third Symposium on Principles of Programming Languages*, 1976. Atlanta, GA.

[Hug82] John Hughes. *Graph Reduction with Super-Combinators.* Technical Report PRG-28, Oxford University Computing Laboratory, Programming Research Group, June 1982.

[Hug84] John Hughes. *Why Functional Programming Matters.* Technical Report PMG-40, Institutionen for Informationsbehandling, Chalmers Tekniska Hogskola, Goteborg, Sweden, 1984.

[IBM84] *APL2 Language Manual.* IBM Corporation, 1984.

[Ive76] Kenneth E. Iverson. *Elementary Analysis.* APL Press, 1976.

[Ive83] Kenneth E. Iverson. *Rationalized APL.* Technical Report, I. P. Sharp Associates, Toronto, Ontario, January 1983. I. P. Sharp Research Report No. 1.

[IW81] Kenneth E. Iverson and Peter K. Wooster. A function definition operator. *APL Quote Quad*, 12(1), 1981. Proceedings of APL81 Conference, San Francisco, CA.

[Met80] R. C. Metzger. Extended direct definition of APL functions. In *APL80 International Conference on APL*, North Holland Publishing Co., 1980. Leeuwenhorst.

[Mor77] L. A. Morrow. Nonce functions. January 1977. Unpublished manuscript.

[Red84] D. Hugh Redelmeier. *Towards Practical Functional Programming.* Technical Report CSRG-158, Computer Systems Research Group, University of Toronto, May 1984.

[Tur79] David A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9, 1979.

[Tur81] David. A. Turner. The semantic elegance of applicative languages. In *Proceedings ACM Conference on Functional Programming and Computer Architecture*, October 1981. Portsmouth, NH.

[Tur84] David A. Turner. Combinator reduction machines. In *Proceedings of the International Workshop on High-Level Computer Architecture*, May 1984. Los Angeles, CA.