On the Power of Applicative Languages

R. J. Lipton and L. Snyder

Research Report #94

Department of Computer Science
Yale University
New Haven, Connecticut 06520

This work was supported by the Office of Naval Research under grant N00014-75-C-0752.

Abstract: The expressive power of applicative structures is investigated (particularly APL one-liners) with the result that all "practically" computable functions are "one-liner" expressible. In particular, the class of functions computable by one-liners contains the elementary functions. The problem of reducing the intermediate storage requirements for evaluating applicative structures is shown to be solvable with only modest execution time degradation. The prospect of improving these results is discussed in connection with an outstanding conjecture concerning a time-space relationship.

Of the volumes of research published in recent years on programming language design and compiler construction and optimization, most of it has concentrated on the "common core" shared by the majority of the several hundred extant programming languages. The constituents of this common core are a set of control structures (goto's, while, if-then-else, etc.) and a set of data types (real, integer, boolean, array) and they are often referred to by the collective term Algol-like. These languages have clearly been immensely successful both for their pratical utility as well as for the large body of theoretical work that they have inspired.

We belive, however, that there are a few languages that have the virtue of being sufficiently different from the Algol-like languages that they raise many new and different questions. Such languages include, for example, SETL [1], MADCAP [2], and APL [3]. We plan to study several questions that arise in the applicative structure of a language like APL.*

As with many languages, APL has both advocates and adversaries and since the latter group is probably larger than the former group, we feel compelled to comment on our interest in APL prior to proceeding to the technical material. Although we refuse to take sides in the discussion as to the merits of APL, we address the criticism:

> Since APL is "clearly" the "wrong direction," any results on such a language are uninteresting.

There are several arguments against this view. First, there is already a large and growing APL user community [4]. Any language with such a large

* Although our development will focus on a general applicative structure, we restrict our attention here to APL to simplify the exposition.

following cannot be discussed without some consideration of its special problems. For example, it seems reasonable to study the efficient implementation of this language, both from a practical standpoint as well as for the new issues raised by the language. Efficient implementation of APL is quite different from efficient implementation of Algol. Secondly, topics such as programming style, "structuring," use of goto's, verification, all take on a different emphasis in the context of APL. Thirdly, one would hope that the study of programming languages - as basic as it is to computer science - is broad enough to include language studies not squarely in the "mainstream" on the presumption that they will ultimately enhance our understanding of programming in general.

The first question which should be asked of any language is

Q1: How expressive is the language?

We know from Turing that the answer here - unless qualified - is trivial: all reasonable programming languages are equal. However, in an applicative language such as APL one can make Q1 more precise: how expressive is an APL one-liner, i.e., an APL expression that calls no functions and uses no branching and no execute operators. It is this question that we propose to study first. The reasons that the question is interesting are several:

1. Those who believe that APL forces one into new ways of solving problems consider a one-liner as the basic unit of APL. Thus it is reasonable to consider just what are the ultimate limits to such expressions.

2. The answer, which demonstrates that such expressions, even if greatly restricted, can compute a huge class of functions, is

perhaps surprising to some. We have heard many conversations such as:

"Can you find an APL one-liner for ....."

We now can answer all (well, almost all) such questions affirmatively.

3. While our results demonstrate the great expressive power of one-liners, they do more to suggest an interesting question about the power of applicative languages and the power of more classic Algol-like languages. We are able, we believe, for the first time to make precise the intuitive feeling to many that APL is inherently slower than Algol.

We now will state precisely our main result that addresses Q1. We assume the usualy semantics and definitions of the APL operators [5].

<u>Theorem 1</u>: Let $f : A_k \to A_k$ be any function from objects of rank k to objects of rank k ($k \geq 0$) that is computable by a random access computer in time $t(n)$ (n = length of the input in some standard encoding). If $t(n) \leq 2^{cn}$ (c a constant), then there is a one-liner E with one variable v such that for all $v \in A_k$,

$f(v)$ is equal to E.

This theorem can be extended immediately to include even larger functions $t(n)$ and functions from many ranks to many ranks. Indeed, if f is any elementary function [6] there is an E for computing it. Moreover, the one-liners need only the operators

+	addition
-	subtraction
=	equality
∧ ∨ ~	logicals
|	residue
ρ	reshape
/	reduction

The chief difficulty with this theorem is that it is still open whether or not the cost of evaluating such an expression as *E* can be done in time $t(n)$ or even, say, $t^2(n)$. By time, here, we mean the cost of evaluating *E* by any of the standard methods of evaluating APL [7]. A more general question is:

> can one add a set of new operators to APL so that Theorem 1 can be improved to where *E* can be evaluated in time, say, $t^2(n)$?

Clearly one must restrict the operators to be "reasonable" in that they do not include, for example, an execute command. Perhaps one can show that no applicative language can be as powerful in this sense as an Algol-like language.

The second question we wish to study is:

> Q2: How can one efficiently implement an applicative language such as APL?

The question here is what can one do about space; in particular, how can temporary storage be reduced? APL and other applicative languages abound with examples of statements that yield small answers (i.e., scalars) from small inputs (i.e., scalars) and yet produce during their executions huge temporaries. A commonly cited example is the expression

$$2 = {}^{-}1 \uparrow + \not{/}\ 0 = \underline{(\iota N)\circ.|\ \iota N}$$

which decides whether or not the integer N is prime. The underscored portion of the expression produces an $N \times N$ array although the input and output are both scalars. Expressions of this type cannot be executed on standard implementations not because they would run too long but rather because they use too much space. Thus, Q2 is really the following question:

> can one evaluate APL expressions in such a way that no large temporaries are created and so that the execution time (over the usual evaluation) is not increased dramatically.

The answer is yes. This is, perhaps, surprising to some, and is made more interesting when it is realized that our methods appear to be practical.

We will now briefly sketch our method of implementation of an arbitrary one-liner E. The method used differs from the method of [7] and [8]. Let E be a one-liner with one free variable v (the generalization to several is easy). Thus E could be

$$+ / A + \phi\ A \leftarrow V.$$

We then show how to inductively construct a module M with the following property:

> for any index i into the position of the answer to E

> *M* will return the value; *M* has the ability to request in <u>any order</u> the value of the j^{th} position of the input V.

The key requirement here is that we enable *M* to access the elements of V in any order. The price we pay, of course, is that *M* must be able to supply similar information, i.e., *M* must be able to supply the i^{th} position to the answer of *E*. This method should be compared with the stream methods of [8]. The insight is that by allowing arbitrary requests, modules such as *M* can be inductively constructed for any APL one-liner. Indeed, the <u>only</u> temporary storage needed in any such module is just the storage needed to retain control information (which is independent of size of inputs) and the storage needed to keep pointers into the structures being handled. More precisely,

<u>Theorem 2</u>: Let *E* be a one-liner with input V. Then *E* can be evaluated in space bounded by the space required for

1. storing the inputs and the answer, and
2. storing a bounded number of pointers of at most log n bits where no temporary structure is created that has size larger than n.

Moreover, the time to evaluate *E* by this method is at most $0(t^2)$ for any one-liner where t is the time required to evaluate *E* by the usual methods and is at most 0(t) when *E* has no occurrence of either dyadic ι nor scan.

Theorem 2 suggests an interesting reason why it remains open whether or not Theorem 1 can be improved to time $t^2(n)$. Such a simulation would yield a positive result to the conjecture of Cook [9]

> Is $O((\log n)^2)$ space sufficient to recognize all languages recognizable in deterministic polynomial time?

and it is thus unlikely. Nevertheless the expressive power of these one-liners is extensive.

REFERENCES

1 K. Kennedy and J. Schwartz
"An Introduction to the Set-Theoretic Language, SETL," in *Computer and Mathematics with Applications*, Vol. 1, Pergamon Press, 1975.

2 M. Wells and C. Cornwell
"A data type encapsulation scheme utilizing base language operators," *Proceedings of ACM Conference on Data: Abstraction, Definition and Structure*, 1976.

3 K. Iverson
A Programming Language.

4 C. Wogrin
Preface to *APL 76: Conference Proceedings*, 1976.

5 L. Gilman and A. Rose
APL, an Interactive Approach, Second Edition, John Wiley, 1974.

6 A. Grzegorczyk
"Some classes of recursive functions," *Rozprawy Mathematyczne*, Polish National Academy, 1953.

7 P. Abrams
"An APL Machine," *Stanford Linear Accelerator Report*, 1969.

8 T. Miller
An APL Compiler, Ph.D. thesis, (in preparation).

9 S. Cook and R. Sethi
"Storage Requirements for Deterministic Polynomial Time Recognizable Languages," *STOC*, 1974 (Seattle).