APL AND THE GRZEGORCZYK HIERARCHY

by

Seth Breidbart*

TR 79-373

Computer Science Department
Cornell University
Ithaca, New York 14853

*Preparation of this paper was supported by NSF grant DCR 75-09433.

APL AND THE GRZEGORCZYK HIERARCHY

Seth Breidbart*

Computer Science Department
Cornell University
Upson Hall
Ithaca, N.Y. 14853

Abstract

We show in this paper that the set of "traditional" APL 1-liners (using arithmetic functions only) compute precisely the set of functions in the class E^4 of the Grzegorczyk hierarchy (the class immediately above the elementary functions). We also show that if we extend the set of 1-liners to include either the "execute" operator, or 1 line programs with gotos, then any partial recursive function can be computed.

* Preparation of this paper was supported by NSF grant DCR 75-09433.

0. Introduction

APL is an array-oriented computer language. Its basic functions operate on rectangular arrays; arithmetic is done element by element, and there are functions that (for example) sum the elements of an array along a given coordinate. For a more complete description see reference [3].

Since the Grzegorczyk hierarchy is a hierarchy of functions on non-negative integers, and APL functions deal with rectangular arrays, we must give a mapping between them. We assume that APL arrays contain only non-negative integers as elements. We assume an elementary mapping between arrays and integers, for example the Goedel mapping (map the array A to the vector (ρρA),(ρA),,A , then use a prime encoding). Note that such a mapping is computable both by an APL 1-liner and by an elementary function. We will identify the integer function f with the APL function F if f(n) is a representation of F(N) whenever n is a representation of N.

We define a class of functions C to be "time-closed" if, whenever f is a function in C and T is a Turing machine that runs in time f, the function computed by T is a member of C.

We define a class of functions C to be "time-honest" if, whenever f is a function in C there is a Turing machine T that computes f and whose running time is also a function in C.

The class E4 of the Grzegorczyk hierarchy is the set of functions obtainable from the constant functions, the successor function, the projection functions, and the function t(n) = */nρ2 by the operations of explicit transformation (substitution) and bounded primitive recursion.

We say that f is obtainable from g, h, and m by bounded primitive recursion if f is defined by

f(0,x1,...,xn) = h(x1,...,xn),
f(y+1,x1,...,xn) = g(y,x1,...,xn,f(y,x1,...,xn)),
and f < m.

The primitive recursive functions can be defined as the set of functions that can be programmed in a language that includes the statements x = 0,

x = x+1
if x=0 then ... else ...
and
Repeat x ... end

where statements must be well nested and Repeat x S end causes S to be executed the number of times specified by the value of x upon entrance to the statement; thus Repeat x x=x+1 end doubles the value of x.

The class E4 is just the set of functions that can be computed by programs in this language with nesting of loops limited to a depth of 3. Other classes of the Grzegorczyk hierarchy can be characterized similarly.

1. Main Result

Theorem 1. All functions computable by APL 1-liners are in E4.

Proof. E4 is closed under composition and contains each single APL arithmetic function.

Theorem 2. All functions in E4 are computable by APL 1-liners.

This is shown by a sequence of lemmas.

Lemma 1. E4 is time-honest.

Let P be a loop program with depth of loop nesting at most 3. Let z be a variable not in P. Modify P to a program P' that has z=0 as its first statement, and z=z+1 following each executable statement of P. Let z be the output variable of P'. Then P' computes the running time of P.

Lemma 2. The set of functions computable by APL 1-liners is time-closed.

To compute T(n), given that T is a Turing machine that runs for no more than f(n) time where f(n) is computable by an APL 1-liner, compose the following steps.

Compute f(n).

Generate all strings of length f(n)*2 in an array. These are the candidates for the computation of T.

Test each row of the array to see if it is a possible computation of T starting on input n. There will be precisely one such row.

Extract that row and return the value of the worktape of T in the last instantaneous description described there.

Lemma 3. If f is in E4 then there is a function g, computable by an APL 1-liner, that bounds f.

This is shown by induction on the length of the definition of f as a member of E4. If f is a basis function then obvious. If f is the composition of g and h, g is bounded by g', and h is bounded by h', then let g"(n) be the bounded sum of g'(i) for i between 1 and n. Then f(n) is bounded by g"(h'(n)+n), which is computable by an APL 1-liner. If f is formed by bounded primitive recursion from g, h, and m then f is bounded by m.

Proof of Theorem 2. Let f be a function in E4. Then there is a g in E4 that is a running time for f by Lemma 2. There is a function h, computable by an APL 1-liner, that bounds g, by Lemma 3. Then by Lemma 1, f is computable by an APL 1-liner.

Corollary. The set of functions computable by APL 1-liners is precisely the set of functions in the Grzegorczyk class E4.

2. Other Grzegorczyk Classes

A subset of the language, which does not include exponentiation, was shown by Lipton and Snyder [5] to include all elementary functions. The obvious analogue of Theorem 1 applies, so that their bound is tight for the subset of APL that they consider. (Please note that this work was performed independently of theirs.)

If APL is extended to include the function t(a,b) = */aρb as a primitive function, then APL 1-liners compute all of the functions in the Grzegorczyk class E5. In general, if APL includes as a primitive a function that defines the class Ek, then APL 1-liners can compute all functions in the class Ek+1. The extra class is caused by the ability to iterate a function by use of function/ ; in the unextended APL, note that multiplication defines the elementary functions (E3).

3. Ackermann Rating

Brown, in [1], proposes assigning to each programming language a rating j which is the largest integer such that the language contains a closed nonrecursive expression for j A n where A is the Ackermann function. He shows that APL without execute has an Ackermann rating of at least 4, and that APL with execute has an Ackermann rating of at least 5. Linden, in [4], demonstrates that APL with the execute operator has an Ackermann rating of infinity.

Theorem 1 shows that APL without execute has an Ackermann rating of precisely 4. The discussion of one-liners with execute below provides an independent proof that APL with execute has an Ackermann rating of infinity; however, both my construction and that in [4] use the execute operator on strings containing the execute operator, and hence might be considered recursive expressions. I conjecture that if argument strings to execute are not allowed to contain further executes, then APL with execute has a finite Ackermann rating. Brown shows that this rating is at least 5.

4. One-Line Functions (with goto)

If we consider the set of APL functions that have precisely one line, but are allowed to use the goto operator, then we find that for any partial recursive function there is a program in this set that computes it.

Let T be any Turing machine. There is an APL 1-liner that takes as argument an instantaneous description of T and returns as result the following instantaneous description, by Theorem 2 above. In the same 1-liner, we can also set a variable (the output variable of the function) to the contents of the output tape of T (the instantaneous description is stored in a local variable). We can also test to see whether the instantaneous description is such that T has halted; if so, the 1-liner evaluates to 0, if not, to 1. A goto then causes the function to compute the same function as T, since it simulates T until T halts.

5. <u>One-Liners With "Execute"</u>

If we allow the "execute" operator (that takes a character string as argument and evaluates it) as part of a 1-liner, then we also have the capability to compute any partial recursive function. The proof is theoretically interesting, in that we use the recursion theorem to show the existence of such a 1-liner.

We first need a function c that takes as argument any APL function, and returns as output a 1-liner that simulates that function for the first 2*2*n steps on input n. (The choice of 2*2*n is arbitrary.) Steps is here defined as the number of APL statements, so that if the argument to c is already a 1-liner, the output can be the same 1-liner. Such a function c exists, by an argument similar to the proof of Lemma 2 above. We do not require that c be a 1-liner, only that it be a total recursive function.

Now construct a function that operates as follows: Given as argument a function F, it creates a 1-liner that takes as argument an instantaneous description I, and runs T on I for one step. If T halts, it returns the output tape of I. If not, it substitutes the new value of I into F wherever F references its argument and evaluates 'execute''c(F)'''.

By the recursion theorem, there is a function that is a fixed point under the operation described in the paragraph above; that is, when the operator is given the fixed point P as an argument, it produces a program functionally equivalent to P.

P simulates T until T halts: after each step, if T has halted, it returns the proper value. If T hasn't halted, it constructs a copy of (a function equivalent in effect to) itself, and executes that. Thus, for any Turing machine T there is a 1-liner with execute P that computes the same function.

References

1. J. A. Brown. Using the Ackermann Function to Rate Programming Languages, APL Quote-Quad, II (1970), pp. 35-36.

2. A. Grzegorczyk. Some classes of recursive functions, Rozprawy Mathematyczne, Polish National Academy, 1953.

3. K. Iverson. A Programming Language. Wiley, 1962.

4. T. A. Linden. The Ackermann Function in APL, APL Quote-Quad, II (1971), pp. 130-131.

5. R. J. Lipton and L. Snyder. On the power of applicative languages, in Formal Descriptions of Programming Concepts, E.J. Neuhold (ed.) North-Holland publishing Company, 1978.

6. A. R. Meyer and D. M. Ritchie. The Complexity of Loop Programs, in Proceedings of the 22nd National Conference, Association for Computing Machinery, 1967.

7. H. Rogers, Jr. Theory of Recursive Functions and Effective Computability. McGraw-Hill 1967.