EQUIVALENCE OF SOME CLASSES OF ALGORITHMS

By: Marvin C. Paul

WORK DONE UNDER:

Contract No. AF 1863

Air Force Office of Scientific Research
Directorate of Mathematical and
Information Sciences
Arlington, Virginia 22209

June, 1971

DCS Technical Report #7

Department of Computer Science
Livingston College
Rutgers University
Equivalence of Some Classes of Algorithms

Introduction and Outline

Our objective is to find large classes of equivalent programs which can be simply characterized so that it is possible to choose from amongst them the program which is optimum by some reasonable criterion. We would like to do this for as high a level of programming language as possible. In this paper, we will consider some easily characterized classes of equivalent programs and some techniques for optimizing over such classes of equivalent programs. In general when we speak of two descriptions of an algorithm being equivalent, we mean that they both produce the same output for the same inputs. The programs we will consider will consist, at first, of ordered sequences of assignment statements of the form:

\[ X = f(X_1, \ldots, X_n) \]

Such a language is not the same as any real programming language. It is more like a flow chart language. It is chosen for simplicity. It will also be seen that for certain kinds of optimizing of more realistic programming languages, we can optimize in the above language and that optimization will carry over when we translate to a more realistic programming language. The classes of equivalent programs we will consider, will each be characterized by an array of "black boxes". Each program in such a class will have the property that it has one assignment statement for each box in the characterizing array of a form which is directly related to that block and each program is equivalent to the characterizing array in that it will produce the same output as the array when presented with the same input. We will call such programs 1-equivalent to their characterizing array. The following example illustrates this equivalence.
Example 1:

\[
\text{Array-A}_1
\]

\[
\text{Array-A}_2
\]

Some 1-equivalent programs to \(A_1\) (all must have five statements corresponding to the five boxes of \(A_1\)).

\[
P_1(A_1): \begin{cases} X_1 = G(I_1, I_2) \\ X_2 = H(I_3) \\ X_1 = F(X_1, X_3) \\ X_2 = G(I_1, I_2) \\ 0 = F(X_1, X_2) \end{cases}
\]

\[
P_2(A_1): \begin{cases} X_1 = H(I_3) \\ X_2 = G(I_1, I_2) \\ X_2 = F(X_1, X_2) \\ X_2 = G(I_1, I_2) \\ 0 = F(X_1, X_2) \end{cases}
\]

Some programs 1-equivalent to \(A_2\):

\[
P_1(A_2): \begin{cases} X_1 = G(I_1, I_2) \\ X_2 = H(I_3) \\ X_3 = F(X_1, X_2) \\ 0 = F(X_1, X_3) \end{cases}
\]

\[
P_2(A_2): \begin{cases} X_1 = H(I_3) \\ X_2 = G(I_1, I_2) \\ X_1 = F(X_1, X_2) \\ 0 = F(X_1, X_2) \end{cases}
\]

Note that arrays \(A_1\) and \(A_2\) are equivalent and so \(P_3(A_1)\) is equivalent to \(P_3(A_2)\), but \(P_3(A_1)\) is not 1-equivalent to \(P_3(A_2)\).
We note that the set of programs 1-equivalent to an array $A_1$ are not all programs equivalent to $A_1$ - but only those equivalent in a very simple way. Since $A_2$ is clearly equivalent to $A_1$, all programs 1-equivalent to $A_2$ are equivalent to $A_1$ even though none is 1-equivalent. So we will not generally consider all programs equivalent to a given array, but only a restricted class of such programs, and this restricted class will be characterized by that array. Two programs which are 1-equivalent to the same array will contain the same number of assignment statements. They can only differ in the order of these assignment statements and the naming of variables. 1-equivalence then does not depend in any way on the properties of the functions in the boxes or even on their identity.

We are interested here in arrays which can be described with a finite number of blocks. Our interest will also center for the most part on arrays which have no loops. For such an array we can define a class of 1-equivalent programs. We can then consider how different members of a class of programs 1-equivalent to a given array differ from each other. It turns out that there are significant differences among such programs which affect the efficiency with which they may be implemented. The work discussed thus far follows not only the spirit but also has considerable overlap with the contents of [1] and [2].

In [1] and [2], most of the questions of efficiency which we will discuss have been considered. Our approach differs mainly in that we try to describe approaches for finding techniques for minimizing various quantities rather than giving such techniques for particular situations. We have also considered some minimization over non-trees. The theoretical development of equivalence classes we consider has been developed many times in many places. Because of the press of time in preparing this report and the fact that much of this material has been independently explored elsewhere, we will not consider it in detail here.
We will, however, consider generalizations of this approach.

One direction in which one can generalize the considerations of equivalent programs, is to introduce arrays with loops in conjunction with some notion of time in the array and consider sets of programs equivalent to such arrays. In this approach, however, we would have to find an equivalence considerably different than 1-equivalence and surely assignment statements would not be adequate. To hold on to a basically simple kind of equivalence, and programs consisting basically only of assignment statements, we have then another approach.

Following the consideration of 1-equivalence classes characterized by a simple array, we will consider sets of arrays, and some of the equivalent sets of programs which are equivalent to such a set of arrays. We will consider sets of arrays in which each member of the set is a finite array which is defined by a set of d-integers, where d is called the dimension of the set of arrays. The kind of array-sets we will define are pictured as being sub-arrays of an infinite array having an identical set of blocks at each point in a d-dimensional space. We are basically interested in such arrays which do not have loops. However, within the framework we will allow for defining such array-sets, it is possible to define arrays with loops. We will give a test for determining if there are any loops in the defined array. If there are no loops, the array is equivalent to a recursive function definition which can be derived directly from the description of the array.

Corresponding to each array-set with no loops, we can in some cases find a class of program-sets. The kinds of program-sets which correspond to an array-set will be describable by a number of indexed assignment statements of the form:

\[ X(i_1, i_2) = F(Z(i_1 + 1, i_2 - 1), X(i_1 - 1, i_2)) \]

and directions as to the range of values that the indices should go through, as well as the order in which the indices should go through these values. (This
is analogous to a nested DO of assignment statements). To one array-set there may be a number of program-sets which are equivalent. We will investigate those program sets which are equivalent to an array-set which differ from one another only in the order or sequence in which their indices run through their values. This is a direct generalization of the fact that two programs can be 1-equivalent to each other if they differ only in the ordering of their assignment statements. We will also consider conditions under which variable names may be altered and still have the programs equivalent. The significant alterations here involve removal of indices. In connection with these possible alterations we will consider some questions of optimization. See example 2.
Example 2:

An Array-Set AS-1

\[
\begin{align*}
\{ & I \quad F(0) \quad F \quad F(1) \quad 0 \\
& I \quad F(0) \quad F \quad F(2) \quad F(2) \quad 0 \\
& I \quad F \quad F \quad \ldots \quad F \quad 0
\end{align*}
\]

Equivalent Recursive Definition:

\[
\begin{align*}
F(i) &= F(F(i - 1)) \\
F(0) &= I
\end{align*}
\]

Equivalent Program

\[
\begin{align*}
F(0) \\
\text{DO } I,N \text{ ( } F(1) = F(F(i)) \text{ )}
\end{align*}
\]

Note here \( F(i) \) is thought of as an indexed variable, whereas in \( F(F(i)) \) the first \( F \) is the function performed by the box \( F \).
Defining Array-Sets:

Here we will consider sets of arrays, but not all such sets. To restrict the sets of arrays to be considered, we will define a restricted system in which arrays are to be described. We call such a system an array-grammar. In fact, we will define a number of kinds of array-grammars. These will differ from each other in the amount of detail with which one can define an array-set using them.

An unrestricted array-grammar consists of:

1. (a) An integer \( d \) called the array dimension.

   (b) The set of all integer vectors of \( d \) dimension; \( V \).

2. A finite set of integer vector transformations, \( T \), such that if \( t \in T, v \in V \), then \( t(v) \in V \). (We will obtain a variety of grammar types as we restrict the nature of the transformation.)

3. A finite set of blocks \( B \) (these are the identities of the blocks located at each point in the array.)

4. A finite set of block functions or labels, \( L \) (these are the set of functions performed by the blocks).

5. A function from \( B \) to \( L \) which assigns a unique label in \( L \) to each block in \( B \); i.e. if \( b \in B \), \( \ell(b) \) is \( b \)'s label.

6. A connection function \( C \) which assigns to each block \( b \in B \), a unique \( d \) set of pairs (connections) \( \{(p_1,t_1),(p_2,t_2), \ldots, (p_n,t_n)\} \) with \( p_i \in B \), and \( t_i \in T \).

The number of connections to two blocks having the same label should be the same. Each label is assumed to have one output so if the same block provides inputs to more than one block, it comes from the same output. The interpretation of this function is that it describes the input connections to each block. If \( C(b) \) includes \( (p,t) \) it means that the block \( b \) at the position given
by a vector \( v \) has an input from the block \( p \) at position \( t(v) \).

We will illustrate the use of the use of the grammar to describe an array using a few examples. Note that:

**Grammar 1**

1. \( d = 1 \)
2. \( T = \{t_1, t_2\} \) where \( t_1(i) = i - 1, t_2(i) = i - 2 \)
3. \( B = \{1, 2, 3\} \)
4. \( L = \{F, G\} \)
5. \( l(b_1) = F, l(b_2) = G, l(b_3) = F \)
6. \( C(b_1) = \{(1, t_1)\} \)
   \( C(b_2) = \{(1, t_2), (2, t_1)\} \)

An alternate pictorial representation of Grammar 1 is:

```
  F
 / \
1--- 1-2
 \
  G
 / \
2--- i-1
 \
  F
 / \
3--- i-3
```
The purpose of an array-grammar is to define a set of arrays. Given an array-grammar \( G \) with dimension \( d \) and any \( n \)-dimensional positive integer vector, \( \mathbf{v} = [n_1, \ldots, n_d] \), called the boundaries of the array, the array generated by \( G \) is defined as:

The set of blocks \( B \) appears at every positive integer (\( >1 \)) point given by a vector \( \mathbf{v} = [v_1, \ldots, v_n] \) in which \( 0 \leq v_i \leq n_i \). These blocks are said to be within the array, or within the boundaries. The connections then, are given by the connection function. The connection function may require that an input to a block in the array (at one of the allowed points) come from outside the array (say from a point given by a vector one of whose components is \( 0 \) or negative or whose \( i \)th component is \( >1 \)). All such inputs from points outside the array are considered to be array inputs. All block outputs will be considered array outputs when not otherwise specified.

So for any positive \( d \)-dimensional vector the grammar defines an array. For example, the array generated by \( G_1 \) above for the vector 3 is pictured below:
The above example illustrates how we will designate array inputs. As another example, consider the 2-dimensional grammar:

(1) $d = 2$

(2) $T = \{t_1, t_2\}$ where $t_1([i, j]) = [i - 1, j]$

$t_2([i, j]) = [i, j - 1]$

(3) $B = \{1, 2\}$

(4) $L = \{F, G\}$

(5) $G(1) = F_1$, $G(2) = G$

(6)
The array generated by $[2,3]$ would be:

The examples we have shown are for very simple vector transformations. These vector transformations have the property that if we think of our array as arranged in a d-dimensional space with a set of blocks $B$ being located at each integer component point, the branches representing connections can be represented as vectors with integer components, and if such a vector terminates on some integer point, the same vector terminates on all integer points in the array. Or in terms of part(2) of the grammar specifications, each vector transformation $t \in T$ has the property that if $v$ is a vector $t(v) = v + q$ where $q$ is a constant vector each of whose components is a positive or negative integer.
Grammars restricted to having such transformations are called constant interval grammars and the arrays are constant interval arrays. In the next section, we will consider a test to determine if the array-set specified by an array-grammar has any loops in its connections. If the grammar is a constant interval grammar, that question is decideable.

Recursive Array Representation:

There are a number of ways one may represent an array-grammar. We can just write down all the six parts of the grammar specification or we may condense some of the information in a picture or some other way. We will give an alternate representation now which also has an alternate interpretation. The recursive representation condenses parts 1 through 6 into a list of assignment statements. These are obtained from the grammar as follows:

Corresponding to each connection function for each block \( j \in P \) in an array grammar \( G \) of the form:

\[
C(b) = \{(k_1, i_1), (k_2, i_2), \ldots, (k_i, i_1)\}
\]

and letting \( Z = 1(j), I = i_1, i_2, \ldots, i_d \), \( W^1 = 1(k_1), t_1 \in T \),
we define a corresponding indexed assignment statement:

\[
z_j(I) = z_j[ w_{k_1}^1 (t_1(I)), w_{k_2}^2 (t_2(I)), \ldots, w_{k_n}^n (t_n(I)) ]
\]

and call \( z_j(I) \) and \( w_{k_j}^j (t_j(I)) \) indexed variables.

The list of all such indexed assignments statements from the given array grammar is called the body of the recursive representation of the array grammar in question.
The two example grammars we have given may be alternately represented as:

\[
\begin{align*}
G_1 & \quad F_i(i) = F_1(F_1(i-2)) & \quad F_i(i,j) = F_1(F_1(i-1,j), G_2(i,j-1)) \\
G_2 & \quad G_2(i,j) = G_2(F_1(i-2), G_2(i-1)) & \quad G_2(i,j) = G_2(F_1(i-1,j), G_2(i,j-1))
\end{align*}
\]

To complete these recursive representations, we need terminal conditions. These correspond to the boundary conditions of the array grammar in the obvious way. In general, \(Z_j(I) = \text{INPUT}(Z_j(I))\) when any component of \(I\) is outside the boundaries. In our examples, we need:

\[
\begin{align*}
G_1 & \quad F_1(0) = \text{INPUT}(F_1(0)) & \quad F(i,0) = \text{INPUT}(G(i,0)) \text{ for } i \geq 1 \\
F_1(-1) = \text{INPUT}(F_1(-1)) & \quad F(0,1) = \text{INPUT}(F(0,1)) \text{ for } i \geq 1 \\
G_2(0) = \text{INPUT}(G_2(0)) & \quad \\
F_3(0) = \text{INPUT}(F_3(0)) & \quad 
\end{align*}
\]

Given an array-grammar, then we can always get a corresponding recursive array representation which contains all the information in the array-grammar. We can independently define a recursive array representation essentially as a finite set of indexed assignment statements, and show that each corresponds to an array-grammar. We started by considering a recursive array representation as an alternate way of representing an array-grammar. We might also consider the recursive array representation as the recursive definition of a function and look at the array-grammar as an alternate representation of such a definition. In trying to view the recursive array representation as a definition, the question arises as to its legitimacy. Does it in fact give a unique value to each of the indexed variables in its
range for all positive integer values given to the indices?

To better define what we require, we give a method - the same kind often used in compilers which accept recursive definitions - to attempt to evaluate a recursive representation like the one we are considering. To evaluate such a set of equations in the boundaries \([n_1, n_2, \ldots, n_m]\) one substitutes the vector \([n_1, n_2, \ldots, n_m]\) for \(I\) and computes the vectors \(t(I), t \in T\). These values are then substituted for \(I\), and \(t(I)\) respectively in all the assignments of the recursive representation. If as a result of this evaluation we still have indexed variables on the right of the equation in which the value of the indices are still inside the boundaries; say \(W(J)\) with \(J\) still inside the boundary, then we substitute for \(W(J)\), the left side of the recursive assignment which results when \(J\) is substituted for \(I\) in the assignment in which \(W(I)\) is on the left. As long as there are indexed variables on the right of the resultant equations with their indices still within the boundaries, substitutions continue. When all the indexed variables on the right have indices outside the boundaries, then we have a set of equations which can be evaluated - since the values for the variables outside the boundaries are presumed to be known inputs, as are the functions which appear in the equations.

If this evaluation procedure can in fact be carried out to completion in a finite number of steps under the assumption that the functions and variable values outside the boundaries are known we call the recursive representation legitimate. The only eventuality that would prevent the evaluation from being carried out is if for some reason the substitution described above would never stop. It is easy to see that this in turn could only happen if the array represented by the recursive representation had a loop in it. In that case, the substitutions would result in
range for all positive integer values given to the indices?

To better define what we require, we give a method - the same kind often used in compilers which accept recursive definitions - to attempt to evaluate a recursive representation like the one we are considering.

To evaluate such a set of equations in the boundaries \([n_1, n_2, \ldots, n_m]\)

one substitutes the vector \([n_1, n_2, \ldots, n_m]\) for \(I\) and computes the vectors

\(t(I), t \in T\). These values are then substituted for \(I\), and \(t(I)\) respectively in all the assignments of the recursive representation. If as a result of this evaluation we still have indexed variables on the right of the equation in which the value of the indices are still inside the boundaries; say \((W(J)\)

with \(J\) still inside the boundary, then we substitute for \(W(J)\), the left side of the recursive assignment which results when \(J\) is substituted for \(I\) in the assignment in which \(W(I)\) is on the left. As long as there are indexed variables on the right of the resultant equations with their indices still within the boundaries, substitutions continue. When all the indexed variables on the right have indices outside the boundaries, then we have a set of equations which can be evaluated - since the values for the variables outside the boundaries are presumed to be known inputs, as are the functions which appear in the equations.

If this evaluation procedure can in fact be carried out to completion in a finite number of steps under the assumption that the functions and variable values outside the boundaries are known we call the recursive representation legitimate. The only eventuality that would prevent the evaluation from being carried out is if for some reason the substitution described above would never stop. It is easy to see that this in turn could only happen if the array represented by the recursive representation had a loop in it.

In that case, the substitutions would result in
a variable with the same index value appearing on the left periodically during evaluation. On the other hand, if there were no loop then this will not happen and eventually all index values on the right will be outside the boundaries.

An array having a loop might actually produce a stable output so we do not say an array having loops is not legitimate, but only that the corresponding recursive definition is not legitimate. This seems reasonable because the legitimacy of the recursive definition is related to a somewhat different view of evaluation than for the array.

To test for legitimacy of a recursive representation of an array then, requires a search for a loop in the array. This can be done if we have sufficient knowledge of the integer vector transformations \( t \in T \). We think of the array consisting of sets of blocks, each set consisting of the same blocks. At each integer point inside the boundaries of the array, there will be one of those sets of blocks. Now if we assume that the array grammar is a constant interval array grammar, i.e. for each \( t \in T \), there is a constant integer vector \( c \) such that \( t(1) = I + c \). This is true if for example \( I = [i_1, i_2], \forall I = [I+1, I-5] \), because then \( I = [i+1, i-5] + [-1, 5] \).

Under our assumption then, the inputs to blocks of the array can be represented by sets of constant vectors, i.e. the same set of vectors enters the same block \( b \in B \) at each point within the boundaries of the array. In such an array, a loop would consist of a set of constant vectors whose sum was zero.

If \( v_1, v_2, \ldots, v_n \) are all the constant vectors that enter some block of the array, then a necessary condition that there be a loop in the array is that there is a set of integers \( x_1, x_2, \ldots, x_n \) for which:

\[
(1) \quad x_1 v_1 + x_2 v_2 + \ldots + x_n v_n = 0
\]
The existence of such a solution insures that there is a path from some block at point $p$ to some other block at point $p$. We need to insure that the same block at point $p$ is involved in the loop. To set up this other constraint we note that we can describe the possible paths by a 'state diagram', in which we have nodes for each block $b_1, b_2, \ldots, b_n$ and we have a branch from $b_i$ to $b_j$ labelled $v_k$ if the vector $v_k$ terminates on block $b_j$ and originates on block $b_i$. Then for any $j$, any path between $b_j$ and $b_j$ can be traced on this diagram. In fact, it can be shown that all such paths can be given as 'regular expressions' in $v_1, v_2, \ldots, v_k$. Furthermore, by specialization of Parikh's Theorem\(^1\) the number of occurrences of $v_1, v_2, \ldots, v_n$ on a path from $b_j$ to $b_j$ can be given as the solution to one of a finite number of sets of linear equations. Let these be $S_1(j), S_2(j), \ldots, S_n(j)$ written in the unknowns $s_1, x_2, \ldots, x_n$. If then we can find a solution with at least one $x_i \neq 0$ to the sets of equations consisting of (1) and $S_k(j)$ for any consideration of $j$ and $k_j \leq n_j$, then there is a loop - otherwise there is no loop. Of course this approach is more general than is often needed since if for example (1) has no solution that is sufficient to insure the absence of loops, or if the state diagram has no loop from $b_j$ to $b_j$, that guarantees no loop from $b_j$ to $b_j$.

Furthermore, in many instances one can establish the absence of loops even if the array grammar is not a constant interval grammar. In the next section, we will consider nested DO's equivalent to an array. If there exists a legitimate nested DO loop for an array, then it follows that the recursive representation is also legitimate, i.e. the array has no loops. On the other hand, using the definitions of the various terms given here, there are arrays whose recursive representation is legitimate and which have no legitimate nested DO-representation. This subject will be expounded
further in the next section.

Nested DO Representation

Another possible representation of an array grammar is the nested-DO-representation. A nested-DO-representation for an array grammar G consists of the recursive representation together with two kinds of orderings:
(1) an ordering on the components of the d-dimensional vector and
(2) an ordering on the values of each of these components.

The information about the array is completely contained in the recursive representation - the permutation is superfluous for the purpose of representing the array. The permutation, however, will be interpreted as part of the prescription for evaluating a DO-representation. It is clear that for every DO-representation there is an array-grammar, and vice-versa. But now we wish to view the DO-representation independently. We will view it as a nested DO computer program. Again, as with the recursive representation, the question of legitimacy will arise. To study that question further, we must define how a DO-representation viewed as a nested DO program is to be evaluated. Consider a DO-representation of our example 2. It consists as we said of the recursive representation, together with orderings on the vector components and their values. The recursive representation without unessential subscripts is:

\[ F(i,j) = F_1(F(i-1,j),G(i,j-1)) \]
\[ G(i,j) = G(F(i-1,j),G(i,j-1)) \]

with inputs defined as:
\[ G(i,0) \text{ = INPUT}(G(i,0)) \]
and let the ordering be:

\[
\text{ordering} = \begin{cases} 
\text{the first component first} \\
\text{the second component second} \\
\text{each component is to range through}
\end{cases}
\]

\[
\text{ordering}_2 = \begin{cases} 
\text{its values in increasing order, i.e.} \\
\text{the order will be 1, 2, 3, ...}
\end{cases}
\]

Now the evaluation of this nested DO-representation is defined to occur as follows given that the boundaries are given by \([n_1, n_2]\): set \(i=1, j=1\) and evaluate \(F(1,1)\) and \(G(1,1)\) by substituting these values on the right of each assignment in the recursive representation.

That is:

\[
F(1,1) = F(F(0,1), G(1,0)) \\
G(1,1) = G(F(0,1), G(1,0))
\]

Since all the arguments on the right are defined as inputs, eg. \(F(0,1)\) in an input, this step is considered legitimate. Next set \(i=2, j=1\), the evaluation this time is of:

\[
F(2,1) = F(F(1,1), G(2,1)) \\
G(2,1) = G(F(1,1), G(2,1))
\]

and in this case all right side arguments have been obtained in a previous step of evaluation so again the evaluation step is considered legitimate.

Similarly we continue through giving the vector \([i, j]\) the succesive values:

\[
... [3,1], [4,1], ... , [n_1, 1], [1,2], [2,2], ... , [n_1, 2], ... , [1, n_2], [2, n_2], ... , [n_1, n_2]
\]
At each step the evaluation is legitimate if and only if the right side arguments are either defined as inputs or have been previously computed.

Thus we may view the evaluation as being done in the same way as would the program:

```plaintext
DO i = 1 to n1 by 1
DO j = 1 to n2 by 1
F(i,j) = F(F(i-1,j),G(i,j-1))
G(i,j) = F(F(i-1,j),G(i,j-1))
END
END
```

If instead of the above ordering, we had had the component ordering:

```
{ 
  second component first  
  first component second 
}
```

with the ordering of the values of the components unchanged, the evaluation done would be equivalent to the above program with the two DO statements interchanged. If on the other hand, the components were not reordered but let's say the values of the first component were reversed, i.e. i will take the values \( n_1, n_1 - 1, \ldots, 1 \) in that order, then the comparable program would be as above with the first DO statement replaced by \( \text{DO } i = n_1 \) to 1 by -1.

Although for the previous cases, the evaluation would be legitimate for any of the boundary vectors \( [n_1, n_2] \), i.e. each step would be legitimate, in this last case the DO-representation would not be legitimate because we
would start with \([i,j] = [n_1,1]\) and get the first evaluation:

\[
F(n_1,1) = F(F(n_1-1,1), G(n_1-1,1)) \\
G(n_2,1) = G(G(n_2-1,1),1)
\]

and \(F(n_1-1,1)\) is neither an input nor a previously evaluated block output.

The example gives the essentials of our definition of the evaluation of a DO-representation, but now we will state the definition somewhat more generally and develop some notation useful in stating our results.

Now assume the assignment statements in the DO-representation are given symbolically in terms of the vector \(I = [i_1, i_2, \ldots, i_d]\). Assume then the component ordering is given by a function on the integers 1 through \(d\), \(\pi\), such that the component \(i_{\pi(1)}\) (or \(i_{\pi_1}\)) is the first component in the component ordering, and in general the component \(i_{\pi(j)}\) is the \(j^{th}\) component in the ordering. In other words the \(\pi(j)^{th}\) component is the \(j^{th}\) component in our component ordering. Assume further that \(\sigma_j(ij)\) is a function of the component \(ij\) such that \(\sigma_j(ij) = k\) when \(ij\) takes on its \(k^{th}\) value. Then for any boundary vector \([n_1, n_2, \ldots, n_d]\), the DO-representation generates an ordering of all the points in the array such that the evaluation is to be carried out at each of these points in the specified order. That order is given as follows: if \(I = i_1, i_2, \ldots, i_d\), let:

\[
\pi(I) = [i_{\pi_1}, i_{\pi_2}, \ldots, i_{\pi_d}]
\]

and

\[
\sigma(I) = [\sigma_1(i_1), \sigma_2(i_2), \ldots, \sigma_d(i_d)]
\]

and

\[
\pi(\sigma(I)) = [\sigma_{\pi_1}(i_{\pi_1}), \sigma_{\pi_2}(i_{\pi_2}), \ldots, \sigma_{\pi_d}(i_{\pi_d})]
\]
1st value of \(\pi(0(I)) = [1,1, \ldots, 1]\)
2nd " " " = [2,1, \ldots, 1]

\(n_{\pi_1}^{th}\) value of \(\pi(I) = [n_{\pi_1}, 1,1, \ldots, 1]\)
\(n_{\pi_1}^{+1}\)th " " " = [1,2,1, \ldots, 1]

\(2n_{\pi_1}^{st}\) value of \(\pi(I) = [n_{\pi_1}, 2,1, \ldots, 1]\)
\(2n_{\pi_1}^{+1}\) " " " = [1,3,1, \ldots, 1]

\(\ldots\)
\(n_{\pi_1} n_{\pi_2} n_{\pi_d}^{th}\) value of \(\pi(I) = [n_{\pi_1}, n_{\pi_2}, \ldots, n_{\pi_d}]\)

In the case in which the values of the \(j^{th}\) index component take the values 1,2,3, \ldots, \(n\) or in other words \(c_j(ij) = ij\) for all \(j\) and \(k\), the component we have that \(0(I) = I\), so we can replace \(0(\pi(I))\) in the above with \(\pi(I)\).

In any case, the DO-representation imposes an ordering on the values of the components of the vector \(I\) in terms of which the assignment statements in the DO-representation are given. This ordering is given above.

We call the ordered set of \(\pi(0(I))\) values \(\text{VEC}[\pi, \sigma]\).

In general, the form of the assignment statements in the DO-representation is:

\[Z(I) = Z[W_1(t_1(I)), W_2(t_2(I)), \ldots, W_m(t_m(I))]\]
If one substitutes the component from the $k^{th}$ value of $\pi(0(I))$ in VEC[$\mu,\sigma$] to form the $k^{th}$ value of $I$ and substitutes the resultant value of $I$ in each assignment statement of the DO-representation then the condition for the legitimacy of the $k^{th}$ evaluation step of the DO-representation can be given as follows:

The $k^{th}$ step is legitimate if the $k^{th}$ value of $\pi(0(t_j(I)))$ either occurs as the $k-r^{th}$ vector in VEC($\pi,\sigma$) with $r \geq 1$, or is outside VEC($\mu,\sigma$) for each assignments statement and each $t_j(I)$ appearing on the right of an assignment statement, and a DO-representation is legitimate if and only if for every boundary $[n_1,n_2, \ldots , n_d]^{*}$ every evaluation step is legitimate.

Stated another way if $t(I)$ is any vector transformation expression that appears on right of an assignment in the DO-representation, then the expression $\pi(0(t(I)))$ when evaluated with any set of positive integers assigned to the components of $I$, must appear earlier in the sequence VEC($\pi,\sigma$) than does the corresponding evaluated $\pi(0(I))$.

To decide then whether a DO is legitimate, we must know the vector transformations $T$ well enough to make the kind of judgement described. Actually the judgement can be made fairly simply in terms of the corresponding components of $I$ and $V_1$.

We consider first the case where the order of values specified for each component $i_j$ of $I$ is simply $1,2, \ldots , n_j$= boundary on $i_j$. In this case $\pi(0(I))=\pi(I)$ and the sequence of values of $\pi(I)$ is VEC($\pi,\sigma$) given in the previous section. Notice that the ordered set VEC($\pi,\sigma$) can be viewed

---

*We note that for a given DO-representation if for some boundary $[n_1,n_2, \ldots , n_d]$ and some value of $\pi(I)$, the corresponding value of $\pi(0(t_j(I)))$ has all components $\geq 0$ and does not occur in the sequence VEC($\pi,\sigma$), then there will be some boundary $[m_1,m_2, \ldots , m_d]$ with the $m$'s larger than the corresponding $n$'s in which for the same value of $\pi(0(I))$, $\pi(0(t_j(I)))$ will be in VEC($\mu,\sigma$).
By similar reasoning, we can extend our argument to the case that \( \pi(O(I)) \neq \pi(I) \), that is, where the components \( i_j \) of \( I \) may take values in an order different from \( 1, 2, 3, \ldots, n_j \). We are still dealing with \( \text{VEC}(\pi, \sigma) \) and may still view it as a sequence of numbers low order digit leftmost.

Let:

\[
\pi(O(I)) = O_{\pi_1} (i_{\pi_1}) O_{\pi_2} (i_{\pi_2}) \ldots O_{\pi_n} (i_{\pi_n})
\]

\[
\pi(O(\tau(I))) = O_{\pi_1} (q_{\pi_1}) O_{\pi_2} (q_{\pi_2}) \ldots O_{\pi_n} (q_{\pi_n})
\]

Again when viewed as a number representation \( \pi(O(I)) \) will progress in order through its values. So, as above, \( \tau(I) \) will always be lower than \( I \) in the ordering if and only if there exists \( 1 \leq u \leq d \) such that \( \sigma_{\pi_j} (i_{\pi_j}) = \sigma_{\pi_j} (q_{\pi_j}) \), \( j = u+1, u+2, \ldots, u+q=d \) \( O_{\pi_u} (i_{\pi_u}) > O_{\pi_u} (q_{\pi_u}) \).

Example: Let \( \sigma_2(i_2) = n_2 - i_2 + 1 \); \( \sigma_1(i_1) = i_1 \); \( \pi(I) = [i_2, i_1] \)

So \( \sigma(I) = [i_1, n_2 - i_2 + 1] \)

\( \pi(O(I)) = [n_2 - i_2 + 1, i_1] \)

Similarly let \( \tau(I) = [i_1 - 1, i_2 + 1] \)

So \( \pi(O(\tau(I))) = [n_2 - i_2 + 1, i_1 - 1] \)

Since \( i_1 < i_1 - 1 \) the condition is met for the given \( \tau(I) \).
as an increasing sequence of numbers whose leftmost digit is their low order digit. Using this fact, the determination of whether a particular \( t(I) \) appearing in a DO-representation is always earlier than the corresponding \( I \) in the ordering of vectors in \( \text{VEC}(\pi, \sigma) \) can be done simply in terms of the components of \( I \) and \( t(I) \).

As we sequence through the evaluation of the DO-representation, the vector \( \pi(I) \) will sequence through \( \text{VEC}(\pi, \sigma) \) taking on values in numerical order if the vector is viewed a representing a number (written in reverse order). In this sequence the digit \( i_{\pi_j} \) starts with 1 as its lowest value and ranges up to \( \eta_{\pi_j} \) as its highest value. Similarly \( \pi(t(I)) \) can be viewed as a number in the same representation. With this in mind we can see that the value of \( \pi(t(I)) \) will always be lower than the corresponding value of the corresponding \( \pi(I) \) if the number represented by \( \pi(I) \) (which = \( N(\pi(I)) \)) is always greater than the number represented by \( \pi_1(t(I)) \) (which = \( N(\pi(v_1)) \)), i.e. if \( N(\pi(I)) > N(\pi(v_1)) \). This in turn will be true under the following conditions: there exists \( 1 \leq u \leq d \) with \( i_{\pi_j} = q_{\pi_j} \) for \( j = \{u + 1, u + 2, \ldots, u + q = d \} \) and \( i_{\pi_u} > q_{\pi_u} \).

Example: If \( I = (i_1; i_2) \), \( \pi(I) = (i_2, i_1) \), \( t(I) = (i_1 + 1, i_2 + 1) \) \( (t(I)) = (i_2 + 1, i_1 - 1) \), so \( i_{\pi_2} i_1 > q_{\pi_2} = i_1 - 1 \) and so the condition is met for the given \( t(I) \).
Conclusions

We conclude the report at this point. We have not given details on all the points raised in the introduction but we have given details on the following points.

(1) We have defined a class of computation arrays.

(2) For each such array, we have defined a recursive function which if evaluated gives the same results together with a test for determining whether the definition is legitimate, i.e. can be evaluated in the traditional way. There are some such arrays for which the corresponding recursive definition is not legitimate, i.e. those having loops.

(3) For each legitimate recursive function definition in (2), a class of equivalent nested DO programs together with tests for determining whether each of these can be evaluated, i.e. will have arguments available when the functions requiring those arguments are available, was defined. We note that there is no nested DO program of the type of (3) it is trivial to find the equivalent recursive definition of the form of (2) and thence to find other equivalent nested DO programs.

References


