The Definition and Validation of the Radix Sorting Technique

John A. N. Lee
Department of Computer and Information Science

Technical Report 71B-1

University of Massachusetts
Amherst, Mass. 01002

(January, 1972)

Paper presented at the Conference:

"Proofs of Assertions about Programs"
New Mexico State University
January, 1972

THE DEFINITION AND VALIDATION OF THE RADIX SORTING TECHNIQUE

John A. N. Lee
Visiting Professor of Mathematics
University of Denver*

Abstract

Based on the formal definition techniques of Lucas et al[1] which have been applied to the description of programming languages, this paper defines the algorithm of radix sorting and develops a proof of its validity.

Introduction

The techniques of formal definition as developed by Lucas and Walk[1] have, in general, been applied to the definition or programming languages [2,3], although formal proofs of validity or equivalence have been applied to specific algorithmic methods of representation of the elements of the language or its compilers. For example, Lucas [4] showed the equivalence of the realization of block concepts as initially employed in the formal definition of PL/I. Related to the same definitional project, Lim[5] proved the equivalence of two versions of the update and search mechanisms. Both of these proofs utilized the method of Recursive Induction proposed by McCarthy[6].

Based on the premise that the method of definition is applicable over the whole domain of algorithms, since a definition of a programming language is fundamentally a co-ordinated system of algorithmic definitions, this paper presents the formal description of a simple algorithm and its validation on the basis of formal logic rather than the previously used technique of Recursive Induction. It is assumed that the reader is familiar with the language of formal definition developed by Lucas and Walk**, as well as the fundamental elements of that system. That is, the definition of objects, predicates, conditional expressions, the mutation operator and the search function. Herein, we shall give formal definitions of those elements of the definitional system necessary for this example and some of the properties of these elements without proof, the development of these proofs being left to the reader.

The Technique of Sorting by the Radix Method

The method of sorting a list of character strings (which correspond to the keys of a set of records) by the technique of radix sorting, predates electronic computers by almost half a century, being one of the techniques used by the Bureau of the Census on the early punched card equipment. The technique is totally mechanical and is not necessarily a technique which should have been carried over for use in computers as we know them today. It is interesting to note that the first program developed by Von Neumann[7] for a stored program computer was a program for sorting a list of integers, and that the standard against which he was able to judge the efficiency of the stored program would have been a mechanical sorter using the technique of sorting described here.

First let us describe the mechanical sorter and the sorting technique which has been fashioned for use on the available equipment. A typical sorter consists of a collection of indexed *pockets* (or stacks) which are fed records from a hopper by a series of selectable chutes. The selection of a particular chute, and hence the particular pocket into which a record from the hopper will be fed, is triggered by the sensing of a representation of the key in the record corresponding to the particular pocket. However, the sorting machine is capable of sensing only one character at a time in each record from the hopper. Hence the development of a sorted order from a set of elements in which the key consists of more than one character, requires several passes through the machine. If the pockets are indexed 0 through 9, and the keys over which sorting is to be performed are purely numeric (digital), then a single pass through the machine of the set of records to be sorted will develop at least a partial sorting. For example, if the key consists of n digits, $d_1d_2d_3\ldots d_{n-1}d_n$, then sorting with respect to the i-th digit would place in the same pocket each element in which $d_i=p$ where p is the index of the pocket. From this *distribution* phase of the sorting process, a partially ordered list may be organized by collecting the contents of each pocket in such a manner as to preserve the relationships between the i-th digit of each record. Thus if an ascending ordering is required, a new list may be formed by *collecting* the elements of the pockets in the order 0 through 9, whilst at the same time maintaining any order already existing in the pocketed elements. It can be shown by example, that when the key is a positional number representation, then distributing according to the low order digit, collecting back into a single list, and continuing to distribute and collect according to successively higher order digits until all digits have been utilized, will develop a completely sorted list. For example, consider the list of keys

<39,42,16,53,49,12>

which are to be sorted into ascending order (left to right on the page). Distributing according to the low order digit in the keys as shown in Fig. 1, will develop 10 sublists corresponding to each pocket in the machine, or which 4 are non-empty;

*Now at the University of Mass., Dept. of Computer Science.
**Variously known as Universal Language Descriptor (ULD) or Vienna Definition Language (VDL)

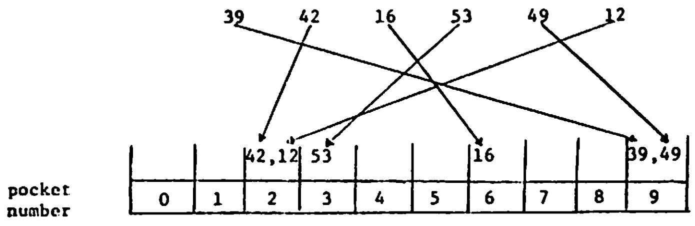

Fig. 1. Distribution according to low order digit

Collecting these elements into a single list whilst maintaining the partial order of the distribution by pocket number and the order of arrival in the pocket (signified by left to right ordering above), a new list <42,12,53,16,39,49> is constructed. Distributing next with respect to the high order digit (Fig. 2), and collecting so as to maintain both the order of arrival and the ordering amongst pocketed elements, the list <12,16,39,42,49,53> is obtained. Since all digits in the keys have been used in successive distribution and collection phases, the list must now be in the required order.

The Environment and Properties of Objects in the Environment

The method of formal definition developed by Lucas and Walk, assumes that there exists an environment over which the algorithm operates, and which is part of an abstract machine which executes (or interprets) the algorithm. We shall define the environment E as part of the state ξ such that E is selected from the state of the abstract machine by the selector function s-E. The structure of the environment is then defined by the structured predicate

is-E = (<s-A:is-A>,<s-P:is-P>)

where the s-A-component of the environment represents the hopper of the mechanical sorter and is represented by a simple list of elements, and where the s-P-component represents the ten pockets of the sorter (numbered 0 through 9) which is a collection of lists of elements. That is,

(1) is-A = is-number-list

and

(2) is-P = ({<s(i):is-number-list>| $0 \leq i \leq 9$})

where

(3) is-number = is-digit-list

This definition of the components of the environment is based on the following simplification and assumption;

a) the records to be sorted are represented only by the keys which conform to the predicate is-number, and

b) the pockets may contain empty lists and hence the s-P-component cannot be represented by a list (as defined by Lucas and Walk), since a list is composed only of non-null elements and the referencing indices of the elements of a list are required to be greater than 0. See definition (4) below.

(4) is-list = ({<elem(i):not-null>| $1 \leq i \leq n$}) v is-<>

where

(5) is-null = is-Ω v is-<> v is-{},

and where Ω is the null object, <> is the empty list, and {} is the empty set. Since, in the object which conforms to the predicate is-P, we require that the components may be empty lists (corresponding to empty pockets), we introduce the object conforming to the predicate ({<s(i):is-number-list> | $0 \leq i \leq 9$}). Such an object is not constrained by the previously defined attributes of an object which conforms to the predicate is-list.

is-number-list is a predicate which defines not only the structure of an object but also the structure of the components of that list. Thus we define

(6) is-number-list(X) = is-list(X) &
($\forall$i) (is-number(elem(i.X)))†

Although definition (3) defines the predicate is-number as a list of objects conforming to the predicate is-digit, the algorithm which is defined later assumes that each element (key) of the list to be sorted is of a common length. That is, each element contains the same number of digits, with zero left (high order) fill if necessary. Thus the definition of the predicate is-number

†elem(i)(X) is also written elem(i,X)

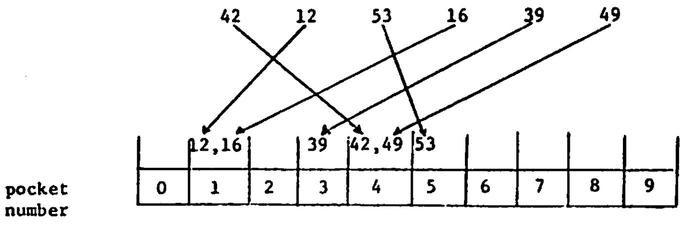

Fig. 2. Distribution according to high order digit

may be revised to the form;

(3a) is-number = ({<elem(i):is-digit> | $1 \leq i \leq m$})

where m is a constant to be predefined in the environment. On this assumption, the length of each element is determined within the algorithm by the function expression length(head(A)) where A is the list of elements being sorted.

Similarly it is assumed that the list of objects which compose an object conforming to the predicate is-number represent a positional representation of a numeric value such that the elem (1)-component contains the low order digit and the elem(length(head(A)))-component contains the high order digit. All numbers are assumed to be positive integers for the purposes of this algorithm definition.

If the following definitions are assumed;

(7) head(list) = elem(1,list)

(8) last(list) = elem(length(list),list)

(9) tail(list) = μ_0({<elem(i):elem(i+1, list)< | $1 \leq i \leq$ length(list) - 1})

where

(10) length(list) = is-list(list) → (is-<>(list) → 0, T→(ιi)(not-null(elem (i,list)) & is-null(elem(i+1, list))))

and the following property of a structured predicate is assumed;

(11) is-pred = ({<K_i:is-pred_i>}) ⊃ K_i(is-pred)=is-pred_i

then it may be shown that

(12) is-number(head(is-number-list))

(13) is-number(last(is-number-list))

(14) is-number-list(tail(is-number-list))

Further based on the definition of the function length over an object conforming to the predicate is-list,

(15) is-list(X) ⊃ length(tail(X)) = length(X) - 1

The fundamental operation over lists is the operation of concatenation represented by ⌢ which we shall define by the conditional expression:

(16) X ⌢ Y = is-list(X) & is-list(Y) → μ(X;{<elem(length(X) + i): elem(i,Y)>| $1 \leq i \leq$ length(Y)})

from which we may deduce:

(17) is-list(X) & is-list(Y) ⊃ is-list(X ⌢ Y)

and

(18) is-list(X) & is-list(Y) ⊃ length(X ⌢ Y) = length(X) + length(Y)

Properties of Instructions in the State of the Machine

The definition of the algorithm of radix sorting which is given here uses an extended version of the definitional system[8] over that developed by Lucas and Walk. In this version the component of the state over which mutations are to be performed is explicitly defined. This component is known as the *subject argument* of the instruction.

Definitions of instructions in the definitional language take the form of macro expansion groups, which define the set of instructions which are to be executed (interpreted) as a means of executing a particular instruction, or consist of a list of mutations over the subject argument.* A group of instructions which define an instruction, replace the instruction in the control part of the state of the machine, thereby performing the macro expansion. Mutations defined over the subject argument consist of a set of selector-object pairs which are to be the arguments of a mutation operator expression. Arguments within these instructions must depend directly on the arguments of the instruction being executed or on the state of the machine. Within the extremely limited set† of instructions utilized in the definition of the radix sorting method the following properties of the state of the machine can be developed.

If Γ defines a mapping from state ζ_a to state ζ_b, $\zeta_a \xrightarrow{\Gamma} \zeta_b$, by the execution of a single instruction contained in the state ζ_a as the K-component then it may be shown that:

a) If that instruction is defined by a macro expansion group X, then

(19) ζ_b = μ(ζ_a;<K:eval(X,K(ζ_a))>)

where eval(X,K(ζ)) represents the macro instruction group X in which the parameter expressions of the embedded instructions have been evaluated with respect to the arguments of the instruction K(ζ_a).

b) If the instruction is defined by a mutation group of pairs ({s_i:exp_i}), where k is the selector over the state ζ_a of the subject argument of the instruction being executed, then

(20) ζ_b = μ(ζ_a;<K:Ω>,{<s_i.k:eval'(exp_i, K(ζ_a))>})

where eval'(exp_i,K(ζ_a)) yields the value of the expression exp_i with respect to the arguments of the instruction K(ζ_a).

Now examination of the mutate operator μ will show that the following properties hold;

Where is-pred_A = ({<K_i:is-pred_i>}) such that <K_B:is-pred_B> ε is-pred_A then

(21) is-pred_A(μ(A;<K_B:B>))

Further,

(22) is-number-list(X) & is-number(Y) & $1 \leq i \leq$ length(X) + 1 ⊃ is-number-list (μ(X;<elem(i):Y>))

and

(23) is-list(Y) ⊃ is-list(μ(X;<I:Y>))

Consider the execution of a macro expansion definition of a recursively defined instruction;

```
inst(X;Y) =
      ...
      ...
      P_i → inst(f(X);g(X,Y));
          group(h(X,Y))
      ...
      ...
```

*For a detailed description of instruction definition groups, see Ref.[1] .

†That is, macro expansion definition groups contain no values to be passed from one instruction to the argument list of another, mutation definition groups contain no PASS: line and no mutations are defined which delete an element of a list, although the deletion of a complete list is included.

where X is the subject argument of the instruction inst being executed, and Y is a list of arguments. The instruction group has an argument list which is a function h over the arguments X and Y. It is required [8] that the function f is an element of the set S* (where S is the set of selector functions) since the subject argument must be a direct component of the state. Function g is not restricted in the same manner. Then it may be shown that, so long as p_i = T and for all j < i, p_j = F, the execution of the instruction inst with the arguments X and Y is equivalent to the group of instructions

```
(24)  p_i → inst(f^n(X);g^n(X,Y));

            group(h(g^{n-1}(X),g^{n-1}(X,Y)));

            group(h(f^{n-2}(X),g^{n-2}(X,Y)));
            ...
            ...
            group(h(f^1(X),g^1(X,Y));

            group(h(f^0(X),g^0(X,Y)))
```

where $f^0(Z) = Z$ and $g^0(S,T) = T$ and $f^n(Z) = f \cdot f \cdot f \cdot \ldots \cdot f(Z) = f(f(f(\ldots f(Z)\ldots)))$

Similarly, the definitions (chosen at a particular instant in the process of transition between states such that the conditions, under which the macro-expansion groups are chosen, do not change)

```
inst(X;Y) =
      ...
      ...
      inst-2(f_2(X);g_2(X,Y);
      inst-1(f_1(X);g_1(X,Y))
```

and

```
inst-1(A;B) =
      ...
      ...
      inst-12(f_12(A);g_12(A,B));
      inst-11(f_11(A);g_11(A,B))
```

are equivalent to the single definition

```
(25)  inst(X;Y) =
        ...
        ...
        inst-2(f_2(X);g_2(X,Y));
        inst-12(f_12.f_1(X);g_12(f_1(X),
                  g_1(X,Y)));
        inst-11(f_11.f_1(X);g_11(f_1(X),
                  g_1(X,Y)))
```

provided that none of the functions f_i and g_i are dependent on the state of the machine or components thereof.

The Algorithm

Following the description of the algorithm given earlier, let us define the problem of formal definition as that of defining a mapping,

$$(26) \quad A_0 \xrightarrow{\Gamma} A^*_n$$

such that A^*_n is an ordered list of elements developed from the unordered list A_0. The definition of the mapping is to be divided into 2n intermediate mappings

$$(27) \quad A_0 \xrightarrow{\Gamma^d_1} P_0 \xrightarrow{\Gamma^c_1} A_1 \xrightarrow{\Gamma^d_2} P_1 \xrightarrow{\Gamma^c_2} \ldots \rightarrow A_{n-1} \xrightarrow{\Gamma^d_n} P_{n-1} \xrightarrow{\Gamma^c_n} A_n$$

The composite mapping $\Gamma_i = \Gamma^d_i \cdot \Gamma^c_i$ is a single "pass" or "cycle" through the set of elements, Γ^d_1 is the mapping of the elements in the hopper (object A, or the s-A-component of the environment) into the pockets of the sorter (the s-P-component) according to the i-th digit in the element, and Γ^c_i is the mapping of the ten lists in the pockets into a single list in the hopper. The mapping Γ^d_i is represented by the execution of the definitional instructions distribute and the Γ^c_i mapping by the execution of the instructions collect.

Assuming that the control part of the state of the machine contains the instruction distribute (s-P(E);s-A(E),1) initially, we define the following instructions as the definition of the sorting technique or algorithm

```
distribute(P;A,i) =
   i > length(head(A))→ null
   is-<>(A) → collect(s-A(E);P,0,i)
   T → distribute(P;tail(A),i);
       pocket(P;head(A),i)
pocket(P;value,i) =
   s(elem(i,value)):s(elem(i,value),P)⌒
    <value>
collect(A;P,j,i) =
   j > 9 → distribute(s-P(E);A,i+1)
   T → collect(A;P,j+1,i);
         collect-2(s(j,s-P(E)););
          collect-1(A;s(j,P),j)

       collect-1(A;list,j) =
         j = 0 → I:list
         T → I:A⌒list

         collect-2(p;) =
          I:<>
```

The instruction distribute initially has the subject argument of the s-P-component of the environment, into which the elements of the argument A are to be placed. Initially the argument A is identical to the object s-A(E), but in successive executions of the distribute instruction this parameter value is replaced by the tail of the argument A in the previous execution. The argument i is the index of the selector of the digit in an element of the argument A over which the distribution pass is to be executed. Initially the value of the object i is 1, since it is assumed that the low order digit of each element is contained in the elem(1)-component of the element. Similarly it is assumed that the number of digits in each element is a constant, equal to the length of any element in the s-A-component of the environment. Hence the high order digit of each element is contained in the elem(length (head(A)))-component of the element.

The distribute instruction is defined as a conditional expression with the ordered propositions; i > length(head(A)), is -<>(A) and T. The first proposition is true only when the index to be applied to the elements being sorted is greater than the number of digits in the elements.

This proposition terminates* the execution of the algorithm. The proposition is-<>(A) tests for the completion of a pass through the elements of the argument A. When this proposition is true then all the elements of the s-A-component of the environment have been distributed to the pockets of the s-P-component. In this case, the elements are to be re-assembled into a new list in the s-A-component by the execution of collect instructions. Otherwise, (the proposition T) the instruction distribute is replaced by the group

```
    distribute(P;tail(A),i);
     pocket(P;head(A),i)
```

The instruction pocket is a value returning instruction and thus has no effect on the control part of the state of the machine (other than its own removal) during or after its execution. This instruction operates over the subject argument P with the arguments value and i. value was selected from the argument A of the instruction distribute and is an element of s-A(E). The argument i is passed directly from the instruction distribute and is the index of the digit in the elements of s-A(E) over which this pass of the sort is being accomplished. By the above definition, the list <value> is appended to the list s(elem(i,value), P), which is the pocket in the subject argument P which has the index elem(i,value). That is, the index corresponding to the i-th digit in the argument value. This augmented list is then returned to the state as the s(elem(i,value))-component of the subject argument P.

The collect instruction contains only two definitions, corresponding to the propositions j > 9 and T. The value of the argument j was passed from the execution of the instruction distribute under the condition is-<>(A), and has an initial value of 0. This argument (j) is used as an index over the components of P, corresponding to the pocket indices in the machanical sorter. So long as j is less than 9, then the execution of the instruction collect is achieved by replacing the instruction in the control part of the state of the machine by the group.

```
    collect(A;P,j+1,i);
      collect-2(s(j,s-P(E)););
        collect-1(A;s(j,P),j)
```

The latter instructions in this group (to be executed *prior* to the execution of the embedded collect instruction) are value returning instructions acting over the state of the machine. collect-1 (which is always executed prior to the collect-2 instruction) takes the argument list with the value s(j,P) and either replaces totally the subject argument A (which is the s-A-component of the environment) when j = 0 or augments the subject argument by list. Hence, the collect-1 instruction executed successively over the range of $0 \leq j \leq 9$, assembles the components of P into a single list as the new s-A-component of the environment. The instruction collect-2 initializes the j-th pocket in the s-P-component in preparation for the next pass.

*The final state of the machine is defined as that in which the control part of the state contains no candidate instructions for execution.

When the collect instructions have been executed over the range $0 \leq j \leq 9$, then it assumed that a new s-A-component has been created and the next pass may be commenced by the execution of the instruction distribute(s-P(E);A,i+1). In the argument list of this instruction, the index of the selector to the digit position in each element of the list being sorted has been incremented by one over that for the last pass. Hence, the pass being initiated will occur over the next higher order digit.

The Definition of Ordering

In order to provide a basis for the validation of the mapping function from A_0 to A_n^* (eqn. 26) it is necessary for us to define the qualities of the predicate which has the value T when a list of numbers is ordered. Let us first define an ordering relation over the domain of objects which conform to the predicate is-number, based on the assumption that the relations of "less than", "greater than" and "equal" are predefined over the domain of digits;

$$\begin{aligned}(28)\quad & \text{rel}(a,b,i) = \\ & \quad \text{is-number}(a)\ \&\ \text{is-number}(b)\ \&\ \text{length}(a) = \text{length}(b)) \to \\ & \qquad (i{=}0 \lor \text{elem}(i,a) < \text{elem}(i,b) \to T, \\ & \qquad \text{elem}(i,a) = \text{elem}(i,b) \to \text{rel}(a,b,i-1),\ T \to F)\end{aligned}$$

This definition of relation is defined only over the i low order digits of the two arguments a and b. Thus when i = length(a) the function rel is equivalent to the usual "less than or equal to" relationship between two numeric values. In particular, we have chosen to define that when i = 0 the two numeric values are in the correct relationship with each other. Further, when the i-th digits of the two arguments are equal then the function rel is dependent of the relationships over the (i-1) low order digits. The following properties of the rel function may be derived;

$$(29)\quad \text{rel}(a,b,i) \supset (\exists j,\ j \leq i)(\text{rel}(\langle\text{elem}(j,a)\rangle, \langle\text{elem}(j,b)\rangle, 1))$$

$$(30)\quad \text{rel}(a,b,i)\ \&\ \text{rel}(\langle\text{elem}(i+1,a)\rangle, \langle\text{elem}(i+1,b)\rangle, 1) \supset \text{rel}(a,b,i+1)$$

Having established a relationship over the domain of pairs of numbers in our representation, let us now extend the concept to that of orderedness;

$$\begin{aligned}(31)\quad & \text{ordered}(A,i) = \\ & \quad \text{is-number-list}(A) \to \\ & \qquad (\text{length}(A) > 1 \to \mathop{\text{Et}}\limits_{j=2}^{\text{length}(A)} \\ & \qquad \text{rel}(\text{elem}(j-1,A),\text{elem}(j,A),i),\ T \to T)\end{aligned}$$

where $\mathop{\text{Et}}\limits_{j=m}^{n} p_j = p_m\ \&\ p_{m+1}\ \&\ \dots\ \&\ p_{n-1}\ \&\ p_n$

This definition of the function ordered, in common with our definition of the function rel, defines partial ordering over the low order i digits of the elements of the list A, and only when i = length(head(A)) does the function conform with the notion of "complete ordering". In particular the function is not defined if the argument A is not a list or its components (elem(i,A)) are not in conformance with the predicate is-number. Where the list is an empty list or contains only one element (length(A) = 1), then the list is said to be ordered.

The properties of this function include;

(32) ordered(A,i) ⊃ (∀j,2≤j≤length(A))
(∀k,k<j)(rel(elem(k,A),elem(j,A),i))

which follows from the transitive property of the rel function;

(33) rel(a,b,i) & rel(b,c,i) ⊃ rel(a,c,i)

and

(34) ordered(A,i) & ordered(B,i) & rel
(last(A),head(B),i) ⊃ ordered(A⌒B,i)

The Execution of the Algorithm

Initially the control part of the state of the machine contains the instruction distribute (s-P(E);s-A(E),1) where the s-P-component of the environment is preset to be a collection of empty lists and where the s-A-component contains the list of elements which are to be sorted. Hence the dummy parameters P and A of the definition of distribute have the values of a collection of empty lists and the list of elements to be sorted respectively. Additionally, the subject argument P is selected from the state by the composite selector s-P·s-E.

By the definition of an equivalent instruction group (24), it may be shown that the definition of the instruction distribute may be replaced by the group

```
(35)  distribute(P;A,i) =
        i > length(head(A)) → null
        is-<>(A) → collect(s-A(E);P,0,i)
        T → distribute(P;<>,i);
              pocket(P;last(A),i);
              ...
              ...
              pocket(P;head(tail(A)),i);
               pocket(P;head(A),i)
```

It is important to note that the groups corresponding to the propositions is-<>(A) and T cannot be combined since the instant of assigning values to the parameters would not develop the same values. That is, the value of the parameter P would be a collection of empty lists is the definition of distribute were to be changed to

```
distribute(P;A,i) =
  i > length(head(A)) → null
  T → collect(s-A(E);P,0,i);
        pocket(P;last(A),i);
        ...
        ...
```

since arguments are evaluated at the instant the instruction is placed in the control part of the state of the machine. However, in the equivalent group given previously, the value of the parameter P in the collect instruction would be that at the instant that the collect instruction were added to the control part; that is, the s-P-component of the state updated by the previously executed pocket instructions.

The value of the parameter value in the definition of the pocket instruction is successively head(A), head(tail(A)),...,last(A); that is, elem(1,A), elem(2,A),...elem(length(A),A). This succession of elements is not directly related to the ordering of the elements over the function rel, but rather will depend on any preveious history of the argument A. Initially, the object P is a collection of empty lists, and hence by our definition of ordered, each component of P is an ordered list.

Since the instruction group in definition (35) is a linear group, (that is, a degenerate tree containing but a single branch), then the order of execution of the pocket instructions is fixed, being sequential from bottom to top in that definition. Thus the order in which the arguments elem(k,A) are applied to the subject argument P is similarly fixed.

The s-P-component is defined to be a collection of lists (def. (2)) each of which is preset to be an empty list, prior to execution of the distribute instruction. Execution of a single pocket instruction maintains the ordering of the s(elem(i,value))-component of P provided that condition (34) is met. Then by execution of successive pocket instructions, the list of elements in s(elem(i,value))-component of P is such that

(36) (∀k, 0 ≤ k ≤ 9)(∀j, 1 ≤ j ≤ length
(s(k,P)))(elem(i,elem(j,s(k,P))) = k)

since the argument value is only appended to that sublist in which the i-th digit of each element equals the i-th digit of value.

The condition (36) may be proved by examination of the sequence of instructions

```
pocket(P;last(A),i);
...
...
pocket(P;head(tail(A)),i);
 pocket(P;head(A),i)
```

and their definition is terms of the mutate operator (20). Let

X(a) = s(elem(i,a))·s-P·s-E,

Then the execution of the sequence of pocket instructions above is equal to the evaluation of the compound mutate expression, the result of which is the new state;

μ(μ(...μ(μ(ξ;<X(head)(A)):X(head(A))(ξ) <head(A)>>);
<X(head(tail(A))):X(head(tail(A)))(ξ)⌒<head
(tail(A))>>);...);
<X(last(A)):X(last(A))(ξ)⌒<last(A)>>)

where mutations over the control part of the state have been omitted. From (17) it may be shown that each newly formed component X(a)(ξ) conforms to the predicate is-list. That is, is-list(s(k,s-P(E'))), where E' is the mutated object E after execution of the sequence of pocket instructions.

Now if the list in the s(k)-component of P is ordered with respect to the (i-1) low order digits of the elements of the list, that is

ordered(s(k,P),i-1)

then by property (30),

(37) ordered(s(k,P),i-1) &
(∀j, 1 ≤ j ≤ length(s(k,p)))(elem
(i,elem(j,s(k,P))) = constant) ⊃
ordered(s(k,P),i)

Let us assume that at the instant that the distribute instruction is placed into the control part of the state of the machine, ordered(A,i-1) then, any sublist formed from the elements of A

(38) <elem(m,A),elem(n,A),...,elem(r,A)>

such that m < n < ... < r, is also ordered with respect to the i-1 low order digits of each element, since

ordered(A,i-1) ⊃ (∀m < n)(rel(elem(m,A),
i-1))

But we have already shown that the order in which elements are formed into sublists which are the components of P, is the order of appearance of the elements in A. Further since the execution of a pocket instruction creates a sublist such that the new element is appended as the last element (in the sense of the last function), then the ordering specified in (38) is maintained. Thus the mapping Γ^d_i

$$A_{i-1} \xrightarrow{\Gamma^d_i} P_{i-1}$$

where ordered(A_{i-1},i-1), develops a collection of lists P_{i-1}, where for each component s(k,P), ordered(s(k,P),i)

After execution of the sequence of pocket instruction defined in (35), the control part of the state of the machine will contain the instruction distribute(P;<>,i) which will be the only candidate for execution. Definition (35), under the condition that i > length(head(A)) is not true while is-<>(A) is true defines that the instruction distribute(P;<>,1) is to be replaced in the control part of the state of the machine by the instruction

collect(s-A(E);P,0,i)

Note that the value of the function expression length(head(A)) is undefined under the condition is-<>(A) and hence the proposition i > length (head(A)) is not true.

At the instant that the instruction collect (s-A(E);P,0,i) is placed in the control part of the state of the machine, we know the following conditions hold:

(39a) $(\forall k, 0 \leq k \leq 9)$(ordered(s(k,P),i))

(39b) $(\forall k, 0 \leq k \leq 9)(\forall j, 1 \leq j \leq$ length $(s(k,P)))$(elem(i,elem($\overline{j}$,s($\overline{k}$,P))) = k)

where the value of the argument i is that in the argument list of the collect instruction.

By the equivalence relation (24) and the relation (25), we may show that the definition of the instruction collect may be expanded to the form:

```
collect(A;P,j,i) =
    j > 9 → distribute(s-P(E);A,i+1)
    T → collect(A;P,10,i);
        collect-2(s(9,s-P(E)););
        collect-1(A;s(9,P),9);
        ...
        ...
        collect-2(s(1,s-P(E)););
        collect-1(A);s(1,P),1);
        collect-2(s(0,s-P(E)););
        collect-1(A;s(0,P),0)
```

where

```
collect-1(A;list,j) =
    j = 0 → I:list
    T → I:A ⌒ list
```

and

```
collect-2(p;) =
    I:<>
```

It can be seen from the sequence of instructions in the group corresponding to the proposition T in the definition of the instruction collect that the execution of instructions is order dependent since the argument s(k,P) in a collect-1 instruction is the subject argument in the sequentially following collect-2 instruction. Similarly, the order of execution of collect-2;collect-1 pairs is order dependent since the definition of the collect-1 instruction organizes a new object (which conforms to the predicate is-list) which is composed of the arguments s(k,P) in the order $o \leq k \leq 9$. By the definition of value returning instruction groups (def. 20), the sequence of collect-2;collect-1 instructions which define the collect instruction are executed by the evaluation of the mutation expression

μ(...μ(μ(μ(μ(ξ;<a:b(0)(ξ)>);<b(0):<>>);<a:a(ξ) ⌒ b (1)(ξ)>);<b(1):<>>);...)

where a = s-A·s-E and b(k) = s(k)·s-P·s-E and where the mutations over the control part of the state of the machine have been omitted. From this expression, it is possible to construct the state of the machine at the instant that the condition j > 9 is true. That is, the s-A-component of the environment is the list s(0,P)⌒s(1,P)⌒s(2,P)⌒...⌒s(9,P) where P is the s-P-component of the environment at the instant the instruction collect(A;P,0,i) was placed into the control part of the state of the machine by the execution of the distribute instruction.

From the conditions (39a) and (39b) we may investigate the properties of this new list. We know from condition (39b) that elem(i,last(s (k,P))) = k and further that elem(i,head(s(k+1,P))) = k+1. Hence from the definition of the rel function

rel(last(s(k,P)),head(s(k+1,P)),i)

and further from condition (34)

ordered(s(k,P),i & ordered(s(k+1,P),i) & rel(last (s(k,P)),head(s(k+1,P)i) ⊃

ordered(s(k,P) s(k+1,P),i)

Hence over the set of compound concatenations in the formation of the s-A-component of the environment, we may conclude

ordered(s(0,P) ⌒ s(1,P)⌒ ...⌒s(0,P),i)

At this point in the execution of the algorithm, the instruction collect(A;P,10,i) is the instruction which is the candidate for execution. Obviously, the parameter j, in the definition of this instruction is assigned a value which is greater than 9 and hence the execution of this instruction results in its replacement by the instruction distribute(s-P(E);A,i+1). This completes the operation of mapping from P_{i-1} to A_i;

$$P_{i-1} \xrightarrow{\Gamma^c_i} A_i$$

and further completes the mapping cycle Γ_i;

$$A_{i-1} \xrightarrow{\Gamma_i} A_i$$

The conditions which existed at the beginning of the cycle have been re-established in part;

· the s-P-component of the environment is a collection of empty lists

the s-A-component is a partially ordered list

the control part of the state of the machine contains the instruction

distribute(P;A,i)

with the following variations;

the s-A-component is ordered over the low order i digits whereas at the beginning of the cycle ordered(A,i-1), and the value of the parameter i is the distribute instruction has been incremented by 1.

Hence, over the mapping Γ_i the ordering of the s-A-component of the environment has been improved by one digit.

That is,

ordered(A_{i-1},i-1)> ordered($\Gamma_i(A_{i-1})$,i) but we defined that the mapping from Λ_0 to Λ is composed of a series of cycles Γ_i. Further we specified that complete ordering of a list of multidigit keys is accomplished when the ordering covers all the digits of a key. That is, ordered(A,length(head(A))).

Examination of the definition of the algorithm will show that we may rewrite the definition of the distribute instruction as two definitions;

```
          distribute(P;A,i) =
              i > length(head(A)) → null
              T → repeat(P;A,i)
where
          repeat(P;A,i) =
              is-<>(A) → collect(s-A(E);P,0,i)
              T → repeat(P;tail(A),i);
                  pocket(P;head(A),i)
```

Initially, the value of the parameter i is set to 1, and as can be shown from the definition of the collect instruction is incremented by 1 before the distribute instruction is placed into the control part of the state of the machine again. Thus the instruction repeat is executed successively with the parameter i taking the successive values 1,2,3,...,length(head(A)). When the value of i, returned from the execution of the repeat instruction is greater than length(head(A)), the control part of the state of the machine contains no further candidates for execution and hence the execution of the algorithm is complete.

If at the initiation of execution of the algorithm, the s-A-component of the environment contains a list of elements such that

ordered(A,0)

the s-P-component contains a collection of empty lists, and the control part of the state of the machine contains the instruction

distribute(spP(E);s-A(E),1)

then after one mapping operation over the environment Γ_1,

ordered(A,1).

By induction we may easily see that following mapping $\Gamma_{\text{length(head(A))}}$ then it is true that

ordered(A,length(head(A))).

That is, the algorithm defined is validated to develop an ordered list provided that the initial conditions specified above are met.

Conclusions

Previous validations of algorithms which were defined in terms of the method of definition developed by Lucas and Walk were based on the proof of equivalence of two functions by the method of recursive induction. This proof shows that where a predicate can be developed to describe the attributes of the final state of the abstract machine, then the method of predicate or quantitative calculus is applicable. The example chosen is that of a simple sorting algorithm where there is no indeterminacy with respect to the order of execution of instructions.

In the case of the definition of an algorithm in which there exists a number (presumably finite) orders of execution of instructions, it may be necessary to establish the equivalence of all possible permutations of execution orders before the question of validity of the algorithm can be tested.

References

[1] Lucas, P. and Walk, K., On the Formal Description of PL/I, Annual Review in Automatic Programming, Vol. 6, Part 3, 1969, Pergammon Press, Oxford, U. K.

[2] Formal Defnition of PL/I:

Fleck, M. and Neuhold, E., Formal Definition of the PL/I Compile Time Facilities. IBM Laboratory Vienna, Techn. Report TR 25.080, 1968.

Walk, K., Alber, K., Bandat, K., Bekic, H., Chroust, G., Kudielka, V., Oliva, P. and Zeisel, G., Abstract Syntax and Interpretation of PL/I, IBM Laboratory Vienna, Techn. Report TR 25.082, 1968.

Lucas, P., Alber, K., Bandat, K., Bekic, H., Oliva, P., Walk, K. and Zeisel, G., Informal Introduction to the Abstract Syntax and Interpretation of PL/I. IBM Laboratory Vienna, Techn. Report TR 25.083, 1968.

Alber, K., Oliva, P. and Urschler, G., Concrete Syntax of PL/I. IBM Laboratory Vienna, Techn. Report TR 25.084, 1968.

Alber, K. and Oliva, P., Translation of PL/I into Abstract Text. IBM Laboratory Vienna, Techn. Report TR 25.086, 1968.

Lucas, P., Lauer, P. and Stigleitner, H. Method and Notation for the Formal Definition of Programming Languages. IBM Laboratory Vienna, Techn. Report TR 25.087, 1968.

[3] Lee, J. A. N., The Formal Definition of BASIC (to be published).

[4] Lucas, P., Two Constructive Realizations of the Block Concept and their Equivalence, Techn. Rep. TR 25.085, IBM Laboratory Vienna, 1968.

[5] Lin, A. L., Proof of Equivalence of the Update and Search Mechanisms by Recursive Induction, Techn. Note, LN 25.3.047, IBM Laboratory Vienna, 1968.

[6] McCarthy, J., A Basis for a Mathematical Theory of Computation in Computer Programming and Formal Systems, (Braffort, P. and Hirschber, D., eds.), North-Holland Publ. Co., Amsterdam, Holland, 1963.

[7] Knuth, D. E., Von Neumann's First Computer Program, Computing Surveys, Vol. 2, No. 4, 1970.

[8] Lee, J. A. N. and Wu, D., Vienna Definition Language—A Generalization of Instruction Definitions, Techn. Note TN/CS/00011, Computer Science Program, University of Massachusetts, Amherst, Massachusetts, 1969.