19. CONTINUOUS EVALUATION

J. K. ILIFFE

International Computers and Tabulators Ltd., London, England

1. Introduction

In Chapter 18, the possibility of segmenting a program into individual *routines*, and separating the processing into several *phases* was noted. One objective of such partitioning is to match, as far as possible, the degree of generality in a program at a certain stage of processing to the amount of precise information which a programmer can supply. Greater generality would lead to inefficient programs, and less would make the programs inflexible.

Complete flexibility is gained by carrying out at execution time ('dynamically') processes which might otherwise be performed during translation, e.g. storage allocation, parameter evaluation. The present chapter is concerned with the reverse process, i.e. recognizing during translation situations in which action, otherwise delayed until the execution phase, might take place. In this way, both time and space may be saved in executing a program. For want of a better term, we shall call this intermixing of the translation and execution processes *continuous evaluation.*

Programming abounds with recognizable instances of continuous evaluation. An optimized compiler specifically seeks out situations where the object program may be simplified by preliminary calculation (see Chapter 9), and a list processing system (see Chapter 20) mixes the translator-like action of symbol manipulation with the interpretive activity of the '*APPLY*' operator. With such good precedents our attempt to formulate certain rules for evaluation which *automatically* recognize executable situations is justified, and a brief account follows of an investigation along these lines.

2. Evaluation Rules

A calculation proceeds by assigning *values* to certain *operands* or *variables.* Values are either given *a priori* in the form of initial data, or are derived from given values by the application of certain *operations.* We shall consider for the time being only the binary operations represented by '$+$' and '$\times$', applicable to values taken from the domain of integers. When the calculation stops, a subset of the values derived for the operands is termed the *result* of the calculation.

The calculation is usually represented as a series of consecutive steps, at each of which a value is assigned to a particular variable. The value derived at one step is typically determined by a *formula* involving the application of one or more operations, and the complete value assignment involved in one step may be represented by an *equation* obtained from the schema:

$$\mu = \mathrm{F} \tag{2.1}$$

by substituting a variable name for μ and a formula for F, e.g.:

$$A = B + C \times D \times (2 + B) \tag{2.2}$$

The interpretation of equation (2.2), when it is executed, is that A is to be assigned the value obtained by evaluating the right-hand side of the equation using the current values of B, C, D (and '2').

It is fairly clear that in order to evaluate a formula F three sets of information must first be provided: (i) a full value assignment for each operand named in F; (ii) a set of precedence rules which determines the order in which the operations in F are to be evaluated, and the operands to which each is applied; (iii) a set of evaluation operators, one for each operation, which determines for given (pairs of) operand values, the result of applying each operation to these. We shall denote the evaluation operator for '+' by ϵ^+ and represent its application to two integers μ, ν with result ρ by the schema:

$$\epsilon^+(\mu, \nu) \to \rho \tag{2.3}$$

Thus, a particular application of ϵ^+ gives:

$$\epsilon^+ (3, 4) \to 7 \tag{2.4}$$

Similar remarks apply to the evaluation operator $\epsilon^\times$ corresponding to the '$\times$' operation.

It is of interest to note that the ability to evaluate F depends in turn on the ability to apply the ϵ operators, which may themselves be defined in terms of similar formulae. This circular mode of definition is stopped, however, by the particular form of ϵ operator exemplified by the multiplication and addition tables for the integers 0-9, on which the multiplication and addition of other integers is made to depend. Another set of ϵ operators on which an evaluation may ultimately rest is the function set of a particular computer, which is, of course, of immense practical importance in the present context.

For the principal task of a compiler is to reduce a calculation represented in a problem-oriented language to a sequence of single applications of machine functions, or of operators (e.g. subroutines) defined directly in terms of machine functions. It does this for a particular formula with reference to the precedence rules (ii) of the formula, and the list of available machine functions and subroutines corresponding to the set ϵ; normally, no reference is made to any value assignments (i) which may hold at the time of compilation. When a calculation is obeyed by a machine, it is normally followed on the *assumption a*: that all value assignments required for the evaluation of each formula have been made. The role of continuous evaluation may perhaps be clarified with reference to the statement *a*. In 'normal' compilation, a converse of *a* is assumed; in evaluation, we have noted that *a* is assumed to be true; under a regime of continuous evaluation, however, the truth or falsity of *a* must strictly be determined at each attempt to apply an operator ϵ.

In practice, the necessary control information is supplied by associating with each variable in the context of a calculation one or more bits which determine its 'state of definition', σ. Although in theory one bit is sufficient it is convenient to consider four possible definition states of a variable:

$\sigma = 1$. Undefined.
$\sigma = 2$. Defined as a function of one or more parameters and other variables.
$\sigma = 3$. Defined formally in terms of other variables.
$\sigma = 4$. 'Numerically' defined.

In each state, we assume that the domain in which the variable finds its values (e.g. integer, word, complex number etc.) is known, hence the definition state refers strictly to the representation of its value in that domain. In state 1, no information concerning the variable is known; in state 4, its full value is known, a condition we term *numerically* defined, although it may equally apply to a variable in the domain of character strings. States 2 and 3 correspond to intermediate degrees of knowledge in which the state of definition of the variable depends in turn on the definition states of others. For an arbitrary variable ν, we shall denote its definition state by $\sigma(\nu)$.

Thus, in applying the operator ϵ^+ to two variables μ, ν sixteen possibilities have to be accounted for. If $\sigma(\mu) = \sigma(\nu) = 4$, then ϵ^+ can be applied normally, as in (2.3). If $\sigma(\mu) = 1$ or $\sigma(\nu) = 1$ then ϵ^+ cannot be applied. In each of the other eight cases, applicability of ϵ^+ depends on the resolution of the state of μ or ν. Specifically, in case $\sigma(\mu) = 3$, a numerical value of μ can be obtained only if all variables appearing in the formal definition of μ are themselves numerically defined. Also, if $\sigma(\mu) = 2$, then a numerical value of μ can be obtained only if all parameters required by μ and all variables required by its functional definition are numerically defined.

In devising a continuous evaluation system, therefore, it is important to be able to scan as rapidly as possible the additional information required in case $\sigma = 2$ or $\sigma = 3$. We must describe the action of the machine when confronted with any one of the above situations, or give reasons to believe that certain situations could not arise in practice. Before doing so, however, we must develop a simple method of describing a calculation which enables continuous evaluation techniques to be more fully exploited.

3. Definition Sets

Most current programming languages rest heavily on the traditional *sequential* mode of procedure description derived from the earliest concepts of stored program machines. In this, although permitting the use of formulae has introduced the idea of implicit rather than explicit sequencing, the essentially temporary character of the value assignment made by a single calculation step remains unaltered. It is now recognized that the dynamic value assignment characterized by (2.2) and its interpretation must be supplemented by static assignments which are not part of the sequential description of a procedure. Most obvious candidates for this group are definitions of pre-set parameters, macro-operations, and subroutines.

Accordingly, we shall decompose a description of a calculation initially into a finite set of definitions. Each definition in our simplified system will have either the form of an equation (2.1) or a function obtained from the schema:

$$\phi(\pi) = F \qquad (3.1)$$

by substituting distinct variable names for $\not p$ and π, and a formula containing the name π for F, e.g.

$$f(x) = x + x \times y \times z \tag{3.2}$$

We shall permit formulae to contain function names (e.g. 'f' in (3.2)) with actual parameters in the usual way. We shall also extend formulae to include predicates defined by the elementary arithmetic relations and Boolean operations '*and*', '*or*', and '*not*', used in conditional equations, thus:

$$y = x - 1 \text{ if } A,\ x \text{ if } x<0,\ 1 - x \tag{3.3}$$

Finally, we shall permit function definitions obtained from the following schema, which we shall call a 'routine':

$$\phi\ (\pi) = \{G_1;\ G_2 \ldots ;\ G_n\} \tag{3.4}$$

by substituting distinct variable names for ϕ and π, and an equation for $G_j, j = 1, 2, \ldots, n$, where at least one equation has a formula involving π in the right-hand side, and at least one equation has the name ϕ on the left-hand side. The form (3.4) is obviously an elementary species of problem-oriented program. From another point of view, the right hand side of (3.4) is a formula with rather involved precedence rules which are used to determine the order in which the equations are applied, for we assume that the application of an equation G_j determines also the index j' of the next equation to be applied, or else stops the calculation. The function (3.2) can be represented as a program in the following way:

$$f(x) = \{\underline{u} = x \times y;\ \underline{v} = \underline{u} \times z; f = x + \underline{v};\ STOP\} \tag{3.5}$$

The use of programs in definitions introduces complications in naming which in practice are extremely complex. Here we shall distinguish names local to a program (and hence meaningless outside that definition) from other names in the definition set by underlining, as in (3.5). We shall disallow side effects: i.e. the left hand side of any $G_j, j = 1, 2, \ldots, n$, must either be the function name (ϕ) or an internal name of the definition. A sufficient condition for the applicability of a program is that its parameter and all non-internal names which occur in it should be numerically defined.

We are now in a position to consider the reduction of an elementary definition set by hand. Given *a priori* the integers, denoted by '1', '2', . . . etc. and the truth values '*TRUE*', '*FALSE*', consider the following set D of definitions:

$$\begin{aligned}
&1.\quad v(t) = 0 \text{ if } t \leqslant 0,\ t \times (u + a \times t) \\
&2.\quad a = 15 \\
&3.\quad u = 50 \\
&4.\quad w = v(10) + 2 \times a
\end{aligned} \tag{3.6}$$

For ease of reference, we have numbered the four definitions. The information available in this set is summarized in Table 1. Definitions 2 and 3 consist of value assignments to the variables a and u respectively. Definition 1 is of the function v, and w is formally defined in 4. In the table we list numerical values

TABLE 1

Def.	v	$\sigma(v)$	F	Dependent on:
1	v	2	0 if $t \leqslant 0$, $t \times (u + a \times t)$	t(parameter), u, a
2	a	4	15	—
3	u	4	50	—
4	w	3	$v(10) + 2 \times a$	v, a.

where they are known, otherwise listing the defining formula F and the names appearing in each definition. Our intuitive evaluation of the definition set then proceeds as follows:

(i) v depends only on u and a and is therefore a known function;
(ii) w, depending on v and a, can therefore be evaluated to give the value 2030;
(iii) Since the function v cannot be further reduced, this completes the reduction of the definition set.

Obviously this trivial example in a simplified language can do no more than illustrate a method of argument. It is a method, however, which can be carried over into much richer languages, with powerful results. What is particularly needed, apart from a larger set of elementary operations, is a type of operation leading to the cyclic or iterative behaviour observed in sequentially written programs. The mathematical expression of iterative calculation is contained in recurrence relations, and we may, for example, consider the following as a definition of the variable r, where i is an integer greater than 1:

$$r_i = 2r_{i-1} - r_{i-2},\ r_o = 1,\ r_1 = 0$$

Evidently this is a special case of a function r with parameter i, and the appearance of say 'r_{10}' in a formula would receive essentially the same treatment as '$v(10)$' in (3.6) above. This and other extensions to the definition set, and their efficient encoding, are beyond the scope of the present discussion.

4. Definition Set Processing

A definition set is presented to a machine, or, for that matter, read by a human being, as a linear string of symbols. Consequently, if only part of the machine's computing capacity is absorbed by the reading process, some attention can be given to a parallel transformation of the definitions. In practice, as applied experimentally on the Rice University machine (Ref. 1), this transformation amounts to:

(i) Recognizing numerically defined operands, and where possible, applying operations between them.
(ii) Transforming all function definitions (including programs) into sequential machine code.
(iii) Forming a table, similar to Table 1, to assist subsequent processing.

Some of these effects are illustrated in the example given by the definition set in Table 2, containing a program *P* written out with the usual line con-

TABLE 2

```
1.      $i = 1$
2.      $j = 2$
3.      $A = TRUE$
4.      $B = FALSE.$
5.    $P(C) = \{$
      $\underline{x} = i + j + k$
      $\underline{y} = j$ if $A$, $i$ if $B$, $0$
      $\underline{z} = i$ if $B$ and $C$, $j \times k$
      $\bar{P} = \underline{x} + \underline{y} + \underline{z}$ if $B$, $\underline{x} \times \underline{y}$ if $A$ and $C$, $0$
      $STO\bar{P}\}$
```

vention. As the set is read in, *P* can be transformed to an internal representation of the program *P'* given in Table 3. This form of processing is obviously

TABLE 3

Def.	ν	$\sigma(\nu)$	F	Dependent on:
1	i	4	1	—
2	j	4	2	—
3	A	4	*TRUE*	—
4	B	4	*FALSE*	—
5	P	2	$\{\underline{x} = 3 + k$ $\underline{y} = 2$ $\underline{z} = 2 \times k$ $\bar{P} = \underline{x} \times \underline{y}$ if C, 0 $STO\bar{P}\}$	k, C (parameter)
6	k	1	—	—

dependent on the sequence of definitions in the set, since if *P* preceded the definitions of *i, j, A* and *B* a similar reduction would not be possible. Clearly the result of applying *P* in either case would be the same, and what is achieved is merely a degree of optimization. The above example typifies a situation which arises frequently in the design of systems of programs where a very general program is written initially (e.g. *P*), leaving as variables parameters which are ultimately to be fixed. As the design proceeds *P* becomes more refined, and by recompiling rather than rewriting (and introducing new errors) it is brought to its final form. A particular example is provided by optional print-outs which are inserted in a program for debugging purposes, finally to be eliminated.

The definition set constitutes a very satisfactory unit of information for an operating system to handle. Indeed, the set we have described bears a strong resemblance to the collection of *documents* needed to define a *job* in the Atlas operating system, and both, of course, rest on the fundamental idea of a calculation as a procedure supplemented by sufficient numerical definitions to make it applicable. Further developments of similar systems will undoubtedly be rewarding; methods of building up new sets from old; of cross-referencing between sets; and (for mathematicians) of nesting one set of definitions inside another must be established: We note that the data-oriented storage control systems mentioned in Chapter 18 are ideally suited to the control of data sets by extending the interpretion of lock-out bits to include an indication of the definition state of each variable: in fact, a data region which is locked out in the conventional sense is effectively undefined at that time ($\sigma = 1$), in the sense we have been discussing. Compare particularly the Index Region (Fig. 4, Chapter 18) with our tabular presentation of the definition set.

By analogy with the behaviour of a time-sharing machine, we can therefore envisage the reduction of a definition set proceeding in several definitions at once. When an undefined variable is encountered in one definition, control switches to another, and so on, until by a continuous scanning process the set is reduced to its simplest form. The application of several scanning and computing devices to the same set is theoretically possible.

5. Conclusion

In this very brief introduction to the idea of continuous evaluation, it was hoped to illustrate the advantage of basing a problem solution on a set of definitions rather than a sequential procedure description. This has already been recognized to some extent in the design of operating systems and of commercial compilers; it is believed to be equally advantageous for descriptions of mathematical procedures, which have become too closely enslaved to the early sequential forms of computer language. It is strongly felt that machines of the future will require to break away from sequential activity, and we must beware of having to face the task of automatically breaking up a sequential code into parallel activities: it seems easier to start from a 'parallel' code, and make it sequential when necessary.

The potentiality of definition sets remain to be fully realized. We have a great deal to learn about their properties and interpretation, but they appear to be capable of influencing appreciably both language and machine design in the future.

Reference

1. J. K. Iliffe (1961). The Use of the Genie System in Numerical Calculation. *Annu. Rev. Automat. Progr.* **2**, 1.