INTRODUCTION TO SYSTEM PROGRAMMING

THE ROLE OF ADDRESSING IN PROGRAMMING SYSTEMS

J. K. ILIFFE

18. THE ROLE OF ADDRESSING IN PROGRAMMING SYSTEMS

J. K. ILIFFE

International Computers and Tabulators Ltd., London, England

1. Introduction

One of the most important aspects of computing activity is that of ensuring that the operands of a routine are correctly situated for its use at the time it requires them. Failure in this respect is, of course, one of the main sources of programming errors. Here, as in other aspects of programming, the correct procedure to adopt is to follow a well-defined set of rules designed to minimize such errors. The rules may form part of the programming system, or they may be built into the hardware of a machine. In either case, in what follows they will be termed *addressing rules.*

A simple addressing rule, for example, would be to assign a fixed location number $L(x)$ to each variable x occurring in a routine. Whenever the value of x was required as an operand, use of location number $L(x)$ would be sufficient to call it out of storage. Whilst this rule is satisfactory for elementary routines and simple operating systems, it is well known that it fails to meet present-day requirements for a number of reasons. Amongst these are the fact that a routine may be written to operate on blocks of data of varying size, in which case an efficient assignment of location numbers may become difficult; another problem is that most routines are written to be run in conjunction with a set of independently written routines, where a feasible assignment of location numbers for one routine may conflict with one or more of the others.

Independence of location number assignment in programming is achieved to a greater or lesser extent by the use of symbolic references to operands. In its simplest application, this is no more than a means of delaying the actual assignment of location numbers until the routine is read into the machine for execution. More generally, the assignment may be delayed even up to the point in time immediately prior to the use of the specific operand.

In discussing addressing, therefore, a problem of translation may be involved, and care must be taken to distinguish the form taken by a reference at various stages of the translation. We shall term a symbolic reference as used by the programmer a *name*; a reference as it appears in a routine held in store and awaiting execution will be called a *routine address*; the actual information sent to a memory bank to enable it to read out a particular operand is the *location number.*

The class of addressing rules we seek defines a translation from a name to an operand. In the following section on Symbolic Addressing, the desirable properties of names are discussed. In Section 3 a variety of devices for passing from a routine address to an operand is examined: these, of course,

are generally understood to be part of the 'hardware' addressing facilities of a machine. Given the general requirements of naming systems, and a range of possible tools, we then proceed to discuss in Section 4 a variety of addressing rules which have been, or might be, exploited.

2. Symbolic Addressing

1. Program Structures

The use of a name for an operand has two distinct interpretations. It may denote a location number by immediate substitution, as in fact we might imply in our previous example by writing down the symbolic addresses '$L(x)$', '$L(y)$', etc., wherever reference to x or y was implied. In this context, it becomes meaningful to use a variant of the symbol, e.g. '$\#L(x)$', meaning the integer location number associated with x, and to use this in arithmetic operations, e.g. '$\#L(x) + 2$', to derive in effect further location numbers of data. More generally, however, immediate substitution is impossible since $\#L(x)$ may in fact vary without the knowledge of the programmer, and consequently only restricted forms of address arithmetic are possible. The second interpretation of a name is independent of the location number concept, and here it is used merely to denote an operand whose value is to be substituted for the name upon *execution* of the expression in which the name appears.

Unless stated otherwise, we shall consider names from the second point of view. This is obviously more satisfactory when the needs of problem-oriented languages are taken into account, since it avoids the introduction of a concept foreign to the problem. Only when the problem-oriented language falls short of the ideal is it necessary to introduce the idea of a location number.

In a program written in symbolic code, the obvious intent in using a name 'A' is to denote the current value of the same operand A upon executing various parts of the program. This apparently simple requirement has, however, been found to fail in only moderately complex programming situations. One common cause of failure, for example, is that a program may be written by several individual coders, who must be protected from using the same name for different operands. Another, less serious, difficulty is that different names may be used, on occasion, for the same operand. In fact, we are quite short of names, and want to use them as efficiently as possible.

It has therefore become necessary to look at the program as a whole, and give rules for determining the extent of text over which a name may be assumed to refer to the same operand: such text is called the 'scope' of the name of ALGOL terminology. Clearly the rules which are given must introduce as few arbitrary boundaries in the text as possible, and one looks to see what natural divisions of the text can be used for this purpose.

The most natural boundaries to choose are those of the routines which may be embedded in the program text: natural because they are usually very clearly marked, because they define logical units at least partly independent of the rest of the text, and because they depart from the sequential ordering of execution which is normally implicit in the text as written. The routine

boundaries are indeed chosen as scope delimiters in most programming systems; in ALGOL a finer scope structure is chosen, down to the level of *blocks* within routines (procedures). The latter extension has few advantages, and will not be discussed further here since it has no new implications in the problem of addressing.

A peculiar property of many logical units of a program text, however, is that they themselves may contain instances of similar logical units, and so on *ad infinitum*. This is true of routines, which may contain subroutines, etc. Having decided on using routine boundaries as scope delimiters, the problem arises: should the scopes of names be nested one within another in the same way as routines, or should we accept only the highest level routines for the purpose of defining scope? For practical purposes, it should be remembered that we rarely want to go to a depth beyond one or two routines, so the choice is not of major importance. One would be tempted, therefore, to choose the ruling with the greater mathematical nicety, and this in fact is what has been done in ALGOL. In FORTRAN II, on the other hand, only a single depth of routines is allowed and hence scopes are not nested.

Obviously, it is not sensible to cut through all scopes at every routine boundary since the continuity in meaning of names is necessary to give coherence to the program as a whole. One must distinguish, therefore, at (say) the *entry* to a routine, just those names whose scopes are bounded at that point and, by implication, at the corresponding *exit* point of the routine: the remaining names have continuity of meaning through the routine boundary. Such distinctions may be made by implication in a particular programming system, often supplemented by declarations at the beginning of the routines. We shall not detail here any particular methods of giving this information: once aware that they exist, it is usually easy to spot them.

Again adopting ALGOL terminology, we shall say that if the scope of a name A extends to the boundaries of routine R but not beyond them, then A is *local* to R. Within the text of R there may be a subroutine Q: if A occurs in Q and the scope of A extends from R into Q, then A is non-local to Q; on the other hand, A may not occur in Q, or it may be defined as local to Q, and hence distinct in meaning from the name A local to R.

Once the extent of text over which a name has a consistent meaning has been defined, there is usually no need to retain the program text in its original form. For further discussion it will be convenient to detach it into its separate routines, and consider the properties of a typical one of these, R. From what has been said already, it should be clear that any name appearing in the program text is local to one and only one routine.

Let R contain the local names $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_n$. Any of these, say Q, may stand for a routine with local names, $\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_m$. Let us suppose, moreover, that the name of a routine is always local to the routine in which it is immediately enclosed. In this case, the name relationships in a program can be represented in the form of a tree (Fig. 1) in which each routine name represents a node from which grow branches corresponding to its local names. The value of this Figure is that it indicates exactly those operands which may be referred to in a given routine: namely those local to the routine,

or accessible to it in the tree by downward and horizontal paths only, with the proviso that if any name is encountered two or more times in this process, then the first occurrence is always taken to define the name. In the Figure, reference in Q may be made only to operands named in the following set:

$$\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_m, \mathbf{R}_1, \mathbf{R}_2, \ldots \mathbf{R}_n, \mathbf{T}_1, \mathbf{T}_2, P \qquad (2.1)$$

e.g. not to $\mathbf{U}_1$ or $\mathbf{U}_2$. The set of names which have meaning within a routine will be termed its *context*.

Our discussion so far has been based on the 'static' program description. It is well known that certain local names in a routine are not used to denote the same operands at all times, but are destined to be given meaning each time the routine is *activated*: these are the *parameter names*. In general they are defined by the calling routine, which gives an expression to be evaluated each time the parameter value is required—some important subclasses of definition exist ('call by value', 'call by simple name'), but in the interest of generality we shall consider only parameters defined by expressions in the context of the calling routine. Other names denote not operands but *functions* which are to be used in obtaining the value of an operand. The representation of a function is a routine either in the same program or in an assumed 'library' of common routines. A function name may also be a parameter name.

The difficulties raised by parameters and functions are, of course, aspects of the same problem: the dynamic sequencing of program execution. Only rarely does the static program description bear any similarity to the hierarchy of routines which is built up during the process of applying a program.

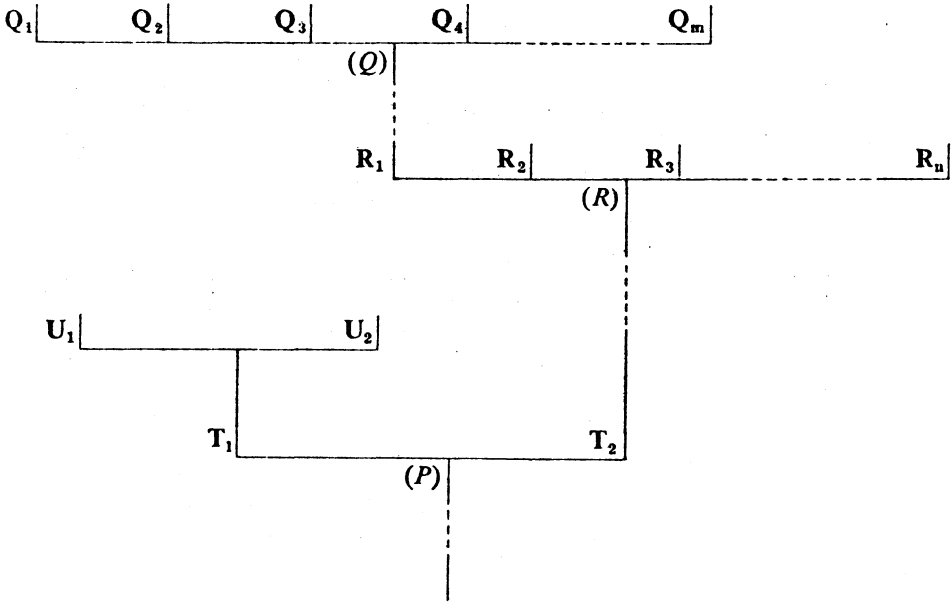

Fig. 1

Consider, for example, a situation in which Q is active (Fig. 1). The context (2.1) is defined, and we may envisage a situation in which from within Q the routine T_1 is activated, its parameters being defined by means of expressions in context (2.1), and control passes to to T_1. At this point, the context becomes:

$$\mathbf{U}_1, \mathbf{U}_2, \mathbf{T}_1, \mathbf{T}_2, P \tag{2.2}$$

Thus execution of T_1, whilst normally based on (2.2), may involve reference to (2.1) whenever a parameter is called for. Clearly, one of the $\mathbf{Q}_i$, $i = 0, 1, \ldots, m$ may itself be a parameter of Q, so a further change of context may be required when evaluating a parameter expression, and so on.

The situation is still further complicated by the fact that Q may directly or indirectly activate itself. In our example, T_1 may activate T_2, and thence R and Q. One might ask, on entering Q for the second time although the first activation of Q is not completed: are the variables local to Q to retain their values from the first activation, or should they be defined anew? A case may be made for either ruling, depending on the intentions of the programmer, and the greatest generality is achieved by partitioning the local variables into two groups: those whose identity is preserved through all activations of the routine (the **own** variables of ALGOL), and those which are redefined on each activation.

It remains to be said, when working in a given context, which activation of the routines defining the context should determine the variables referenced at a given time. When a routine R is activated from a context C, it should be clear from our previous remarks that a new context C^R is determined, comprising a subset C of C plus the local variables of R. The activations referred to in C are by definition the same as those referred to in the corresponding portion of C; this activation of R creates a new set of non-own variables, which is destroyed when the corresponding termination of R is encountered. With this information, and an initial context C^0 which is supplied by the programming system, it is possible to follow through a program and determine without ambiguity the correct context at any point in time.

2. *Data Structures*

A brief note should be made of the logical structure of data which is used in a programming system. Whereas the complexities of program structure generally derive from the requirements of scientific systems, complications in data structure derive from commercial demands or, quite often, from system programmers themselves.

Elementary operands are defined as numbers, words, character fields, etc. These may be grouped together to form *records*. Records may also include other records. Thus a name A within a routine may denote a single elementary operand, or a set of elementary operands and records. Some operations may be performed on A as a whole. Others require access to individual items within A; this may be achieved either by naming them and using an appropriate *compound name* in the routine ('*AGE OF EMPLOYEE*' etc.), or by using a suitable numerical index system ('A_i' etc.). The first method involves some

elaboration of the idea of a context as developed above. Either can be considered as extensions of the tree referencing system (Fig. 1) allowing restricted paths to be taken *up* the branches as well as down.

Unlike routines, data structures may vary quite freely during the execution of a program. Records of various sizes may, for example, be read from a file on magnetic tape, and some means must be provided for detecting whether or not the last item in a record has been processed or not. This leads us to note, in passing, that actually naming an operand is not the only way of referring to it, for when the data possesses a certain structure one may obtain an operand by specifying its position in the structure relative to that of another item: indexing may be taken as an example of such referencing, and the various operations of list processing yield further examples. One must distinguish rather carefully, however, between referencing based on the logical data structure, and that based on the memory structure on which a program is realized. For obvious reasons, the first has a certain invariance, or is under control of the programmer, whilst the second is under control of the operating system and may vary arbitrarily during the course of a program.

Much of data processing is concerned with transmitting data from one medium to another, often with a change in structure. The programming system may require this to be done entirely by writing the appropriate routines, or by allowing the use of *data descriptions* which are used by the compiler and possibly by the operating system whilst a program is running. The latter possibility seems to be generally more satisfactory, since data structures are often the most fluctuating parts of the problem.

3. The Operating System

A complete problem description generally consists of a program, a set of data descriptions, and a set of data. The regime under which these three items are processed constitutes the operating system of a machine. Fig. 2a illustrates the flow of information we have in mind, when the operating system is conceived as one with the machine. This simple picture, however, is the source of many troublesome requirements of an addressing system.

For in most programming languages the degree of complication is such that an initial phase of program translation is necessary: certainly the names must be translated into routine addresses. The translation may or may not involve the data descriptions. This scheme is represented by Fig. 2b, in which 'translation' and 'execution' are two separate processes, probably on the same machine. Typically, program text is supplied to the translator, data is supplied to the executive machine, and data descriptions may be supplied to both. It now becomes an objective of system design to minimize the total time spent in the translation phase of any one program, including reruns due to program errors, changes in the problem requirements, and so on. The horns of this dilemma are quite well known: a fast compiler produces a slow-running object program, and vice versa. One way towards a more efficient operating system is to segment the translator into a number of separate stages which follow one another, such that those parts of the problem description most likely to be subject to variation are only taken into account

at the later stages of translation. In this way, a minor change in the problem may involve only repeating part of the translation, or repeating the translation for only part of the problem.

This is the situation in the scheme based on a loading routine (Fig. 2c). Here the program is segmented into routines which are translated separately,

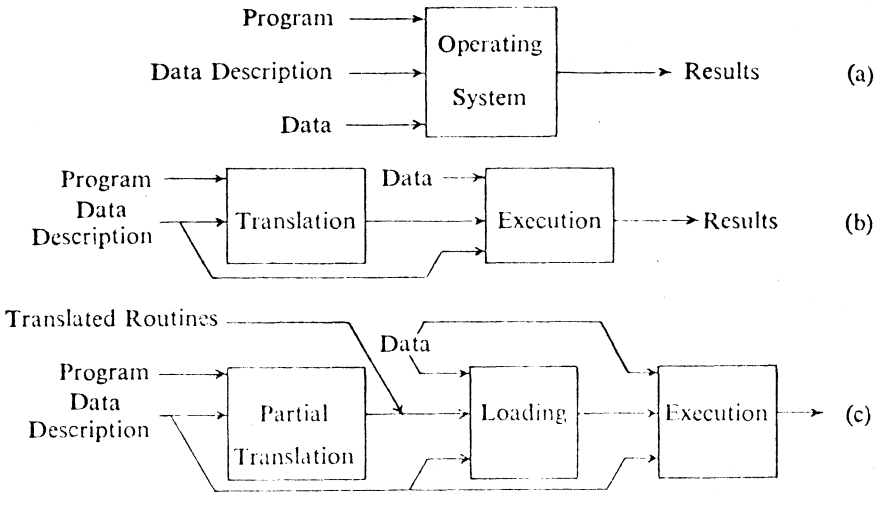

Fig. 2. Elementary operating systems.

but combined together by a loading routine immediately prior to execution. One of the main tasks of the loader is to establish the correct context, in the sense discussed above, for each routine as it is loaded. We shall return to this problem in Section 4.

To summarize our findings up to this point, we may say that the use of a name in a program text has been found to involve following certain rules which determine its scope. Conversely, within any routine of the program a certain set or context of meaningful names is defined. When a routine is obeyed, one must be sure to establish the correct context for each routine, which means, when recursive use of routines is involved, establishing reference to the correct variables in each activation of each routine. Use of a name may involve not only direct reference to an operand, but also reference through a data structure along a path chosen by one or more modifiers of the name (which may themselves be named); or reference to an expression (parameter) which is to be evaluated in another context, or again, reference to a procedure (function) whose evaluation will eventually lead to the value of the desired operand. Finally, any chosen realization of names must allow for the types of modification which are most frequently made in programs or data descriptions. Before examining possible solutions to these problems, we must examine the techniques at our disposal.

3. Routine Addresses

Some years ago, in order to refer to a certain item of data in memory it was considered sufficient to allow the programmer to place a location number in the instruction word of his program. In our sense, the routine address was identical with the location number. That these days are past is not so much due to the demands of symbolic addressing outlined in the last Section as to the observed patterns of access to storage in a large selection of programs during execution. A major question, which remains largely unanswered at present, is the extent to which patterns of access to storage have changed as the result of the growth of symbolic programming techniques, either because of the change in style of translated programs, or because of the demands of the translators themselves. In spite of changes in demand, however, the basic economics of storage which have dominated computer design in the past seem likely to do so in the future, so we shall continue to be faced with realizing a problem description on a hardware configuration which is at first sight ill-adapted for this purpose.

Let us list, first of all, some of the forms which a routine address may have.

1. Implicit Address

Certain functions of the machine involve registers which are not explicitly given in an instruction. For example, the Accumulator in a single address machine, or certain Indicator registers.

2. Truncated Address

The full range of location numbers may demand a field length of 15 or more bits to specify an operand. Part of the store can be addressed by a shorter field if an assumption is made concerning the value of the missing bits. Locations 0–31 of the store may be addressed through 5 bits, for example, if the remaining bits are always assumed to be zero.

3. Relative Address

A routine address may be added to a pre-assigned 'base address' in order to determine a location number. This technique is often combined with method 2.

4. Modified Address

This is similar to the relative address, except that the base address is contained in a modifier register (B-register, index register) selected by part of the routine address. A variant of this technique places the base address in the instruction and the relative part in the modifier register.

5. Block Address

This is similar to truncated addressing, but the high order bits of the location are obtained from a register selected by a block number, which is given as part of the routine address.

6. Detached or Indirect Address

A routine address may be used to indicate a register whose content is to be used as a routine address. A B-modifier used alone is a special instance of a detached address. The facility may be extended to any register in the machine.

7. Immediate Address

The routine address, or part of it, may be used 'immediately' as an operand.

8. Associative Address

This is the only instance in which the location selected depends on the value of its content. In an associative memory, input of a bit pattern and a mask results in the selection of one or more locations in which the stored information coincides with the given pattern in the positions determined by the mask. Such a memory is potentially very useful in certain applications, but it has not yet reached the point of large scale economic use. The Atlas page-address registers may be regarded as a very special form of associative memory, although in this case the output is a selected location number.

* * * *

Thus the object of machine addressing, which is to determine an operand from a routine address, is pursued in a variety of combinations of *addition* and *substitution* operations; the sequence of operations may be fixed rigidly in the hardware of a computer, or it may be allowed to vary according to information found in routine addresses. In the latter case, this information must indicate whether the present routine address gives the operand, or the location number of the operand, or whether it is to be used to give a new routine address. As usual, the penalty of increased flexibility is the provision of selection bits in the routine address, but this is a cost which it has been found increasingly worthwhile to pay. On the other hand, machine addressing systems which require appreciably more time than a single main memory cycle to obtain an operand require careful justification.

We may distinguish trends in machine addressing systems towards four main objectives, to which we shall return in Section 4 when we examine overall system requirements. These trends are as follows.

1. To increase the 'information content' of an address, and hence the efficiency of instructions.
2. To increase the mobility of programs, so that storage allocation is simplified.
3. To allow efficient use to be made of the more expensive (i.e. high speed) parts of the store.
4. To permit flexible communication links to be set up between routines and data.

Problems of economy in storage have led to many further complications in machine addressing. We have already noted that a given range of location numbers may cover different levels of store of varying access speeds. There

may in addition be different location numbering systems associated with different categories of storage (e.g. B-register number 3 and location number 3 may mean different registers in the same machine). One of the greatest problems is that part of the storage system may not be directly addressable, and the efficiency of a problem realization which exceeds the capacity of addressable store then depends critically on the way transfers of information to and from the 'backing store' are organized. Such transfers are generally restricted to blocks of words: the longer the block length the less time, per word, is spent in the mechanics of arranging for data transfers to take place, but the more difficult becomes the problem of efficient use of store.

In contemporary machines, the addressing system must also permit the addressable store to be shared by several activities (programs, backing store transfers, peripheral transfers, etc.) at once, and prevent one activity from interfering with another. This can be achieved either by 'locking in' each activity to its own region of store or 'locking out' each activity from all others —a combination of the two methods may be used. In practice, therefore, any location number obtained during machine addressing may be subject to one or more comparisons with boundary markers before it can be used, and a number of high-speed registers must be deployed to contain the location numbers limiting the regions into which the store is divided.

To summarize the present position in machine addressing, we have noted that a series of memory devices of increasing size but decreasing speed is likely to remain the most efficient basis for a computer store. In the past, order codes have been arranged so that the hand coder could take maximum advantage of the different levels of store. More recently some of the high speed registers have been used for specialized purposes such as lock-out registers, block or page-address registers, and so on. As computer programs increase in size and complexity more and more of the storage allocation problem will have to be handed over to the programming system. The problems which will have to be solved automatically relate to the use of high-speed registers, the presentation of a core store and one or more levels of backing store as an effective 'single level store', and the mutual protection of several activities using the store simultaneously.

4. Translation Processes

Let us formalize the system of names used in a program P in the following way. Let R be a routine in P. Denote by $\mathbf{R}$ its set of local names, i.e. $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_n$, where n is a number dependent on R. Let the first $l (0 \leqslant l \leqslant n)$ of these names stand for parameters, and let the next $m (0 \leqslant l + m \leqslant n)$ stand for non-own variables; the remainder (if any) stand for own variables.

Let R be local to the routine λR, i.e. R is a name in the set $\lambda\mathbf{R}$; more strongly, assume that R is an own variable name in $\lambda\mathbf{R}$. Further assume the name λR is local to $\lambda(\lambda R) = \lambda^2 R$, and so on. Then the context of names defined in R consists of the union of sets:

$$\mathbf{R}, \lambda\mathbf{R}, \lambda^2\mathbf{R}, \ldots, \lambda^k\mathbf{R} \tag{4.1}$$

where $\lambda^k R$ is the highest level routine in P.

In general, corresponding to each routine R and set $\mathbf{R}$ of local names there will at any time be defined a series of sets of operands, each corresponding to one of the current activations of R. The number, h, of such sets depends not only on R but also on the time at which this observation is made. These sets of operands will be denoted by $\omega^{(1)}\mathbf{R}, \omega^{(2)}\mathbf{R}, \ldots, \omega^{(h)}\mathbf{R}$. Each set has the same number of elements, and we know (by definition), that the operands corresponding to the subsets of own variables are identical between any two sets in the series. Since similarly defined sets exist corresponding to each activated routine, the actual operands corresponding to the context (4.1) of a particular activation of R can be described as the union of the following sets:

$$\omega^{(j_0)}\mathbf{R}, \omega^{(j_1)}\lambda\mathbf{R}, \omega^{(j_2)}\lambda^2\mathbf{R}, \ldots, \omega^{(j_k)}\lambda^k\mathbf{R} \tag{4.2}$$

where the index j_i selects the particular set of operands corresponding to $\lambda^i\mathbf{R}$ at this time.

When R calls a new routine, it must be a member of one of the sets in (4.1), say T in $\lambda^i\mathbf{R}$, $0 \leqslant i \leqslant k$. When control is transferred to T the new context of names will be

$$\mathbf{T}, \lambda^i\mathbf{R}, \lambda^{i+1}\mathbf{R}, \ldots, \lambda^k\mathbf{R} \tag{4.3}$$

Corresponding to these are the operands

$$\omega^{(h')}\mathbf{T}, \omega^{(j_i)}\lambda^i\mathbf{R}, \omega^{(j_{i+1})}\lambda^{i+1}\mathbf{R}, \ldots, \omega^{(j_k)}\lambda^k\mathbf{R} \tag{4.4}$$

i.e. a new set of operands $\omega^{(h')}\mathbf{T}$ is activated for $\mathbf{T}$, and the operands common to the contexts of T and R are retained.

When T calls for parameters defined by R, reference is made to an expression using terms in the sets (4.2). When T terminates, the context (4.1) of R is re-assumed, with the operands (4.2). It should be noted that although T cannot change directly any operand named in the sets $\mathbf{R}, \lambda\mathbf{R}, \ldots, \lambda^{i-1}\mathbf{R}$, (since it has no name for them), it may change the values of operands named by $\lambda^i\mathbf{R}, \lambda^{i+1}\mathbf{R}, \ldots, \lambda^k\mathbf{R}$, an effect which may or may not be considered desirable ('side-effect' in ALGOL).

The discussion in Section 2 showed that each defined operand may be an 'elementary' item of data, or a structured piece of information with named sub-items or numerically indexed lists of sub-elements. With regard to the 'fine structure' of elementary operands, we remark that once a word has been read from memory, an item of data may be retrieved from this by further manipulation. Some machines include special functions in the order code for packing and unpacking data and addressing sub-elements of words. Analysis of such operations is beyond the scope of this discussion, which is primarily aimed at giving rules for deriving, from a name, single word units of information from storage. We have to take some account, however, of the extent to which data structures may vary during the course of a calculation.

The foregoing paragraphs set out the broad requirements of a naming system. We shall now discuss various aspects of its realization on the machine configurations of Section 3, and consider what modifications to our general requirements are necessary to attain efficient program execution.

1. Mapping onto Addressable Store

Consider the execution of routine R, with defined operands (4.2). Suppose that the addressable store is sufficient to contain representations of most, if not all, of the operands in the context of R. If the store is composed of several units of differing access times, the problem arises of how to dispose the operands of R in the best possible way in the store. More exactly, we may state this requirement as that of minimizing the contribution to program running time in a certain time interval, arising from accesses to the store in the same interval. Efforts to solve this problem depend very much on the chosen time interval; it may be a fixed period in the activity of a machine, or the time devoted to executing R, or the total time of program execution for a given set of data, or the time accumulated in translating R and running it for many sets of data with certain characteristics. The basic information necessary to formulate a solution is the sequence of access to operands over the chosen time interval.

In FORTRAN, for example, much effort might be spent in optimizing the use of B-registers over the execution time of a given program. The necessary basic information is derived in part from an analysis of the program and partly from extra statements which may be made by the programmer; assuming the accuracy of this information, effectively optimized programs may be generated. On enlarging the time interval of optimization to cover the compilation time as well, it has often been found that gains in execution time have been more than offset by the increased time of compilation.

Another device which has been used when a small number of fast registers is involved is to name these and include them as own variables local to the highest level routine in the program (Ref. 1). This technique applies only to single word operands. In this way the task of optimizing is left in the hands of the programmer, though he is still free to use problem-oriented input languages. The difficulty of introducing fast-register names local to any but the highest level routine is that the task of saving and restoring them when control is switched from one routine to another can easily annul any speed advantage gained from their use.

One is frequently asked: given a machine with a main core memory but with an additional memory of, say, $\frac{1}{10}$ the access time, how much fast memory should be at the disposal of a compiler? It is probably fair to say that without outside help the number of fast registers a compiler can use efficiently for named operands is quite small. Allocation within a short series of statements is comparatively easy and pays good returns with up to four or five words of fast store; allocation over loops and between different routines requires much more work and often involves placing undesirable restrictions on the source language. An average compiler would find six to ten fast registers all it could cope with. The fact remains that a good programmer can make effective use of many more fast registers in certain classes of problem, and if a machine has them he should not be prevented by the source language from using them. The technique of including them in the local names of the highest level routine is to be recommended.

One aspect of machine addressing systems that is not desirable from the point of view of translation is the possibility of two or more different ranges of location numbers. We have already noted three distinct classes of local variables: if each of these can be translated into several classes of storage, compilation quickly becomes intolerably complicated. The ideal addressable store is given by a single unbroken range of location numbers, a part of which, at least, may correspond to a series of fast access registers.

2. Mapping onto an 'Apparent' Single Level Store

If addressable store is insufficient for a complete problem representation use must be made of a suitably sized backing store of drums, discs, etc. As in the case of optimization of high speed store, there are three main lines of attack on the problem associated with a backing store.

(i) Structural, i.e. by building into the source language commands to be used by the programmer for transferring data to the addressable store.

(ii) Analytical, i.e. by allowing the translator to examine the flow of the program and make the decisions on where transfers should be made. As with other analytical methods, this demands for success either complete and accurate information on a problem or a series of lucky guesses, neither of which can be guaranteed to a programmer.

(iii) Dynamic, i.e. the assignment of transfers as they are required during the running of a program. This may hold up the program, but if another can be run until the transfer is completed the actual time wasted may be negligible.

Most first generation programming systems make structural provision for organizing transfers of information to and from backing stores. Present trends of large machine systems indicate that even if a programmer can confidently predict the storage demands of his own program, it will be executed in conjunction with others of which his knowledge is zero. Clearly, if efficient storage control is to be attained over a mixture of programs sharing a machine, the only possible source of control is the supervisor program. The philosophy of dynamic storage allocation has been adopted on many machines of the present generation, an example being the Ferranti Atlas, where the page-addressing system is specifically designed to allow flexible allocation of space over drum and core stores (Ref. 2).

There are two primary requirements of dynamic storage allocation schemes: (a) that from a routine address it should be quickly deducible whether an operand is in (say) core or drum store; (b) that the information representing a program should be partitioned into sections which can easily be relocated individually to any part of the store, without causing major re-addressing operations to take place. Requirement (b) finally removes the possibility of location numbers being used as routine addresses: at least one stage of addition or replacement is essential, and at the same time a check can be made in the presence of the operand in core store. If it is present, the program should proceed with no appreciable loss of time; if it is not, a transfer of the data from backing store must be arranged.

We noted earlier that the time taken by a backing store transfer can be reclaimed by switching control to another program or another part of the

same program. The time taken to organize the transfer, i.e. finding vacant space in the core store, which may involve reorganizing data in the core or transferring data from core to drum, represents a positive loss in computation time which must be balanced against overall system efficiency. A division of the store into blocks of fixed length seems to offer the best possibility for organizing inter-level transfers efficiently.

An alternative division of the store, based on the natural lengths of blocks into which representations of routines and data fall, has been implemented (Ref. 4). The advantage of this technique is that it makes for efficient packing of core store, and it solves at the same time some of the problems associated with storage maps (see (*4.4*), below).

3. Reservation and Lock-out

It was noted above (*4.2*) that between a routine address and the corresponding location number at least one substitution or addition must take place if a routine is relativized, and at the same time a check may be made on the presence of the operand in addressable store. It is possible to make further tests at this point to ascertain that protected regions of store are not being violated.

Storage protection schemes fall into two main groups: 'block reservation' and 'limit register'. In a store organized for coping with dynamic storage allocation problems, the blocks into which the store is divided may be individually protected at the expense of one bit in the word through which the block is addressed, and sufficient hardware to detect its status. In the case of fixed length blocks, choice of a length which is some power of 2 automatically (in a binary address) avoids the possibility of exceeding the limits of the block. In the case of variable block sizes, having selected a block and checked that its use is permitted, a further check must be made to ensure that the element selected lies within the block: comparison of an index value with a given block length is implied.

In the absence of a given storage block structure, limit registers may be used to indicate which storage areas are within bounds. These are pieces of hardware which are liable to be expensive, and once again complete generality must be relaxed in order to get an economical system. Ideally, one would like to have *available* to a routine those parts of store occupied by its operands (e.g. corresponding to (4.2)), *less* those regions engaged at any time in other activities with higher priority, e.g. drum and peripheral transfers. In practice, a routine may only have a single reserved region available (into which (4.2) must be mapped), and a fixed number of limit registers which can be used to lock out subsidiary regions involved in other activities. If the latter are insufficient, either the routine must be held up, or the activity can proceed at the programmer's risk, or the supervisor may be able to allocate space to the activity outside the reserved region.

Obviously there are various *degrees* of protection which can be offered by the programming system. The minimal practical requirement is that a routine should not interfere with the supervisor itself. Beyond that, it should not be able to refer to any operands outside the program in which it is used. More

strongly, it should not be able to refer outside the operands (4.2) defined by its context.

There is, however, an even stronger possible degree of protection. For within (4.1) we can list (at translation time) all operands which are *actually* referred to by R (this is normally a proper subset of the context of R) and arrange that reference to any other operand is impossible. This strong form of protection is in some respects the easiest to effect, for one can guarantee *during translation* that certain routine addresses refer during execution to operands known to be present (i.e. guaranteed by the supervisor) in specific parts of the store. This is particularly true of simple operands and known elements in arrays of known structure (e.g. $A_{3,4}$ in a 10×20 matrix A): it is not possible when unknown data structures are involved, or when named subscripts are used (e.g. $A_{i,j}$). Similarly, the assembly of a routine generally ensures that control cannot pass accidentally outside the routine area and further hardware checks on the content of the control register are superfluous (except in the case of transfer through a 'switch vector'). This type of protection has long been an implicit benefit from symbolic assembly systems: its potential application in overall system design has perhaps been neglected when considering protection problems in time-sharing machines.

4. *Storage Maps*

We return to the problem of representing the operands of a routine R in what appears, at least, to be a homogeneous, random access, store. Let Π denote at a given time the totality of operands defined in a program. Of Π, the sets:

$$\alpha = \bigcup_{i=1}^{h} \omega^{(i)}\mathrm{R} \text{ where } \mathrm{h} = \mathrm{h}(R)$$

and

$$\beta = \bigcup_{i=0}^{k} \omega^{(i_i)} \lambda^i \mathrm{R},$$

constitute subsets. Here α denotes the union of local operands of each current activation of R, and β denotes, as in (4.2), the context of operands defined in a given activation of R.

Our discussions have shown that when control passes from R to a new routine T, a new set of operands local to T is defined. Hence, from (4.4):

$$\Pi' = \Pi \cup \omega^{(h')}\mathrm{T} \tag{4.5}$$

When T terminates, we have:

$$\Pi'' = \Pi \tag{4.6}$$

Clearly, therefore, Π can be represented as an expanding and contracting list of operands in which items are only added and deleted from one end. Provided, once defined, operands do not vary in structure, it is then possible

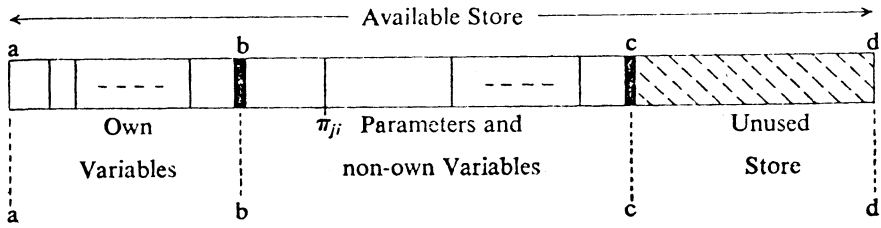

Fig. 3. Stack organization.

to represent Π in store by the familiar 'push-down' list or 'stack' (Fig. 3). We note that own variables local to R are common to all the sets $\omega^{(i)}$R, and hence may be stored more efficiently in a permanent section (*ab*) at the head of the list. In the same section *ab*, representations of the routines themselves can be stored. If the limit *c* of used store exceeds the limit *d* of available store, this storage mapping fails, but this happens sufficiently infrequently to allow the stack to have been applied successfully and elegantly by many designers (Ref. 3).

Several programs can share a store if it is divided appropriately into regions of the type *ad*. In terms of routine addresses, it is clear that own variables can be referenced relative to the base *a* (or *b*); other variables must be addressed relative to an appropriate base in the region *bc*. For, let $\pi^{(j_i)}\lambda^i$R denote the subset of $\omega^{(j_i)}\lambda^i$R consisting of its parameters and non-own variables, which is therefore represented in a section of the region *bc*. Let π_{j_i} denote the starting location of this section. Then to refer to the *n*th word in this region the pair (π_{j_i}, n) must be given. But π_{j_i} is unknown to the translator, which must give the pair (i, n), the actual value of π_{j_i} being derived at execution time by reference to a table (the 'display' if Dijkstra) which is maintained by the supervisor program. Evidently, a change of context ((4.2) — (4.4)) can be achieved simply by changing this tabular information.

The stack concept is not incompatible with a two-level store organized on a page-addressing system: after the address corresponding to (i, n) has been derived, the page-address comparisons can be made. There are, however, a number of defects to the 'pure' stack.

1. It is difficult to share two or more activities within the same program, since one may get out of phase with another and wish to overwrite a part of the list Π which is still in use.

2. Data structures cannot easily vary in size, once defined.

3. Rather more information than is strictly necessary is kept in the addressable part of the store.

None of these difficulties makes use of a stack intolerable, but their removal is of more than academic interest in large machines.

Some progress is made by maintaining an 'index' with one entry for each distinct operand used in the program. At a given time, the index entry will give the location number (or its equivalent) of the current representation of each operand (Fig. 4). When a new routine is entered, new blocks are activated

for its non-own variables, the previous representation being chained on to the new one; when a routine terminates, its last set of non-own variables is discarded, and the set before that (if any) becomes accessible through the index. The use of a push-down list can now be restricted to parameters, link data, and working stores, although it is sometimes also convenient to use this area to contain scalar non-own variables, which can then be referred to relative to a local pointer. All non-scalar items are referenced through the index region and are represented by independent blocks of storage; not only may they vary in size dynamically to suit the programmer, but they may vary in position to suit the requirements of the storage allocation system, provided the index is kept up-to-date.

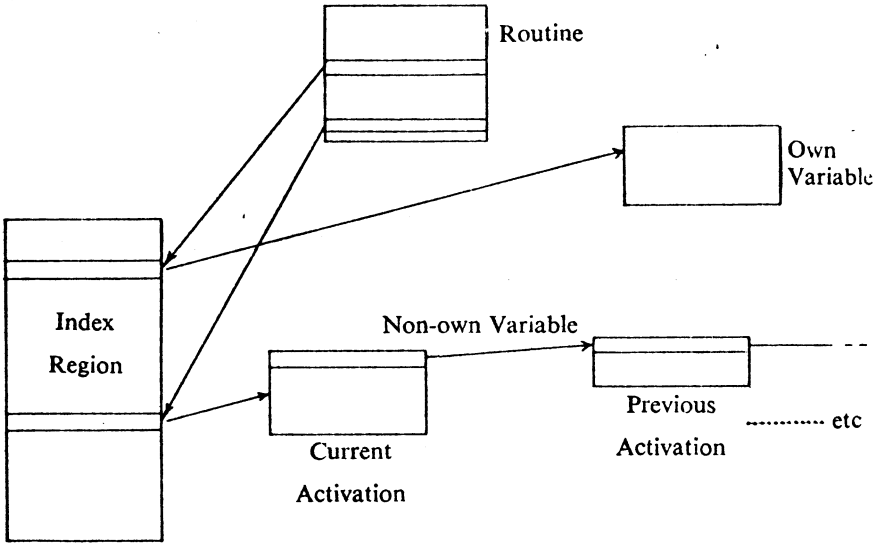

Fig. 4. Addressing through an index region.

It is not difficult to see the similarity between the index region proposed for storage mapping and the set of storage control registers required by a block-addressed reservation scheme. The combination of both functions can be performed by one set of registers in a storage system based on variable block lengths (Ref. 4).

Control of storage through an index removes most of the difficulties left by the stack. Storage areas can be handed over to peripherals for action without interrupting the program activity; data structures can be allowed to vary; and at any given time the only essential items in the addressable store are the current routine, the index region, and the stack. The above representation has to be elaborated slightly to deal with time-sharing branches of the same program, but it can in principle deal also with that situation.

5. Communication

One of the ideals of system design is to be able to add to, delete and modify parts of a program without taking undue time in retranslating the program. We have noted that this is assisted by segmenting the translation process into distinct phases. One of the last phases is concerned with establishing appropriate channels of communication between independently compiled routines and the data on which they operate. If information is relocated dynamically, this phase must link up to the system used in the supervisor.

The most flexible solution to this problem is provided by the index region of (*4.4*). For to any named operand corresponds a unique item in the index, and a routine can be translated (and hence re-translated) independently provided the positions in the index corresponding to its context (4.1) are known. It is rather obvious that if R is re-translated, then any routine local to R must also be re-translated.

Quite often, the actual index positions cannot be supplied at translation time. In this case, a communication region may be set aside at the end of a routine, with an entry for each non-local name used in the routine. It is the function of the loading routine to insert correct references to the index in the communication regions of each routine in a program being set up for execution.

At first sight, FORTRAN possesses a flexibility in communication not given to ALGOL. In fact, independent translation is generally more difficult in ALGOL, but not impossible, and if restrictions are placed on the nesting of procedures at least as powerful and flexible a system can be derived. One can anticipate such restrictions being necessary should a large scale user build ALGOL into a programming system.

6. Pointers

The routine address, as we have described it, 'points' to an operand in a language which can be interpreted by a machine addressing system. It may point to another 'pointer' which is defined at the last stage of translation, or placed in a modifier register by the routine. A list structure, based on the use of pointers, is the ultimate in variability and indirectness. Since at each stage of an indirect addressing chain a location number has to be derived, it is tempting, having reached the end of the chain, to keep the last location number formed (that of the operand) and derive from it the location number of the next operand (e.g. by addition). Such action is dangerous unless the block which is being operated on is somehow 'locked down' into fixed locations by the supervisor program.

In general, therefore, a pointer held in an index register, or in a list element, should have the same form as a routine address; in particular, it must be independent of location numbers.

5. Summary and Conclusions

The conflicts of addressing arising in programming system design derive in the main from the following: (i) the name structure imposed by source

languages; (ii) the requirement to use two or more levels of addressable store; (iii) the desirability of using automatically a backing store as an extension of the addressable store; (iv) the need to share the store amongst several activities; (v) the need to retain programs in a flexible form throughout most of their working lives.

Mere statement of the problems involved is incomplete without some assessment of their relative importance. This is perhaps most easily seen by considering the consequences of dispensing with some generality. In name structure, for example, little is lost by removing the possibility of recursion, and corresponding to the context (4.1) we then have the sets of operands:

$$\omega\mathbf{R},\ \omega\lambda\mathbf{R},\ \omega\lambda^2\mathbf{R},\ \ldots,\ \omega\lambda^k\mathbf{R} \tag{5.1}$$

More simplicity is gained by limiting the degree of nesting of scopes to two levels only. The variables of R are then either local (ωR) or non-local ($\omega\lambda$R): this, in effect, is all that is provided in FORTRAN.

We may also restrict array structures to be invariant during execution of a program, and limit parameter specifications to simple names or values, and once again the majority of source languages in regular use would be within our scope.

Hence the generalization we have attempted serves mainly to signpost land beyond the practically useful territory available to today's programmers, and to classify a few lone trails. Academic programmers may enjoy questioning some of the assumptions we have made: Should the partition of local variables into own/non-own/parameter classes be identical at each activation of a routine? Should the routine itself have the ability to 'generate' names dynamically? Is the 'tree' hierarachy of names adequate for all programming problems? How might it be generalized? The answers to some of these questions have been anticipated by list processing systems, but it should be obvious that in spite of advances in traditional programming methods, we still face list processing across a considerable gap in addressing technique. Ideally, one would like to buy the full flexibility of list processing only when it is essential; future translators should recognize the degree of generality called for in a particular situation. Far too often the steam hammer has to be used on every nut.

The most significant trend in machine addressing is to impose at least one stage of substitution or addition between a routine address and the corresponding location number. To maintain speed, this is done by placing the most frequently used substitutes or addenda in special fast access registers. Besides giving in effect location numbers, these registers may also be used to select a modifier, to indicate the presence or absence of an operand from the addressable store, and to provide reservation facilities. In a data oriented system, i.e. one in which the block sizes in store are chosen to correspond to individual pieces of data, substitution provides further power in the ability to monitor selected operands, and to provide for alternative modes of definition, which is important, for example, when general parameter specifications are considered. Control of storage through an index region seems to offer much of the graded flexibility that efficient programming demands.

REFERENCES

1. J. K. ILIFFE (1961). The Use of the Genie System in Numerical Calculation. *Annu. Rev. Automat. Progr.* **2**, 1.
2. T. KILBURN *et al.* (1963). One Level Storage System. *I.R.E. Trans. Electronic Computers* **EC-11**, 223.
3. E. W. DIJKSTRA (1961). An ALGOL Translator for the XI. *ALGOL Bull. Suppl.* **10.**
4. J. K. ILIFFE and J. G. JODEIT (1962). A Dynamic Storage Allocation Scheme. *Computer J.* **5**, 200.