McCarthy, J. (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", *Comm. ACM*, Vol. 3, p. 184.
Collins, G. E. (1960). "A Method for Overlapping and Erasure of Lists", *Comm. ACM*, Vol. 3, p. 655.
Schorr, H., and Waite, W. M. (1965). *An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures*, Research Rept. RC-1450, International Business Machines Corp.
Comfort, W. T. (1964). "Multiword List Items", *Comm. ACM*, Vol. 7, p. 357.
Ross, D. T. (1961). "A Generalized Technique for Symbol Manipulation and Numerical Calculation", *Comm. ACM*, Vol. 4, p. 147.
Wilkes, M. V. (1964). "An Experiment with a Self-Compiling Compiler for a Simple List Processing Language", *Annual Review in Automatic Programming*, Vol. 4, p. 1.
Orgass, *et al.* (1965). *WISP—A Self Compiling List Processing Language*, Technical Rept. 36, Basser Computing Department, University of Sydney, Sydney, Australia.

Data compression and automatic programming

By A. G. Fraser*

Data compression is defined as the reduction of the volume of a data file without loss of information. Methods of obtaining this effect are considered and the implications for automatic programming systems are discussed.

In certain commercial undertakings it is necessary to process data which is not always of fixed format or size. Commonly, names and addresses of exceptional length are held, although the majority are of only modest proportions. Also a data record may contain the occasional but bulky item of additional information which, for the normal case, is totally irrelevant. Variations such as these do not normally present serious problems in a manual data processing system but they must be viewed with some concern when mechanization is introduced.

Operating costs for an automatic data processing system depend heavily upon the volume of data held and the extent of the computing activity which is required per event. It is usually possible, however, to reduce the volume of data but only at the cost of increased complexity in the computing process, and there are a number of ways in which this is commonly done.

(*a*) *Redundancies*

It may be possible to reduce the volume of redundant information carried by the system, but it may then be necessary to re-compute certain values when they are required.

(*b*) *Coding*

By the use of special codes it may be possible to reduce the size of a data item without loss of information. This method may be worthwhile where the number of distinct values that can be taken by an N-bit item is considerably less than 2^n but its use will probably involve table accesses during processing.

(*c*) *Special cases*

By the use of special representations for special values it may be possible to take advantage of known patterns in the data. One example is the special treatment of zero items and "not applicable" groups. Another example is the storage of only the significant characters in an alphanumeric string or the storage of only the significant members of a multi-occurrence group.

The user must strike a balance between data volume and procedural complexity in order to get maximum return from his data processing machinery. The choice of method is one that can only be made in the light of a full understanding of the nature of the job and the demands to be met. Such a choice can only be made by the user and cannot be made automatically on his behalf without risk of severe inefficiency. However, automatic programming methods could well assist the user once the basic decision has been taken, since the application of these techniques for data handling is a well defined operation.

Compiler source language

Methods of data compression vary but for many there are two clearly defined stages in their application. First, the method is defined by a series of rules which must be obeyed rigorously if the method is to work. Secondly, each and every data reference must be handled in accordance with the rules applicable to the item concerned. An automatic programming system can be of service to its user by rigorously applying such rules. Indeed the mechanical application of rules of this type is almost

* *University Mathematical Laboratory, Corn Exchange St., Cambridge, England.*

essential since program bugs introduced by failure to apply a rule correctly can be very difficult to diagnose. An automatic programming system could also be of assistance with some of the problems which arise when data compression is employed. For example:

The record layout will be a function of the sizes and values of the particular items involved so that diagnostic printouts will require translation. Test data (and even live data) may have to be assembled in compressed form and this also will involve some translation. In a data processing application there will be many files which are used by more than one program. Each program must therefore be sure to assume the same file format and must obey the same compression rules. Space recovery might be expensive if done thoroughly, and the user could well expect to be able to select thorough space recovery for only some of his programs (e.g. "end of the month" runs.) Compatibility between different files will also be required in order to avoid expensive transformations when two files are brought together. Dependencies of this nature can form quite complex inter-relationships within one set of files, and a well-organized programming system could be of immense value by providing the necessary administrative controls.

In so far as data compression methods have been used in general purpose compilers it has been usual to build into the system knowledge of a limited set of methods. The following is a list of some methods which might be used.

1. Item value stored in coded form. Either or both of two translation rules must be specified; one for obtaining the true value and the other for assigning a new value. The translation might possibly be defined in tabular form.
2. Less significant (and/or most significant) spaces (or other) characters suppressed and the item size adjusted accordingly.
3. Less significant (or otherwise identifiable) occurrences of a multi-occurrence group suppressed.
4. Special arrangements made for storing the zero elements in an array (or table).
5. Erased (or otherwise identifiable) form of an item, or group of items, represented in compact form.
6. Item value not stored but obtained when required by evaluating a specified expression.
7. Advantage taken of the knowledge that two data items are mutually exclusive and cannot both be meaningful at any one instant.

One might also consider a system in which the user is able to declare a data compression method of his own. There would be four components to such a declaration:

(*a*) A rule for recognizing an item that is to be handled by the method and a description of how the fixed parameters of the method are to be obtained from the item specification.
(*b*) A rule for space allocation including details of the additional items that are needed for control purposes.
(*c*) A rule for obtaining the value of the item at object time and including rules for obtaining information about the state of the item (e.g. suppressed) if the system recognizes such states.
(*d*) A rule for assigning a value to the item at object time together with details about certain special cases, e.g. how the item can be suppressed explicitly and whether or not it can be updated *in situ*.

A realistic choice of the facilities to be provided in an automatic programming system will show some concern for compiling speed and object program efficiency. No doubt, also, compiler size will be a serious consideration. Indeed, it is most likely that the software designer will limit the range of facilities provided since, as will be shown, the demands on the compiler are heavy and optimization of the object program is not a trivial task. One effective way of simplifying the system is to impose a restriction upon the number, or juxtaposition, of items that employ data compression methods. Alternatively one could abandon any pretence of concern for object program speed when such items are used.

There is one other aspect of compiler design that is brought to the fore when data compression methods are automatically compiled. This concerns the user's awareness of the costs which result from his own decisions. A high level source language may so insulate the user from the stresses of machine coding that he loses all feel for the machine on which he is constrained to operate. In this way a fully automatic system for handling data compression could be turned from an aid to optimization into a weapon with very high end costs. Some form of feedback is essential.

Compiling process

Increased object program complexity is the price paid by those using the methods of data compression which are described above. It is therefore not surprising that the introduction of these methods into an automatic programming system should result in increased size and complexity in the compiler itself. Apart from the extra software which is required to provide the special functions made necessary by the use of compressed data formats, special problems are posed for the construction of the central compiling process.

Several additional factors are brought into play when data compression is used. The most important follows from the fact that item addresses do not remain constant when more than one variable item is permitted in one data record. Under such circumstances the address of one item, say *A*, may vary with each variation in the length of some other item, say *B*. This fact alone can have wide repercussion on the design of the compiling process. For example, in order to give effect to the statement COPY *A* TO *B* it is necessary for the object program to find the length of *A*, adjust the length of *B*

and then find the new position of A before the conventional transfer operation can be obeyed.

Similar complications arise as a result of the introduction of items whose attributes are not constant during the course of program execution. The length of an alphanumeric string, the number of occurrences of a multi-occurrence item and the possibility of a totally suppressed item are all cases in point. Not only must the compiler have an answer to the situations created in this way but the attributes themselves will probably need to be given recognition at source language level.

Another form of complexity is peculiar to those machines which have a fixed word length. The user of such a machine may well attempt to store more than one item per computer word and could reasonably expect the compiler to handle such items, dealing automatically with the necessary shifting, collating and checking for field overflow. To do this the compiler must check for and compile the field extraction and field insertion instructions every time it handles a data reference. Furthermore the store address, as used in the compiled instruction, is no longer sufficient to identify a single item of data. These are matters which concern the central operations of any compiler.

Field access, checks on item status and address calculation are the three principal activities which are automatically inserted into the object program in order to provide a data compression capability. Each may be compiled for every item reference and it is probable that some of this coding will be redundant. Where this is the case one must consider the possibility of including some form of optimization during compilation. Consider, for example, the statement COPY $X + Y$ to Z. If X, Y and Z occupy subfields of different computer words then collating and shifting would be required. The simple approach would be to shift into a standard position before adding and then shift again before collating into Z. Clearly, however, no shifting need take place if X, Y and Z were identically placed in their respective computer words. Furthermore if X and Y were contained within one group G whose position could vary then, in the simple approach, the address would be calculated twice, whereas this could have been avoided. Similar effects arise in relation to checks on item status which may be compiled into the object program.

Optimizations of the type described above can take two distinct forms. In the above example the redundancy was contained entirely within the scope of one source language statement. Similar redundancies can arise within a string of statements even though there is no apparent redundancy when each statement is considered separately. To handle this latter type is a far more exacting task for any compiler, since it involves a process in which the compiler seeks to discover the strategy of the program that it is compiling, a process which is not always possible and nearly always expensive. It involves following the possible control paths through the source language program and noting where the user has referred to items of interest. If an address calculation, for example, is to be omitted then the compiler must be sure that the object program will continue to operate properly whatever path the program takes. This type of analysis is sometimes carried out in connection with repeated operations on arrays, but it is unfortunate that commerically useful data records are unlikely to be as simply constructed and the interrelationship between items can be quite complex. Any solution is almost certain to involve what amounts to a simulation of the item status calculations in sections of the object program.

Conclusion

There is clearly a place for some form of data compression in programs compiled for use in a commercial environment, and it also seems clear that techniques of automatic programming could be used to advantage in applying data compression methods. Unfortunately it is far from clear that this can be done without significant loss of performance both at compile time and in the object program.