Building a mobile programming system

W. M. Waite*

* Department of Electrical Engineering, University of Colorado, Boulder, Colorado, U.S.A.

The techniques of abstract machine modelling and macro processing can be used to develop programs of great mobility. This paper describes the concepts and construction of a system which has been implemented on nine different computers. In no case was more than one man-week required, and most implementations went more rapidly than that.

(Received June 1969)

1. Introduction

The *mobility* of a program is a measure of the ease with which it can be implemented on a new machine. A user of a highly mobile program suffers minimum disruption of his work when his computer is upgraded or he moves to a new installation. Because his program is mobile, only a small amount of effort is required to get it running on the new machine before he can continue using it. Applications programs written in high level languages such as FORTRAN, ALGOL or COBOL have reasonable mobility, provided that the author has taken some pains to avoid local idiosyncrasies. For systems software, however, the picture is much grimmer. There have been many attempts to devise high level languages for compiler writing (Feldman and Gries, 1968) and a few to use such languages for the production of operating systems (Glaser, Couleur and Oliver, 1965). Unfortunately, none of these attempts has yet met with widespread success.

A FORTRAN user requires a large programming system, consisting of the compiler and associated runtime routines. The mobility of his programs is determined by the number of installations which support this system. One can therefore argue that the mobility of a program is completely dependent upon the mobility of the programming system on which it rests. In this paper, I shall discuss a technique which I believe is fundamental to the improvement of programming system mobility. The background and general concepts from which the technique was derived are presented in Section 2, while the remainder of the paper is concerned with a specific system which has been designed and built using this technique. Section 3 describes the basic bootstrap on which the system rests, and Section 4 discusses the implementation of the common macro processor. The advantages and disadvantages of the approach are presented in the Section 5.

2. General approach

The technique rests on the fact that it is possible to identify two components of any program: the basic operations and the algorithm which coordinates these operations. Given a particular task, it is possible to define a set of basic operations and data types needed to perform the task. These operations and data types define an *abstract machine*—a hypothetical computer which is ideally suited to this particular task. The program to perform the task is then written for this abstract machine. To run the program on a real machine, it is necessary to realise the abstract machine in some way.

The abstract machine is an embodiment of the basic operations required to perform a particular task. Theoretically, it has no connection with any real machine, but practically we must always keep a wary eye on reality when designing an abstract machine. Many abstract machines can be formulated for a given task. The trick is to choose one of the right ones. Three considerations must be kept in mind—

1. The ease and efficiency with which the algorithm for accomplishing the task can be programmed in the language of the abstract machine.
2. The ease and efficiency with which the simulation of the abstract machine can be carried out on machines available currently and in the forseeable future.
3. The tools at hand for the realisation of the abstract machine.

Balancing these three considerations is very much an engineering task. If one is stressed at the expense of the others, there will be trouble.

An early attempt to use the abstract machine concept was the UNCOL proposal (SHARE, 1958). UNCOL was to be a *UN*iversal *C*omputer *O*riented *L*anguage which would reduce the number of translators for n languages and m machines from $n \times m$ to $n + m$. Effectively, the UNCOL proposal created a single abstract machine. In view of the three considerations mentioned above, it is easy to see why this attempt was unsuccessful—try to design an abstract machine which comes close to satisfying condition 1 above for FORTRAN, LISP (McCarthy, 1960) and SNOBOL (Griswold, Poage and Polonsky, 1969) simultaneously! The mobile programming system circumvents this difficulty by allowing a multiplicity of abstract machines.

It does not attempt to solve the $n \times m$ translator problem. The advantage of the mobile programming system lies in the fact that it reduces the specification of an algorithm to the specification of its basic operations, thus drastically reducing the amount of effort needed to implement it.

One way to carry out the realisation of an abstract machine is to devise for it an assembly language whose statements can be expressed as macros acceptable to the real machine's assembler. This approach was suggested by McIlroy (1960) and used in the implementation of L6 (Knowlton, 1966) and SNOBOL (Griswold, *et al.*, 1969). Unfortunately, the assemblers for different machines may require quite different input formats, and their macro processors often vary widely in power. The ease of transferring a program written in this way thus depends critically on the existence of a suitable assembly language for the target machine.

Recent developments in language-independent macro processing (Strachey, 1965, Waite, 1967, Brown, 1967) suggest that it is possible to make available a common macro processor. If the assembly language statements for the abstract machine are acceptable to this macro processor, then the realisation does not depend upon the macro capabilities of the target machine, but rather on the availability of the common macro processor. An example of this approach is the implementation of WISP (Wilkes, 1964). There the WISP compiler is the common macro processor, and is itself built up by bootstrapping from simpler macro processors.

3. The bootstrap

One of the design goals of the mobile programming system was that implementation on a new machine should not require a running version on another machine. The base of the system was to be easily implemented by hand if necessary. Once this base was available, it could be used to implement a common macro processor which would handle the realisation of the various abstract machines in the system. The common macro processor itself was written in the assembly language of an abstract machine, and hence the base of the system must include a simple macro processor.

An adequate macro processor can be expressed as a 91 statement program written in a restricted form as ASA FORTRAN. This program, known as SIMCMP, is described in detail by Orgass and Waite (1967). SIMCMP is written in FORTRAN for two reasons:

1. Since FORTRAN is a widely-used language, it may be available on the target machine. This means that SIMCMP can be implemented trivially.
2. FORTRAN was originally designed to be quite close to machine code, and hence an algorithm expressed in FORTRAN is easy to translate to machine code by hand. (Translation of SIMCMP to machine code for two different machines required about 4 man-hours in each case.)

The primary criterion used in the design of SIMCMP was simplicity. Only those features considered to be absolutely necessary were incorporated. SIMCMP has only one purpose: to realise the abstract machine used for the common macro processor. This approach is not the one which has been taken by most designers of mobile systems, who prefer to assume that a working version of the common processor is already available on *some* machine. They argue that if a working version is available on machine M, then a new version can be created on machine N by the following procedure:

1. Code macros which translate the source code of the processor to the assembly language of N.
2. Expand the common processor, using the macros developed in (1) and the processor existing on M. The result is an assembly language program for N.
3. Run the program resulting from (2) on N.

I must reject this procedure on the basis of my own experience. More often than not, the machines M and N are remote from one another. Since it is virtually impossible to write the macros correctly the first time, steps (2) and (3) must be iterated and the distance between the machines makes this a slow and costly business. Also the machines often have incompatible peripherals and/or different character sets; at each iteration of (2) and (3) a tedious translation must take place.

By eliminating the need for a working version, SIMCMP avoids these problems. All work is done on one machine. The abstract languages being translated are defined using a restricted character set available on most machines (43 characters of the FORTRAN set on the IBM 026 card punch). The character set used for the real machine's assembly language is completely arbitrary. It is determined by the macro definitions, which are written specifically for that machine and translated on it. This is not true for the procedure outlined above. There, machine M must be capable of writing assembly code for machine N. If machine M cannot output all of the characters needed by the assembly language of N, a translation must take place at each iteration of steps (2) and (3) above. On the other hand, suppose that all of the work is being done on N using SIMCMP. If N cannot recognise all of the 43 characters used in the abstract language, a translation need only be done once to put the source code into an acceptable form.

4. The common macro processor

As pointed out in the previous section, SIMCMP was designed primarily for simplicity and is certainly not adequate to serve as a common processor for realising abstract machines. It is impossible to produce good code using SIMCMP because decisions cannot be made and alternate expansions provided. Because there is no iteration facility, macros with argument lists of indefinite length cannot be handled. A second processor, known as STAGE2, was therefore designed as the common processor for the system. It provides all of the features normally associated with a general purpose macro processor (McIlroy, 1960). In many respects, it is quite similar to LIMP (Waite, 1967). Its input recognition procedure is language-independent, employing the LIMP type of scanning mechanism to recognise macro calls and isolate parameters. The code body, however, differs from that of LIMP. It does not include the 'grouping' concepts nor the SNOBOL interpreter. Conditional expansion, iteration and the like are provided by *processor functions* rather than by an explicit program structure. The ability to perform different parameter conversions has been retained and extended. A complete

manual describing the use of STAGE2 is available (Waite, 1968).

The design of STAGE2 required a balancing of two conflicting objectives:

1. It must be translated by SIMCMP.
2. It should be as flexible and as general as possible.

My overall philosophy dictated that (1) should be given most weight, and any clear choice had to be resolved in favour of it. Thus, STAGE2 does not provide *all* features which one would like to see in a macro processor, it is relatively slow, and it requires a fair amount of data space. Its purpose is to provide a common macro processor for realising a variety of abstract machines, and not to act as a processor for day-to-day use by applications programmers. For this latter use, we are preparing versions of ML/1 (Brown, 1967) and LIMP.

STAGE2 is written in a language called FLUB (*F*irst *L*anguage *U*nder *B*ootstrap). FLUB has 28 machine operations and 2 pseudo-operations, each of which can be expressed as a macro acceptable to SIMCMP. A complete description of the characteristics and design of the FLUB language has been given by Waite (1969).

The procedure for implementing STAGE2 on a new machine, *N*, is thus:

1. Implement SIMCMP on *N*.
2. Write 28 macro definitions which translate FLUB into the assembly language of *N*.
3. Translate STAGE2, using SIMCMP and the definitions of (2), and assemble the resulting program.

Once STAGE2 is running, it is possible to rewrite the 28 macros, taking advantage of the added flexibility of STAGE2. Using the version of STAGE2 already available, an optimised version of STAGE2 can thus be produced:

4. Write 28 macro definitions which translate FLUB into the assembly language of *N*. These macros use the features made available by STAGE2.
5. Translate STAGE2, using the version of STAGE2 implemented in (1)–(3) and the definitions of (4), and assemble the resulting program.

The operations of the FLUB machine are rather simple to describe and can be coded in a straightforward manner. Unfortunately, a well-known axiom in computer programming states that it is impossible to write even the simplest code correctly the first time. The macro definitions are the 'hardware' of an abstract machine, and an incorrect macro definition is analogous to a writing error in a real computer. No manufacturer tests a computer just off the production line by running a batch of FORTRAN jobs through it, and the implementor of an abstract machine should not be forced to attempt a similar feat. Two test programs have been developed by Mr. E. H. Henninger and Mr. R. C. Dunn to facilitate checkout of the macros. These test programs were quite tedious to construct, and are subject to most of the problems of normal hardware test programs (Bashkow, Friets and Karson, 1962).

Using the test programs, steps (2) and (4) above have two sub-steps:

2,4a Write 28 macro definitions which translate FLUB into the assembly language of *N*.

2,4b Translate, assemble and run the test programs. Correct any errors in the macros which were detected, and repeat step (b) until no errors remain.

Testing the macros in this manner simplifies and speeds the implementation considerably, because it means that the implementor need not be familiar with the inner workings of STAGE2 to be able to debug his macros. Since the test programs were made available, STAGE2 has been implemented on six different machines. Four of those implementations were done by people who were not familiar with STAGE2, but in no case was such familiarity needed. The test programs detected all of the macro coding errors, and STAGE2 ran perfectly when compiled.

I cannot overstress the importance of a comprehensive macro test program in the successful implementation of a machine independent system. Macro coding errors can be very subtle, requiring hours of debugging to trace them down if the machine independent program is the only test case. A conservative estimate based on our experience is that lack of a good test program will increase the time required to complete an implementation by a factor of five when the author of the system is available to debug the macros. When he is not available, the task is almost hopeless.

5. Advantages and disadvantages

Use of the abstract machine concept allows an impressive reduction in the amount of coding necessary to establish a processor on a new machine. In the case of SNOBOL4 (Griswold, *et al.*, 1969), for example, the compiler/interpreter contains roughly 4500 lines of code, but only approximately 100 primitives must be recoded for a new machine. The FLUB machine has 28 primitives, and STAGE2 is over 850 FLUB statements. A LISP (McCarthy, 1960) system without numeric capabilities requires just 8 primitives operating on 6 data types. Although the reduction in sheer bulk of coding is a factor in the success of this approach, the decrease in code complexity is more significant. Each primitive can be specified thoroughly, and is generally quite easy to implement.

Whenever the term 'machine independence' is mentioned, questions of efficiency are bound to arise. Of course, a program implemented as described in this paper will not be as fast and tight as a hand-coded version. Two arguments can be brought to bear, however. First, extreme efficiency of the generated code may not be an issue, as long as extreme inefficiency is avoided. An interactive BASIC (Dartmouth, 1966) system which can be up and running with less than one man-week of effort may be preferable to one which runs twice as fast but requires 10 man-months to complete. Second, it is possible to optimise the program on several levels. The source language is designed to match the problem well, and a great deal of time can be put into producing an efficient program for the abstract machine. Careful construction of the macros can result in surprisingly good code for the target machine. Once the assembly language version is available, critical parts can

be rewritten at leisure to further improve the program's performance.

In addition to SIMCMP and STAGE2, a comprehensive text manipulation program and two small editors have been produced using the system described in this paper. We are currently working on interactive BASIC (Dartmouth, 1966) and SNOBOL4 (Griswold, *et al.*, 1969). A 'logic analyser' (which includes automatic flowcharting) and a paginator for producing reports are in the planning stage. The list processors WISP (Wilkes, 1964), LISP (McCarthy, 1960) and L6 (Knowlton, 1966) are being considered.

The system has proved extremely mobile in practice, having been implemented on nine different machines. In each case the total effort was less than one man-week, and most went more rapidly than that. A team of two people, one thoroughly familiar with the system and the other with the target machine, have had it running in one day.

Acknowledgements

The basic ideas for the system came from some unpublished comments by T. R. Bashkow on the duality of hardware and software. Techniques were developed through the implementation of LIMP and a number of WISP compilers, in collaboration with M. V. Wilkes (Cambridge University), H. Schorr (IBM) and R. J. Orgass (IBM). Thanks are due to R. E. Griswold, J. F. Poage and I. P. Polonsky of Bell Laboratories for their willingness to discuss the inner workings of the SNOBOL 4 compiler/interpreter. Dr. P. C. Poole of the U.K. Atomic Energy Authority and Prof. Wilkes were of material assistance in criticising several drafts of this paper. Computing time was kindly made available at different times by the following institutions: Computer Center, Columbia University; University Mathematical Laboratory, Cambridge; Graduate School Computer Center, University of Colorado; Culham Laboratory, U. K. Atomic Energy Authority.

References

BASHKOW, T. R., FRIETS, J., and KARSON, A. (1963). A Programming System for Detection and Diagnosis of Machine Malfunctions, *IEEE Trans. on Electronic Computers*, Vol. EC–12, p. 10.

BROWN, P. J., (1967). The ML/1 Macro Processor, *CACM*, Vol. 10, p. 618.

DARTMOUTH COLLEGE COMPUTER CENTER (1966). *BASIC*, 3rd ed. Hanover, N.H.: Dartmouth College.

FELDMAN, J., and GRIES, D. (1968). Translator Writing Systems, *CACM*, Vol. 11, p. 77.

GLASER, E. L., COULEUR, J. F., and OLIVER, G. A. (1965). System Design of a Computer for Time Sharing Applications, *AFIPS Conf. Proc.* Vol. 25, p. 197

GRISWOLD, R. E., POAGE, J. F., and POLONSKY, I. P. (1969). *The SNOBOL4 Programming Language*, Englewood Cliffs, N. J.: Prentice-Hall, Inc.

KNOWLTON, K. C. (1966). A Programmer's Description of L6, *CACM*, Vol. 9, p. 616.

MCCARTHY, J. (1960). Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1, *CACM*, Vol. 3, p. 184.

MCILROY, M. D. (1960). Macro Extensions of Compiler Languages, *CACM*, Vol. 3, p. 214.

ORGASS, R. J., and WAITE, W. M. (1967). *A Base for a Mobile Programming System*, Yorktown Heights, N.Y.: IBM Corp. (to appear in *CACM*)

SHARE AD-HOC COMMITTEE ON UNIVERSAL LANGUAGES. (1968). The Problem of Programming Communication with Changing Machines: A Proposed Solution, *CACM*, Vol. 1, p. 12.

STRACHEY, C. (1965). A General Purpose Macrogenerator, *Comp. J*, Vol. 8, p. 225.

WAITE, W. M. (1967). A Language Independent Macro Processor, *CACM*, Vol. 10, p. 433.

WAITE, W. M. (1968). *The STAGE2 Macro Processor*, Boulder, Colorado: Department of Electrical Engineering and Graduate School Computing Center, University of Colorado.

WAITE, W. M. (1969). *Building a Mobile Programming System*, Boulder, Colorado: Graduate School Computer Center, University of Colorado.

WILKES, M. V. (1964). An Experiment with a Self-Compiling Compiler for a Simple List Processing Language, *Ann. Rev. in Automatic Programming*. Vol. 4, p. 1.