Using a macro processor to aid software implementation

By P. J. Brown*

A method of software implementation is described whereby the logic of the software is described as a series of machine-independent macro calls. When it is desired to implement the software on a given machine these macro calls are mapped into the assembly language of that machine. The advantages and limitations of these techniques are discussed and some practical results are presented.

(Received February 1969)

There are at least three internationally accepted machine-independent high-level languages in which algorithms involving purely numerical manipulations can be encoded. Efficient compilers for these languages are widely available and it is rare that one is forced to resort to machine code for encoding numerical algorithms. The situation in this field, though not perfect, is therefore reasonably satisfactory.

However, for algorithms involving a good deal of non-numerical manipulations and in particular for software, the situation is much less satisfactory. (The term 'software' is used here to mean compilers, assemblers, etc.—*not* the languages they compile.) It is possible to use some existing high-level languages for software writing or to use other tools, such as syntax directed compilers, but the resultant object code is normally large and inefficient. Non-numeric features are often treated interpretively. Hence these methods, though very suitable if one wants to get something working quickly, are not suitable for production software. There do exist some honourable exceptions to this, i.e. machine-independent high-level languages with non-numerical facilities and efficient compilers. These are sometimes extensions of well-known languages, such as Burroughs' extended ALGOL. However, these languages are not widely accepted and compilers often only exist for one machine or range of machines. Hence the advantages of machine-independence are largely lost.

There is, in fact, a much more fundamental disadvantage in using predefined high-level languages for software writing over and above the fact that compilers tend to be inefficient. However general a language might be, it will never contain quite the right facilities needed for describing a particular piece of software. For example very few high-level languages provide an efficient way of implementing a dynamic stack in which data of varying types is freely intermixed. The basic problem is that for numeric working there is a fairly standard set of possible operations, data types and aggregates, whereas for non-numeric working there is a huge range of possible operations, data types and aggregates and it is not possible to represent these efficiently in terms of a few primitives. Hence when writing software in a predefined high-level language one is forced to adapt the design of the software to fit the facilities that happen to be available in the language, i.e., one tailors the software to the software-writing language.

Because of the disadvantages described above, production software is still largely written in assembly language or in other machine-dependent languages suitable for software writing, such as PL360 (Wirth, 1968). If it becomes necessary to implement the software on a new machine, it needs to be re-coded, which is a most undesirable task.

A DLIMP

The purpose of this discussion is to describe a method of software implementation which uses a machine-independent high-level language but surmounts the problems described above. The method uses the ML/I macro processor (Brown, 1966, 1967) and is called a DLIMP, which stands for *D*escriptive *L*anguage *I*mplemented by *M*acro *P*rocessor.

Assume, therefore, that it is required to implement some software *S* on several different machines or, alternatively, some software *S* is to be implemented on one machine and it is thought it may be necessary to implement it on other machines, as yet undefined, at some time in the future.

The first step of the implementation process is to separate the logic of *S* into its machine-independent part and its machine-dependent part. Typically the latter might include I/O, hashing algorithms, data type conversion, routines dependent on individual character representations, etc., i.e., routines where the very logic is dependent on the object machine.

Of course, the word 'machine-independent' is a rather loosely used (and much abused) term, and the above division must be made in a somewhat *ad hoc* manner. Moreover a distinction must be made between theory and practice. In theory if some software is expressed as a Turing machine it is machine-independent. However, this is not much use in practice. This paper is concerned with software that is machine-independent in practice and hence there is an added requirement of efficiency in the use of machines.

The technique to be described below is only applicable if *S* is reasonably machine-independent. It is inapplicable if the machine-dependent part of *S* is as large as or larger than its machine-independent part, in which case *S*, by its very nature, needs to be largely rewritten for each object machine.

The descriptive language

After the logic of *S* has been separated out into its two parts, the next step of the implementation process

* *Computing Laboratory, University of Kent at Canterbury*

is to design a *descriptive language* for the machine-independent part of *S*. This will be represented as *DL(S)*. *DL(S)* is a machine-independent language with semantics designed especially for describing *S* and with syntax designed to be translatable by ML/I into the assembly language of any machine and preferably also into any suitable high-level language. Loosely speaking, *DL(S)* is a collection of machine-independent ML/I macros corresponding to all the primitive operations used in *S*.

Note that both the operations and the data types in *DL(S)* are specially tailored to *S*. Thus if *S* involved a dictionary facility, *DL(S)* would contain operations for manipulating the exact type of dictionary used in *S*. Alternatively if *S* involved list processing, *DL(S)* might contain list data together with suitable operations for manipulating it.

The designing of *DL(S)* would probably be concurrent with the last stages of the designing of *S* and would proceed incrementally, until a comprehensive and well-rounded set of macros for the description of *S* was accumulated. *DL(S)* would be an aid to the documentation of *S* and, in isolating the primitives needed, should be a help in the design of *S*. If one uses flow-charts, the macros in *DL(S)* would probably mirror the operations one writes in the boxes of a fairly detailed flow-chart.

The degree of sophistication in *S* is entirely at the choice of its designer. A reasonable level would be to make the level of *DL(S)* in its field roughly the level of FORTRAN IV in the field of numerical problems. An example of a descriptive language is given in **Fig. 1.**

```
      SET STAKPT = STAKPT − OF (LNM)
      SET VALUE = IND (STAKPT) NM
      IF LEVEL = 0 Λ STAKSW = FALSE THEN
          SET LEVEL = 1
          MOVE FROM BLOCK (SDB) TO STAKPT
                    LENG OF (6*LPT+LNM)
      END
      // NOW TEST WHICH OPERATOR //
      CHARMATCH IDPT, '+' GOING ADD, '−'
                     GOING SUB, '*' GOING MPLY
      GO TO ERROR 3
[ADD] SET RESULT = RESULT + VALUE

      . . .
```

Fig. 1. This shows typical statements in *DL*(ML/I) to give the reader a general idea of the scope of the language. The first two statements remove a variable from a stack. IND(...)NM means the *NuM*ber pointed at by the argument and OF(...) stands for the number of units of storage occupied by its argument. For example OF(LNM) means the *L*ength of a *NuM*ber and would be mapped into 1 on a word machine, or, say, 4 on a byte machine.
The remaining statements illustrate some of the branching statements (IF, GO TO and CHARMATCH) and a block move (MOVE).

Generating an implementation

When it is desired to implement *S* for a given machine *M*, an *object language* is chosen. This can be any language for which a compiler or assembler exists for *M* and into which *DL(S)* can be mapped. Normally the object language is the assembly language of *M*. The implementation of *S* then proceeds in two stages. This is illustrated in **Fig. 2**, and **Fig. 3** provides a concrete example.

The first stage is performed using any machine for which ML/I has been implemented. Macros are written to map *DL(S)* into the object language. These *mapping macros*, together with the machine-independent part of *S* encoded in the language *DL(S)*, are fed to ML/I and the output (once the mapping macros have been debugged) will be the machine-independent part of *S* encoded in the object language.

The second stage is performed on the object machine *M*, and simply involves the compilation or assembly of the output of stage one and its incorporation with the machine-dependent part of *S*, which, of course, needs to be coded by hand in the object language on each implementation.

In order to ease the debugging stages, it is desirable to write a comprehensive test program in *DL(S)* which uses all the facilities of *DL(S)* and tests all the machine-dependent routines, and to get this working before embarking on the larger job of mapping the entire machine-independent part of the logic.

Typically an implementation might take about six man weeks though there will be considerable variations dependent on the relative sophistication of *DL(S)* and the object language.

To take a very simple example of a mapping macro, assume that a statement in a descriptive language had the form

IF *variable* = *variable* THEN GO TO *label*

For IBM System/360 the mapping macro for this statement might be

```
MCDEF IF = THEN WITHS GO WITHS TO NL
AS<   L     RA, ~A1.   Load first variable.
      C     RA, ~A2.   Compare with second.
      BZ    ~A3.       If equal, jump to label.
>;
```

(where NL stands for 'newline', and '~A*n*.' means 'insert the *n*th argument here'), whereas for a PDP–7 mapping the macro might be

```
MCDEF IF = THEN WITHS GO WITHS TO NL
AS<   LAC   ~A1.      Load first variable.
      SAD   ~A2.      Skip if it differs from
                        second.
      JMP   ~A3.      Jump to label.
>;
```

In practice, no doubt, a rather more sophisticated IF statement would be needed, but the above should illustrate the general technique.

Advantages of DLIMPs

The advantages of DLIMPs are manifold. Among them are the following:

(*a*) Implementation time is considerably reduced, partly because there will be no isolated coding bugs. Any errors in the mapping macros tend to be manifestly obvious since they will arise in many places in the logic and will make it horribly wrong. Once the mapping macros are debugged the entire logic will be mapped correctly. Only if extensive optimisation is attempted by making macros interdepend in elaborate ways does a danger of isolated bugs arise, and, in fact, in practice no difficulties have been found in this area since it has been found that optimisation is best performed by an extra pass through ML/I with special macros for eliminating redundancy.

(*b*) Once one mapping has been performed, the implementing of changes and improvements to *S* is trivial. The logic of *S* as described in *DL*(*S*) is updated and is re-mapped into each desired object language.

(*c*) Once one descriptive language, *DL*(*S*1), has been designed, it is much easier to design a descriptive language for a different piece of software, *S*2. Most descriptive languages will have a common kernel (e.g. conditionals, looping, simple arithmetic) and *DL*(*S*2) could be derived from *DL*(*S*1) by deleting facilities peculiar to *S*1 and replacing them by ones specially tailored to *S*2. The concept of 'inner and outer syntax' (Wilkes, 1968) is relevant in this respect.

(*d*) The technique is specially suitable for a new machine with little software of its own.

(*e*) The implementor does not need to learn all the details as to how the logic of the software works, i.e. there is less of a problem in implementing someone else's software.

These advantages are such that it may be worth while to implement software by means of a DLIMP even if it is only to be implemented on one machine. Indeed, one approaches a DLIMP when making extensive use of a macro-assembler in software implementation, a practice which is becoming quite common.

Disadvantages

The disadvantage of a DLIMP is the fact that the generated object code is less efficient than would be produced by hand-coding in assembly language. However this inefficiency is much less than that which arises when software is coded in a predefined high-level language. This is because *the software-writing language is tailored to the software and not vice versa,* the thesis on which this whole technique is based. In practical cases DLIMPs have involved inefficiencies of between 3% (on a PDP-7) and 20% (on an IBM System/360) in speed and size compared with assembly language coding. (There is a further, usually very small, inefficiency involved in organising the logic of software in a machine-independent way, even if its encoding is perfectly efficient.) The degree of inefficiency in a DLIMP depends to some extent upon the effort put into optimising the mapping macros, but to a much larger extent on the complication of the order code of the object machine. The simpler an order code the easier it is to generate optimum code, a factor that should be borne in mind in machine design given that more and more code is artificially generated.

Ambitiousness

It should be noted that a DLIMP is a relatively unambitious method of software implementation compared with some automatic software generation techniques in that the designer of the software is entirely responsible for its logic. He gets no help (and no constraints) in the form of automatic scanning or translation algorithms. The end product runs entirely independently of ML/I. ML/I does not act as a generalised compiler.

Proposals such as Halpern's (1968) and software such as the Compiler Compiler (Brooker and Morris, 1962) are therefore entirely different in concept from a DLIMP.

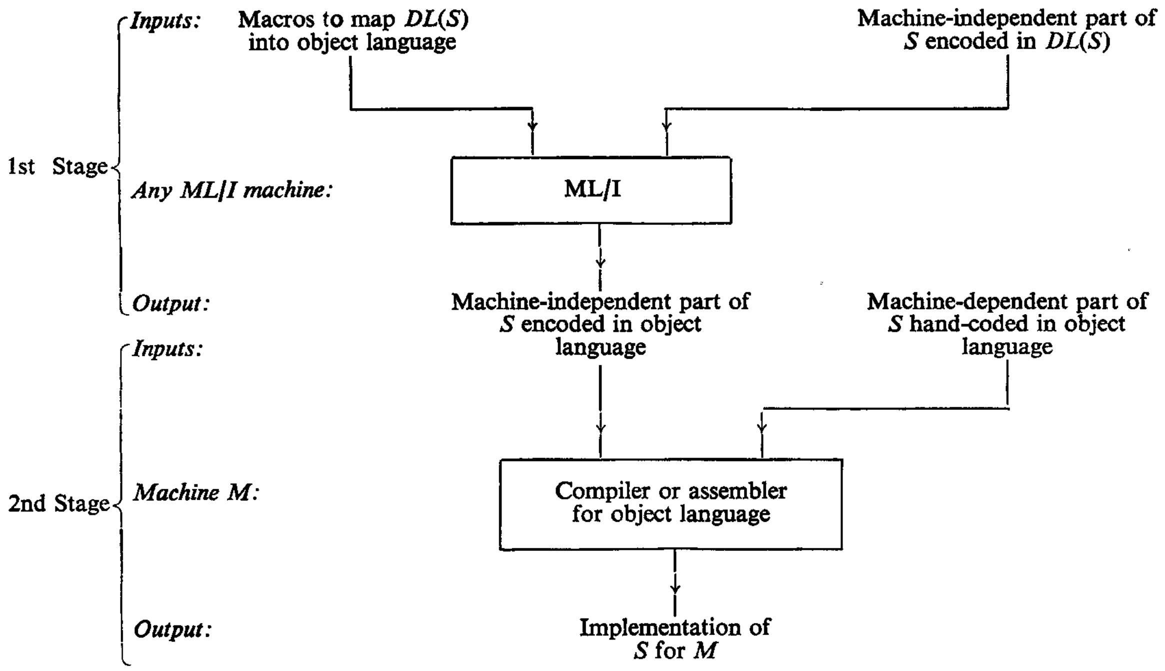
1st Stage
Inputs:
Macros to map DL(S) into object language
Machine-independent part of S encoded in DL(S)
Any ML/I machine:
ML/I
Output:
Machine-independent part of S encoded in object language
Machine-dependent part of S hand-coded in object language
2nd Stage
Inputs:
Machine M:
Compiler or assembler for object language
Output:
Implementation of S for M

Fig. 2. General organisation of a DLIMP. The software *S* is being implemented for a machine *M*.

On the other hand, work similar in concept to a DLIMP includes that of Wilkes (1964), Ferguson (1966) and Waite (1968).

Results of DLIMPs

ML/I itself has been implemented on several machines by means of a DLIMP. The way this was done is illustrated in Fig. 3. About nine-tenths of the logic of ML/I went into the machine-independent part. A full description of *DL*(ML/I), which is called, simply, *L*, is given by Brown (1968a) and detailed descriptions of some of the mappings that have been performed are given by Brown (1968b). Fig. 1 is a short extract from the logic of ML/I, and should give the reader an idea of the form of *DL*(ML/I).

Up to now, *DL*(ML/I) has been mapped into one high-level language, PL/I, and into the assembly languages for the following machines: PDP-7, ICL Titan, ICL 1900 series, ICL 4100 series, Honeywell 200 series, IBM System/360.

All of these mappings have produced successful implementations of ML/I except the mapping into PL/I. The PL/I implementation turned out to be so large and so slow that it was virtually useless. The main reason for this is that PL/I, although better than other high-level languages, has not quite the right facilities for describing the logic of ML/I and a number of operations had to be performed in a very clumsy way. This is evidence to support the contention that production software should not be written in predefined high-level languages. Another reason for the slowness of the PL/I implementation was the heavy overhead on a subroutine call, a fact that I had not sufficiently appreciated when designing the mapping macros.

Of the assembly language mappings, none has produced code that is more than 20% worse than would be produced by hand, and the average inefficiency is about 10–15%. These figures for inefficiency were derived by encoding randomly selected parts of the logic by hand and comparing the number of instructions produced with the number produced by the macro-generated equivalent. The hand-coding was done in a straightforward manner, using no programming tricks. To some extent the figures for efficiency reflect the amount of effort put into optimising the mapping macros.

No mapping has necessitated a change to the descriptive language of ML/I and hence it can reasonably be claimed to be machine-independent. The only reservations about the descriptive language arose from the Honeywell Series 200 mapping. Series 200 are character machines with data fields and instructions delimited by word marks and item marks. The logic of ML/I, on the other hand, has been written in terms of data fields with explicit lengths and hence did not make good use of the facilities of the Series 200. The Series 200 was the only mapping where the inefficiency due to describing ML/I in a machine-independent way (as distinct from the inefficiency due to using macros rather than hand-coding to implement it) was significant. On the IBM System/360, on which character manipulation instructions have explicit lengths, much better use was made of the hardware.

The average time taken for these implementations of ML/I was six to ten man weeks.

In addition to the DLIMPs that have been used to implement ML/I, a project is under way to implement Dartmouth BASIC by means of DLIMPs. This is being undertaken by W. H. Purvis of the University of North Wales.

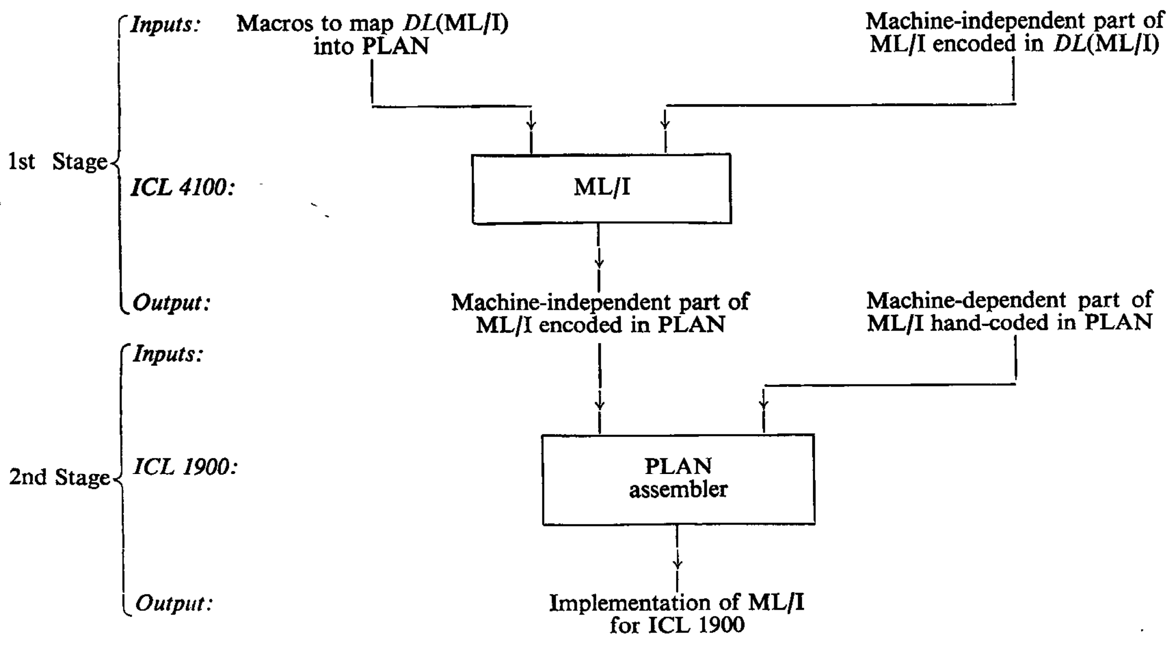
Inputs:
Macros to map DL(ML/I) into PLAN
Machine-independent part of ML/I encoded in DL(ML/I)
1st Stage
ICL 4100:
ML/I
Output:
Machine-independent part of ML/I encoded in PLAN
Machine-dependent part of ML/I hand-coded in PLAN
Inputs:
2nd Stage
ICL 1900:
PLAN assembler
Output:
Implementation of ML/I for ICL 1900

Fig. 3. Specific example of a DLIMP. This illustrates a DLIMP to implement ML/I for the ICL 1900 series using the ICL 4100 implementation of ML/I to perform the mapping.

Reasons for using ML/I

Almost any general-purpose macro processor can be used to perform a DLIMP but it is claimed that ML/I has considerable advantages because:

(*a*) it allows freedom of notation in writing macro calls and sophisticated macros can be built up relatively easily (e.g. arithmetic expressions, ALGOL-like IF statements);
(*b*) it allows macro calls within macro calls (e.g. a constant macro within an arithmetic expression macro within an IF macro);
(*c*) it does not require any trip character to precede a macro call.

Of these (*c*) is probably the most important. It means that ML/I is capable of performing systematic editing and it is unnecessary to specify in the descriptive language what is a macro and what is not. This is decided only when a mapping is made. The following example illustrates the advantages of this.

In *DL*(ML/I) identifiers of up to six characters are used for program labels. In most mappings these were copied without alteration into the object program, i.e. they were not treated as macros. However, the PLAN assembler used on the ICL 1900 Series mapping only allowed five character labels, and so it was necessary to change all six character labels to five. It was possible to do this by including, within the mapping macros for PLAN, macros to scan for all six character labels and generate macros to replace them by five character labels, checking each for uniqueness. No messy hand editing was needed.

Examples like this arose on nearly every mapping. In the PL/I mapping, for instance, it was necessary to map a series of data declarations into an array with an initial value and to replace references to the data accordingly.

This facility in ML/I for systematic editing (i.e. editing involving commands to replace systematically one piece of text by another throughout a document) is the main reason why mappings have avoided most of the unforeseen difficulties that normally plague exercises in machine-independence because of the arbitrary differences between machines and between various compilers and assemblers.

Acknowledgements

The following people have helped me with the various individual implementations and I very much appreciate their excellent work:

A. G. Adams of Glaxo Laboratories.
L. H. Bouchard of the University of Essex.
H. Brown of the University Mathematical Laboratory, Cambridge (now of the University of Kent).
B. C. Chapman and R. G. Gray of Honeywell.
S. Clelland and I. Duncan of Ferranti.
D. F. Howe and P. Seaman of IBM.
W. H. Purvis of the University of North Wales.
Several others have also been of great assistance.

A good deal of the work described above was performed at the University Mathematical Laboratory, Cambridge, to whom I am very grateful, together with the S.R.C., who provided my grant while I was there.

I should also like to thank the referee for useful criticisms.

References

Brooker, R. A., and Morris, D. (1962). A general translation program for phrase structure languages, *JACM*, Vol. 9, No. 1, pp. 1–10.
Brown, P. J. (1966). *ML/I user's manual*, University Mathematical Laboratory, Cambridge.
Brown, P. J. (1967). The ML/I macro processor, *Comm. ACM*, Vol. 10, No. 10, pp. 618–623.
Brown, P. J. (1968a). *The use of ML/I in implementing a machine-independent language in order to bootstrap itself from machine to machine*, Technical Memorandum No. 68/1, University Mathematical Laboratory, Cambridge.
Brown, P. J. (1968b). *Macro processors and their use in implementing software*, Ph.D. thesis, Cambridge University (partial copies available from the author).
Ferguson, D. E. (1966). The evolution of the meta-assembly program, *Comm. ACM*, Vol. 9, No. 3, pp. 190–193.
Halpern, M. I. (1968). Towards a general processor for programming languages, *Comm. ACM*, Vol. 11, No. 1, pp. 15–25.
Waite, W. M. (1968). *The STAGE 2 macro processor*, University of Colorado Computing Center, Boulder, Colorado, Preliminary Edition.
Wilkes, M. V. (1964). An experiment with a self-compiling compiler for a simple list-processing language, *Annual Review in Automatic Programming*, Vol. 4, pp. 1–48, Pergamon Press, Oxford.
Wilkes, M. V. (1968). The outer and inner syntax of a programming language, *Comp. J.*, Vol. 11, pp. 260–263.
Wirth, N. (1968). PL360, a programming language for the 360 computers, *JACM*, Vol. 15, No. 1, pp. 37–74.