Levels of Language for Portable Software

P. J. Brown
University of Kent at Canterbury

An increasing amount of software is being implemented in a portable form. A popular way of accomplishing this is to encode the software in a specially designed machine-independent language and then to map this language, often using a macro processor, into the assembly language of each desired object machine. The design of the machine-independent language is the key factor in this operation. This paper discusses the relative merits of pitching this language at a high level or a low level, and presents some comparative results.

Key Words and Phrases: portable software, level of language, machine independent, macro processor, efficiency

CR Categories: 4.12, 4.21, 4.22

Author's address: Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, England.

A commonly used way of implementing portable software is to encode the software in a machine-independent *descriptive language* and then to map this descriptive language, normally using a macro processor, into a number of different assembly languages. In particular, the STAGE2 [1] and ML/I [2] macro processors have been used in this way to implement a number of pieces of software, and the technique has also been used in implementing WISP [3] and SNOBOL4 [4]. (Poole and Waite, who have been concerned with STAGE2, use the term "abstract machine" in place of "descriptive language," but this paper will use the latter term, partly for reasons of consistency and partly because it is the langugage aspect that is of interest here.)

Of these portability projects, the ones where ML/I has been used to implement itself [5] and a symbolic logic package [6] have been exceptional in that the descriptive languages used have been set at a very high level. The languages have more the flavor of, say, Algol than an assembly language. For example, the following is a typical piece of code in the descriptive language used for ML/I itself. The language is called, simply, L.

```
      / / THE FOLLOWING IS CODE IN L / /
      IF NTYPSW = 6 & IDPT + IDLEN GR SIZE THEN
        SET DIFF = IND(SPT + 1)NM − IDLEN + 1
        STACK NTYPSW (SW) ON FSTACK
        MOVE FROM IDPT TO STAKPT LENG 13
                        BACKWARDS
      END
[LAB] CALL SCAN (IDPT)PT
      SETSW NTYPSW = TYPSW & MASKSW
      BACKSPACE ARGPT
```

It can be seen that L is a mixture of specialized statements needed to implement ML/I, such as STACK, MOVE, and BACKSPACE and familiar statements such as IF, SET, and CALL. Even the latter, however, are specially tailored to the needs of the software to be described; this is the very purpose of a descriptive language.

The main purpose of this paper is to examine the relative merits of using high-level descriptive languages and to present some comparative results.

Implementing via High-Level Language

If a descriptive language is set at a high level it is possible to implement it by mapping into a suitable high-level language that is available on the object machine instead of mapping it into an assembly language.

An early attempt was made to implement ML/I on IBM System/360 by mapping L into PL/I. It was very easy to write macros to map L into PL/I, but the resulting implementation of ML/I on System/360 was so large and slow that it was totally unusable. PL/I compilers have improved since this implementation was performed, but it is still true that using PL/I or any predefined high-level language with similar capabilities would be almost certain to produce an implementation at least twice as slow as an assembly language one. The very purpose of the use of descriptive languages is to produce an efficient implementation—typically ML/I implementations produced by mapping L are about 10 percent worse than a hand-coded implementation would have been—and hence mapping into a high-level language loses the whole point of the exercise (except, perhaps, where it is used as the first stage of a two-stage bootstrapping operation). It is almost imperative that to produce a production piece of software the *object language*, i.e. the language that the descriptive language is mapped into, should be an assembly language (or else a language like PL360). This applies even when the descriptive language is high level. Although the use of a low-level descriptive language may have the apparent disadvantage that it cuts out the use of a high-level object language because the resultant inefficiencies are very bad indeed, this is no great loss in practice.

It will be assumed in the rest of this paper that the object language is an assembly language.

Machine-Independence

An encoding of a piece of software is completely machine independent if it can be mapped into an efficient implementation on all possible machines. Clearly no software is absolutely machine independent, but machine independence is a relative quality and some encodings of software are more machine independent than others.

It is obvious that encoding software in a high-level descriptive language is more machine independent than in a low-level descriptive language. Looking down from on high it is possible to see over a wider area than when one is close to the ground.

The difference in practice, however, is not as great as it might appear. This is because successful machine designs are usually conservative, largely because the machine users are also conservative, and one machine is much like another. Machines that are significantly different, such as ILLIAC IV or CDC STAR, are often so different that they even knock the machine independence out of high-level languages.

Hence if a low-level descriptive language is used, very few machines will be lost. There is, however, likely to be some loss of efficiency on each implementation. Some figures for this are given by an experiment using ML/I.

The Experiments

A low-level descriptive language called LOWL has been designed. LOWL contains all the primitive facilities needed in L and in the descriptive language for the symbolic logic package. (These two high-level descriptive languages are quite close to one another, and it was easy to encompass them both within LOWL.) LOWL basically looks like an assembly language for a machine with two special registers, A and B. The former is the accumulator, and the latter an index register. The following statements give a flavor of LOWL. They are encodings of the first two L statements shown in an earlier example. The parenthesized comments to the right of each statement give the meaning of the operation code.

```
NB      THE FOLLOWING IS CODE IN LOWL
LAV     NTYPSW   (Load A with Variable)
CAL     6        (Compare A with Literal)
GONE    GL20,100 (GO if Not Equal)
LAV     IDPT
AAV     IDLEN    (Add to A a Variable)
CAV     SIZE     (Compare A with Variable)
GOLE    GL20,96  (GO if Less or Equal)
LBV     SPT      (Load B with Variable)
LAM     1        (Load A Modified)
SAV     IDLEN    (Subtract from A a Variable)
AAL     1        (Add to A a Literal)
STV     DIFF,X   (STore A in Variable;
                  X means A need not be preserved)
```

ML/I and the symbolic logic package have been encoded in LOWL, and LOWL has been mapped into the assembly languages of the three following machines:

1. ICL 4130 (a scientific machine with a large instruction repertoire, except for character manipulation).
2. IBM System/360.
3. PDP-11.

L has also been mapped into each of the three assembly languages, and the purpose of the experiment was to compare the results. The sizes of the logic of ML/I (excluding hand-coded machine-dependent parts, which should be identical whether the mapping was via L or via LOWL) are shown in Table I. Comparative figures for the symbolic logic package are only available for the ICL 4130, and in this case the percentage overhead for LOWL was 2.75 percent as against the 2.51 percent above.

All in all, therefore, the inefficiencies introduced by

Table I.

	ICL 4130	System/360	PDP-11
Mapped via L	3672 words	14808 bytes	5343 words
Mapped via LOWL	3764 words	15484 bytes	5665 words
Percentage overhead for LOWL	2.51%	4.57%	6.02%

the use of the low-level descriptive languages are very small.

The following points are relevant in interpreting the figures in the table.

a. The percentages in Table I relate to the increased size of ML/I. The LOWL implementations of ML/I would also be expected to run more slowly by the same percentage and tests on the ICL 4130 confirm this.

b. For the ICL 4130 and the PDP-11, the form of data areas and stacks is identical whether ML/I is mapped via L or LOWL. On System/360 some data items (so-called "switch" data in L) are stored in one byte on the L mapping and in one word on the LOWL mapping. This means that the LOWL version of ML/I is up to 5 percent wasteful in its use of work areas, as well as being 4.57 percent larger itself.

c. The encoding of ML/I in L was mapped into a LOWL encoding using ML/I itself, and the LOWL encoding of the symbolic logic package was generated in a similar way. A hand coding in LOWL might have been slightly better, which means that the above figures are overestimates.

Description of LOWL

LOWL itself has some features which may be of interest. The two registers A and B are never used simultaneously and they may correspond, in an implementation of LOWL, to the same physical register. This would apply on an object machine with no index register and only one accumulator. The only way the two registers interact is through the LAM statement, which loads A with the address given by B modified by a numerical operand. If A and B are physically the same the LAM statement might map into two or three instructions in the object assembly language, whereas if they are different it might be accomplished in one.

There are two data types in LOWL: numerical and character. On machines with inadequate character manipulation instructions it is necessary to store one character to a word and there is no need to differentiate the data types.

Each LOWL statement consists of a mnemonic operation code followed by a sequence of operands. Almost all operators essentially need only one operand, but extra information about the first operand is often given by succeeding operands. On the GO operator, for example, the first operand is the name of a label to be gone to and the second operand gives the distance of the label from the current statement. Thus

```
GO    REPEAT, 26
```

means GO to label REPEAT which is 26 LOWL statements ahead.

These redundant operands are a valuable feature of LOWL, and they cost nothing to an implementor who does not need them. The second operand on the GO statements would be very useful in producing an implementation for a computer with a special instruction for short distance jumps.

As can be seen, the structure and format of LOWL statements is simple and it should be possible to map it into a desired assembly language using ML/I, STAGE2 or a powerful macro assembler on the object machine. In the latter case some minor preprocessing might be necessary to change statement formats. This property of LOWL has been copied from the descriptive language for SNOBOL4.

LOWL contains just over 50 different operation mnemonics, though many are very similar to one another. There are, for example, nine variants of the load operator, and some of these may be the same on some implementations of LOWL (e.g. load A, load B, load numeric, load character). The number of fundamentally different operation mnemonics is less than 25.

Sources of Inefficiency

An analysis of the inefficiencies of LOWL over L on the three test cases shows a wide variety of causes. In no case does one source account for more than one-sixth of the total. Most of the inefficiencies are because it is easier to make use of specialized instructions on the object machine when mapping from a high-level language.

An example of this is given by the ICL 4130, which has an instruction, called DECS, that decreases a given storage location by one. On the mapping from L it was fairly easy, within the macro corresponding to the assignment statement in L, to look for the special case

SET $X = X - 1$

and use the DECS instruction when it occurred. On the

mapping from LOWL it would have been necessary to recognize sequences of statements of form

```
LAV  X
SAL  1
STV  X
```

Since three separate macro calls are straddled, this is rather tedious to do so it was not done.

Implementation Time

The prime advantage of LOWL over L is that it is much easier to write macros to map it into assembly language. Consider, for example, the L statement

```
SET SIZE = IND(IDPT)NM + OFFSET - 6
```

(where IND(IDPT)NM means the NUMber INDirectly addressed by the pointer IDPT). The mapping macro for SET has to cater for an argument which may be any arithmetic expression involving constants and variables, possibly indirectly addressed, connected by addition and subtraction operators.

The equivalent in LOWL is:

```
LAI   IDPT     (Load A Indirectly)
AAV   OFFSET   (Add to A a Variable)
SAL   6        (Subtract from A a Literal)
STV   SIZE     (STore A in Variable)
```

These represent calls of four simple LOWL macros.

It is hard to quantify how much easier it is to map LOWL than L, but a factor of four in favor of LOWL is no exaggeration. The actual computer time taken for the ML/I macro processor to perform a mapping is certainly measurably improved by a factor of four.

Comparing an implementation via LOWL with one via L, the LOWL implementor gains several weeks. If he has these weeks to spare, he can spend them optimizing his generated code. Taking the ICL 4130 implementation as an example, it would be possible to write macros to recognize sequences of instructions that could be mapped into a DECS instruction. Thus the LOWL implementor should be able to wipe out all the inefficiencies during the extra time the L implementor would take to get his software working. If a very large amount of time was spent over an implementation, the L implementor would always win, since some optimization can only be performed at a high level. In most practical cases, however, there is little difference in efficiency between the two methods.

Conclusions

The advantages of high-level descriptive languages, namely extra machine independence and efficiency, are not in practice very great, and are usually outweighed by the advantages of low-level descriptive language would be the better choice for 9 out of 10 implementations.

An encoding of a piece of software in a high-level descriptive language is, however, useful to have in reserve. It gives a better chance of producing a good implementation if an unusual object machine is encountered. A high-level description is also extremely valuable as documentation.

There are, therefore, great merits in using a hierarchy of descriptive languages, as suggested by Poole [7].

Acknowledgment. R.C. Saunders, working under a Science Research Council grant, supplied the figures quoted here for the PDP-11.

Received August 1971, revised December 1971

References

1. Waite, W.M. The mobile programming system: STAGE2. *Comm. ACM 13*, 7 (July 1970), 415–421.
2. Brown, P.J. The ML/I macro processor. *Comm. ACM 10*, 10 (Oct. 1967), 618–623.
3. Wilkes, M.V. An experiment with a self-compiling compiler for a simple list-processing language. In *Annual Review in Automatic Programming, Vol. 4*, Pergammon Press, Oxford, England, 1964.
4. Griswold, R.E. *A Guide to the Macro Implementation of SNOBOL4*. Bell Telephone Labs., Murray Hill, N.J., 1969.
5. Brown, P.J. Using a macro processor to aid software implementation. *Comput. J. 12*, 4 (Nov. 1969), 327–331.
6. Brown, P.J., and Lowe, J.D. A computer program for symbolic logic. *Bulletin Inst. Maths. Applics. 7*, 11 (Nov. 1971), 320–322.
7. Poole, P.C. Hierarchical abstract machines. Proc. Software Engineering Conf., Culham, England, HMSO, London, 1971.