Interlibrary Loan Request Form

TransactionNumber: 521659

Call #: QA76.5 . S653

Location:

Article Information

Journal Title: Software---Practice and Experience

Volume: 11 **Issue:**
Month/Year: / 1981
Pages: 963-973

Article Author: Paul Klint

Article Title: Interpretation Techniques

Loan Information

Loan Title:

Loan Author:

Publisher:
Place:
Date:
Imprint:

Customer Information

Username: nramse01

Norman Ramsey
Faculty - Computer Science
161 College Ave

Medford, MA 02155

Article Delivery Method: Hold for Pickup
Loan Delivery Method: Hold for Pickup
Electronic Delivery? Yes

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 963–973 (1981)

Interpretation Techniques*

PAUL KLINT

Mathematical Centre, P.O. Box 4079, 1009AB Amsterdam, The Netherlands

SUMMARY

The relative merits of implementing high level programming languages by means of interpretation or compilation are discussed. The properties and the applicability of interpretation techniques known as classical interpretation, direct threaded code and indirect threaded code are described and compared.

KEY WORDS Interpretation versus compilation Interpretation techniques Instruction encoding Code generation Direct threaded code Indirect threaded code.

1. INTRODUCTION

Two major subjects are covered in this paper. Section 2 contains a discussion of the relative merits of compilation and interpretation as implementation methods for high level languages. Issues like efficiency, flexibility, error detection and portability are treated. Section 3 contains a detailed description of three common interpretation techniques. Some measurements of the time and space requirements of these techniques are presented.

2. COMPILATION VERSUS INTERPRETATION

In the past there have been hot debates on the question whether implementations of high level languages should be based on compilation or interpretation. In the extreme cases, an *interpreter* inspects the source text of a high level language (HLL) program and performs actions as prescribed by that source text, whereas a *compiler* translates an HLL program into machine code, which can be executed directly by a computer. Figure 1 illustrates these extreme cases, (1) and (4), together with two mixed compiler/interpreter systems, (2) and (4). These systems can be distinguished by the semantic level at which the HLL program is finally executed. See Reference 1 for a more elaborate discussion of this subject. Going from left to right in this figure one encounters four types of high level language implementations:

(i) Type-1 systems: Direct interpretation of the source text. The source text is inspected character-by-character and actions are performed accordingly. For example, when at a certain moment the character sequence 'g', 'o', 't', 'o', ' ', 'L' is encountered, the interpreter initiates a search for the occurrence of 'L' followed by ':'. Most macro processors (STAGE2,[2]ML/I[3]) operate in this way. (Disregarding

*This research is registrated at the Mathematical Centre as IW110/79.

0038–0644/81/090963–11$01.10

Received 12 June 1979

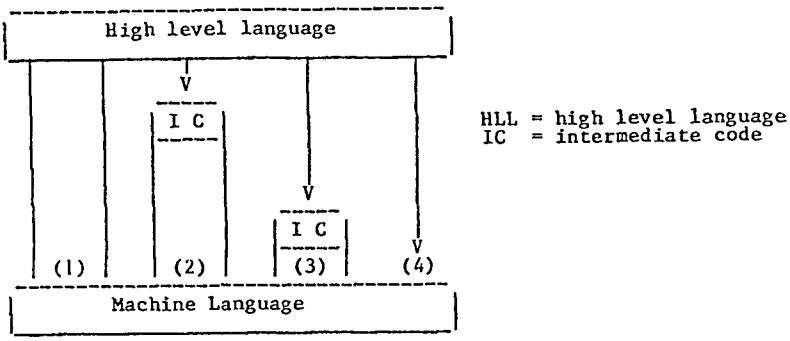

Figure 1. Four types of high level language implementations

the fact that such macro languages can hardly be called 'high level'.) Format specifications in input/output statements in programming languages are in the vast majority of cases implemented by means of pure interpretation. Type-1 systems are based on repeated lexical and syntactic analysis of the source text. Accordingly, these systems are inefficient and have the undesirable property that syntax errors can only be detected at run time.

(ii) Type-2 systems: Interpretation of a high level intermediate code. The source text is compiled into a high level intermediate code by means of lexical and syntactical preprocessing. Typically, there is a one-to-one mapping from statements in the high level language to statements in the intermediate code. Most APL and SNOBOL4 systems are implemented in this way. Most LISP systems use only lexical preprocessing of the source text.

Type-2 systems have the advantage that syntax errors are detected before the execution phase of the HLL program. The intermediate code is of a high level and depends only on the high level language and not on an underlying (low level) machine language. As a consequence, maximal flexibility is possible at run time and it is easy to provide diagnostics and debugging facilities at the source language level.

(iii) Type-3 systems: Interpretation of a low level intermediate code. The source text is compiled into a low level intermediate code, which is close to the machine language level. There is a one-to-many mapping from statements in the high level language to statements in the intermediate code. The Pascal-P compiler,[4] for example, performs a lexical, syntactic and (modest) semantic analysis of Pascal programs and transforms these into low level P-CODE programs. P-CODE programs are then executed by a very simple interpreter.

Type-3 systems have the advantage that both syntactic and certain semantic errors (for instance, multiplying a floating point number by a file) can be detected before the execution phase of the HLL program. At run time one loses flexibility but gains efficiency. These systems tend to be rather portable, since implementation of a (simple) interpreter is sufficient to port the whole system.

(iv) Type-4 systems: Execution of machine language. This method is the same as the previous one, with the exception that an intermediate code is used that can be interpreted ('executed') directly by a computer. Traditional compilers operate in this manner. It is now immediately clear why type-4 systems are more 'efficient' than type-2 or type-3 systems: in type-4 systems interpretation is done at a level that is supported by (fast) hardware facilities.

With this subdivision in mind, we can try to analyse the arguments pro and contra interpretation (type-2) and compilation (type-3 and type-4).

As indicated above, compilers allow more *efficient* execution of HLL programs, since they transform the HLL program to a form which can be interpreted directly by the hardware. Another reason for this efficiency is the fact that most compilers determine once and for all certain static properties of the HLL program. Static analysis leads to the detection of some classes of semantic errors and allows the generation of optimized code. However, static ('compile time') analysis of a source program is not the prerogative of compilers. This kind of analysis does not depend on the way the program is finally executed on a computer. Two well known programs performing this kind of analysis ('verifiers') are PFORT[5] and DAVE.[6] Both programs try to determine certain semantic properties of FORTRAN programs, but are deliberately unconcerned about the way these programs will be executed. In the same way, one can envisage interpreter based systems that contain a static analysis phase for the detection of semantic errors and for the gathering of information that can be used by the interpreter later on.

In many situations, compiler based systems are superior to interpreter based systems, with regard to execution speed. But compiler based systems have their drawbacks. Compilers are complex programs with corresponding development and maintenance costs. Interpreters tend to be much simpler. The reason for this difference in complexity is the following. A compiler transforms a program in a high level language (defined by a fixed set of semantic rules) into a program in a low level language (defined by another set of semantic rules). This transformation is supposed to preserve the meaning of the HLL program. An interpreter deals only with the semantic rules of the high level language. There is no need for a difficult meaning preserving transformation to a lower semantic level. The only task of the interpreter is to realize the semantic rules of the high level language.

How does this difference in complexity affect *portability*? Two separate problems must be solved to achieve a successful transportation of a high level language implementation. First, the language processor itself must be ported from a host machine to a target machine. Second, the functional specifications of the language processor must be adapted to requirements imposed by the target machine. In general, the first step is easier for interpreters than for compilers. Above, we saw that interpreters are simpler programs than compilers and it is evidently easier to transport a small, simple program than a large, complex one. For a compiler the second step consists of modifying the code generator, which should now produce code for the target machine and not for the host machine. A significant effort may be needed for these modifications, because the code generation phase tends to be a substantial part of a compiler. (See Reference 7 for a survey of problems likely to be encountered during such a transportation.) For interpreters the second step is not needed at all, since there is no dependency on any machine language whatsoever. One may conclude that interpreters are easier to port than compilers.

For a number of years it seemed that this debate had come to an end with a conclusion in favour of compiling. But in recent years the interest in interpreter based systems has been growing. There are various reasons for this.

With the advent of (a bewildering variety of) microprocessors the ratio of software cost to hardware cost is rising steeply. This implies that the importance of software portability will increase and the importance of efficiency (in the traditional sense) will decrease.

A second reason for this revived interest in interpretation techniques is the current trend toward 'very' high level languages. In such languages the primitive operations are so complex and time consuming that it is irrelevant (with regard to execution time) whether they are compiled or interpreted. This being the case, one can profit from the advantages of interpretation (flexibility, debugging tools).

In the light of these arguments I want to make a plea for a way of implementing high level languages that combines static analysis with interpreter based (type-2) execution. In this way one obtains:

(a) Detection of syntax errors and semantic errors.
(b) Flexibility (inherent in interpreter based systems).
(c) Portability (no target machine code dependency).
(d) Programming aids at the source language level.

3. THREE INTERPRETATION TECHNIQUES

The preceding considerations justify a study of the qualitative and quantitative aspects of existing interpretation techniques. Three common methods will be compared here:

(i) Classical interpreter with opcode table
(ii) Direct threaded code
(iii) Indirect threaded code

In Section 3.1 some global properties and basic differences of these techniques are described. Sections 3.2–3.4 give detailed descriptions of each technique together with general remarks on their applicability. Section 3.5 treats instruction fetch timing and contains the result of some measurements on a PDP11 and a CDC CYBER73. These measurements are placed in the right perspective by considering the influence of instruction fetch and instruction complexity on the total execution time for each instruction. Section 3.6 deals with space requirements.

3.1. Overview

There are two significant differences among the three interpretation techniques: the means by which the operation code is represented and the means by which the dynamic (run time) operand type can influence operator selection. The classical interpretation method (CLASS) uses compactly encoded operation codes, which must be looked up in a table in order to identify the operation; type-dependent classification of the operator, if necesssary, occurs by explicit testing.

With direct threaded code (DTC), the operation code is simply the address of the required subroutine to execute the operation. This eliminates one table-lookup. It is easy to add new operation codes for specific operations a specific program may need. It is possible to combine static operator and operand information in a single address; for example, a new opcode/address could easily be constructed to 'add to the value of variable V', in contrast with the classical method where it is necessary to parameterize the 'add' operation code with variable V. If the interpreter routine is dependent on the operand type, this must be analysed within the interpreter routine just as for CLASS.

With indirect threaded code (ITC), the address of the routine to be used is obtained from the value itself. Each value contains the addresses of the routines for the common unary operations on values of that type, such as 'fetch', 'store' and 'call'. Compared to the CLASS method, the opcode table is as it were distributed over all values that occur in the program. The advantage of this method is that almost no overhead is required to select an interpreter routine depending on the operand type: the selection is performed

when the value is constructed and not when an operation on it is performed.

The following discussions are based on the assumption that a high level language program is translated to an *IC-program*, which consists of *instructions* and *data*. Interpretation of this IC-program requires a set of *interpreter routines*. The IC-program and the interpreter routines reside in a linear array *MEM* of heterogeneous cells. *MEM* models the whole memory that is available for the execution of the IC-program. A global variable *PC* (program counter) keeps track of the current instruction in the IC-program. All interpreter routines have access to this variable.

One example will be used throughout to illustrate each technique: an IC-program for the addition of two real variables *A* and *B*. This example is adapted from an example in PDP11 assembly language given in Reference 10.

3.2. Classical interpretation (CLASS)

3.2.1. *Description*

This technique is so old and so ubiquitous that it is hard to give a unique reference to the literature. The general scheme for classical interpretation is:

```
CLASS. 1: INCREMENT(PC);
CLASS. 2: execute routine at location opcode_table[MEM[PC]];
          (this routine returns to CLASS. 1)
```

This interpretation loop is explicit in a classical interpreter, as opposed to the two other techniques. Step *CLASS*. 2 may be more complex in reality, when it is difficult to isolate the actual opcode from the value *MEM*[*PC*].

The generated code and required interpreter routines for the addition of the real variables *A* and *B* are:

```
{ IC-program and data (compiler generated)                }
start: Ipush      { index in opcode_table                 }
       A          { index in MEM                          }
       Ipush      { index in opcode_table                 }
       B          { index in MEM                          }
       Iaddreal   { index in opcode_table                 }
{ data }
A:     0.0        { variable A; initial value 0.0         }
B:     0.0        { variable B; initial value 0.0         }

{ interpreter loop and routines (permanent)               }
{ assume:                                                 }
{      opcode_table[Ipush] = push                         }
{      opcode_table[Iaddreal] = addreal                   }

interpreter_loop:
          INCREMENT(PC);
          goto opcode_table[MEM[PC]];
push:     INCREMENT(PC);
          stackreal(MEM[MEM[PC]]);
          goto interpreter_loop;
addreal:  stackreal(popreal()+popreal());
          goto interpreter_loop;
```

All named entities, except *PC* and *TMP*, are integer constants. Variables are denoted by the index in *MEM* where their value is stored. In this notation all memory references are explicit. When Algol-like notation uses '*A*' to denote the value of a variable, we write '*MEM*[*A*]' instead. The entities *PC* and *TMP* behave like ordinary variables. (Purists could prefer to replace all occurrences of *PC* and *TMP* by *MEM*[*pc*] and *MEM*[*tmp*].) The IC-program consists of a sequence of integer values, which are either indices in the opcode table (*Ipush, Iaddreal*) or operands (*A*, *B*). In the examples '*label:contents*' stands for '*MEM*[*label*] = *contents*'. When such a label specification is omitted, the next memory locations is intended.

Execution of the IC-program starts at the label '*start*'. At that moment the condition $PC = start - 1$ holds.

3.2.2. *Remarks*

(a) The format of CLASS instructions can be chosen freely, may be even differently for each instruction. As a consequence, Huffman encoding can be used to reduce program size at the cost of increased instruction decoding times. See Reference 8 for an application of this technique to the design of B1700 S-languages (special intermediate languages which are generated by high-level language compilers and interpreted by B1700 microprograms). In this particular case, a 39–43 per cent size reduction and a 2·6–17·2 per cent decoding time penalty were observed. Special encoding has the disadvantage that creation of arbitrary bit patterns is needed for the generation of CLASS instructions.

(b) The CLASS instructions do not contain information that depends on a particular version of the interpreter. Compare this with the two other techniques, where addresses of interpreter routines are part of the instructions. All interpreter routines, whether used or not, are part of the combination interpreter/IC-program (see 3.3.2.(b)).

(c) The CLASS method does not accommodate operations, which depend on the operand type. The meaning of all occurrences of a particular instruction is the same and discrimination on operand type can only be accomplished by explicit testing in the interpreter routine.

The CLASS method makes it easy to change the meaning of all occurrences of a particular instruction by modifying the opcode table. This can be used for the implementation of procedure call tracing.

3.3. Direct threaded code (DTC)

3.3.1. *Description*

Direct threaded code (DTC) was first described in Reference 9, but had been in existence before that time. It has, for example, been used on the IBM 1620, where sequences of addresses turned out to be faster than subroutine jumps. The basic scheme for DTC is:

DTC. 1: *INCREMENT*(*PC*);
DTC. 2: *execute routine at location MEM*[*PC*]
(*this routine returns to DTC*. 1)

DTC was as it were 'invented' for the PDP11. On that machine DTC can be realized very efficiently by means of the instruction '*jmp* @ (*r*)+', where *r* stands for some PDP11 register that acts as *PC*.

In the CLASS case, instructions are connected by their sequential ordering. In threaded code, the instructions are as it were threaded together like beads on a chain.

DTC and ITC programs in this paper are represented as sequences of addresses, in accordance with the techniques as originally published. It may well be better on some machines to implement them as a series of subroutine jumps.

The DTC version of the example given previously is:

```
{ IC-program and data (compiler generated)                    }
  start:   pushA    { index in MEM = start address of         }
                    { push-value-of-variable-A routine        }
           pushB    { index in MEM = start address of         }
                    { push-value-of-variable-B routine        }
           addreal  { index in MEM = start address of         }
                    { routine to add two real numbers         }
{ data }
A:         0.0      { variable A; initial value 0.0           }
B:         0.0      { variable B; initial value 0.0           }
{ variable access routines (compiler generated)               }
pushA:     TMP := A; goto L;
pushB:     TMP := B; goto L;
L:         stackreal(MEM[TMP]);
           INCREMENT(PC);
           goto MEM[PC];
{interpreter routines (permanent)                             }
addreal:   stackreal(popreal()+popreal());
           INCREMENT(PC);
           goto MEM[PC];
```

It is assumed that $PC = start$ holds before execution starts. The IC-program consists of addresses (in our case: indices in *MEM*) of routines, which are generated by the compiler (*pushA*, *pushB*) or are part of the interpreter (*addreal*). The use of special access routines for variables is not essential for DTC. It is only more efficient, since this eliminates the need for an extra operand as is the case when a general push routine is used.

3.3.2. *Remarks*

(a) The format of DTC instructions is fixed and consists of addresses. Sometimes, however, immediate operands are part of the IC-program. These can be fetched via *PC* and it is the responsibility of the current instruction to give *PC* a proper value for his successor. This fixed instruction format renders special encoding impossible. Creation of hardware instructions and addresses is needed for the generation of DTC instructions.

(b) Instructions depend on a particular version of the interpreter (i.e. addresses of interpreter routines). A linkage editor is required to bind IC-programs to the interpreter routines. In most operating systems, a linkage editor makes *copies* of the modules that are to be tied together. This has two consequences:

   (i) The disadvantage that the IC-program is bound to copies of interpreter routines, which may cause proliferation of copies of obsolete or erroneous interpreter routines. More disk space may be required to store the resulting object modules.

(ii) The advantage that only copies of *used* interpreter routines are added to the IC-program.

(c) DTC does not accommodate operations, which depend on the operand type, but the fixed instruction format makes it feasible to add new instructions.

(d) As indicated above (3.3.1.), DTC is most efficient when special routines for loading and storing variables are used. Such routines are specific for each IC-program and can not be part of a (shared) interpreter that executes several IC-programs simultaneously.

3.4. Indirect threaded code (ITC)

3.4.1. *Description*

Indirect threaded code (ITC) was first described in Reference 10 as used for the implementation of MACRO SPITBOL. ITC is also used for the implementation of FORTH.[11] See Reference 12 for a description of a FORTH system with microcoded (built-in) subroutines. The basic scheme for ITC is:

ITC. 1: *INCREMENT*(*PC*);
ITC. 2: *execute routine at location MEM*[*MEM*[*PC*]]
(*this routine returns to ITC*. 1)

It must be noted that ITC is in fact a generalization of CLASS. In the CLASS case, the opcode table is fixed and is part of the interpreter. In the ITC case, the opcode table is scattered through the IC-program and can be modified.

The result of using ITC for our sample expression $A+B$ is:

```
{ IC-program and data (compiler generated)                          }
. start:     A        { index in MEM                                }
                      { MEM[A] = address of routine push            }
                      { MEM[A+1] = value of variable A              }
             B        { as above for B                              }
             addreal  { index in MEM = address of address           }
                      { of routine to add two reals                 }
{ data }
A:           push     { index in MEM = start address of push        }
A+1:         0.0      { value field of variable A; initial 0.0      }
B:           push
B+1:         0.0      { value field of variable B; initial 0.0      }

{ interpreter routines (permanent)                                  }
push:        stackreal(MEM[TMP+1]);
             { TMP is always equal to MEM[PC] }
             INCREMENT(PC);
             TMP := MEM[PC];
             goto MEM[TMP];
addreal:     addreal+1 { points to actual entry point of addreal routine }
addreal+1:   stackreal(popreal()+popreal());
             INCREMENT(PC);
             TMP := MEM[PC];
             goto MEM[TMP];
```

It is assumed that $PC = start$ and $TMP = MEM[PC]$ hold initially. Interpreter routines to which the IC-program refers directly (*addreal*), contain as first entry a pointer to their actual code. This is a consequence of the extra level of indirection in ITC.

3.4.2. *Remarks*

(a) The instruction format is fixed (see DTC). In this case, there is less need for immediate operands, since even constants can be treated in the same way as variables, without the need to introduce additional access routines. For the generation of ITC code it is sufficient to generate addresses only (compare this with DTC where both addresses and machine code have to be generated). This improves the portability of a code generator for ITC code. The latter is demonstrated by the fact that the MACRO SPITBOL system,[13] which uses ITC, has been implemented on nearly twenty different computers in a very short time.

(b) ITC accommodates operations, which depend on the operand type. This has the advantage that special cases (such as value tracing of variables, input/output associations) can be treated without additional overhead. However, there are problems with the modification of *all* occurrences of a particular instruction, which can be done so easily in the CLASS case. References to routines that contain an extra level of indirection at their entry point (like addreal) can be changed easily, but a scan through the IC-program is needed to change the references to all routines without such an entry point (push).

3.5. Time considerations

In this section we consider the time behaviour of the various techniques. First we pay attention to *instruction fetch* time. The techniques have been implemented both on a PDP11/45 with cache memory and a CDC CYBER-73. For each technique the effect of keeping the program counter in a machine register or in a variable (memory location) was considered. The way in which these techniques were implemented favours classical interpretation and CYBERs. It is assumed that no bit handling is required to isolate the opcode in the case of classical interpretation, and all CYBER programs use 60 bit words for each opcode for the same reason. Tables I and II show the results of these measurements. The time per instruction is given in microseconds. The additional entry HARD indicates the time needed for hardware instruction decoding, i.e. execution of a program consisting of dummy (noop) instructions.

It is very tempting to try to relate these figures to the architectural differences between PDP11 and CYBER. If we compare the cases HARD and CLASS, it is remarkable that the execution time decreases with a factor 0·09 on the CYBER and only with 0·41 on the PDP11. Relatively slow memory accesses on the CYBER are probably responsible for this phenomenon. In reality, this effect will even be enhanced by the fact that, regarding instruction fetch, byte addressing can be used to advantage on the PDP11, while bit manipulation is needed on the CYBER.

A second point that should be noted is the effect of keeping *PC* in a machine register in the DTC case. On the PDP11, execution time decreases with a factor of 0·33, on the CYBER only with a factor 0·54. This is probably due to the fact that the PDP11 can realize DTC in one instruction, provided that *PC* is kept in a register. On the PDP11, the difference between the three techniques disappears completely, when *PC* is not kept in a register!

Table I. Timing on PDP11/45

	Program counter in variable		Program counter in register		
	Time/ instr	Ratio CLASS	Time/ instr	ratio CLASS	Ratio reg/var
CLASS	7·59	1·0	3·58	1·0	0·47
DTC	6·86	0·9	2·24	0·63	0·33
ITC	7·86	1·04	3·50	0·98	0·45
HARD	—	—	1·45	0·41	—

Table II. Timing on CYBER-73

	Program counter in variable		Program counter in register		
	Time/ instr	Ratio CLASS	Time/ instr	Ratio CLASS	Ratio reg/var
CLASS	10·96	1·0	5·62	1·0	0·51
DTC	7·93	0·72	4·31	0·77	0·54
ITC	9·08	0·84	5·90	1·05	0·65
HARD	—	—	0·49	0·09	—

Several conclusions can be drawn from these figures:

(a) PDP11s are faster than CYBERs (when applied to the task of software instruction decoding).
(b) DTC with program counter in a register is the fastest technique.
(c) On the PDP11, it is crucial to keep the program counter in a machine register. If not, all methods show the same performance. This observation has implications for the choice of higher level system implementation languages: *if* the language does not allow you to place global entities (i.e. the program counter) in a register *then* do not worry about the time properties of your interpretation technique.

It must be emphasized that we have measured times for *instruction fetch* and not for *instruction execution*. This means that the significance of these measurements decreases when the execution time per instruction increases. Table III shows the effect of adding an increasing number of dummy operations to the instruction fetch cycle. These measurements show a not very surprising effect: the difference between the three techniques disappears when the execution time per instruction increases. The measurements were done on a PDP11/45 and noops were added to the interpretation cycle.

In a typical Pascal interpreter (type-3 system), only a few machine instructions are needed to implement operations like 'push value', 'assign', 'copy byte', that occur frequently in the IC-program. Under these circumstances, instruction fetch overhead can be substantial. In a typical APL interpreter (type-2 system), many (10–1000) machine instructions are needed to implement frequently occurring operations. In this case instruction fetch overhead is irrelevant.

3.6. Space considerations

It is difficult to give general estimates for the space requirements of programs using one of the three interpretation techniques. These depend very much on each specific

Table III. Effect of increasing instruction execution times

	1	2	3	4	5	6	7	8	9	10
CLASS	4·48	5·44	6·10	6·88	7·92	8·94	9·84	10·78	11·42	12·22
DTC	3·14	3·94	4·84	5·66	6·54	7·62	8·60	9·48	10·26	11·06
ITC	4·42	5·16	6·60	7·30	7·82	9·10	9·72	10·52	11·28	12·66

application. One can conjecture that program sizes increase according to: CLASS, ITC, DTC, HARD. This conjecture is based on the following facts:

(a) The CLASS method allows very concise encoding schemes.
(b) In ITC, the IC-program does not have to contain operations for explicit runtime checks, since such checks can be made part of the routines associated with individual values. For example, it does not have to influence the size of the IC-program whether array bound checking is enabled or not; this difference can be realized by associating a different (i.e. checking or not checking) array access routine with each array value.
(c) DTC requires separate routines for accessing each individual operand in an IC-program. In all these cases ITC contains only two addresses and one (common) access routine.
(d) In general, machine language is not geared to the implementation of any particular high level language. There is no way to take advantage (in space *or* time) of properties of a high level language. It is not possible to introduce new machine language instructions for frequently occurring sequences of operations. (This can be done by microprogramming, but that is a different story).

ACKNOWLEDGEMENTS

I want to thank Hendrik Boom, Jan Heering, Marleen Sint and Arthur Veen for reading earlier versions of this paper. Their comments and suggestions have resulted in substantial improvements.

REFERENCES

1. G. J. Myers, *Advances in Computer Architecture*, Wiley, 1978.
2. W. M. Waite, *Implementing Software for Non-numeric Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1973.
3. P. J. Brown, *Macro Processors and Techniques for Portable Software*, Wiley, London, 1974.
4. K. V. Nori, U. Amman, K. Jensen and H. H. Naegeli, 'The PASCAL (P) compiler: implementation notes', *report 10*, ETH Zurich, 1974.
5. B. G. Ryder, 'The PFORT verifier', *Software—Practice and Experience*, **4**, 358–377 (1974).
6. L. J. Osterweil and L. D. Fosdick, 'DAVE—A validation error detection and documentation system for FORTRAN programs', *Software—Practice and Experience*, **6**, 473–486 (1976).
7. A. S. Tanenbaum, P. Klint and W. Bohm, 'Guidelines for software portability', *Software—Practice and Experience*, **8**, 681–698 (1978).
8. W. T. Wilner, 'Burroughs B1700 memory utilization', *AFIPS FJCC*, **41**, part I, 579–586 (1972).
9. J. R. Bell, 'Threaded code', *CACM*, **16**, 370–372 (1973).
10. R. B. K. Dewar, 'Indirect threaded code', *CACM*, **18**, 330–331 (1975).
11. C. H. Moore, 'FORTH: a new way to program a mini-computer', *Astron. Astrophys. Suppl.*, **15**, 497–511 (1974).
12. J. B. Philips, M. F. Burk and G. S. Wilson, 'Threaded code for laboratory computers', *Software—Practice and Experience*, **8**, 257–263 (1978).
13. R. B. K. Dewar and A. P. McCann, 'MACRO SPITBOL— a SNOBOL4 compiler', *Software—Practice and Experience*, **7**, 95–113 (1977).