CLM-P258

United Kingdom Atomic Energy Authority

RESEARCH GROUP

Preprint

THE DESIGN OF PORTABLE ABSTRACT MACHINES

P. C. POOLE
W. M. WAITE

Culham Laboratory
Abingdon Berkshire

1971

CLM - P 258

THE DESIGN OF PORTABLE ABSTRACT MACHINES

by

P.C. POOLE
W.M. WAITE*

A B S T R A C T

Portability is a measure of the ease with which a program can be transferred from one computer to another. The techniques of abstract machine modelling and macro processing can be used to construct software which is highly portable. The key to the whole process is the design of suitable abstract machines and, in this paper, three such machines are described to illustrate the design process.

* Dept. of Information Science, Monash University, Clayton, Victoria 3168, Australia.

U.K.A.E.A. Research Group
Culham Laboratory
Abingdon
Berks

January, 1971.

C Ø N T E N T S

ERRATA

Page 1, line 5 - "algorithm"

Page 6, line 25 - "characters or fixed-length strings"

Page 12, line 24 - FREG A = IREG B

1. INTRODUCTION

Portability is a measure of the ease with which a program can be transferred from one computer to another. We say that a program is portable if the effort required for such a transfer is considerably less than the effort required to recode the program from the original alogorithm. Programs written in Fortran or Algol are portable if one assumes that the effort required to implement the compiler and run-time routines is spread over many programs. We have concerned ourselves mainly with programs for which these languages are not suitable. Our technique for producing portable software [1,2] is based on the design of a set of abstract machines, each of which can easily be realized on existing computers.

The fundamental concept of our approach is that, given a particular task, it is possible to postulate a special purpose computer (an abstract machine) which is well designed for carrying out that task. A program to perform the task is then written in the language of the abstract machine. The abstract machine exists only as an imaginary model of the basic operations and data types required to solve the problem. To actually run the program, we must realize the abstract machine on some available computer.

We favour a macro processor as the tool to carry out the realization, and have used STAGE2 [3] for this purpose. STAGE2 is a flexible, powerful processor which is itself highly portable. Currently it has been implemented on 20 different computers, requiring about one man-week of effort to obtain a running version in each case. It provides all the features normally associated with a general purpose macro processor - conditional expansion, iteration, parameter conversion, etc. In fact, it is powerful enough to translate Algol-like algebraic

languages [4].

The key to the whole technique is the design of suitable abstract machines. Three points must be taken into consideration:

(1) The convenience of the abstract machine language and its suitability for expressing the particular algorithm for which the machine is designed.

(2) The relationship between the abstract machine language and the structure of available computers.

(3) The limitations imposed by the tools used to convert the abstract machine language to a language for the real machine.

In this paper we shall discuss three of the abstract machines which have been designed so far. Our purpose is to illustrate the design process by example, and to show how the above three points have affected it.

Section 2 is devoted to FLUB, the machine designed to implement STAGE2. FLUB is unique because of the severity of the limitations imposed by the tools used to realize it. TEXED and INTERP, the two machines discussed in Section 3, are not subject to these constraints. We conclude with some general remarks on the design process.

2. THE FLUB MACHINE

FLUB (First Language Under Bootstrap) is the abstract machine designed specifically for the task of constructing the STAGE2 macro processor. STAGE2 uses three types of data: strings, trees, and integers. Thus the basic organization and operations of the FLUB machine must be suited to manipulating these data types.

The representation chosen for a tree was a slight modification of that presented by de la Briandais [5]. This representation

determined the basic composition of the FLUB word. Figure 1 shows a tree containing the strings CAT, COT and DOT. Notice that each word is divided into three fields. The flag (FLG) field contains indicator bits, the value (VAL) stores one character, and the pointer (PTR) is used as an address.

Given the structure of the FLUB word, a string is easily represented as a linked list of words. The VAL field of each word contains a character of the string, and the PTR field addresses the word containing the next character. Any substring of such a string may be specified by a word whose PTR field addresses the first character of the substring and whose VAL field contains the length of the substring. The VAL field must thus be long enough to hold either the largest character or the length of the longest substring.

There are several ways to represent an integer in a FLUB word: use the full word, use a combination of fields, or use a single field. At the moment there is no clear reason for choosing one of these representations over the others. Let us therefore defer the question temporarily and consider the operations required on each field.

The FLG field is used as an indicator, and thus operations which test and set this field are important. Because the VAL field is used to hold a string length, addition and substraction of VAL fields must be possible. It is also reasonable for the VAL field to be used in character I/O operations, since it holds characters for both trees and strings. Since the PTR field must hold an address, it can be used to hold the return address for a subroutine call. Addition and subtraction must be possible on the PTR field to provide for sequencing through a tree.

ADDRESS	FLG	VAL	PTR	
100	0	C	107	(Root of the tree)
101	0	A	104	
102	0	T	0	
103	1			(End of CAT)
104	0	O	0	(Continuation of COT)
105	0	T	0	
106	1			(End of COT)
107	0	D	0	(Beginning of DOT)
108	0	O	0	
109	0	T	0	
110	1			(End of DOT)

Figure 1

Representation of a Tree

Considering the operations, it seems reasonable to use the PTR field to represent an integer. This field is already the largest, and addition and subtraction operations are defined for it. Thus only multiplication, division and a test for relative magnitude must be added. The length of the pointer field will determine the range of integers allowed in a particular implementation.

So far, only the first of our three points has been used in the design of FLUB: we have considered only its suitability for macro processing. Now we must look at various real computers to see how their structure relates to that of FLUB.

The operations which we have proposed (integer arithmetic, conditional branching, etc.) are almost universal. It is quite uncommon, however, to find a computer whose words are partitioned into three fields of the types making up the FLUB word. There are two choices open to us: pack the fields of the FLUB word into one or more words of the target computer, or allocate one target computer word to each field. The first choice will minimize the space required to store information, but will result in large overheads to unpack and repack the fields for each operation. Opposite results (low overhead, maximum space) can be expected from the second choice.

We can escape this dilemma by providing the FLUB machine with a small set of 'registers', on which almost all operations take place. These registers can be implemented on the target computer with one word allocated to each field. If the number of registers is small, the space requirements are not prohibitive. The memory can then be packed to conserve space. Since memory is not accessed by most operations, the overhead of packing will not be excessive.

A register organization requires memory/register transfer instructions, and a way of specifying the memory address. Since the PTR field of a register can hold an address, it is reasonable to use this field as a memory address. Thus a memory/register transfer instruction will specify two registers: one participates in the transfer (either receiving or transmitting information), the PTR field of the other addresses the memory location.

Each FLUB memory word may occupy several locations in the memory of the target computer. In the implementation of STAGE2 for System/360, for example, each FLUB word is 8 bytes long; on the CDC 6400 each FLUB word is a single machine word. We have chosen to interpret a PTR field which addresses the FLUB memory as containing the target computer address of the FLUB word. To obtain the address of the next word of the FLUB memory, we must therefore increment this PTR by the number of addressable units per FLUB word.

The final consideration in the design of an abstract machine is the limitation imposed by the tools used to realize it. FLUB is unique in this regard, because it implements STAGE2. STAGE2 is therefore not available for the realization of FLUB. Instead, we must use a trivial macro processor called SIMCMP [6]. SIMCMP is only capable of handling simple substitution macros whose parameters are single characters. Unlike STAGE2, it has no internal memory or conditional expansion facilities and can only make one pass over the input text.

Because of the limitations of SIMCMP, we must restrict the operands of FLUB statements to single characters of fixed-length strings of characters. Two types of operand are required: register names and program labels. We have therefore given the FLUB machine 36 registers, and named them A-Z and 0-9. All program labels must consist of two

digits. Constants cannot be used as operands in FLUB. Instead, the registers named 0-9 are initialized by an external process to the values shown in Figure 2. Thus the instruction

```
PTR A = A + 1
```

increments the PTR field of register A by 1.

A complete list of FLUB statements is given in Figure 3. A single apostrophe represents a register name, and may be replaced by any letter or digit. Two successive apostrophes represent a program label, and may be replaced by any two digits. There are many facilities missing which one might expect to see even in a machine as simple as this one. It must be clearly recognised that the omission of such a facility is not due to any oversight in the design. Rather, it is indicative of the fact that such operations are not essential to any of the algorithms in STAGE2. The design of the machine has been deliberately kept as simple as possible commensurate with the requirement that such algorithms could be expressed both adequately and conveniently.

Remember that the main objective in designing the FLUB machine was the construction of the portable macro processor STAGE2. The only programs other than STAGE2 written in this language have been two test programs which validate the realization of the abstract machine and a simple editor to assist in the maintenance of distributed versions of the system.

Register	Initial value of FLG	VAL	PTR
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	-	4	-
5	-	5	10
6	-	6	-
7	-	7	Address units per FLUB word
8	-	8	First FLUB word address
9	-	9	Last FLUB word address

Figure 2

Initialization of Digit Registers

I. Data Transfer operations

A. Register - Register	FLG ' = ' VAL ' = PTR ' PTR ' = VAL '
B. Register - Memory	GET ' = ' STO ' = '

II. Integer arithmetic operations

A. VAL field	VAL ' = ' + ' VAL ' = ' - '
B. PTR field	PTR ' = ' + ' PTR ' = ' - ' PTR ' = ' * ' PTR ' = ' / '

III. Control operations

A. Unconditional	STOP TO '' TO '' BY ' RETURN BY '
B. Conditional	
1. FLG field	TO '' IF FLG ' = ' TO '' IF FLG ' NE '
2. VAL field	TO '' IF VAL ' = ' TO '' IF VAL ' NE '
3. PTR field	TO '' IF PTR ' = ' TO '' IF PTR ' NE ' TO '' IF PTR ' GE '

IV. I/O operations

A. Character transfers	VAL ' = CHAR CHAR = VAL '
B. Record transfers	READ NEXT ' WRITE NEXT ' REWIND '

V. Pseudo operations

A. Program label definition	LOC ''
B. End of text	END PROGRAM

Figure 3

FLUB Statements

3. TEXED AND INTERP

In the previous section we discussed the considerations which led to the design of the FLUB machine. The final form of the statements was strongly influenced by the limitations of SIMCMP. Once FLUB has been realized on the target computer, STAGE2 is available and SIMCMP may be discarded. Thus SIMCMP does not influence the design of other abstract machines. Let us suppose that STAGE2, rather than SIMCMP, is used to implement FLUB and see what design changes would result.

The most obvious change is to broaden the allowable operands for FLUB statements. Parameters of STAGE2 macros are not restricted to single characters, but may be any character strings. Also, STAGE2 has conditional expansion facilities which can be used to make the generated code depend upon the parameter types. Thus the operands of the FLUB statements might be constants as well as register names.

Because STAGE2 has a memory, it is possible to have declarations which associate attributes with variable names. These attributes can then be inspected and used to control the expansion of a macro. It is therefore possible to name single words or arrays in memory and use these names in memory/register data transfer operations.

A more significant feature of STAGE2 is its ability to handle nested macro calls. SIMCMP is unable to do this, and thus FLUB operations must be directly translatable into machine code. By using nested macro calls, however, we may regard an existing abstract machine as the 'real' machine, and design a new abstract machine which is realized in terms of the existing one. The macros for the new abstract machine are defined in terms of calls on macros for the existing abstract machine which, in turn are defined in terms of operations on the real machine. Nesting of abstract machines can be carried to any

depth, but care must be taken to avoid a 'cumulative mismatch' - the situation in which each machine in the nest introduces significant inefficiencies.

The TEXED machine is built on FLUB by defining single operations corresponding to frequently-used sequences of FLUB statements. TEXED takes advantage of the declarative power of STAGE2, incorporating named constants, variables and arrays. It was designed to implement a comprehensive text manipulator, and hence provides a richer set of I/O operations than FLUB. The most important extension of FLUB was the addition of operations on pushdown stacks. These operations can be defined in terms of FLUB statements and array declarations, and are therefore 'macro' operations in FLUB.

When abstract machines are 'nested', with one being realized in terms of another, the implementor is free to bypass one or more steps in the expansion of a particular instruction. For example, the TEXED machine has been implemented on the ICL KDF9. This computer has hardware operations for manipulating pushdown stacks. Thus TEXED's stack manipulation instructions are realized directly in KDF9 machine code, bypassing the translation to FLUB. On a more conventional computer, where there would be no serious inefficiencies encountered, the realization of the stack operations could be left in terms of FLUB statements.

Consider a nest of abstract machines M_1, M_2, ..., M_k such that M_{j+1} is realized in terms of M_j for all $1 \leq j < k$. M_1 is realized in terms of some real computer. The definitions of the M_j $(j \neq 1)$ are independent of the particular computer on which M_1 is implemented. Thus implementing M_1 provides 'free' implementations of all the other M's. By making an additional effort, some of the macros defining

M_j ($j \neq 1$) could be rewritten in terms of the machine code of the target computer for efficiency. The INTERP machine provides an example of this approach.

INTERP has evolved as part of a project to develop a range of portable interpreters for procedure-oriented languages. It is quite similar to the model described by Randell and Russell [7] for the Whetstone Algol interpreter. The basic arithmetic instructions are carried out on the top cells of a stack. For example, the INTERP instructions ADD adds the top 2 cells of the stack, nests down 2 levels and then places the result in the top cell. It assumes that the hardware of the abstract machine includes a polymorphic adder which takes care of type conversions automatically. (The type of a quantity, as well as its value, is held in the stack). If the program were moved to a machine equipped with such an adder, then the arithmetic operations could be converted directly into machine code to take full advantage of this facility. However, since current machines are not equipped with such an adder, the expansion of these operations directly into machine code could be complex.

To avoid this complexity, INTERP is actually realized as a nest of abstract machines. The next 'lower' machine is equipped with a number of integer and floating point registers. If a number is moved between registers of different types, then type conversion is automatically carried out, i.e.

FREG A = IREGB .

takes the integer value from register B, converts it to floating point and loads it into register A. Since the machine is also equipped with conditional instructions of the form

IF TYPE(N') IS ' THEN GOTO ' .

which allows it to test the type of N1 and N2, then ADD could be carried out by first moving the operands from the stack to the registers, changing type if necessary, and executing instructions of the form

FREG A = B + C. or IREG A = B + C.

The former carries out floating point addition and the latter integer addition. Such operations can easily be realized on existing machines. It is important to note that without this nesting one would have great difficulty in taking full advantage of changes in machine architecture when the program is moved from machine to machine. Thus if the high level operations were omitted and the program was coded entirely in the low level operations, then one could only take advantage of special hardware features by modifying the code. The probability of introducing errors during such an operation is much higher than it would be if one were merely replacing machine independent macro expansions by machine dependent ones.

4. CONCLUSIONS

We have discussed the considerations involved in the design of abstract machines for the implementation of programs which are easily moved from one computer to another. Because we know of no algorithms for arriving at such a design, our approach has necessarily been one of presenting 'case studies'. Unfortunately these case studies are somewhat artificial, in that they cannot show all of the blind alleys, agonizing reappraisals and major failures which led to the final design of each of these machines. Suffice it to say that abstract machine design is still an art, but one in which it is possible (we hope) to become proficient.

5. REFERENCES

1. Poole, P.C. and Waite, W.M. Machine independent software. Proc. ACM Second Symposium on Operating System Principles, Princeton, N.J. October 1969.

2. Waite, W.M. Building a mobile programming system. Computer J., 13, 28 (1970).

3. Waite, W.M. The mobile programming system: STAGE2. Comm. ACM, 13, 415 (1970).

4. Gebala, S.G.E. Macro Parsing of Context - Free Language Ph.D Thesis, U. of Colorado, June, 1970.

5. de la Briandais, R. File searching using variable length keys. Proc. WJCC, 295. (1959).

6. Orgass, R.J. and Waite, W.M. A base for a mobile programming system. Comm. ACM, 12, 507 (1969).

7. Randell, B. and Russell, L.J. ALGOL 60 Implementation. (Academic Press, New York, 1964).