Bootstrapping the BCPL Compiler using INTCODE

by M. Richards

Abstract:

For a compiler written in its own language, there is the problem of choosing a good strategy for bootstrapping it onto a new machine. The method explored in this paper is the preferred mechanism for transferring BCPL and involves the use of an interpretive machine code called INTCODE. INTCODE is designed specifically for this purpose. Its design and the general strategy of using it in a transfer are described.

Computer Laboratory,
University of Cambridge,
Corn Exchange Street,
Cambridge,
England

August 1973

Bootstrapping the BCPL Compiler using INTCODE

The portability of a programming language is strongly influenced by its design, the structure of its compiler and the mechanism used to transfer it from one machine to another. Although the prime concern of this paper is to discuss a method of easing the bootstrapping problem, it is in order to survey the effects on a language of requiring it to portable, since decisions in this area have a considerable bearing on the subsequent bootstrapping problem.

A programming language is always a compromise between the differing and usually conflicting requirements of a large number of constraints and design aims. For instance, one often wishes to incorporate powerful high-level facilities into a language without, at the same time, jeopardising the efficiency of the compiled code. Alternatively, one may be under pressure (from users) to provide language extensions at a time when the compiler is already too large to fit comfortably in the machine.

The main effect of the portability constraint on a language is a reduction in the number of primitive facilities provided and the removal of most machine dependencies. Small machine independent languages are inherently portable, and only gross errors in compiler design will prevent such languages from being transferred easily. It is worth noting that much of the effort required to transfer a compiler is in the rewriting of the code-generator for the new machine, and that the size of the machine independent parts of the compiler are of little relevance. This suggests that portability is enhanced mainly by reducing the more fundamental facilities of the language such as the variety of data and storage types, the complexity of the calling mechanism for procedures, and the number of primitive expression operators, while many of the higher level features such as conditional commands and the scope rules of identifiers may, on the other hand, be left in without portability suffering significantly. BCPL [1,2] was designed to be inherently portable and, as a result, it has rather few primitive facilities. It has, for instance, only one data type, three storage types, a very simple procedure calling mechanism, and few expression operators, and it is possible to describe the language in terms of a very simple abstract machine whose machine code is a simple and natural interface between the machine independent part of the compiler and the code generator. Since there is only one data type, all variables, values of expressions, anonymous results, and arguments are of the same size and it is reasonable for the allocation of space for items in the run-time stack to be done by the machine independent part of the compiler, with a consequent simplification of the code-generator and an improvement in portability. The language has a wide variety of non-primitive linguistic facilities such as conditional commands and syntactic constructions to reduce the need for GOTO commands, but the only expression operators available correspond to the fixed point, logical and relational instructions common to most computers. Two additional operators provide facilities for forming and using machine addresses, and since these operators, like all the others, cannot check the types of their operands, they are dangerously powerful. In many respects, BCPL can be regarded as a clean machine code in high level notation.

The interface language - OCODE

OCODE [3] is the name of the assembly language for the abstract BCPL machine. Its design is important since it is the interface language between the first phase of the compiler and the code-generator, and, like any other language, it must satisfy a number of constraints, the main one being that it must be capable of efficient code-generation. The OCODE form of an expression is basically the reverse polish translation with separate OCODE statements for each operation. For example, if x, y and z are local variables in positions 4, 5 and 6 of the current stack frame, then the OCODE translation of x/y + z would be:

```
LP 4 LP 5 DIV LP 6 PLUS
```

There are three fundamental operations for BCPL local variables: loading the value, loading the address of the variable and updating the variable, and LP, LLP and SP are the corresponding OCODE keywords. Similarly, the other two storage types, global and static, each have three OCODE statements for their translation. Thus, there are only 9 statements, in all, for accessing variables; in addition to these, there are 19 for the arithmetic, relational and logical primitives, and one for indirection which is also used for subscripted expressions and data structure selection. There are 5 statements for loading the various kinds of explicit constants available in the language, and remaining statements are mainly directives to the code-generator, or are concerned with procedure calls and jumps. Thus, the abstract BCPL machine can be programmed in a language containing fewer than 60 different simple statements.

It is instructive, at this stage, to consider the effect of language extensions on the complexity of OCODE. We have seen already that each storage type requires three OCODE statements; however, for each additional numerical data type in the language the effect is far more disastrous. We would require three new statements for each of the storage types and about 12 new statements for expression operators defined for the new data type. Unfortunately, the situation is likely to be even worse than this since it may be necessary to leave the space allocation to the code-generator which will, in consequence, require a more complex version of OCODE and a proportional increase in effort required to write the code-generator. For a BCPL-like language extended to contain real and long-real arithmetic, one would expect the corresponding OCODE to contain nearly 120 different statements. Many applications do not require real arithmetic and the improvement in portability resulting from its omission is attractive.

OCODE makes no provision for optimisation based on the analysis of the flow structure of the program, but optimisation at the local level is certainly possible and is performed by most code-generators. Particular care was taken in the design of the OCODE primitives for procedure definitions and calls so that there would be as wide a choice as possible in the details of the actual calling mechanism used.

Before INTCODE was developed, OCODE was the basis of the mechanism used to transfer the compiler to a new machine. At that time, the bootstrapping kit consisted of the source form of the compiler and a character representation of the corresponding OCODE form. To bootstrap the compiler, one first had to write a simple non-optimising code-generator for OCODE and then use it to generate code for the entire compiler from its OCODE form supplied in the kit. The first stage of the bootstrap was completed by combining this code with suitable interface routines to provide input, output and other operating system facilities. An optimising code-generator for the new machine could then be produced by suitably modifying an already existing one for some other machine; this being far less work than writing one from scratch.

OCODE is thus effective not only as an interface between the two halves of the compiler, but also as the basis of a method of bootstrapping. However, after completing several transfers using OCODE, it was found that the bootstrapping capability could be improved. OCODE makes more provision for optimisation than is necessary for bootstrapping purposes and, although a simple code-generator could be written, it required more knowledge and understanding of BCPL than was absolutely necessary. Thus, when the implementation of the bootstrap code-generator was undertaken by a programmer with no previous experience with BCPL, it often took longer than expected and frequently contained strategic errors in design. The solution was to take OCODE and to compile it into the assembly language of a second, even simpler, machine code for the BCPL abstract machine. The assembly language that was designed for this purpose is called INTCODE and it could be used in place of OCODE in the BCPL kit.

The INTCODE machine

Unlike a conventional computer, the INTCODE machine is not fully specified, and such details as the word-length, byte-size, and instruction format are left undefined. The machine has 6 control registers as follows: A and B are accumulators for computing expressions, C is the sequence control register giving the location of the next instruction to be obeyed, D is a register used to hold the computed address of the current instruction, and P and G are index registers. All these registers are the size of a machine word.

An instruction has a 3 bit function field, and an address field of unspecified size, 2 bits for index modification and an indirection bit. These fields may be laid out in the word in any way that is convenient for the interpreter. An instruction is executed as follows. Firstly, it is fetched from the store and C is incremented, then, the computed address is formed by assigning the address field to D, conditionally adding P or G as specified by the modification field, and indirecting if required. Finally, the operation specified by the function field is performed.

The 8 machine functions are: LOAD, ADD, STORE, JUMP, JUMP ON TRUE, JUMP OF FALSE, CALL, and EXECUTE OPERATION, and they are denoted in the assembly language by the single mnemonic letters L, A, S, J, T, F, K, and X, respectively. LOAD will assign the computed address to A after saving its previous contents in B. ADD will add D to A, and STORE will assign A to the storage location addressed by D. The effect of JUMP is to assign D to C, thus causing a transfer of control. JUMP ON TRUE and JUMP ON FALSE are conditional transfer instructions that test the value held in A. For these instructions, zero represents false and any non-zero value represents true. CALL is used in the compilation of a BCPL function or routine call. It increments P by the amount specified in D, saves the old value of P and the return address, and then jumps to the entry point held in A. The final instruction EXECUTE OPERATION provides a miscellaneous collection of arithmetic, relational, logical, and control functions, the actual function being determined by D. Most of the functions operate on B and A, usually leaving a result in A. For example, X7 will cause the remainder after the integer division of B by A to be assigned to A. There are 23 execute operations in the basic INTCODE machine, but for practical use, a further 5 to 10 are needed in order to provide an adequate interface with the operating system.

The assembly form of an INTCODE instruction consists of the mnemonic letter for the function, followed by 'I' if indirection is specified, followed by 'P' or 'G' if P or G modification is specified, and finally followed by the address. The address is either given explicitly as a decimal integer or as a reference to a label. A label reference is denoted by 'L' followed by the label number. A number not preceded by a letter is interpreted as a label setting directive and causes the specified label to be set to the address of the next item to be assembled. As an example, the following piece of BCPL program:

```
IF SW DO X := 126
Y := Y REM X
```

could be translated into the following INTCODE:

```
LIG103               / load SW
FL73                 / jump on false to label 73
L126                 / load 126
SP3                  / assign to X
73                   / set label 73
LIP4                 / load Y
LIP3                 / load X
X7                   / form remainder
SP4                  / assign to Y
```

In this example, SW is assumed to global 103, and X and Y to be the third and fourth local variables.

Data may be assembled using various data statements. For instance, the statement D163 will cause a word to be allocated with initial value 163, and DL46 will allocate a word holding the value of label 46 as initial value. String data can be assembled using character statements; for instance, the BCPL string "ABC" might compile into:

```
LL493    ...
493  C3  C65  C66  C67
...
```

In this example, the instructions LL493 will load the address of a region of store where the bytes 3, 65, 66, and 67, representing the string, are stored. They are packed according to the word-length and byte-size of the particular implementation.

Other facilities in INTCODE include directives for initialising global variables and marking the ends of segments of code, and a comment facility.

We can see from this description that INTCODE is an easy assembly language to learn and use, and that its assembler and interpreter are simple to write. The INTCODE kit of the BCPL compiler consists of the source form and the corresponding INTCODE translation of the compiler and that part of the library that is written in BCPL. The documentation includes a detailed description of INTCODE and a BCPL version of the assembler and interpreter. To bootstrap the compiler using this kit, one first rewrites and tests the assembler and interpreter in some suitable language. Next, one constructs a library to provide the necessary input and output routines. This library consists partly of hand-written INTCODE and partly of the compiled form of the BCPL library suitable modified (if necessary) for the new word-length and byte-size. These corrections are simple as most of this library is machine independent. Finally, the compiler can be assembled and tested. To simplify this last stage, several debugging aids are incorporated permanently into the compiler. Many of these are in the lexical and syntax analysers which are usually the first sections to be tested. There is, for instance, an option to print the current input character on every call of the lexical analyser, and there is another option to print the integer code of basic symbols as they are recognised. Once the lexical analyser works, the rest of the compiler usually works immediately and the options to print the syntax tree and the intermediate object code are provided mainly for their educational value in helping implementers to understand the compiler.

Summary and Conclusions

The OCODE mechanism provides a reasonable mechanism for portability, since its bootstrapping capability is good and, once the bootstrap is complete, it is possible to write (or modify) a code-generator to compile adequately efficient code. However, it was found that time could be saved by using INTCODE and the reasons for this are listed below.

a) Less knowledge and less work is required to construct the first bootstrap.

b) INTCODE is easier to learn and is more convenient to write or modify than OCODE, and so it is reasonable and useful to include many of the machine dependent parts of the library in the kit.

c) Bootstrapping an INTCODE version of the BCPL compiler is a useful educational exercise. It allows the implementer to learn BCPL, the specification of OCODE, and how the compiler works before he needs to write a new code-generator.

d) Little of the programming of the initial bootstrap need be discarded when the production code-generator takes over, since much of the original code is concerned with library routines which will still be required. Also, it has frequently been found that the interpretive system has sufficient advantage in size and convenience to merit its continued existence even after the production system is available.

e) The text of the INTCODE form of the compiler is more compact than the corresponding OCODE text and this reduces the amount of material comprising the kit. When using magnetic tape, this advantage is small, but when using cards or paper tape, it is too important to ignore.

In conclusion, INTCODE is a useful aid to simplifying the bootstrapping problem for BCPL, but it should be remembered that it, in itself, does not make the language portable. For a larger language such as Algol 68, portability is a much harder problem, since the abstract machine is larger and more complicated, reflecting the greater variety of data types and primitive operations. Furthermore, to obtain a reasonable level of optimisation, a larger proportion of the compiler will have to be machine dependent. Although many of the arguments for using an interpreter in the initial bootstrap are still valid, they hold less weight since the scale of the job is so much larger. Even so, the use of an interpretive scheme may prove beneficial in some circumstances.

References

[1] Richards, M. BCPL - A tool for compiler writing and systems programming. Spring Joint Computer Conference, 1969.

[2] ----------- The BCPL Programming Manual. The Computer Laboratory, University of Cambridge, 1973.

[3] ----------- The Portability of the BCPL compiler. Software, Practice and Experience, Vol. 1, No. 2 (1971).

[4] ----------- INTCODE - An interpretive machine code for BCPL. The Computer Laboratory, University of Cambridge, 1972.