PROJECT MAC

Computer Systems Research Division

December 2, 1975

Request for Comments # 96

THE IMPLEMENTATION OF ALGOL 68C ON MULTICS

from J. H. Saltzer

The attached memo was received from Professor Maurice Wilkes of the University of Cambridge. It describes a project, undertaken by M. Richards and J. Grant, to import an ALGOL 68 compiler to Multics using the ARPANET from the London TIP. Although the project was not completed, some interesting ideas are contained here, and the files (including an ALGOL 68 manual) are available for future use.

This note is an informal working paper of the Project MAC Computer Systems Research Division. It should not be reproduced without the author's permission, and it should not be referenced in other publications.

UNIVERSITY OF CAMBRIDGE

COMPUTER LABORATORY

THE IMPLEMENTATION OF ALGOL68C ON MULTICS

In early 1974 shortly after a connection to the ARPA net had become available to us, machine time was made available on Multics for us to embark on a project to transfer the Cambridge implementation of ALGOL68 (ALGOL68C) to the Honeywell 6180 computer which supports Multics. It was hoped that this project would give us some experience of the remote use of Multics and would also give useful practical experience of the transfer of the ALGOL68C compiler, one of the design objectives of which had been that it should be as portable as possible. Dr.Martin Richards was in charge of the project and he was assisted by Mr.J.Grant. It was realised at the outset that the task was an ambitious one to attempt with the limited manpower and resources available, but it was hoped that at the very least useful experience would be obtained.

Connection to the ARPA net

Connection is by means of a leased line to a TIP in London. At the Cambridge end there is a GT40 with a display and keyboard. The GT40 is programmable and equipped with cassette tapes for the storage of short files. The software for it was developed at Cambridge in connection with another project and has proved to be completely reliable and reasonably convenient to use. It provides facilities for driving the cassette tapes and the keyboard, and also for driving an Olivetti typewriter. It may be connected either to the TIP or to a local IBM 370/165 computer. The latter is not a host to a TIP and bulk data transfer must therefore be made via the cassette tapes. This limits the amount of information that may be transferred and the maximum data rate obtainable amounts in practice to about 30 characters per second. There would in any case have been the fundamental limit imposed by the fact that the connection across the Atlantic has a bandwidth of only 7.2K bits per second.

ALGOL68C

ALGOL68C is a substantial subset of ALGOL68 and its compiler has been under development at Cambridge for several years. It first ran on the Cambridge Multiple Access System which was implemented on an Atlas 2 computer, and it has since, after further expansion, been transferred to the IBM 370/165, a PDP 11/45, and the Cambridge CAP computer, as well as to other machines elsewhere. The compiler was designed under the direction of Dr. Bourne and much of the implementation work was done by himself and his two research students Andrew Birrell and Ian Walker. The resulting system has proved to be portable, particularly between machines whose architecture is similar to that of Atlas or the IBM 370. This portability was achieved by splitting the compiler into two parts, one being machine independent and concerned among other things with syntactic issues and the other being a code generator which is necessarily highly machine dependent. The interface between these two halves is a language called Z-code which resembles a typical assembly language in form but is, in fact, specifically designed for ALGOL68C.

Unlike the intermediate object codes of most portable systems, Z-code can be modified to take advantage of the characteristics of the target machine. A parameterised specification of the target machine must be supplied to the machine independent part of the compiler for each transfer to a new machine. This has the advantage of allowing much of the routine optimization of register and storage allocation to be done in a general way by the machine independent part of the compiler, thus making the highly machine dependent code generator smaller and easier to write than it would otherwise be. The main part of the parameterised machine specification gives the number of each kind of register and what types of value they can contain. With this information the machine independent first part of the compiler is able to generate appropriately optimised Z-code for machines with multiple registers, such as the IBM 370 or PDP 11. The optimising capability is made less use of in the case of a single accumulator machine such as the Honeywell 6180.

Implementation Strategy

The ALGOL68C compiler was more or less complete and working in the summer of 1974 and by January of 1975 it was sufficiently free of bugs to be fit for distribution. The first step of the transfer to Multics was to construct the parameterised specification of the Z-code that we proposed to generate and then compile the entire ALGOL68C compiler into Z-code using this specification. The only problems at this stage resulted from the large size of the compiler and the organisation of the many datasets involved. It turned out, for instance, that we needed our own private version of the compiler on the IBM 370 with one of the main tables slightly enlarged. The source code of the compiler amounted to about 12,000 lines of text and when compiled into Z-code it expanded to approximately 40,000 lines. This quantity of data was far too great to transmit to Multics using the net because of the low speed of our line and the small capacity of the cassette tapes, and so we chose to transfer the material on a magnetic tape that was physically sent to M.I.T. For this purpose we used standard IBM 9 track tape, since we did not consider that an attempt to write a standard Multics tape on our computer would be likely to succeed. The tape format chosen was one used when transferring BCPL to non-IBM installations and consisted of 80 character card images in which 36 EBCDIC characters of IBM data were represented by pairs of hexidecimal digits (0-9,A-F) in positions 1 to 72, the remaining character positions holding a sequence number. With the expert assistance of a Multics systems programmer during a brief visit to Project MAC, we were able to read and decode this tape with the aid of a PL/1 program and a newly available IBM magnetic tape package.

We were in this way able to transfer the entire source and Z-code forms of the compiler to Multics together with files containing among other things the language manual and a variety of test programs. It was clearly most desirable that the Z-code on the tape should be correct, since it would be both slow and inconvenient to send replacement tapes. The compiled Z-code was therefore checked by eye with great care. However, one problem with the specification of the target machine was not spotted at this stage. In the original specification we had allowed only one register (MQ) for reals and integers. However, it was soon realised that the compiler would only work if at least two such registers were allocated and so a second was added. At the time it was thought that this would cause no problem to the code generator since, if necessary, the second register could be implemented as a storage location. However, a problem did arise since most arithmetic operations needed to be done in the machine accumulator and so its current contents frequently had to be saved. Unfortunately, it was not always possible to determine from the Z-code whether the current value of the accumulator was an integer or a real and for this reason the code generator could not be designed so as to compile satisfactory code. It turned out, however, that the compiler used no floating point arithmetic at all, and so it was decided that for the initial bootstrap only integer arithmetic would be provided.

Having generated the Z-code for the entire compiler we had three possible strategies available to us. First, one could implement an assembler and interpreter on Multics. This was ruled out because of the expected inefficiency of the interpreter bearing in mind the size of the compiler to be interpreted. Secondly, a code generator could have been written and run on the IBM 370 at Cambridge. This would have allowed for easy program development since we were, of course, familiar with our own computing service and could take advantage of its excellent facilities to the full, including the simple but valuable ability to produce line-printer listings conveniently. However, this approach was not chosen since one of the primary aims of the project was to gain first hand experience with Multics and this approach would have meant that most of the development work would have been done on our own machine. There would also be a problem with the debugging of the massive quantities of assembly code generated since it would have had to be transmitted across the Atlantic before it could be test properly. Any errors found in this code would usually have indicated an error in the code generator and re-transmission of the corrected assembly code would have been necessary. We therefore chose the third possible strategy which was to implement the code generator on Multics having transferred the entire Z-code of the compiler once and for all by way of magnetic tape.

The Code Generator

It was clear that the code generator for Z-code should be written in a high level language and in retrospect perhaps we should have adopted a more coventional approach and used PL/1, since this is the standard Multics programming language. However, we had a particular interest in BCPL and chose to use it for the job. BCPL is a much simpler language than PL/1 and generates more straightforward code which is less well optimised but easier to follow. It was useful to study the code produced by this compiler to help us plan the code generator for ALGOL68C, and it was also interesting as an example of how to incorporate a foreign language into the PL/1 environment of Multics. One of the problems with BCPL on Multics is that its library is undocumented and so it took more time than we expected to learn how to use it effectively.

We designed the code generator to translate from Z-code into ALM, the assembly language for the Honeywell 6180, rather than into binary since we wished to avoid the need to learn the detailed format of binary object modules. Although the syntax of ALM is well documented, few Multics users find they need to use it and one needs to be a Multics expert to program in ALM effectively. Our basic documentation was the GE 645 Processor manual which was later augmented by a supplement. We chose, at an early stage, to generate very simple code in the hope that we could keep within the time scale available. For instance, we chose to use none of the Extended Instruction Set (EIS) instructions available on the H6180 and also chose to compile code so that all the ALGOL68C data were held in one segment. This allowed addresses to be held in the index registers of the machine but, of course, it meant that much of the multi-segment structure of Multics would not be accessible directly to this first version of the compiler.

Conclusions

From our point of view the project was generally successful and most educational to us. However, like so many projects it overran its planned time period and did not reach the level of development that we had hoped for initially.

When we chose to write a code generator rather than an interpreter we made the conscious decision to expend very little effort on optimisation, since we were only planning to perform the initial level of bootstrap. As work on the code generator progressed it became clear that the compiled code would be substantially more bulky than we had predicted. This was not a serious problem, since Multics was well capable of supporting large programs, but it did have indirect drawbacks. For instance, the cost of machine time to generate and assemble the code was higher than we had expected, and it was also a problem to find sufficient disc space for the files involved. We now realise that the cost of running an interpretive compiler, although high, would not have been substantially greater than the cost of constructing and debugging the compiled compiler. Thus, in retrospect, we might have done better to implement an interpretive system. We also found that Z-code was not nearly as well matched to the H6180 as it was to the IBM 370 and that in consequence the code generator itself was much larger than we expected.

Debugging the code generator at the BCPL level was not a difficult task since it was a fairly simple program, but debugging the assembly code that it produced was quite another matter, since we had to become experts in the rather complex inner structure of Multics. For instance, it turned out to be necessary to have a detailed knowledge of the implementation of many of the PL/1 data types, the coding of the standard calling sequence, and the layout and organisation of the stack and linkage segments. This knowledge was certainly of great interest and improved our understanding of Multics, but its acquisition took a long time and contributed to the delays in the implementation of the code generator.

At the conclusion of the project the code generator was virtually complete and compiling correct but inefficient code. The parts that were left unimplemented involved rare special cases that were thought never to occur in practice. In order for the compiled code to be run a package of input/output routines has to be provided to serve as an interface to the operating system. Much of this package is itself implemented in ALGOL68C and consists of two segments of code: system code, and user code. The former provided very primitive input/output procedures, many of which were defined with the aid of sections written in assembly code. The latter contained higher level ALGOL68C transput procedures defined in terms of the primitive ones. Both the system and user code segments were based on the corresponding segments in the implementation of ALGOL68C on the Cambridge CAP computer; however, their conversion for Multics was not completed in time.

A list of files remaining in Multics is given in the appendix in case anyone is able to take the project up. The ALGOL68C compiler, in machine-independent form, is currently supported by the University of Cambridge Computing Service.

APPENDIX

This appendix gives the names, purpose, and status of the more important files used in the implementation. Unless otherwise stated they all resided in the directory of J. Grant.

a) mr10>rrf.ex

This is the exec_com file that was used to extract documents from the ALGOL68 tape. The tape is currently in the Multics machine room and the use of the exec_com is most easily deduced by typing the file. The first file on the tape is called JCLADUMP and is the IBM job that wrote the tape. It is useful as a reference since it specifies the position and names of all the other recorded files.

b) mr10>manual

This is a copy of the ALGOL68C manual written by S.R.Bourne, A.D.Birrell, and I. Walker.

c) system.A68 and system.zcode

These are the source and Z-code files of the library of primitive input/output routines.

d) user.A68 and user.zcode

These are the source and Z-code files of the part of the library that provides the high level transput routines.

e) A68CTXT

This is the entire ALGOL68C source of the compiler.

f) A68CMPLZ

This is the entire Z-code compilation of A68CTXT compiled on the IBM 370/165 at Cambridge.

g) cg.bcpl

This is the source of the Z-code to ALM code generator

h) cg

This is the executable program corresponding to cg.bcpl. It is called with one argument which is the name of the Z-code segment to be compiled.