Short Note

A Transportation of Multiphase Compiler

SATORU KAWAI* and KIYOSHI ISHIHATA*

Some consideration on multiphase compilers aroused during a project of compiler transportation is presented. The concepts of phase, pass, and program separation are examined from the viewpoint of practical transportation. An implementation of Algol 68 compiler is described in order to clarify the actual process of transportation. Some other points which affected the project are also shown.

1. Phase and Pass of Compilers

Compilers of programming languages usually have several phases during translation. The source text of a program is converted step by step towards the final form. The process of the elementary conversion can be defined as a single phase of the compiler. It is worth noting that **phase** is a different concept from **pass** which is generally used for the characterization of compilers. A one-pass compiler need not be a single phase one. The usual cases are that more than one phase is "interleaved" with each other within a particular pass of compiling. The parts of the compiler corresponding to these phases work cooperatively like the elements of pipe-line control in the architecture of large scale computers. That interleaving is intended for the minimization of the amount of information which is to be stored in main or external storage for the purpose of communication between the phases.

Typical examples of phases are; lexical analysis, syntax analysis, and code generation. In the compilers for carefully designed languages, these three (and probably some other) phases can be interleaved into a single pass, forming a one-pass compiler. For some other languages such as Algol 68, on the other hand, it is impractical to get all the phases interleaved into one mainly because the information obtained in a particular phase is required, as a whole, to commence its successor phase.

Phase separation, on the other hand, has its own advantage against the interleaving described above. Though the required amount of and the number of access to the main and/or external storage would increase when the phases are separated, that separation would also increase the independency of phases. The subparts of the compiler corresponding to the phases might be programmed almost independently of each other, provided that the format of interface information is completely predetermined. When the whole compiler is to be modified, for example, because of the changing of the specification of the language or of the working environment under which the compiler works, the program segments corresponding to only few phases must be modified without any modification of other part of the compiler.

A transportation of compiler from one installation to another can be regarded as a task in which the changing of working environment of the compiler must be properly dealt with. If the phase of code generation is clearly defined in the compiler, that phase is to be modified so that the resultant code suits the new environment. Many compilers written in its own language have been transported in this way. Usual transportation process is as follows.

1) The part for code generation is replaced.
2) The modified compiler is compiled in the old environment by the old compiler.
3) The modified compiler is compiled in the old environment by the new (modified) compiler.

This process is illustrated in Fig. 1.

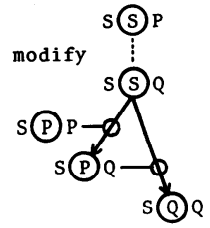

Fig. 1 A Bootstrapping.
S: source code
P: machine code of P (original machine)
Q: machine code of Q (target machine)
U (V) W: a converter from U to W which works on V-processor.
M —Φ: a conversion by M

2. ALGOL 68C and Its Transportation

ALGOL 68C is a dialect of Algol 68 developed at the University of Cambridge, England. The implementation for the processor of ALGOL 68C has two characteristic features; the two step compilation using an intermediate code, and the separate compilation facility.

The intermediate code is called ZCODE. It is gener-

*Department of Information Science, Faculty of Science, University of Tokyo, Tokyo 113, Japan.

ated by the first pass of the processor, named the **compiler**, and is used to generate final machine code by the second pass, named the **translator**. The reason of employing ZCODE is multifold. First, a single "compiler" can be used commonly for several target machines provided that the "translators" for these machines are prepared. Note that the complexity of a "translator" can be (and should be) much less than that of a "compiler". Second, local and global optimization of code can be done more easily by this pass separation. For this purpose, the user is allowed to specify some characteristics of "ZCODE machine", such as the size of unit storage and the number and the usage of general registers, when compiling programs. The final purpose of employing ZCODE is to make the processing system as portable as possible. The following steps are expected to be carried out for a transportation of the system.

1) The characteristics of ZCODE machine is determined so that the difference in architecture between it and the target machine becomes as small as possible.
2) The compiler is compiled by itself, using the specification of ZCODE in 1), to obtain a ZCODE version of itself.
3) A translator for the target machine is written on the old machine.
4) The ZCODE version of the compiler is translated by the new translator into the object code of the target machine.
5) The new translator on the old machine is also compiled and translated in the same way as the compiler.

By the separate compilation facility of the implementation of ALGOL 68C, any program can be split into pieces. The scheme of splitting is top-down manner, i.e., each piece of program except for the root is supposed to fill a hole in its parent segment. Any **unit** in terms of Algol 68 may be replaced by a hole which is called a **handle** in ALGOL 68C. The resultant form of a split program is a set of tree-structured program segments. They are compiled in the order of root-first tree traversal. A link edit program combines the compiled object modules into one to make the program executable.

3. Transportation Making the Use of Loader-Level Language

We had a plan to transport ALGOL 68C system from Cambridge to our installation. The followings are the points which had to be considered for the transportation.

As far as the code generated by the compiler (or more precisely by the translator) is concerned, there is no practical difference in the architecture and the set of instruction between IBM370/165 on which ALGOL 68C system was developed at Cambridge, and HITAC 8700/8800 which is our target machine. It was planned, therefore, to make the most of the translator on IBM, called Z370, for the transportation. However, the process was not straightforward because the input format of our linkage editor is completely different from that of IBM's. IBM's is card image format and then can be handled by the output facility of Algol 68, though "binary characters" must be "punched" on object module files. On the other hand, our linkage editor accepts only the files of special format for which it is very difficult to prepare an interface with Algol 68 output facility. It was, therefore, almost impossible to modify Z370 so that it produced directly the object modules of our machine.

Z370 has three phases; the input of ZCODE, the creation of program on an internal buffer, and the output of the program in linkage editor format. The whole text of the translator is segmented correspondingly. Considering this phase structure of Z370, we decided not to carry out the transportation using ZCODE (which was expected to be used by the original designer) but rather use a loader language as the tool for bootstrapping. The separate compilation facility also played an important role because the interface information among (subdivided) segments can also be handled within the loader language.

The key of our transportation is to determine the specification of the interface language, named **pseudo-loader code** (PLC). PLC must be closely related to the structure of our object module and, at the same time, must be generated by Z370 with little modification. We adopted the character representation of subroutine calls with parameters as the format of PLC. The explicit format is as follows.

O s_1 s_2	**open** the object module "s_2" in the file with definition name "s_1". s_1 and s_2 are strings of size less than or equal to 8.
P x y f	**put** data x of length y. f is the flag for absolute/relocatable.
N s x	create **entry** name "s" at location x.
X s x	create **external** reference "s" from location x.
C	**close** the object file.

An example of PLC is shown in Fig. 2.

Actual transportation process follows.

1) Replace the segment of Z370 corresponding to the final phase by i) code generation main routine, and ii) routines for output of pseudo-loader code (PLC).
2) Compile and translate (by Z370) the modified translator, named Z8800, on an IBM machine.
3) Compile and translate (by Z8800) both the compiler and Z8800. We then obtain the PLC version of the compiler and Z8800.
4) Make two routines on our machine one of which, named OBJ, is a utility subroutine for creating object modules and the other is the interpreter of PLC invoking OBJ.
5) Using the routines described in 4), convert the compiler and Z8800 from PLC form into the object module of our machine. Note that the compiler and Z8800 thus converted consist of a number of object modules corresponding to the original subdivision.
6) Link the objects of the compiler together.

```
O  A68OBJCT   PRINTSEG
P  183F  2  0
P  51430FFF     4  0
P  50C02108     4  0
P  D203210C     4  0
P  52F04EF7     4  0
P  51E02180     4  0
P  0D1F  2  0
P  98152004     4  0
P  07F1  2  0
P  0000  2  0
P  0001  2  0
P  03E0  2  0
P  0000000A     4  1
P  00000100     4  0
P  00000144     4  0
E  PRINTTOP   0000
E  SBRETURN   0018
X  SSREADER   0028
X  SSWRITER   002C
C
```

Fig. 2 Example of pseudo loader code.

7) Link the objects of Z8800 together, replacing the PLC output module by OBJ.
This process is illustrated in Fig. 3.

Besides the programming of the modification of Z370 and that of OBJ routines, the transportation required very little machine time for compiling, translating, and interpreting PLC. The "compiler" and the "translator" Z8800 are running successfully on our machine.

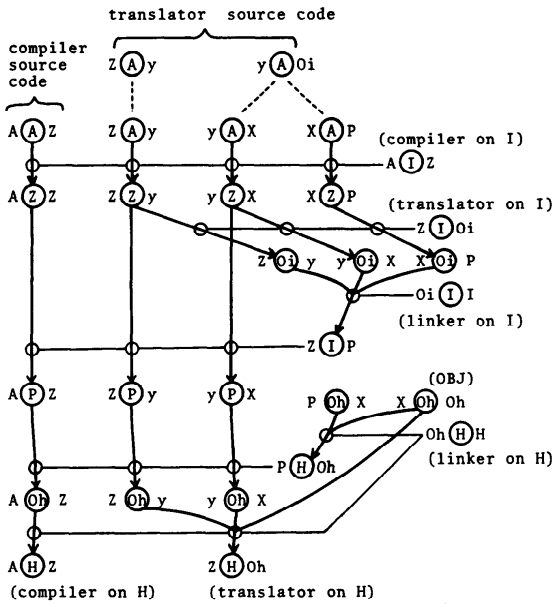

Fig. 3 The Transportation Scheme.
A: ALGOL68C
Z: ZCODE
y: an internal code
X: subroutine call sequence
P: PLC, pseudo loader code
I: IBM machine code
Oi: IBM object code
H: HITAC machine code
Oh: HITAC object code

It would be worthwhile to note the following points.
a) As described earlier, the special file format of object modules in our machine made the transportation rather complicated and less comprehensible. If it were not, as in OS/MVS, we could have modified Z370 so that it would generate directly the object modules of our machine, and the transportation scheme would be greatly simplified by omitting the PLC stage.
b) The procedure we adopted is considered very useful and efficient for the transportation between "OS-incompatible" systems. The two programming tasks necessary at each installation, translator modification and implementing OBJ, can be proceeded independently, after the specification of the intermediate code (PLC in our case) and that of the routine calls are determined.

4. Miscellaneous Comments

The original compiler is capable of handling all of the case, upper, quote, and point stroppings. That was a very helpful feature for the transportation because we had only restricted facility for handling lower case letters. We had to deal with another problem with respect to lower case letters caused by the unreasonable replacement of lower case letters by Kana characters in the code system of EBCDIK (not EBCDIC!).

The separate compilation facility of ALGOL 68C strongly recommends its users to use "collective files" in order to reduce the number of data definition statements in job control commands. This is because a user is obliged to manage three sets of files if the separate compilation facility is to be used intensively; source files, object files, and environment files which contain necessary information at all handles. In the IBM OS/360 or OS/MVS environment, a kind of files called partitioned data set is available for this purpose. In our case (OS-7), a similar kind of files exist. The difference between the two is whether a member of a file can be defined as a data set in JCL (IBM) and not (ours). This restriction imposed by our system influenced the transportation to a certain extent.

References

1. Lindsey, C. H. and Boom, H. J. A Modules and Separate Compilation facility for ALGOL 68, *Algol Bulletin* No. 43 (December 1978) 19–53.
2. Bourne, S. R., Birrel, A. D. and Walker, I. *ALGOL68C Reference Manual*, University of Cambridge Computing Service (1975).
3. Kawai, S. Lattice Structure Segmentation of ALGOL-like Programs, *Software—Practice and Experience*, Vol. 9, No. 6 (1979) 485–498.

(Received April 19, 1979)