A technological review of the FORTRAN I compiler

by F. E. ALLEN
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

ABSTRACT

The FORTRAN I compiler functions and organizations are described and shown to form the basis for many of the techniques used in modern compilers.

INTRODUCTION

Early in 1954, the FORTRAN I project was formed by John Backus. A fundamental question posed by the project was " . . . can a machine translate a sufficiently rich mathematical language into a sufficiently economical program at a sufficiently low cost to make the whole affair feasible?"[1] A major goal was to provide an automatic programming system which " . . . would produce programs almost as efficient as hand coded ones and do so on virtually every job."[1] This seemingly impossible goal was met to an astonishing degree. In some cases, it produced code which was so good that users thought it was wrong, since it bore no obvious relationship to the source. It set a standard for object program efficiency that has rarely been equalled. The FORTRAN I compiler, completed in 1957, established modern compiler tasks, structure, and techniques.

The compiler was developed for the 704, an IBM machine introduced in 1954 featuring built-in floating point and indexing capabilities. It compiled the FORTRAN I language, which was defined as part of the project and evolved considerably as the project progressed. In order to achieve its efficiency goals, the high-level arithmetic statements in the source program had to be translated to minimize storage references, and, even more important, subscripts and their control had to make maximal use of the machine's three index registers. The way in which this was achieved is described by Backus, and other project members;[2] formalized by Sheridan;[3] and reviewed by Backus and Heising[4] and Backus.[1] The latter paper, presented at the 1978 Conference on the History of Programming Languages, contains a penetrating analysis of the project, its origins and development. The purpose of this paper is to assess the technological impact of the FORTRAN I compiler on compiler construction, theory, and practice as it has evolved over the last 25 years.

COMPILER FUNCTIONS AND ORGANIZATION

The basic function of the FORTRAN I compiler was, of course, to translate the source program to an object program for loading and executing on the target machine. However, confronted with a belief that compilers could only turn out code intolerably less efficient than hand coding and confronted with a machine that would make small lapses in arrangement of coding show up as sizeable inefficiencies, the primary goal of the compiler, and indeed of the whole project, was to produce very efficient code.

The greatest potential source for inefficiencies was believed to be the address calculations rather than the code generated for the arithmetic expressions. Thus, while the translator part of the compiler was designed to produce excellent code for the arithmetic expressions, the design and organization of the entire compiler was driven by the need to produce nearly perfect code for array addressing on the three-register 704. Consider the FORTRAN program fragment in Figure 1.

```
      DIMENSION A (10,10)
      DIMENSION B (10,10)
      .....
      DO 1 J = 1,10
      DO 1 I = 1,10
    1 A(I,J) = B(I,J)
```

Figure 1—FORTRAN program to move array B to array A

Remembering that FORTRAN stores arrays column-wise, the expansion of the subscript on array B (as well as on A) is $(I-1)+(J-1)*10$. Clearly such a computation inside the DO loops was intolerable—and certainly not two such computations, one for A and one for B. It is interesting to note however that some current, nonoptimizing compilers do perform variants of this computation and are tolerated quite happily.

To achieve a modicum of efficiency, there was also a need to utilize the 704 index register instructions to increment, test and branch to control the execution of the DO loops. Furthermore, the registers had to perform dual functions when possible, controlling the looping and indexing the arrays in the loops. Assembly language programmers did this all the time, and if compiled code was to compete, the FORTRAN translator had to also. The primary criterion which dictated the design of the compiler was, therefore, the need to produce excellent addressing code. In fact it is still the case today that the biggest payoff for optimizing compilers for languages at the FORTRAN level (e.g., PL/I and Pascal) is in generating good addressing code.

The compiler was divided into six sections (phases in today's terminology):

1. A statement identifier and arithmetic statement translator
2. A subscript and DO statement analyzer
3. A transformer which interfaced sections 2 and 4
4. A control flow analyzer
5. A global register allocator
6. Final assembly

As John Backus makes clear,[1] this organization evolved as the problems associated with assigning index registers became clear. The initial intent was to have translation and code generation, including register allocation, complete by the end of Section 2 so that all that was left was final assembly; i.e.,

producing the binary code, load maps, etc. Section 1, the translator, was to classify statements, compile object instructions for the arithmetic formulas, and partially compile or record information about the remaining statements (the I/O, DO, GO TO, IF, DIMENSION, and function definition statements). Section 2 was to compile the instructions associated with subscripting and DOs. When it became clear that the task of Section 2 was too complex, Sections 4 and 5 were created and then Section 3 to glue everything together.

It is worthwhile looking in more detail at what went on here because it presents a model of an approach to solving very complex problems—the use of a divide and conquer strategy. Stated in today's terminology and from today's perspective (after 25 years of problem partition and solution), the problems being solved were the following:

1. (Section 2) Assuming an unlimited number of index registers, to create optimal code for addressing and loop control. In today's terms this meant: (a) reassociating the subscript expansions to collect constants and make them part of the base address and to group subexpressions to minimize the computation required in the loop; (b) finding common subexpressions; (c) moving computations out of loops; (d) performing strength reduction so that subscript calculations become index register increments and decrements; (e) folding constants; and (f) replacing loop tests by tests on registers required for addressing within the loop, i.e., linear function test replacement.
2. (Section 4) To perform the control flow analysis and identify (probabilistically) the relative frequency of program regions.
3. (Section 5) To assign real registers to the symbolic registers in order to minimize, using the control flow based frequency information, storage references and register–to–register moves.

How does the overall organization of the FORTRAN I optimizer (Sections 2 through 5) differ from today's optimizing compilers? Today we would probably do control flow analysis first and use it as a basis for performing Section 2's optimizations. Separating register assignment from the problem of optimizing code involving symbolic registers is now considered a good strategy,[5] though many optimizing compilers have not exposed loads, symbolic registers, and all of the addressing code to their optimizing sections and have ended up with most of the problems originally faced by the FORTRAN project when doing register allocation!

How does the overall organization of the rest of the FORTRAN I compiler compare with today's compilers? The translation phase is typically broken into several subparts today; syntactic analysis, semantic analysis, and code generation are common partitions, although the evolution here is by no means complete.

Overall, the organization of the compiler was surprisingly simple. Most of the complexities arose from the desire to produce object programs competitive with hand code and the consequent need to gather information and postpone producing code until the analysis necessary to produce efficient code had been performed.

We now turn to a closer examination of the significant sections of the compiler (Sections 1, 2, 4 and 5) in order to assess their technological impact in more detail.

TRANSLATION

Today's compilers often use elegant, language-independent translator systems. The theory behind these systems did not really start to develop until the 1960's, but the problem appeared in its full form in this system. Given an arithmetic expression, the translator first created a sequence of arithmetic instructions, then transformed this sequence to eliminate redundant computations arising from the existence of common subexpressions (their term) and to reduce the number of accesses to memory. These transformations have been the subject of numerous investigations (Aho[6] gives a good set of references), and we now know that an optimal solution is inherently hard. It is interesting to note, however, that the compiler designers felt that "the near-optimum treatment of arithmetic expressions is simply not as complex a task as a similar treatment of 'housekeeping operations'."[2]

In addition to parsing and producing good code for the arithmetic expressions, the translator identified the other statements and transformed complex I/O lists into their component DO nests for treatment by the regular mechanisms of the rest of the compiler. The attempt here and in numerous other parts of the compiler to seek common mechanisms rather than create special case mechanisms is interesting in light of the overall complexity of the task and the amount of invention required for every part.

SUBSCRIPT AND DO STATEMENT OPTIMIZATION

The translator did not complete the translation of DO statements and subscripts; that was the function of Section 2. A symbolic index register corresponding to each particular subscript combination of a variable was created by the translator and existed until Section 5 had assigned registers. The function of Section 2 was to optimize the calculation of subscripts and DO control statements. The constant parts of the calculation were incorporated into operand addresses, operations involving DO control variables were transformed into index register increments when possible, loop independent parts of the calculation were removed from the loop, and the loop exit test was transformed to use one of the registers needed for indexing. A nest of DO loops for array calculations was sometimes replaced by a single loop in the generated code! Some of these transformations are now subsumed in more general optimizations, but today's production compilers rarely do as well.

FLOW ANALYSIS

The function of Sections 4 and 5 of the compiler was to assign real registers to the symbolic registers. Except for the symbolic registers and the assumption that they could all be assigned to real registers, the program on entry to Section 4 was complete. The basic task, therefore, was to assign the sym-

bolic registers to real registers in order to minimize the time spent loading and storing index registers. Section 4 of the compiler did a flow anlaysis of the program to determine the pattern and frequency of flow for use in Section 5, where the actual assignment was made.

Basic blocks ("a basic block is a stretch of program which has a single entry point and a single exit point"[2]) were found and a table of immediate predecessor blocks constructed. Here, then, is the beginning of the elegant and fast control flow algorithms of today. Basic blocks and predecessor (successor) relationships are inputs to these algorithms.

The other task performed by Section 4 was the computation of a probable frequency of execution of every predecessor edge. To do this a Monte Carlo "execution" of the program with initial weights assigned to each edge was developed. This method is no longer commonly used to identify frequently executed areas of a program; rather the program topology is used more directly but with less resultant precision.

REGISTER ASSIGNMENT

Using the edge execution frequencies, regions were formed so that registers could be assigned to the most frequently executed areas (usually innermost loops), then to the next most frequently executed areas, etc. until the entire program had been treated. When a region had been processed, its entry and exit conditions were recorded, i.e., the values which needed to be loaded on entry and stored on exit. A processed region was not reexamined when its containing region was processed, but the entry and exit conditions and whether or not it had any unassigned registers were used. The assignment of registers within a basic block used the "distance to next use" criterion to determine which register to displace when out of registers. "Activity bits" were used to determine the necessity of storing a value in a register for subsequent use if the register had to be reused. In case of register assignment mismatches across basic blocks, an attempt was made to permute the assignment.

This register assignment method was a phenomenal piece of work. The displacement algorithm for straight-line code was later proved optimal[7] for the "one-cost model";[8] a displacement costs the same whether you need to store the register contents or not. Until 1980, when Greg Chaitin[9] successfully applied a graph coloring algorithm to the global assignment of registers, most global assignments were essentially variants on the FORTRAN I approach.

RESULTS

Perhaps the best way of demonstrating the results of this project is to show an example of its output—an output which startled this author. The FORTRAN program in Figure 1 moved array B to array A in a double nest of DO loops. The assembly program in Figure 2 is the FORTRAN I compiler's output for this program and shows the move being done with one loop instead of the two expected from the source. (FORTRAN stored its arrays column-wise and backwards; the 704 subtracted the value in the index register from the address.)

```
        LXD ONE,1          load 1 into reg1
LOOP    CLA B + 1,1
        STO A + 1,1
        TXI * + 1,1,1      add 1 to reg1 and
                           goto next inst
        TXL LOOP,1,100     if reg1 ≤ 100
                           goto loop
        ...
ONE       , ,1             data value one
A       BES 100            reserve 100 locs,
                           ending with A
B       BES 100            reserve 100 locs,
                           ending with B
```

Figure 2—FORTRAN I translation of array move

In general the code produced by the compiler was not only locally efficient but globally as well. The output program did not contain long, precoded, predictable sequences but contained code optimized to run efficiently in its context (where the context included the whole program). The project was a success.

Jean Sammet states, "Its major technical contribution was to demonstrate that efficient object code could be produced by a compiler; as a result, it became clear that productivity of programmers could be significantly improved."[10] Another major contribution of the project is the influence it has had on compiler structure and techniques. Overall "... FORTRAN has probably had more impact on the computer field than any other single software development"[10]—because of the language, the technological impact on subsequent compilers, and the impetus it gave to widespread use of higher-level languages.

REFERENCES

1. Backus, John. "The History of FORTRAN I, II, and III." ACM SIGPLAN History of Programming Languages Conference, *SIGPLAN Notices* 13, 8 (August 1978), pp. 165–180.
2. Backus, J. W., R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. "The FORTRAN Automatic Coding System." *Proceedings Western Joint Computer Conference,* Los Angeles, 1957, pp. 188–198.
3. Sheridan, Peter B. "The Arithmetic Translator-Compiler of the IBM FORTRAN Automatic Coding System." *CACM* 2,2 (Feb. 1959), pp. 9–21.
4. Backus, J. W., and W. P. Heising. "FORTRAN." *IEEE Transactions on Electronic Computers,* EC-13,4 (August 1964), pp. 382–385.
5. Auslander, M. A., and M. E. Hopkins. "An Overview of the PL.8 Compiler." *Proceedings of the SIGPLAN Symposium on Compiler Construction,* June 1982 (to appear).
6. Aho, Alfred V. "Translator Writing Systems: Where Do They Stand?" *Computer,* 13,8 (August 1980), pp. 9–14.
7. Belady, L. A. "A Study of Replacement Algorithms for a Virtual-Storage Computer." *IBM Systems Journal,* 5,2 (1966), pp. 78–102.
8. Horwitz, L. P., R. M. Karp, R. E. Miller, and S. Winograd. "Index Register Allocation." *JACM* 13,1 (Jan. 1966), pp. 43–61.
9. Chaitin, Gregory J., Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. "Register Allocation via Coloring." *Computer Languages* 6 (1981), pp. 47–57.
10. Sammet, Jean E. "History of IBM's Technical Contributions to High Level Programming Languages." *IBM Journal of Research and Development,* 25,5 (Sept. 1981), pp. 520–534.