1980

Experience with Transporting Pascal to an Interactive Environment

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

John Garney

Report Number: 80-340
Experience with Transporting Pascal to
an Interactive Environment

Douglas Comer
John Garney

Computer Science Department
Purdue University

April 1980

ABSTRACT

The PCODE implementation of Pascal can be transported to new environments easily, without significant modification to the compiler itself. This paper reports on a transportation effort that used a standard Pascal compiler on a CDC 6000 system to aid in transporting the PCODE compiler to a Digital Equipment VAX system. It also presents measurements of the CPU cost introduced by the lazy evaluation scheme for interactive I/O.
1. Introduction:
Transporting an existing compiler from its source environment to a new target environment is one of the fastest and easiest ways to implement a language processor. Recently, compilers have been designed to minimize the transportation effort [1, 2, 3, 4 et al.]. These portable compilers employ various techniques to insure their machine independence. Of particular interest here is the portable Pascal compiler known as P4 [1] which generates code for a hypothetical, high level, stack architecture computer.

Output from P4 is called PCODE, and the compiler itself is often referred to as the P-compiler [1]. Although PCODE is a low level language, care has been taken to include only the most common kinds of instructions. Thus, it can easily be mapped into the assembly language of most existing machines.

In theory, moving the P4 compiler to a new environment is a trivial task -- one need only write a PCODE to assembler language back-end translator. Because it is written in Pascal, the compiler, running in the source environment, can be used to compile itself into PCODE. The PCODE version is then transported to the target environment, and translated to assembler language using the newly constructed back-end. Assembling the output of the back-end creates a version of P4 running in the new environment, and completes the transportation process.

In practice, the transportation process is not so trivial (as we will show). In particular, minor differences in the source and target environments can lead to difficult transportation problems. The next section describes the pertinent parts of the transportation process, giving the reader insight into the difficulties and technical problems. Later sections discuss the problem of modifying Pascal's standard I/O facilities for use in an interactive environment, and present measurements of the computational overhead for the lazy evaluation scheme.

2. History of the Project:
In November, 1978, the Computer Science Department of Purdue University acquired a Digital Equipment Corporation VAX 11/780 computer system to support research. At the time of delivery only one operating system was available: Digital's VMS. D. Comer decided to implement Pascal on the VAX and started two independent projects, one to acquire the NBS Pascal compiler [6] which runs in PDP 11 compatibility mode, the other to transport the P4 compiler to the VAX. While the NBS compiler ran almost immediately after it arrived, its small address space and language restrictions limited its applicability, and diverted attention to the P4 compiler.

In addition to the VAX system, the Purdue University Computing Center's Control Data Corporation 6000 series equipment was used in the P4 project. The CDC system included a Pascal compiler as well as a version of P4 which ran on the CDC machines but cross compiled code for a PDP 11/70. At first, the PDP 11 cross compiler seemed like an ideal candidate for transportation to the VAX. Careful examination showed, however, that differences between the two machines was so great that little use could be made of the PDP 11 version, except as a model. For one thing, the compiler had been modified to generate threaded code, making it a hybrid compiler-interpreter. For another, the runtime environment of the machines differed considerably (eg., VAX integers and pointers were twice as long as on the PDP 11). It was decided to write a back-

---

*Compatibility mode is a hardware mode that allows a VAX to emulate a PDP 11.
end translator and run-time system from scratch.

S. Groff worked quarter-time from January 1979 through May 1979 building a PCODE to assembler language back-end and a version of the necessary run-time routines to support P4 under the VMS operating system. The back-end translator was coded in Pascal and debugged on the Computing Center's Pascal compiler. The run-time routines were coded in assembler language on the VAX, and were crafted explicitly for the VMS environment. In particular, the input and output routines set up several linked control blocks for each file in order to use the standard VMS I/O interface.

By April, the Department had acquired a version of the UNIX time shared operating system [7]; the project was retargeted for UNIX at the beginning of May, 1979, just as the back-end and run-time systems were being completed. J. Garney took over for Groff and worked half-time to modify the run-time system for UNIX. At the same time, a 9600 baud asynchronous link was established between the machines making it possible to move files from the CDC system to the VAX system directly. Although the maximum transfer rate was close to 1000 characters per second, the systems usually achieved a rate of much less than 400 characters per second. The machine link made it possible to cross compile test routines using a version of P4 running on the CDC equipment, and transfer them to the VAX to test the run-time system before the compiler itself was available on the VAX.

By July, the run-time system was deemed sufficient to support the compiler itself, and the bootstrap process began. Two versions of the P4 compiler were derived from the original source code: one to run on the CDC system and the other to run on the VAX. Similarly, a modified version of the back-end was created which would run on the VAX. To carry out the transportation process, one had to:

1. Compile the version of P4 which would run on the CDC system using the standard CDC Pascal compiler.
2. Compile the back-end translator using the standard CDC Pascal compiler.
3. Compile the version of P4 which would run on the VAX into PCODE using the compiler generated in Step 1 above.
4. Translate the PCODE output from Step 3 into VAX assembler language using the back-end compiled in Step 2 above. The result is a Pascal to PCODE compiler in VAX assembler language.
5. Compile the version of the back-end designed to run on the VAX into PCODE using the compiler generated in Step 1 above.
6. Translate the PCODE output from Step 5 into VAX assembler language using the back-end compiled in Step 2 above. The result is a PCODE to VAX assembler translator in VAX assembler language.
7. Transfer the assembler language files from Steps 4 and 6 to the VAX using the machine to machine link.
8. Assemble the compiler and back-end on the VAX using the standard assembler to produce a Pascal compiler (actually two separate parts -- a Pascal to PCODE compiler and a PCODE to VAX Assembler language translator).
9. Transfer the source files for the P4 compiler and back-end translator to the VAX using the machine to machine link.
10. Use the VAX compiler generated in Step 8 to compile the source files obtained via Step 9 to obtain a truly self-compiling compiler.

The process can be summarized using T-diagrams similar to those of McKeeman et al. [8]. Each "T" represents one translator; the arms are labeled with its input and output; the base is labeled with the machine on which it runs as shown in Figure 1. Figure 2 combines several T-diagrams to show the bootstrap from one machine to the next, and Figure 3 shows how the resulting compiler can be used to compile itself.

A preliminary version of the compiler was running by the end of July 1979. It had many undesirable restrictions, however, and could not compile itself on the VAX. Most notably, it would only accept upper case input, it supported 3 input and 5 output files with fixed names, did not allow external procedures, had a small user stack and heap, and wrote the assembler language file in upper case. Since the UNIX assembler required lower case input, an extra process was introduced to translate the compiler output to lower case. The restriction to fixed file names presented worse problems; it meant that the user could only perform one Pascal compilation or run one Pascal program at a time.

The simple problem of character set incompatibilities between the source and target environments caused many of the difficulties. Although two versions of the compiler were maintained (one to run on the CDC system and one to run on the VAX) their source code originated on the CDC system which did not support lower case characters. Thus, it was nearly impossible to rewrite the lexical routines to accept lower case until the compiler had been bootstrapped onto the VAX. On the other hand, restrictions and bugs in the compiler made it impossible to move onto the VAX until a reliable, expanded compiler had been developed. Even using a machine link, changing the compiler, recompiling and translating it, transferring it across the link, and assembling the resulting files required nearly 12 hours. Changes were tedious, time consuming, and frustrating. For instance, inserting a single debugging statement and recompiling it meant waiting a day to see a test run. To further complicate matters, the project had reached the limits of the standard CDC Pascal compiler. Attempts to enhance the compiler sometimes caused the bootstrap process to abort because the new source exceeded a size or nesting limit. It took August and September to expand and stabilize a version on the VAX which could compile itself.

Once the transportation process was complete, improvements to the compiler followed in rapid succession. Using the existing version, an enhanced version could be compiled and installed in a matter of minutes. The run-time system was rewritten in C, a higher level language, and was tailored to UNIX. The UNIX facilities for diverting input or output to a file, and pipelining of processes became available as a side effect of using the standard UNIX I/O interface. Using C also increased the readability and made subsequent improvements easier.

By November, 11 months after the project started, the compiler running on the VAX could handle almost all of Pascal (with the exception of a goto out of a procedure), and was in use by several researchers and students. The implementation had been carried out by two students, working half-time, with guidance and occasional assistance from a faculty member.

Essentially complete, the compiler still lacked the ability to compile programs that used a terminal interactively. In some systems the lack of interactive I/O would cause no problem. UNIX relies so heavily on interaction, however, that compilers which do not support it are not used. Unfortunately, Pascal was designed for a batch environment; the addition of interactive capabilities requires a close examination of the language and the particular interactive
3. The Implementation of Interactive I/O in Pascal:

Various strategies for incorporating interactive I/O in Pascal have been proposed [9,10,11,12,13,14]. To illustrate the differences, we will use the following Pascal program which reads a real number after issuing a prompt:

```pascal
program echo(input,output);
var x: real;
begin
    writeln(' please enter a number');
    read(x);
    writeln(' you typed in ',x)
end.
```

Strait et al. [9] suggest a scheme for implementing interaction where:

1. The user is required to indicate which files correspond to interactive terminals (to do so, the user places a delimiter ("/"/") in the program heading immediately following each file that corresponds to an interactive terminal).
2. Every call of the procedure "readln" produces a prompt to the terminal, and
3. The eoln function is initially true.

The sample program will not work under the Easton method unless it is modified:

```pascal
program echo2(input/,output);
var x: real;
begin
    writeln(' please enter a number');
    readln; (* to prompt for input *)
    read(x);
    writeln(' you typed in ',x)
end.
```

The use of readln to signal interactive prompting is misleading to beginners. Furthermore, the program can only run interactively, a point to remember.

Clark [10,11] notes that with respect to interactive I/O "any differences from standard Pascal should be kept to a minimum." He proceeds to define a new function, sor (start of record), and then redefines the built-in functions reset and readln. The original form of our sample program will work under the Clark proposal, but programs that use other built-in functions must be changed to run using Clark's definitions.

Bran and Dijkstra [12] propose that the definition of textfiles be changed to use the notion of line separators instead of line terminators. Before reading from an interactive terminal, the user must issue a call to the readln procedure to "skip over the line separator." Again, the proposed implementation makes a change in the semantics of Pascal. In addition, it means that there will be differences in the way one processes interactive and noninteractive I/O.

To understand our goals for an interactive Pascal, one must understand the environment in which the programs were intended to run. Under UNIX, the input or output from a process may go to a conventional file on secondary storage, a device (like the line printer), another process, or an interactive terminal. The destination of I/O is not known at compile time, however; input or output may be redirected when the program is invoked, or when a file is opened.
by name (since some file names correspond to devices). We believe the uniform treatment of files, devices, pipes (process to process links), and interactive terminals to be among the greatest strengths of UNIX. In order to exploit this strength:

1. Distinctions between interactive and noninteractive I/O cannot be made at compile time.
2. Interactive and noninteractive files should be processed identically at run time.
3. The user should issue all prompts for input, and should be able to read from an interactive terminal without prompting if desired.
4. Standard Pascal programs should run without any changes.

The lazy evaluation scheme for interactive I/O noted by Saxe and Hisgan [14] (and others) fits the above criteria. It can best be described as an implementation change (i.e., the definitions of standard functions remain unchanged and no program changes are required). A lazy evaluation of input merely defers actual input until the results are needed. In Pascal, the crucial difference between normal input and lazy input lies in the processing of \texttt{eoln}, \texttt{eof} and \texttt{file\textdagger}. Normally, input is thought of as a stream, and \texttt{file\textdagger} is thought of as a window through which the program may view the "next" object without having to read it. For example, under the conventional implementation of a text file the run-time system prefetches blocks of text into a buffer. As the program reads characters they are retrieved from the buffer. When the buffer empties, the system reads the next buffer load in a single I/O operation. Reading data into a buffer before it is needed improves performance by overlapping I/O and computation. Under the buffered implementation, the value of \texttt{file\textdagger} can be determined by looking ahead in the buffer. So can the appropriate value for \texttt{eof} or \texttt{eoln}.

In standard Pascal, the input file is reset at the beginning of the program automatically, and \texttt{input\textdagger} is assigned a value. Resetting input in an interactive environment means reading from a terminal before the user could issue a prompt. The lazy evaluator simply marks \texttt{input\textdagger} as "unevaluated" and proceeds with the program. The first reference to \texttt{input\textdagger}, \texttt{eof}, or \texttt{eoln} causes the system to read a value and change the status to "evaluated". Subsequent references to \texttt{input\textdagger} find the "evaluated" status and return the old value without reading another. Finally, the procedures \texttt{get}, \texttt{read}, or \texttt{readln} perform the input as usual and leave the file pointer marked "unevaluated" again.

Lazy evaluation does not require a change to the programming language or to existing programs. It can be implemented easily, and does not require the programmer to designate which files are interactive. Therefore, it seems like an ideal solution. The functions \texttt{eof} and \texttt{eoln} do have a side-effect, but it is (almost) transparent to the program. The lazy scheme does have another disadvantage: a program must check to see whether the file pointer has been left evaluated or unevaluated each time an I/O operation is performed. This raises the question, "how much will the additional checking cost?".

4. Measurements of the Lazy I/O implementation:

Several experiments were run to measure the computational overhead of the lazy implementation for interactive I/O. Since the additional CPU time introduced by the lazy scheme is negligible compared to the time required to read input from an interactive terminal, the experiments concentrated on measuring I/O to a file.

Five programs were constructed and run on files of length 1, 4, 8, 30, 50, 100, 300, and 800 thousand characters. The programs tested the cost of
procedures *get*, *put*, *read*, *readln*, *write*, and *writeln*, functions *eoln* and *eof*, as well as references to *input*†. The experiments were conducted on a Digital Equipment Corporation VAX 11/780 computer system (configured with 3 mbytes of memory and 3 RM03 disk drives). Each program was run 3 times using the noninteractive run-time system, and 3 times using the interactive run-time system; the average CPU time for 3 runs was used in all calculations. The VAX was otherwise idle when the experiments were performed (a duplicate set of experiments performed during regular production showed no significant difference in times).

Figure 3 displays the CPU time for three sample programs as a function of input file size. The times marked with a square represent the costs for interactive and noninteractive systems processing a program with 15 references to input character read. The plots marked with a triangle show times when using the procedure *get*, and plots marked with a circle give times when using procedure *read*. In general, our implementation of the lazy evaluation strategy introduces 5% to 15% computational overhead in the I/O costs, depending on which primitives are used, and the frequency of use.

Figure 4 presents another view of the data shown in Figure 3. From the number of I/O references in each program, and the size of the input file, an estimate was obtained of the CPU time per I/O reference. The time per reference was plotted for both the interactive and noninteractive versions of each program to show the relative difference. From the Figure, it can be seen that references to *input*† are the most costly part of the interactive run-time system (about 56% more than the noninteractive system). In the noninteractive version of the system, a reference to *input*† translates into two machine instructions which reference the file buffer directly. Our interactive implementation, which has not been optimised, generates a procedure call to check the evaluation status of *input*†, and introduces another instruction. Actually, the frequency of references to *input*† is normally small, so the most programs do not notice the additional overhead. For example, programs which use the procedure *read*, experience an overhead closer to 5% as illustrated by the third experiment.

5. Conclusions:

Although the P4 compiler is designed for portability, transporting it to an interactive environment can be costly. Minor issues, like the use of explicit character constants, make changing environments difficult. Finally, neither Pascal nor P4 is especially well suited to an interactive environment.

The lazy evaluation scheme for interactive I/O satisfies all of our objectives. It is transparent to users, allows identical treatment of interactive and noninteractive I/O, and does not require changes to the Pascal language or existing programs. We feel that 5% - 15% computational overhead for the lazy implementation is justifiable considering its advantages. The higher overhead for references to the file buffer variable make it less attractive for our users. Experience has shown, however, that programs tend to use only one input file and one output file interactively. In order to reduce the computational costs for noninteractive files, we have adopted the convention that only the files corresponding to the UNIX standard input, standard output, and standard error files have lazy evaluation of I/O. This reduces the CPU requirements for noninteractive programs, yet permits redirection of input and output in a normal manner. This study has shown us that we need to improve the implementation by reducing the cost of references to the file buffer variable.
References

5. R. Berry, "Experience with the Pascal P-Compiler," Software -- Practice and Experience 6 (1976), 617-627.
Figure 1. A T-diagram which corresponds to a translator. The left arm is labeled with the input language, the right arm is labeled with the output language, and the base is labeled with the machine on which the translator runs.

Figure 2. A T-diagram showing the use of a standard Pascal compiler to bootstrap a version of the Pascal P4 compiler (Numbers refer to the bootstrap steps given in the text). Both the compiler and back-end translator are compiled into VAX assembler language before being moved to the VAX.

Figure 3. A T-diagram showing how the compiler and back-end are used in the target environment to produce a self compiling compiler. Compiling the source to P4 reproduces the first part of the compiler (or the next version, if it has been changed).

Figure 4. A plot of the CPU time vs. input file size for noninteractive and interactive run-time systems for three sample programs (indicated by circle, square, and triangle). Two lines are plotted for each program. In each case, the plot showing higher CPU costs corresponds to the interactive run-time system.

Figure 5. A plot of the CPU time vs. number of I/O references for the sample programs. Using the lazy evaluation method increases the cost of references to the file buffer variable is increased by nearly 56%. Other costs are increased by 5% - 15%.
Input Language

Machine on which Translator runs

Output Language