Reliability and Testing in the POGO Compiler Development Environment

Jack W. Davidson

Computer Science Technical Report TR-87-08
May 18, 1987

Reliability and Testing in the POGO Compiler Development Environment

Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

ABSTRACT

Developing, testing, and maintaining a compiler or interpretor for a modern programming language is a major software development task. A software development environment specifically tailored to assist in building reliable compilers and interpretors can make the job much easier. This paper describes such an environment. The environment, called POGO, provides an integrated set of tools for managing, developing, and testing all phases of translators. Unique aspects of the environment are the strategies, tools, and facilities provided for testing the translator. These facilities allow a compiler under development to be thoroughly tested before it is moved to the host machine. After development is complete, POGO provides facilities for efficiently testing the compiler as it evolves.

April 16, 1986

Department of Computer Science

The University of Virginia

Charlottesville, VA 22901

Reliability and Testing in the POGO Compiler

Development Environment

1. Introduction

Developing a compiler or interpretor for a modern programming language is a major software development task. Most translator development efforts concentrate on building translators that are portable, or easily retargeted. There are several reasons for this. From the language designer's point of view, the language should be widely available on a number of different machines to gain exposure and acceptance. From the compiler writer's point of view, a portable compiler allows the total development cost to be amortized across several machines. Further, it is often the case that the compiler will be used in a software development environment for embedded systems. In this case, the compiler runs on one machine (the host) and generates code for a variety of machines (the targets). Such a compiler is often referred to as a *cross-compiler*.

A compiler that is being ported to other machines or being used as a cross-compiler in a development environment is difficult to test and maintain. Consider the case where a compiler is to be ported from a host machine to a target machine. This process, called a half-bootstrap, requires that most initial work be done on the host machine. Assuming that the compiler to be ported is implemented in the language it compiles (L), the first step is to produce a compiler that runs on the host machine, H, and produces code for the target machine, T. Using the abbreviated T-diagram [Brat61, Earl70] notation of Aho, Ullman, and Sethi [Aho86], this compiler is specified as $L_L T$. This compiler is compiled using the compiler $L_H H$ to produce the compiler $L_T T$ which is then moved to the target machine.

If everything went according to plan, we would now have a compiler for language L that runs on machine T and produces code for machine T. Unfortunately, it is rare that

one does things right on the first try. Discovering bugs in the compiler L_TT often requires that fixes be installed on the host machine, and the generation phase be repeated. If the target machine is readily accessible and it is easy to move code from the host to the target, this iterative process may not be a problem. It is, however, often the case that the target machine may not be readily accessible and that moving code from the host machine to the target may take days. An extreme situation occurs when software development is occurring in parallel with hardware development. In this case, the target machine may be an engineering prototype not readily available for software development. In the worst case, the machine may not exist yet! The compiler development environment described here solves this problem by allowing the compiler L_TT to be thoroughly tested on the host machine.

Even if the target machine is readily accessible, the environment provided by the target machine may lack good tools to aid debugging and development of the compiler. This is often the case when a compiler is being developed for a microprocessor as part of an embedded system. There is usually no symbolic debugger or tools for easily testing code. Even if reasonable tools are available, they may be unfamiliar and require a substantial investment of time to learn how to use them. Again, an integrated environment that provides tools for developing a compiler as well as testing and debugging the compiler even when it is targeted for a different machine can be very useful.

A cross-compiler also makes testing and maintenance more difficult. As an example, consider a cross-compiler for a language L that has been retargeted for eight different machines. In effect, there are eight different versions of the compiler. If the compiler is modified, the potential exists that the eight different compilers require testing to ensure that the modification did not "break" one of the compilers. This testing can be time-consuming and expensive.

This paper describes POGO, a compiler development environment that provides solutions to the above problems. The unique aspect of the environment are the strategies and tools

used to build highly reliable translators. These tools allow a compiler for a target machine to be thoroughly tested before it is moved.

2. Compiler Development Tools

POGO is based on UNIX† and PO [Davi84], a retargetable optimizer. Much of POGO's utility in building compilers comes from the use of the quality program development tools available on UNIX such as Yacc, Lex, Make, and Diff. PO provides the ability to rapidly produce back ends that generate efficient code. Figure 1 shows a schematic of a retargetable compiler or cross-compiler that uses PO.

2.1 Building In Reliability

A large portion of the reliability of the compilers developed under POGO is due to the strategies employed when using PO. The following sections briefly discuss the intermediate representations, the various phases of the compiler, and the strategies used to make building compilers simple and easy to do correctly.

2.1.1 Front Ends

The first phase of the compiler, the front end, translates the source language to a simple intermediate language ('IL'). Of course the program generators Lex [Lesk79] and Yacc [John78] are available to help build the lexical analyzer and the parser. In contrast to the front ends of many compilers, front ends developed for use with PO are especially simple to build. Front ends for use with PO emit naive, but semantically correct code. Efficiency is *not* an issue. We rely on the optimizer to transform the naive code to efficient code. This allows the implementor to concentrate on the most important task—generating correct code.

†UNIX is a trademark of Bell Laboratories.

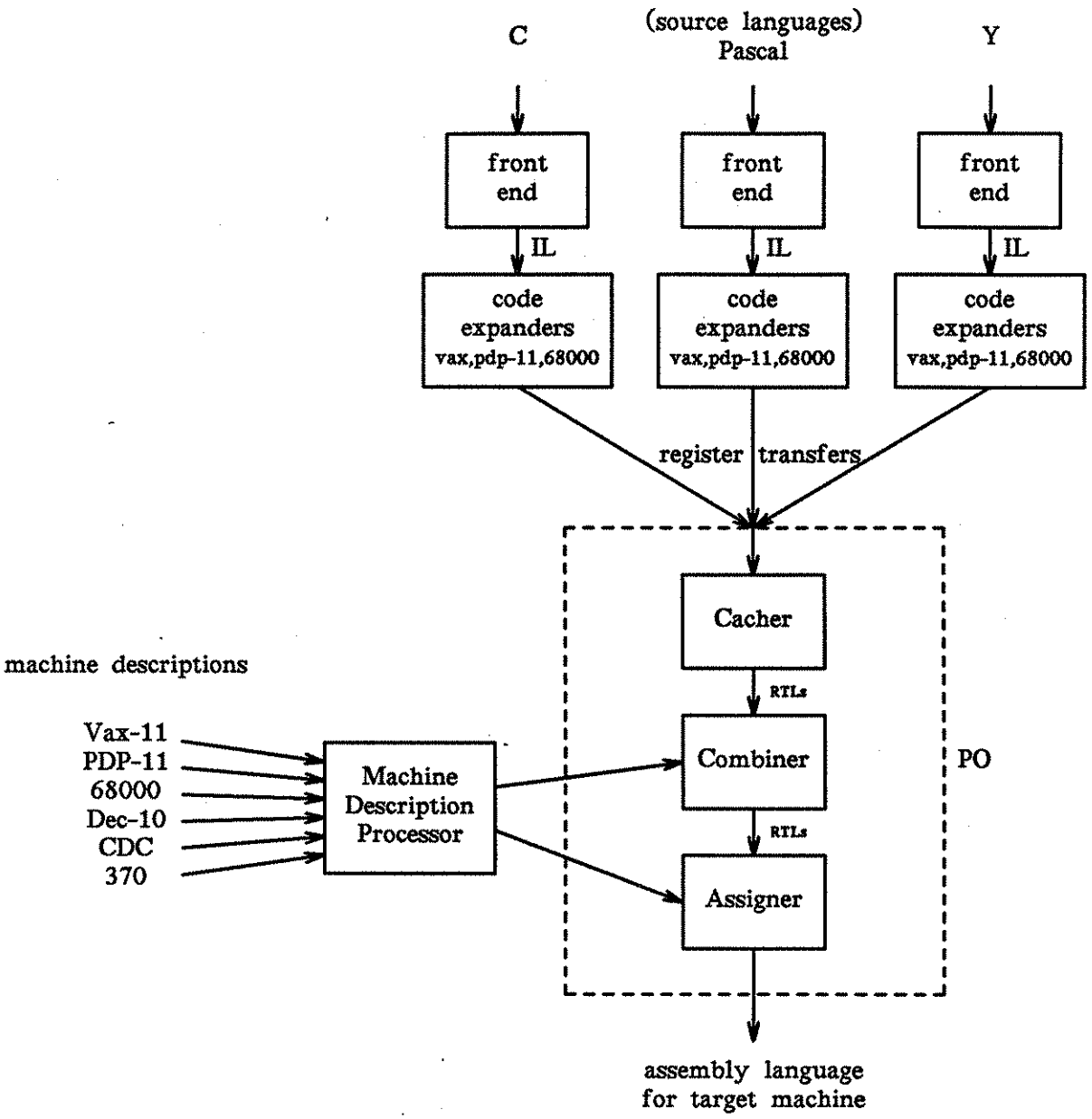

Figure 1. Schematic of the PO Compiler System

2.1.2 Intermediate Languages

The strategy or philosophy of ignoring efficiency and concentrating on correctness also affects the design of the IL. Because efficiency of the code generated by the front end is not an issue, the ILs can be made very simple. For example, the IL language that our C compiler emits contains only 66 opcodes. For most of the compilers developed using PO,

we have chosen to emit code for an abstract stack machine. There are several reasons for this decision. Translating expressions to stack code is a well-understood process, and allows simple code generation algorithms to be used in the front end. In addition, it is relatively straightforward to take stack code and convert it to code for a register machine. Indeed, the code expanders of the next section routinely do this translation. On the other hand, mapping an intermediate language that is tailored to register machines onto a stack machine, while possible, is not as easy to do. A final reason for favoring the use of abstract stack machine code is that it permits the construction of an interpretor for the language if one is desired. The ability to develop an interpretor can be useful if the language needs to be moved to a new machine quickly.

2.1.3 Code Expanders

The second phase, called a code expander, translates the IL to instruction sequences for the target machine. Much like the front end, the code expander is only concerned with correctly translating the IL opcodes to semantically equivalent target machine sequences. Again, efficiency is not a concern. This along with the small size of the IL make writing a code expander quite simple. For most IL opcodes and most target machines, code expansion is a simple one-to-one mapping of an IL opcode to a target machine instruction.

2.1.4 Register Transfer Lists (RTLs)

While a code expander can conceptually be thought of as emitting assembly language for a target machine, it actually outputs register transfer lists ('RTLs') which describe an instruction's effect. The RTL notation is based on ISP [Bell71]. A RTL is simply a machine-independent representation of a machine-dependent operation. For example, a PDP-11 instruction that adds two registers would be expressed in the RTL notation as:

```
r[2] = r[2] + r[3]; NZ = r[2] + r[3] ? 0;
```

The RTL notation acts as a universal assembly language.

Emitting RTLs instead of assembly language has several important benefits. First,

because they describe the effects of instructions they allow programs to be written that can reason about the code. For example, the optimizer, PO, takes sequences of RTLs and determines whether they can be replaced with faster/smaller sequences that are semantically equivalent. Using RTLs also allows fast simulators for instruction sets to be built automatically [Davi85]. These simulators form an important component of the POGO environment.

Finally, the machine-independent nature of RTLs and the similarity of machine operations allows code written for one machine to be easily adapted for another. Consider the previous representation of the register add instruction for the PDP-11. For most register machines, the register add instruction would be expressed similarly. Thus code fragments from the code expander for one machine can often be "borrowed" to help build the code expander for a new machine. Our experience with the environment is that a code expander rarely need be written from scratch.

2.1.5 Back Ends

The job of a conventional back end is largely subsumed by PO. PO's main function is to take the naive (but correct) code produced by the earlier phases and emit more efficient code. PO is retargeted by providing a description of the target machine's instruction set. The Combiner phase of PO uses an automaton automatically constructed from a machine description ('MDs') to determine if an RTL represents a legal instruction on the target machine. The Assigner phase of PO uses a similar automaton to convert RTLs to assembly language for the target machine. For more details on the operation of PO see Davi81, Davi84.

PO has been used by the author to develop retargetable compilers for Y [Hans81], C [Kern78], and Pascal [Jens75]. One major computer manufacturer has successfully used PO in a large in-house compiler project. Because of its continued use and refinement over a period of years, PO has evolved into a stable, reliable tool. We note that the strategy of having the front end generate naive but correct machine code and relying on a thorough

optimizer that operates on a low-level machine-dependent representation is the strategy that has been used successfully to develop the PL.8 family of compilers [Ausl82, Radi82].

3. Testing

Testing and maintaining a family of retargetable compilers creates several logistical problems. Testing a compiler that is being retargeted to a new machine can be difficult if the target machine is not readily accessible or has a poor support environment. For a compiler that has already been tested and retargeted to a number of machines, it is unreasonable to expect that all compilers will be fully retested if the compiler is later modified. The time and resources required would be substantial. The following sections describe the tools and facilities provided by POGO to facilitate testing and maintenance of a family of compilers.

3.1 Instruction Set Interpretor

One of the major tools provided by POGO to facilitate testing of a compiler is a fast instruction set interpretor [Davi85]. The interpretor, which is automatically constructed from the same MD used by PO, allows code generated for another machine to be executed on the host machine. A schematic of the system is shown in Figure 2.

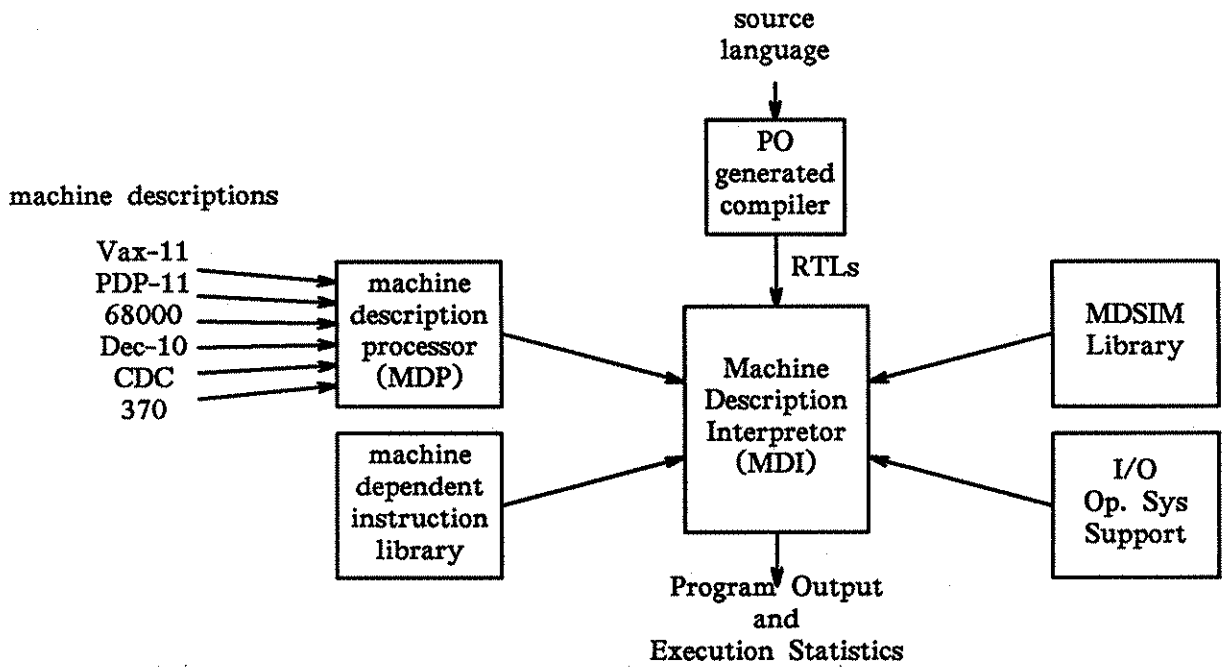

Figure 2. Schematic of the Instruction Set Interpretor

It is the ability to simulate an arbitrary machine's instruction set on the host machine that is fundamental to our testing strategies. This simulator is also the basis for an environment for designing and evaluating instruction sets. For details on the implementation of the interpretor and some of its other uses see Davi85.

3.2 Machine Description Testing

Central to the correct operation of PO, is the accurate specification of a machine's instruction set. Obviously, we would like to convince ourselves that a MD accurately describes the instruction set and assembly language of a machine. There are two strategies used to test MDs—static and dynamic.

The static testing strategy is to make sure that PO, during its operation on a suite of test programs, has correctly processed every addressing mode and instruction contained in the machine description at least once. We do this by constructing the automaton used by PO so that it counts how many times each addressing mode and instruction is used. The idea is that if PO has not used every addressing mode and instruction it is much more likely that a bug in the MD could go undetected. If during testing we discover that all addressing modes and operations have not been used, further tests cases should be devised.

The dynamic testing strategy is to make sure that all addressing modes and instructions in the MD have been executed at least once during testing. This is accomplished by directing the machine description interpretor ('MDI') to record counts of how many times each addressing mode and each instruction is executed. Again, the idea is that if each addressing mode and instruction has been executed at least once, we are less likely to be surprised later by an incorrectly described instruction or addressing mode.

Exhaustively testing a MD could be a time-consuming process. Fortunately, because MDs are factored into a section that describes the addressing modes, and a section that describes the instructions, every combination of addressing mode and instruction need not be tested.

This makes developing test suites for a MD feasible. Further, a MD need only be tested once even if it is being used to support compilers for several languages.

3.3 Building Test Suites

In order to test a compiler thoroughly, a comprehensive suite of test programs is required. Again the MDI can help the compiler writer develop a such a set of test programs. This is accomplished by producing a compiler that is executable under the MDI. As test programs are compiled using the simulated compiler, the MDI gathers data about which portions of the compiler have and have not been executed. Based on this information, additional programs can be devised and added to the suite of test programs.

3.3.1 Target Machine Testing

As discussed in the introduction, testing a cross-compiler can be difficult if access to the target machine is problematic, or the target machine environment is poor. Again, the MDI provides a nice solution to this problem. Code generated for the target machine can be executed on the host. In addition, because the MDI's user interface is similar to a conventional debugger (breakpoints can be set, memory locations can be examined/changed, etc.), debugging of the code can also be performed on the host environment. Testing and debugging on the familiar host machine is clearly preferable to testing and debugging on the target machine.

3.3.2 Regression Testing

After development is complete, a compiler writer is faced with the task of maintaining the compiler. A standard technique for insuring that bugs are not introduced into a compiler is the regression test. To do regression testing, code produced by a validated compiler is saved. The saved code may consist of code produced when the programs in the test suite were compiled. If the compiler is self-compiling, the code produced when the compiler is passed through itself is also saved. This technique is well known and widely used [Aho86, Ausl82, Corn84]. Regression testing a cross-compiler that has been retargeted to

several machines could be time-consuming and expensive if each compiler was individually tested each time a change was made.

The POGO environment provides some relief from this problem. After development is complete, a fixed-point of the output of each phase of the compiler is saved for all programs in the test suite. For example, the output from the front end (the IL), the output from the code expander (RTLs), and the output from each phase of PO is saved. When a modification to the compiler is made, often only a few phases of the compiler require testing before we are convinced that no bugs have been introduced. For example, consider the case where a change has been made to the Cacher phase of PO. Testing begins by generating the output of the new Cacher. If there are no differences, or only trivial ones that are clearly correct, testing need not proceed further and the new fixed-point of Cacher can be saved. In our experience, this is the most common occurrence. If there are substantive differences, subsequent phases of the compiler must be tested until a steady-state is reached. If the compiler is being used as part of the test suite, it is possible that the compiler will have to be compiled three times before a steady-state is reached [Corn84]. POGO provides scripts that perform the testing automatically.

4. Discussion

This paper has described how the POGO environment promotes the development of reliable compilers. There are two fundamental strategies employed. First, build in reliability from the start by using systematic and simple construction techniques. Second, use thorough testing strategies to catch bugs early and keep bugs from being introduced during maintenance phases. A unique feature of the environment is the use of a machine simulator to help validate the MDs used by the optimizer, to develop test suites, and to test cross-compilers using the host environment.

5. References

[Aho86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[Ausl82] M. Auslander and M. Hopkins, An Overview of the PL.8 Compiler, *Proceedings of the ACM SIGPLAN Notices '82 Symposium on Compiler Construction*, Boston, MA, June 1982, 22-31.

[Bell71] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[Brat61] H. Bratman, An Alternative Form of the UNCOL Diagram, *Communications of the ACM 4*, 3 (March 1961), 142.

[Corn84] B. J. Cornelius, I. R. Lowman and D. J. Robson, Steady-State Compilers, *Software - Practice and Experience 14*, 8 (August 1984), 705-709.

[Davi81] J. W. Davidson, *Simplifying Code Generation Through Peephole Optimization*, PhD Dissertation, University of Arizona, December 1981.

[Davi84] J. W. Davidson and C. W. Fraser, Code Selection through Object Code Optimization, *Transactions on Programming Languages and Systems 6*, 4 (October 1984), 7-32.

[Davi85] J. W. Davidson, Fast Interpretation of Instruction Sets: Implementation and Applications, in *Computer Hardware Description Languages and their Applications*, C. J. Koomen and T. Moto-oka (ed.), North-Holland, Amsterdam, August 1985, 179-191.

[Earl70] J. Earley and H. Sturgis, A Formalism for Translator Interactions, *Communications of the ACM 13*, 10 (October 1970), 607-617.

[Hans81] D. R. Hanson, The Y Programming Language, *SIGPLAN Notices 16*, 2 (February 1981), 59-68.

[Jens75] K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, New York, NY, 1975.

[John78] S. C. Johnson, Yacc: Yet Another Compiler-Compiler, *Unix Programmer's Manual 2B*, Section 19 (July 1978), 1-34.

[Kern78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Lesk79] M. E. Lesk, Lex - A Lexical Analyzer Generator, *Unix Programmer's Manual 2B*, Section 20 (January 1979), 1-13.

[Radi82] G. Radin, The 801 Minicomputer, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 39-47.