Debugging a High Level Language via a Unified Interpreter and Compiler Runtime Environment

Jinlong Cai, Marc Moreno Maza, Stephen Watt, Martin Dunstan

Abstract

Aldor is a programming language that provides higher-order facilities for symbolic mathematical computation. Aldor has an optimizing compiler and an interpreter. The interpreter is slow, but provides a useful debugging environment. Compiled Aldor code is efficient, but cannot be debugged using user-level concepts. By unifying the run-time environments of the Aldor interpreter and compiled Aldor executables, we have realized a debugger for Aldor. This integration of the various existing functionalities in its debugger improves the development environment of Aldor in a significant manner, and provides the first such environment for symbolic mathematical computation. We propose that this approach can be useful for other very high level programming languages.

Introduction

Aldor is a high-level computer programming language designed for symbolic computation. It was originally developed by a team under the direction of Stephen Watt at IBM T.J. Watson Research Center at Yorktown Heights, New York. Both INRIA (France) and NAG (UK) have contributed significantly to the libraries of Aldor as well. Today the development of Aldor is led by the Ontario Research Center of Computer Algebra, at the University of Western Ontario, with contributions from various institutions world-wide, especially INRIA. The Aldor distribution is freely available via the web [2].
Currently, the Aldor distribution provides an optimizing compiler and an interpreter, together with a set of libraries for symbolic computation. The interpreter is essential for prototyping symbolic algorithms before presenting them as compiled libraries. The interpreter works by first translating source to an intermediate code, FOAM, using the front end of the compiler, and then this intermediate code is executed by a software interpreter. This interpretive environment provides an excellent context for debugging.
The optimizing compiler for Aldor first compiles the source program to intermediate code, optimizes the intermediate code, and then generates machine code for specific hardware by generating very low-level C. The resulting code is 10 to 1000 times more efficient than the original and is typically as efficient as optimized hand-written C code for the same problem. Because the generated C code is at such a low level, any information about the high-level constructs of the source program is lost, and debugging using the usual tools provides only

arcane information useful only to those with deep knowledge of the compiler's internals.
The management of time and space is a key issue in symbolic computation, where even the best algorithms often have exponential complexities. Therefore, the interpreter is not adequate for large problems. To debug such problems, it becomes necessary to compile instrumented versions of suspect modules and to return to the days of primitive debugging with print statements. This is clearly an unsatisfactory situation.
The purpose of this study is to investigate whether is practical for programs to be developed in a combined compiled/interpreted environment, while preserving the best features of each. This would allow mixed programs with large well-understood parts compiled, and suspect modules or functions to be run interpretively and thus be debuggable at a users semantic level. Our criteria for this study were that:

- production level compiled code need not be re-compiled in order to be run in this mixed mode
- compiled code and interpreted code be freely mixed, both of them reading and writing the same data in the memory
- compiled code remain as efficient as in a purely compiled environment
- interpreted code be as fully debuggable as code in a purely interpreted environment
- values of variables in the environment of either compiled or interpreted programs be displayed using their own high-level methods.

The use of combined compiled/interpreted environments is not new. This has been common practice for two decades in Lisp systems. Our work addresses several new questions, however:
To our knowledge all previous work in combined compiled/interpreted environments has been in the context of a language, such as Lisp, that is normally interpreted and where the efficiency of compiled programs is not critical. In our case, Aldor (like FORTRAN and C) was conceived as a language to be compiled for efficiency and so it is not acceptable to have any overhead to support an interpretive mode in compiled programs.
Secondly, previous work along these lines viewed compiled code as an exceptional case to be loaded into an interpretive environment. In our case, we build compiled executables that contain interpreter support as a form of library support, allowing the (normally few) interpreted modules to masquerade as compiled code.
This paper is organized as follows: Section 1 is dedicated to the design of our Aldor debugger. In Section 2, we described the unification of the runtime environments of the Aldor interpreted code and compiled code. Appendix is a sample session of the Aldor debugger.

1 The Aldor Debugger Architecture

As shown in Figure 1, the architecture of the Aldor debugger can be separated into two main modules:

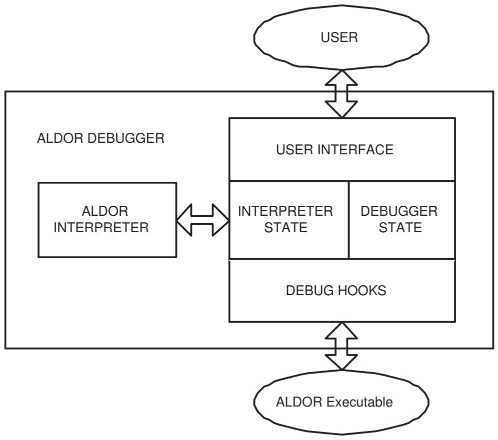

Figure 1: Architecture of the Aldor debugger

- The Aldor interpreter which executes the FOAM code generated by the query commands of the user.
- The debug library which provides the following components:
  - the user interface which is the user command language and its implementation,
  - the debug hooks which are the statements to be added into the Aldor source program in order to control its execution,
  - the debugger state which represents the internal state (breakpoint list, current function and line, ...) of the debugger,
  - the interpreter state which keeps track of the data (current FOAM unit and FOAM program, ...) needed in order to invoke the interpreter.

The debug hooks are inserted into the FOAM code by the Aldor compiler. They allow the debugger state and the interpreter state to collect data from the executable. The debugger state also collects data from the user.
When the debugger option is enabled, the Aldor compiler inserts these debug hooks into the proper positions of functions in generated code. Then it links them together to generate the target executable. When the executable is running, these debug hooks will save debugging information such as which function is executing, which line of Aldor source code is reached or what is the breakpoint list that user has set. These debug hooks can be intercepted by programs written in Aldor and used as the basis of a debugger. In this way, the user can print out these debug info at any time.
An interpreter state saves data including unit and program names of the corresponding FOAM code, line number and lexical environment for the Aldor interpreter. The interpreter state is updated once the Aldor executable enters into a new functional closure.

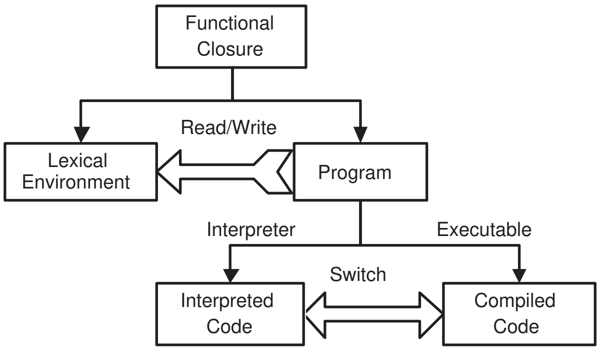

Figure 2: Mixed-mode interpreted/compiled system

2 Unification of the Runtime Environments

One of the requirements for our combined compiled/interpreted environment is that normally compiled code need not be re-compiled in order to run in this mixed mode. In addition, compiled code and interpreted code must be freely mixed, both of them reading and writing the same data in memory. A consequence of this is that the runtime environment of a binary executable and the runtime environment of the interpreter must share the same lexical environment structure.

The runtime environments of the Aldor interpreter and executables are similar for the support they provide for domains and categories. However, they differ on the data representation of language structures including functional closure and lexical environment.

As shown in Figure 2, The program part is unified by the switch between the interpreted code and the compiled code. When the debugger invokes the interpreter in order to query variables, the debugger switches from compiled version of the program to its interpreted version of the program. This switch is by means of the interpreter state, which was described in the previous section.

Because both the Aldor executable and the Aldor interpreter read and write to the same lexical environments, the data structures to represent the lexical environments should be unified. The unification is realized by constructing C structs with memory alignment on the fly for lexical environments and records in the Aldor interpreter

References

[1] Alfred V.Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison Wesley Longman

[2] Stephen M. Watt, *Aldor* in Handbook of Computer Algebra, pp. 154-160. J. Grabmeier, E. Kaltofen, V. Weispfenning editors, Springer Verlag Heidelberg 2003. (See also Aldor, http://www.aldor.org.)

[3] GDB: The GNU Project Debugger
http://www.gnu.org/software/gdb/gdb.html

[4] Guy L. Steele Jr. *Common Lisp: The Language,* Second edition, Digital Press, 1990.

[5] JDB - The Java Debugger
http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html

[6] Jonathan B. Rosenberg. *How Debuggers Work.* John Wiley & Sons, INC.

[7] Norman Ramsey. Embedding an Interpreted Language Using Higher-Order Functions and Types. *In Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators (IVME'03),* June 2003.

[8] Ronald Mak. *Writing compilers and interpreters.* New York: Wiley Computer Publishing, c1996. 2nd Ed.

[9] Rhys Weatherley, Gopal V. Design of the Portable.NET Interpreter, *Linux Conference 2003,* Australia.

[10] Stephan Diehl, Claudia Bieg. A new Approach for implementing stand-alone and web-based Interpreters for Java. *In Proceedings of the 2nd international conference on Principles and practice of programming in Java (ACM PPPJ 2003),* pages: 31-34.

[11] Stephen M. Watt, P.A. Broadbery, P.Iglio, S.C.Morrison and J.M.Steinbach, *Foam: A First Order Abstract Machine, V 0.35,* IBM Research Report RC 19528, 1994.

[12] Stephen M. Watt, *A# Language Reference, V0.35,* IBM Research Report RC 19530, 1994.

[13] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, Robert S. Sutor. A first report on the A# compiler. *In Proceedings of the international symposium on Symbolic and algebraic computation (ISSAC 1994),* August 1994.

Jinlong Cai, Marc Moreno Maza, Stephen Watt
Ontario Research Center of Computer Algebra
University of Western Ontario
{jcai,moreno,watt}@scl.csd.uwo.ca
Martin Dunstan
Department of Applied Computing
University of Dundee, UK
mdustan@computing.dundee.ac.uk

Appendix: Demo of the First Aldor Debugger

A.1 The Source Program deb40.as!

```
1:  #include "axllib"
2:  #include "debuglib"
3:  start!()$NewDebugPackage;
4:  main(p:SingleInteger): SingleInteger == {
5:      import from SingleInteger, String;
6:      import from Array SingleInteger;
7:      local x: SingleInteger := p*p;
8:      y: SingleInteger := foo(x);
9:      z: String := "ORCCA @ UWO";
10:     w: Array SingleInteger := new(2, x);
11:     hin(p1:SingleInteger): SingleInteger == {
12:         print << z << newline;
13:         print << w << newline;
14:         foo(p1);
15:     }
16:     hin(y);
17: }
18: foo(f:SingleInteger): SingleInteger == {
19:     f*2;
20: }
21: main(2);
```

A.2 A Debug Session

```
$ aldor -wdebugger -ldebuglib -laxllib -grun deb40.as
————————————————————————
Aldor Runtime Debugger
v0.60 (22-May-2000), (c) NAG Ltd 1999-2000.
v0.61 (05-Dec-2003), ORCCA @ UWO.
Type "help" for more information.
————————————————————————
main(p:SingleInteger == 2) ["deb40.as" at line 5]
5         import from SingleInteger, String;
(debug) step
main(p:SingleInteger == 2) ["deb40.as" at line 6]
6         import from Array SingleInteger;
(debug) step
main(p:SingleInteger == 2) ["deb40.as" at line 7]
7         local x: SI := p*p;
(debug) step
main(p:SingleInteger == 2) ["deb40.as" at line 8]
8         y: SI := foo(x);
(debug) step
foo(f:SingleInteger == 4) ["deb40.as" at line 18]
18    foo(f:SI): SI == {
(debug) step
main(p:SingleInteger == 2) ["deb40.as" at line 9]
9         z: STR := "ORCCA @ UWO";
(debug) step
main(p:SingleInteger == 2) ["deb40.as" at line 10]
10        w: Array SingleInteger := new(2, x);
(debug) step
main(p:SingleInteger == 2) ["deb40.as" at line 11]
11        hin(p1:SingleInteger): SingleInteger == {
(debug) print x
4
(debug) print y
8
(debug) print gcd(x,y)
4
(debug) print x mod y
4
(debug) print z
ORCCA @ UWO
(debug) print w
array(4, 4)
(debug) print p
2
(debug) update z:="Western Ontario"
(debug) print z
Western Ontario
(debug) update w:=new(5,10)
(debug) print w
array(10, 10, 10, 10, 10)
(debug) step
hin(p1:SingleInteger == 8) ["deb40.as" at line 12]
12                    print << z << newline;
(debug) step
Western Ontario
hin(p1:SingleInteger == 8) ["deb40.as" at line 13]
13                    print << w << newline;
(debug) step
array(10, 10, 10, 10, 10)
foo(f:SingleInteger == 8) ["deb40.as" at line 18]
18    foo(f:SingleInteger): SingleInteger == {
(debug) step
hin(p1:SingleInteger == 8) ["deb40.as" at line 14]
14                    foo(p1);
(debug) step
main(p:SingleInteger == 2) ["deb40.as" at line 16]
16        hin(y);
(debug) next
$ exit.
```