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

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

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 runtime 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.

1 Introduction

- The interpreter works by first translating source code 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 it, and then generates machine code for specific hardware by generating very low-level C. But any information about the high-level constructs of the source program is lost, and debugging using the usual tools (gdb, etc) is not relevant.

2 Objective

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 user's 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.

3 The Aldor Debugger Architecture

The architecture of the Aldor debugger can be separated into two main modules:

- 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.

4 Unification of the Runtime Environments

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.

- 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

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

```
1:  $ aldor -wdebugger -Mdebuglib -laxllib -grun deb40.as
2:  ------------------------------------
3:  Aldor Runtime Debugger
4:  v0.60 (22-May-2000), (c) NAG Ltd 1999-2000.
5:  v0.61 (05-Dec-2003), ORCCA @ UWO.
6:  Type "help" for more information.
7:  ------------------------------------
8:  main(p:SingleInteger == 2) ["deb40.as" at line 5]
9:  5       import from SingleInteger, String;
10: (debug) step
11: main(p:SingleInteger == 2) ["deb40.as" at line 6]
12: 6       import from Array SingleInteger;
13: (debug) step
14: main(p:SingleInteger == 2) ["deb40.as" at line 7]
15: 7       local x: SI := p*p;
16: (debug) step
17: main(p:SingleInteger == 2) ["deb40.as" at line 8]
18: 8       y: SI := foo(x);
19: (debug) step
20: foo(f:SingleInteger == 4) ["deb40.as" at line 18]
21: 18    foo(f:SI): SI == {
22: (debug) step
23: main(p:SingleInteger == 2) ["deb40.as" at line 9]
24: 9       z: STR := "ORCCA @ UWO";
25: (debug) step
26: main(p:SingleInteger == 2) ["deb40.as" at line 10]
27: 10      w: Array SingleInteger := new(2, x);
28: (debug) step
29: main(p:SingleInteger == 2) ["deb40.as" at line 11]
30: 11      hin(p1:SingleInteger): SingleInteger == {
31: (debug) print x
32: 4
33: (debug) print y
34: 8
35: (debug) print gcd(x,y)
36: 4
37: (debug) print x mod y
38: 4
39: (debug) print z
40: ORCCA @ UWO
41: (debug) print w
42: array(4, 4)
43: (debug) print p
44: 2
45: (debug) update z:="Western Ontario"
46: (debug) print z
47: Western Ontario
48: (debug) update w:=new(5,10)
49: (debug) print w
50: array(10, 10, 10, 10, 10)
51: (debug) step
52: hin(p1:SingleInteger == 8) ["deb40.as" at line 12]
53: 12              print << z << newline;
54: (debug) step
55: Western Ontario
56: hin(p1:SingleInteger == 8) ["deb40.as" at line 13]
57: 13              print << w << newline;
58: (debug) step
59: array(10, 10, 10, 10, 10)
60: foo(f:SingleInteger == 8) ["deb40.as" at line 18]
61: 18    foo(f:SingleInteger): SingleInteger == {
62: (debug) step
63: hin(p1:SingleInteger == 8) ["deb40.as" at line 14]
64: 14              foo(p1);
65: (debug) step
66: main(p:SingleInteger == 2) ["deb40.as" at line 16]
67: 16      hin(y);
68: (debug) next
69: $ exit.
```