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

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

Keywords: Aldor, Debugger, High level language, Interpreter, Compiler, Runtime environment, Debug library.

## 1 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: The Aldor interpreter is presented in Section 2. Section 3 is dedicated to the design of our Aldor debugger. In Section 4, we described the unification of the runtime environments of the Aldor interpreted code and compiled code. The implementation of the query commands is explained in Section 5 together with a sample session of the Aldor debugger. Finally, conclusions are reported in Section 6.

# 2 THE ALDOR INTERPRETER

## 2.1 The Aldor Programming Language

Aldor is a type-complete, strongly-typed, imperative programming language with a two-level type system with domains and type categories. These are similar in some ways to classes and interfaces in Java. Types and functions are first class entities allowing them to be constructed and manipulated within Aldor programs just like any other value. Pervasive use of dependent types allows static checking of dynamic objects and provides object-oriented features such as parametric polymorphism.
What does this mean for a normal user? Aldor solves many difficulties encountered with certain widely-used object-oriented programming languages. It allows programs to use a natural style, combining the more attractive and powerful properties of functional, object-oriented and aspect-oriented languages. These features are essential for expressing the rich and complex relations of the mathematical objects involved in symbolic computations.

### 2.2 A Session with the Aldor Interpreter

As shown in Figure 2, the Aldor interpreter can be used in two different ways:

- running Aldor programs without compiling them to executable files,
- and writing Aldor programs in interactive mode.

The interactive mode provides an interactive environment in which it is possible to define functions and domains, to use operations provided by the library, to evaluate functions and to use other features. Below is a session of the interactive mode.

```
%3 >> Residue(R: EuclideanDomain, a:R): Ring == R add {
...      Rep == R; import from Rep;
...      coerce(x: R): % == per(x rem a);
...      (x: %) + (y: %): % == per((rep(x)+rep(y)) rem a);
...      -(x:%): % == per((- rep(x)) rem a);
...      (x: %) * (y: %): % == per((rep(x)*rep(y)) rem a );
...      }
Defined Residue @ (R: EuclideanDomain, a: R) ->
   ( Ring with  == R add () )
                              Comp: 340 msec, Interp: 120 msec
%4 >> a: Integer == 7
Defined a @ AldorInteger
                              Comp: 70 msec, Interp: 990 msec
%5 >> Z7  == Residue(Integer, a)
Defined Z7 @ ? == Residue(AldorInteger, a)
                              Comp: 10 msec, Interp: 0 msec
%6 >> x: Z7 := coerce(13)
6 @ Z7
                              Comp: 0 msec, Interp: 90 msec
%7 >> x * x
1 @ Residue(AldorInteger, a)
                               Comp: 10 msec, Interp: 80 msec
```

### 2.3 First Order Abstract Machine (FOAM)

The purpose of the Aldor interpreter is to interpret FOAM code. FOAM is a High-Level Intermediate Representation (HIR) used by Aldor [11]. FOAM is platform independent, has well defined semantics and can be mapped to

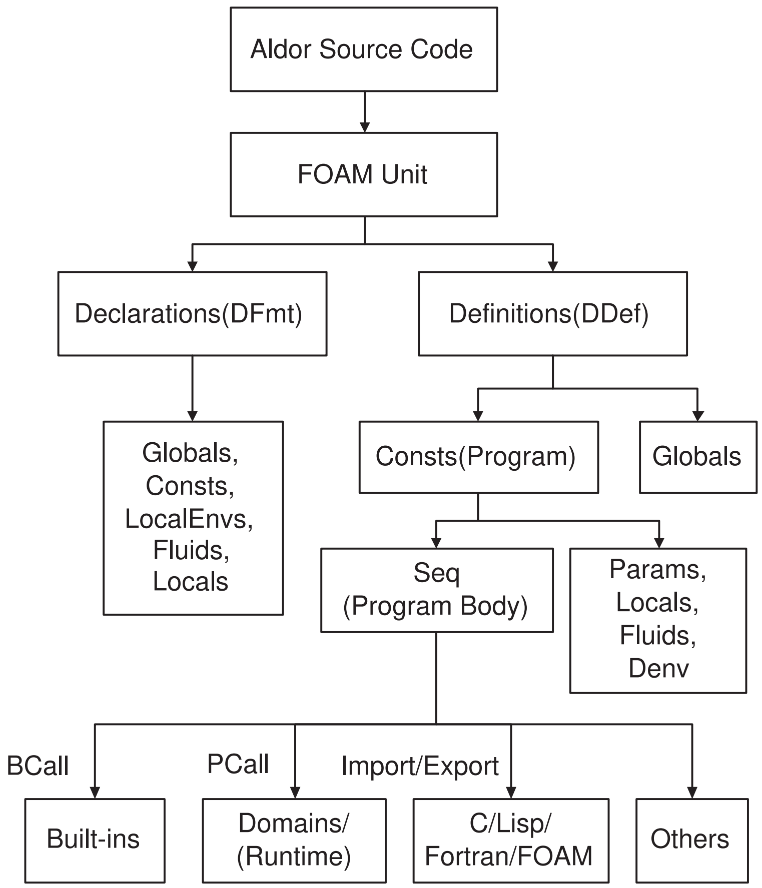


Figure 1: High-level structure of a FOAM unit

ANSI C and LISP efficiently. Various optimizations are implemented as FOAM-to-FOAM transformations. FOAM is first order in the sense that FOAM types are not values.
Each Aldor source program is compiled into a FOAM unit. Figure 1 shows the high-level structure of a FOAM unit. We describe below the parts of a FOAM unit and the FOAM instructions used in the remaining sections of this article.

A FOAM unit consists of a list of declarations (DFmt) and definitions (DDef). Typical declarations include global variables, constants (programs) and localEnvs (local environments). The definitions consist of global variables, programs and initializations. A localEnv declaration lists the lexical variables of a lexical environment. Each program has a local declaration section. For program p, the local declaration section of p lists the local variables, fluids and parameters for use during the execution of p.

The DEnv declaration section (for instance, Line 7 in Figure 5) of a program p lists the declaration indices (4, 6, 7) for the lexical environment levels with respect to p. These indices refer to the (DDecl localEnv) slots in the declaration section of the enclosing unit. The instruction (Lex lev n) (line 11 in Figure 5) returns a reference to the lexical variable at slot n of the lexical environment lev level out of the enclosing program. The Seq instruction denotes the body of a FOAM program which is made up of a sequence of commands such as BCall, OCall, CCall and PCall. The CCall is a closure call which calls the program part (lambda-expression) of a functional closure with the lexical environment portion of the closure as its parameter. The OCall calls a program in a lexical environment. The only way to exit a program body is by a Return instruction.

### 2.4 How the FOAM Interpreter Works

The Aldor interpreter can be divided into three modules:

- The compiler module which compiles the Aldor source code to FOAM code
- The FOAM unit loader which loads the main FOAM unit, the FOAM units in the libraries and the runtime environment. The main FOAM unit is the FOAM program generated from the Aldor source program.
- The execution engine which starts the execution from the first program in the main FOAM unit.

Figure 2 shows the high-level structure of the Aldor interpreter.

We shall now concentrate on describing the execution engine. The execution engine is formed by three main functions: fintExecMainUnit(), fintStmt() and fintEval(). The purpose of the function fintExecMainUnit() is to

- Initialize the runtime stack and the global variable of the interpreter including unit, prog and tape which are current running unit, program and tape.
- Allocate local variables, parameters and lexical environments for the current program of a unit.

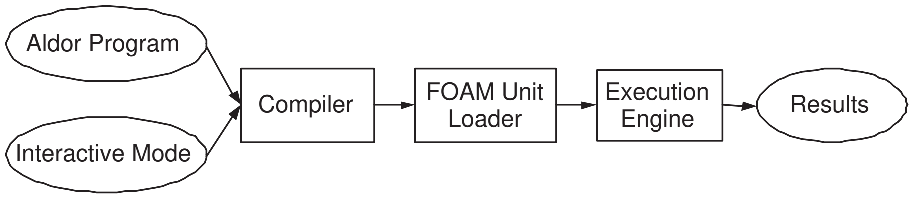


Figure 2: High-level structure of the Aldor interpreter

- Start interpreting by calling fintStmt() function.

The function fintStmt() interprets the byte code of a program line by line to the end of the program. It branches into different procedures by checking the operand of the current statement. Typical operands are Seq, Return and CCall.
The function fintEval() evaluates an expression of the statement which fintStmt() is interpreting. It returns a data object and its data type to fintStmt().
Execution begins with the execution of first const value, a program p of the main unit (mainUnit) under the environment with the format pointed to by slot 0 of the DEnv section of p. If there are lexical variables listed under the format, space is allocated for each environment variable contiguously in the heap. Otherwise there is no environment variable. Either way, a call to fintEnvPush pushes the environment with format p.DEnv[0] onto the stack of environments. Program p is responsible for the initializing the values of environment variables. The interpreter starts the execution by calling fintStmt() from the Seq statement of current program and interprets the program line by line until the Return statement. In each statement, the interpreter calls the evaluation function fintEval() recursively for each FOAM expression.

# 3 THE ALDOR DEBUGGER ARCHITECTURE

In our combined compiled/interpreted environments, each line of the source program to be debugged is run as compiled code, whereas each query command (like update of a variable of the source program) is run as interpreted code. This design satisfies our requirement that compiled code remains as efficient as in a purely compiled environment.

As shown in Figure 3, the architecture of the Aldor debugger can be

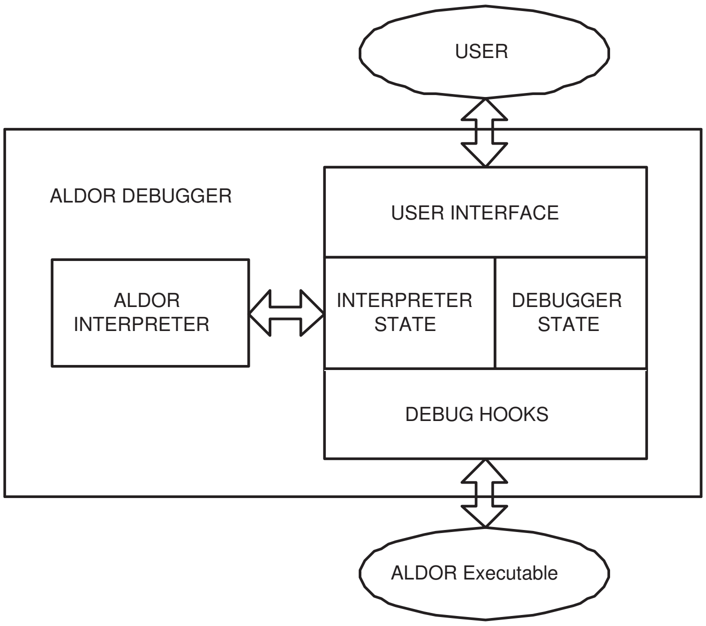


Figure 3: Architecture of the Aldor debugger

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.

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.
Our debugger relies on a preliminary work by Martin Dunstan (University of Dundee, UK) who wrote a library in the Aldor language providing support for debugging. The debug library implements functions, domains and macros that are useful for logging and debugging. The library exports functions which are registered by the Aldor runtime environment. These functions are the debug hooks.
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.
To illustrate, we now give an example of how the debugging system works. Consider the simple Aldor program:

```
myfun(c:Integer):() == {          -- line 23
      bar();                      -- line 24
      local b: Integer := c;      -- line 25
}                                 -- line 26
```

When we compile this file with the debugger option, the Aldor compiler generates code that actually looks like this:

```
myfun(c:Integer):() == {
      -- Create a context while executing "myfun"
      context:Ptr==rtDebugEnter("foo.as", 23, (), "myfun", 1);
      -- Assign function parameters
      rtDebugAssign("foo.as",23,context,"c", Integer, c, -2, 0);
      -- Now executing the context
      rtDebugInside(context);
      -- Execute to the first statement at line 24.
      rtDebugStep("foo.as", 24, context);
      -- About to make a function call
      rtDebugCall("foo.as", 24, myfun, "bar", (), ()->(), 0);
      bar();                                  -- line 24
      -- Execute to the next statement
      rtDebugStep("foo.as", 25, context);
      -- Variable assignment
      rtDebugAssign("foo.as",25,context,"b", Integer, c, -1, 0);
      local b:Integer := c;                   -- line 25
      -- About to exit function with no return value
      rtDebugExit("foo.as", 26, context);
   }
```

In the remainder of this section, we describe the implementation of the interpreter state. The data structure below represents the interpreter state.

```
typedef struct {
        String unit;  //current FOAM unit name
        String prog;  //current FOAM program name
        FiEnv env;    //current lexical environment
        int lineno;   //current line number of source program
} IntState;
```

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.

To update the interpreter state, a list of debug hooks is inserted into the FOAM code of the Aldor program by the Aldor compiler. These debug hooks include rtDebugIntEnter, rtDebugIntStep and rtDebugIntExit. rtDebugIntEnter pushes the old interpreter state to an interpreter stack, and updates current unit and program name in a new interpreter state. rtDebugIntStep updates current lexical environment for this new interpreter state. rtDebugIntExit pops the last interpreter state from the interpreter stack as the current interpreter state.

# 4 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 purpose of a runtime environment is to provide language features for executing a program that cannot be determined at compile time. It maps language structures to machine structures and provides a set of routines to be called by compiled code. 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. Before discussing the unification of the runtime environments, let us recall the notion of a lexical environment and of a functional closure and explain their implementation in the Aldor runtime environments.[4]

A lexical environment contains, among other things: ordinary bindings of variable names to values, lexically established bindings of function names to functions, macros, symbol macros, blocks, tags, and local declarations. The data structure implementing a lexical environment in Aldor is the struct below:

```
typedef struct _FiEnv {
  Ptr level;
  struct _FiEnv *next;
  FiWord info;
}*FiEnv;
```

When invoked on arguments, a function executes its lambda-expression in the lexical environment that was captured at the time of creation of the lexical closure, augmented by the binding of the function's parameters to the corresponding arguments. The corresponding functional closure is the pair consisting of the lexical environment of the function and its lambda-expression. The data structure implementing a functional closure in Aldor is:

```
typedef struct _FiClos {
  FiEnv env;    // lexical environment
  FiProg prog;  // program
}*FiClos;
```

Hence, a functional closure is a function that has partially or fully received its arguments (i.e., lexical environments) and awaits its evaluation. The functional closure can be invoked like a function. When this happens, the associated piece of code is executed with the lexical environments as its arguments. The functional closure is a basic type of function call in the Aldor compiled C code and interpreted code. Thus, the lexical environments can be passed between compiled code and interpreted code for evaluation.
The runtime environments of the Aldor interpreter and Aldor executables are quite different by the nature of execution. Therefore the fields of struct _FiClos and _FiEnv are mapped in a different manner.
As shown in Figure 4, the program part of a functional closure is the lambda expression which reads or writes the lexical variables in a lexical environment. The program parts cannot be unified because it is compiled code in an Aldor executable, whereas it is FOAM byte code in the Aldor interpreter. Actually, 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. Indeed, the interpreter state keeps track of the program name of the functional closure being currently executed such that the Aldor interpreter can invoke the FOAM version of this program.

Another difference is the data structure of the field level in struct _FiEnv. In an Aldor executable, this is a C struct. Here is an example of a lexical environment in an Aldor executable (total size = 12 bytes):

```
struct Fmt6 {
        FiWord X0_p;
        FiWord X1_w;
        FiWord X2_x;
};
```

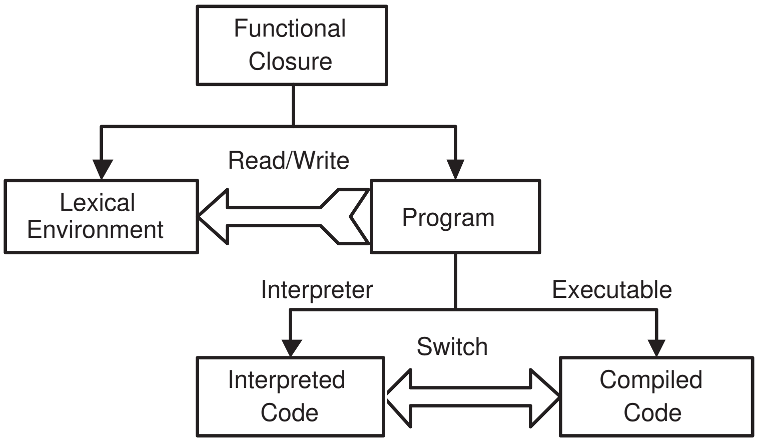


Figure 4: Mixed-mode interpreted/compiled system

However, it is an array of union dataObj in the Aldor interpreter where union dataObj is an 8 bytes long data structure shown below.

```
union  dataObj {
  FiWord    fiWord;
  FiArb     fiArb;
  FiPtr     fiPtr;
  FiBool    fiBool;
  FiByte    fiByte;
  FiClos        fiClos;
  ...
};
```

The corresponding lexical environment in the FOAM code of above C struct is shown below (total size = 24 bytes):

```
(DDecl
  LocalEnv
  (Decl Word "p" -1 4)
  (Decl Word "w" -1 4)
  (Decl Word "x" -1 4))
```

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. We explain now how to construct an equivalent C struct representing a lexical environment in the Aldor interpreter.
Most CPUs require that objects and variables reside at particular offsets in the system's memory. For example, 32-bit processors require a 4-byte integer to reside at a memory address that is evenly divisible by 4. This requirement is called "memory alignment". Thus, a 4-byte integer can be located at memory address 0x2000 or 0x2004, but not at 0x2001. On most Unix systems, an attempt to use misaligned data results in a bus error, which terminates the program altogether. On Intel processors, the use of misaligned data is supported but at a substantial performance penalty. Therefore, most compilers automatically align data variables according to their type and the particular processor being used. This is why the size that structs and classes occupy is often larger than the sum of their members' size.

```
struct member {
    char gender;
    int age;
}
```

As shown above, apparently, struct member should occupy 5 bytes (1+4). However, most compilers add some unused padding bytes after the field 'gender' so that it aligns on a 4 byte boundary. Consequently, member occupies 8 bytes rather than 5. We can examine the actual size of an aggregate by using the expression sizeof (struct member).
To construct this lexical environment in the Aldor interpreter, a block of memory, whose size is the total number of bytes of all its elements, is requested from the heap. Then the memory is initialized with null characters. The following explains how to read and write the elements of the lexical environment.
To write the value of an element of a lexical environment to the memory, if the element is not a pointer, then the value will be written directly in the corresponding offset. For example:

```
case FOAM_Char: *(FiChar *) (ref) = (expr).fiChar; break;
```

If the element is a pointer, then it will be cast to long integer before it is saved into the memory. For example:

```
case FOAM_Ptr:
   FiWord ptr = (FiWord)expr.fiPtr;
   memcpy((FiPtr) (ref), &ptr, sizeof(FiPtr)); break;
```

To read an element from the memory, if the element is not a pointer, then the value will be read directly from the corresponding offset. For example:

```
case FOAM_Char:
   (pdata)->fiChar = *(FiChar*)fintRecGetElem(ref, u, lev, n); break;
```

The function fintRecGetElem() returns the pointer of an element in the record. If the element is a pointer, then it will be cast to proper pointer from long integer after it is read from the memory. For example:

```
case FOAM_Ptr:
   (pdata)->fiPtr =
        (FiPtr) * (FiWord*) fintRecGetElem(ref, u, lev, n); break;
```

By constructing a C struct at runtime, the Aldor interpreter can read and write lexical environments from an Aldor executable directly. Moreover, by rewriting the implementation of FOAM operations on records (FOAM_RNew, FOAM_RElt and FOAM_EElt) in the Aldor interpreter, the records created by the Aldor interpreter can now be read and written by an Aldor executable correctly. Therefore, the variables of domains whose internal representation are records can be queried in the Aldor interpreter properly.

# 5 IMPLEMENTATION OF THE QUERY COMMANDS

One of our requirements was to compile query commands to interpreted code and print or update the values of variables in a purely interpreted environment. To achieve these goals, the following functionalities were developed into our debugger:

- FOAM code generation of debug info,
- FOAM code generation of query commands,
- interpretation of FOAM code of query commands.

In the remainder of this section, we describe the new features related to FOAM code. Then we give a sample session with the Aldor debugger.

## 5.1 Compiling Aldor Source Code to FOAM

In the Aldor debugger, the compiler compiles the Aldor source code in a different way than usual. Normally, when the compiler compiles the Aldor source code to FOAM code, the variables, including parameters of a function, which are in a single lexical scope only are generated as local variables or parameters. Otherwise they are put into a lexical environment so that they can be passed into different lexical scopes. Because it is possible to query any variable from different lexical scopes in the debugger, all debuggable variables and parameters ought to be put into the lexical environments.

## 5.2 Compiling Query Commands to FOAM

In order to query variables in the interpreter, the query commands must be compiled to FOAM code. The query command will be defined in a function query() such that it is easier to find what should be executed after the compilation. Then the function query() is inserted into the Aldor source program in the line where the user issues the command. There are two query commands: print and update. Below is a print example in the debugger session which demonstrates how it is transformed:

```
(debug) print x    //user command
```

The generated Aldor codes to be inserted into the source program:

```
query() : () == { print << x << newline;}
query();
```

Below is an update example:

```
(debug) update x:=5   //the user command
```

The generated Aldor codes is shown below:

```
query() : () == {free x:=5;} // free means x is an existing variable
query();
```

After inserting the query() function to the Aldor source program, the Aldor compiler compiles the source program to FOAM code. It will return to the user if there are compilation errors.

### 5.3 Interpreting the Query Commands

Interpreting the query commands involves the following steps:

- load the interpreted code of the query command to the interpreter,
- pass the lexical environments from the Aldor C executable to the interpreter,
- load the FOAM version of runtime system,
- initialize the lexical environment which contains the functional closures, that is, switch these functional closures from C executable code to interpreted code,
- run query program in current FOAM unit,
- return to the Aldor debugger.

As discussed in Section 2.4, after the Aldor interpreter compiles the Aldor source code with the query command to FOAM code, the FOAM unit loader loads the FOAM code to the interpreter as usual. But the execution engine does not start from the first program of the FOAM unit. Instead, it will interpret the query program only. The function fintExecMainUnit() of the execution engine initializes the global variables prog which is starting program to the query program. The query program does not have any local variables or parameters. It needs the lexical environment passed from the Aldor executable. The lexical environment of query program and the function of Aldor source program where the debugger stops are slightly different. To illustrate how to pass the lexical environment from the Aldor executable to the Aldor interpreter clearly, we use an example of a query program which is generated from the user command "print x" issued between the line 7 and 8 of the source program "deb40.as" in Section 5.4:

```
(Def
 (Const 2 query)
  (Prog ...
   (DDecl Params)
    (DDecl Locals)
    (DFluid)
    (DEnv 4 6 7)
    (Seq
     (CCall
```

```
10:          Word
11:           (Lex 2 8 <<)
12:           (CCall
13:            Word
14:             (Lex 2 12 <<)
15:             (CCall Word (Glo 18 llazyForceImport) (Lex 2 10 print))
16:             (Lex 1 2 x))
17:           (CCall Word (Glo 18 lazyForceImport) (Lex 2 9 newline)))
18:         (Return (Values)))))
```

Figure 5. The FOAM code of a query command

As discussed in Section 2.3, (DEnv 4 6 7) in Line 7 of above FOAM program indicates that the query program has a lexical environment list of indices: 4, 6 and 7. The total number of lexical environments of the query program is one more than that of the function main() where the Aldor debugger stops. Indeed, the query program is one lexical level deeper than the function main(). The local declaration of the lexical environment in main() is shown below:

```
(DEnv 6 7)
```

Since the first lexical environment of query(), (DEnv 4) in the example, is always empty, we push a null environment to the lexical environment list of the query program.

The second lexical environment (DEnv 6) shown below contains the lexical variable which will be queried. So it must be passed to the lexical environment of the query program directly:

```
(DDecl
  LocalEnv
  (Decl Word "p" -1 4)
  (Decl Word "w" -1 4)
  (Decl Word "x" -1 4))
```

The last lexical environment, (DEnv 7) shown below, is the collection of symbols which represent the domains and closures in compiled code format.

```
(DDecl
  LocalEnv
  ...
```

```
  (Decl Word "SingleInteger" -1 4)
  (Decl Word "Character" -1 4)
  ...
  (Decl Clos "<<" -1 4)
  ..))
```

As discussed in the first part of Section 4, this lexical environment (Denv 7) should not be passed to the lexical environment of the query program because compiled code is not executable in the interpreter. Instead, a new lexical environment is created as discussed in the last part of Section 4. Then the FOAM code is loaded for it by executing the first program in the current FOAM unit from the Seq command to the first OCall which is for initializing the lexical environment (DEnv 7) and some global variables. For example, the CCall in Line 12 of the above query program calls the functional closure (Lex 2 12 <<) in Line 14 which refers to the slot 12 of the lexical environment (DEnv 7) to print out the lexical variables (Lex 1 2 x) which refers to the slot 2 of the lexical environment (DEnv 6). Thus the functional closures (Lex 2 12 <<) in (DEnv 7) must be reloaded with the interpreted code.
After passing lexical environments from Aldor executable, the execution engine starts interpreting the query program by calling the function fintStmt() and then fintEval() until it reaches the Return command. As a result, the interpreter prints out the value or updates the value of the variable and returns to Aldor debugger. Finally, the debugger resumes the execution of the compiled code from where it stops.

## 5.4 A Session with the First Aldor Debugger

### 5.4.1 The Source Program "deb40.as"

```
#include "axllib"
#include "debuglib"
start!()$NewDebugPackage;
main(p:SingleInteger): SingleInteger == {
    import from SingleInteger, String;
    import from Array SingleInteger;
    local x: SingleInteger := p*p;
    y: SingleInteger := foo(x);
    z: String := "ORCCA @ UWO";
    w: Array SingleInteger := new(2, x);
    hin(p1:SingleInteger): SingleInteger == {
```

```
        print << z << newline;
        print << w << newline;
        foo(p1);
    }
    hin(y);
}
foo(f:SingleInteger): SingleInteger == {
    f*2;
}
main(2);
```

### 5.4.2 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);
```

```
68:(debug) next
69:$ exit
```

# 6 CONCLUSIONS

As stated in the introduction, Lisp interpreters that allow loading of compiled code have had debugging support for quite some time. These, however, must be seen as mostly interpreted environments, with entirely different trade-offs than our mostly compiled environment.

There are many debugger developed for high-level programming languages. Both GDB for C++[3] and JDB for Java[5] are very famous and widely used debuggers. We compare them with our debugger in terms of the functionality. GDB can only call a member function of a C++ class while the member function is active. Our debugger can call any member operations of an Aldor domain when the domain is active in the current scope. JDB can only print the value of a primitive variable or dump the value of an instance of a class. Our debugger can also update the value of an instance of a domain.

We have successfully created an environment for Aldor that allows compiled and interpreted code to be run in a combined context. Our implementation allows interpreted code to masquerade as compiled code, thus supporting our original objective of not impacting optimized code while providing the debugging support required.

A number of aspects remain to be polished if we wish to bring our implementation close to a production quality environment.

- As discussed in Section 5.2, every time the user issues a query command, this command is compiled together with the Aldor source code of the program to be debugged. Obviously, it would be more efficient to compile the user command only.

- Beside the functionalities such as step, next, breakpoint, where, print and update, we plan to extend the debugger for multi-process and distributed debugging.

- The user interface of the debugger manages user requests and displays results or error messages from the debugger. The interface should be combined to an integrated development center which provides editing, compiling, debugging and executing in one interface.

## 7 ACKNOWLEDGMENTS

Sincere thanks and appreciation are extended to the compiler group of ORCCA, Laurentiu Dragan, Geoff Wozniak, Cosmin Oancea and Michael Lloyd, for their role in the ongoing development of the Aldor compiler and review of an earlier draft of this paper


## 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 standalone 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.