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

by

Jinlong Cai

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
August 2004

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

To my loving parents.
Without their knowledge, wisdom, and guidance.
I would not have the goals I have to strive and be the best to reach my dreams!

ACKNOWLEDGEMENTS

I wish to extend my appreciation and gratitude to my supervisor Dr. Marc Moreno Maza for his guidance, support and encouragement through this entire investigation.

I also wish to thank Dr. Stephen Watt for his great idea about the thesis, and thank Dr. Martin Dustan for his earlier work on the Aldor debugger.

Sincere thanks and appreciation are extended to the compiler group of ORCCA, Laurentiu Dragan, Geoff Wozniak, Cosmin Oancea, Michael Lloyd, Steve Wilson and Xin Li, for their assistance during the course of the studies.

My sincere appreciation also goes to my family for their love and support throughout my life and especially the past few years.

TABLE OF CONTENTS

Chapter 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 [19].

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 implementing them as compiled libraries. The interpreter works by first translating source code to the intermediate representation(IR) of Aldor (First Order Abstract Machine, FOAM for short), 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. Moreover, the interpreter can evaluate expressions which are not in the compiled code (such as GCD in Chapter 6).

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 first generating very low-level C (almost three-address code). The resulting code is nearly the same efficient as carefully written C [18]. 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 it 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 compiled parts, and suspect modules or functions to be run interpretively and thus be debuggable at a user semantic level. Our objectives for this study were that:

Objective 1 compiled code and interpreted code be freely mixed, both of them reading and writing the same data in the memory

Objective 2 compiled code remain the efficiency in a purely compiled environment

Objective 3 interpreted code be as fully debuggable as code in a purely interpreted environment

Objective 4 values of variables in the environment of either compiled or interpreted programs be displayed using their own high-level methods.

Objective 5 high portability - the debugger should be implemented as an application that can be built easily on every platform where the compiler is supported

The use of combined compiled/interpreted environments is not new. This has been common practice for two decades in Lisp systems [3, 6]. Our work addresses several new questions, however:

To our knowledge all Lisp combined compiled/interpreted environments have been in the context where programs were normally interpreted and where the efficiency of compiled programs was 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, Lisp systems view 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.

In [2] a mixed compiled/interpreted environment is used to debug FORTRAN programs in such a way that interpreted code is considered as the exceptional case so that efficiency is preserved. However, this is a case of relatively low level languages, compared to Aldor. Moreover, there is no evidence that the interpretative mode can evaluate expressions that are not in the debuggee. The Aldor debugger allows the user to enter and evaluate expressions (or even statements) in the interpreter during the debugging process. In Chapter 6, we extend the comparison of our study with related work.

The Aldor compiler has been developed for more than 13 years and represents over $200,000$ lines of source code. A preliminary task in our study was to understand the large variety of the involved software components (together with their source code organization): the Aldor programming language, FOAM, the FOAM interpreter, the

Aldor runtime system, code generation and code optimization. In Chapter 2 we review these software components briefly.

The Aldor debugger is organized as a debug library with an embedded interpreter. The debug library consists of the user interface, the debug hooks, the debugger state and the interpreter state. The details are presented in Chapter 3.

One of the main challenges in this design was the unification of the runtime environments of the Aldor interpreted code and compiled code. Because the compiled code and interpreted code have to share the same data, the data structures to represent the data should be exactly the same. However, because of the difference of the running mechanism between the interpreted code and compiled code, the data structures for the same data are different. We explain how we accomplished this unification of runtime systems in Chapter 4

Another challenge is the implementation of the query commands. The query commands have to be compiled and interpreted by the interpreter with the runtime stack. Chapter 5 contains a detailed report on this topic.

In Chapter 7 we discuss how our implementation has reached our initial objectives. We also list some further possible developments for this first Aldor debugger.

The Aldor debugger manual can be found in Appendix A. Appendix B demonstrates how the Aldor debugger works in Emacs.

Chapter 2

The Aldor Compiler and Interpreter

2.1 The Aldor Programming Language

Aldor is an acronym standing for **A** **L**anguage for **D**escribing **O**bjects and **R**elationships. Aldor was originally developed by a tean led by Stephen Watt at IBM Yorktown Heights as an extension language for the AXIOM computer algebra system [17, 18].

The implementation of computer algebra algorithms requires a programming language that allows to represent not only numbers (like integers, rational numbers, complex numbers) but also sets of numbers as values (such that we can say "let Z be the set of integers") and algebraic structures (such as groups, rings, fields). In addition, this language should support functions taking types as parameters and returning new types. For instance, the function $R : \mathit{Ring} \longmapsto R[x] : \mathrm{Ring}$ that associates with a ring R the ring $R[x]$ of univariate polynomials over R. An important feature for these functions is that of *dependent type.* This is the idea of allowing a function like $(R : \mathrm{Ring}, p : R) \longmapsto R/p$ that given R and an element p of R returns the residue class ring R/p.

Aldor was designed to meet these requirements, among others.

Let us summarize the main features of the Aldor language by quoting the Aldor user guide in the next paragraph and the Aldor web site in the next one. See [19] for more detail.

"Aldor is a strongly-typed, imperative programming language with a two-level type system with *domains* and *categories.* These are similar in some ways to *classes* and *interfaces* in Java. Aldor is unusual among compiled programming languages, in that types and functions are *first* class: that is, both types and functions may be constructed dynamically and manipulated in the same way as any other values. This provides a natural foundation for both object-oriented and functional programming styles, and leads to programs in which independently developed components may be combined in quite powerful ways."

"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. The Aldor compiler can produce:"

- stand-alone executable programs,
- object libraries in native operating system formats,
- portable byte code libraries,
- C source.

The object libraries produced by the Aldor compiler can be linked with one another, or with C or FORTRAN code, to form application programs. The byte code libraries can be interpreted, and are used by the compiler for inter-file optimization. The Aldor compiler has been designed for portability and runs in many different environments. Code generated by Aldor will run on 16, 32 and 64-bit architectures.

In the remaining chapters of the thesis, the Aldor executable, compiled code and binary code stand for machine code. The interpreted code and byte code stand for FOAM code [1].

2.2 The Aldor Compiler

The Aldor compiler compiles Aldor source programs to executable target programs. As shown on Figure 2.1, the Aldor compiler consists of well-defined modules including lexical analyzer, syntax analyzer and semantic analyzer, FOAM code generator, FOAM code optimizer, C/Lisp code generator, symbol table manager and error handler.

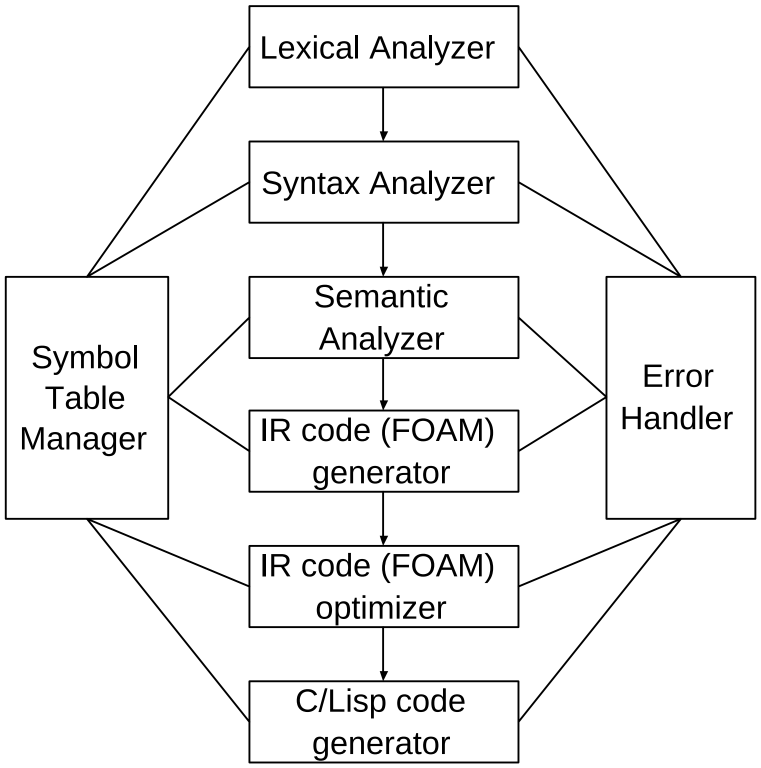


Figure 2.1: High-level structure of Aldor compiler

- The Lexical analyzer groups the input program into tokens;
- The Syntax analyzer recognizes sequences of tokens according to the Aldor grammar and generates Abstract Syntax Trees (ASTs);
- The Semantic analyzer performs type checking (ie, checking whether the variables, functions etc in the source program are used consistently with their definitions and with the language semantics);
- The FOAM code generator translates ASTs into intermediate representation (IR);
- The FOAM code optimizer optimizes FOAM code;
- The C/Lisp code generator translates FOAM code to C/Lisp code;
- The Symbol table manager records the identifiers of the source program and information (called attribute) about them: storage allocation, type, scope, and (for functions) signature;
- The Error handler detects any errors in the process of compiling and reports on them. The compilation will proceed allowing further errors to be detected.

The Aldor compiler compiles the generated C code to executable target programs with machine dependent C compilers. An Aldor program can be run by the Aldor interpreter. However, the binary executable is much faster than interpreted code (see Table 7.1).

From a low-level point of view, the Aldor compiler is divided into three parts: the front-end, the middle and the back-end. These parts can be split into eighteen phases which take input from one phase and pass it onto the next (see Figure 2.2).

Sub-sections 2.2.1, 2.2.2 and 2.2.3 summarize the sucessive steps in compiling an Aldor source file to an executable. Figure 2.2 illustrates this process.

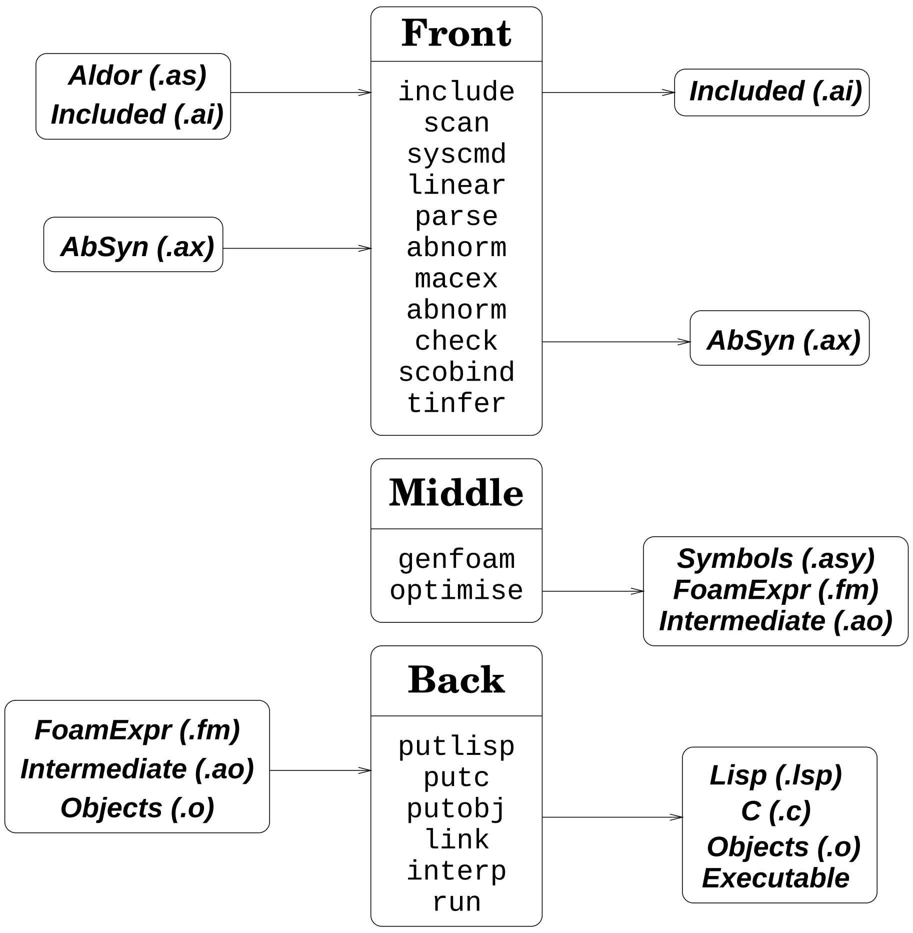


Figure 2.2: Low-level structure of Aldor compiler

2.2.1 Front-end

Files suffixed `.as` (Aldor source programs) and `.ai` (Aldor included files) are passed to to the pre-processing routine `include` which recursively replaces all `#include` directives with the contents of the included files. The result is a `.ai` file that contains no `#include` directives. Next the source code lines are converted into a list of tokens by the `scan` routine using the Lex tool. Pre-processor system commands are then eliminated by `syscmd` and the resulting token list is managed a little more by `linear` to deal with piling. The token list is then converted into an abstract syntax tree by `parse` using a YACC-generated parser from a conflict-free grammar.

Once the abstract syntax tree has been created or loaded, the remainder of the front-end can get to work. First the abstract syntax is given to `abnorm` which ensures that syntactic sugars are eliminated and that a standard form of abstract syntax is obtained. Any macros are replaced by `macex` and another `abnorm` pass is made to ensure that everything is just right. The abstract syntax is then validated by `check` before being written out as a `.ax` file. The remaining two phases of the front-end are performed by the scope binder (`scobind`) and the type-checker/type-inferer (`tinfer`).

2.2.2 Middle

The middle section of the compilation process contains two phases: `genfoam` to generate intermediate FOAM code from an abstract syntax tree, and `optimise` which modifies the FOAM object in situ in an attempt to generate “better” intermediate code. At the end of this latter phase the top-level symbol table may be emitted in a `.asy` file. In addition, a human-readable version of the intermediate code is emitted as a `.fm` file and a FOAM binary object is emitted as a `.ao` file. Note that there are no inputs to this part of the compiler, except for the output of the front-end.

2.2.3 Back-end

The back-end of the compiler is necessary but fairly dull. The input is either a FOAM object from the middle of the compiler, or a FOAM object loaded from a `.fm` or `.ao` file. Native libraries (`.a` files) and native objects (`.o` files) are not examined by the compiler directly and are simply passed to the native linker when needed. The phases here are self-explanatory: `putlisp` is used to convert the FOAM object into a LISP or Scheme file suitable for loading into the AXIOM environment (The original motivation for Aldor was to provide an improved extension language for the AXIOM computer algebra system), `putc` generates a C file which will be compiled and linked against `libfoam` while `putobj` is used to create native object files (by compiling the results of `putc`).

Note that all the necessary phases up to, and including, `putc` are applied to each source file that was specified on the command line in succession. Once these phases are complete the compiler proceeds to the `link`, `interp` and `run` phases. If a native executable is required then `link` performs the necessary work. Interpreting FOAM objects (using the `-Ginterp` option of the Aldor compiler) is performed by `interp` while running native executables (the `-Grun` option) is handled by `run`.

2.3 The Aldor FOAM Interpreter

One of the more unusual features of Aldor is that it provides an interpreter. Interpreters are quite common and some languages are designed specifically with them in mind (*e.g.* Java), but constructing an interpreter for a compiled language such as Aldor is not easy.

Programs running under the Aldor interpreter, or more precisely, the FOAM interpreter, are intended to behave in exactly the same way as if they were running natively or under a LISP system such as AXIOM. Of course with the statically linked system used by the current Aldor compiler, this equivalence can only hold provided that there are no calls to foreign functions. The foreign functions are C functions which are not implemented in the Aldor interpreter. The exceptions are foreign calls to the runtime libraries which the interpreter knows about, and calls to a selection of standard C functions such as `fputs`. Calls to any foreign function that the interpreter does not recognize will cause an exception to be thrown instead.

2.3.1 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 [16]. FOAM is platform independent, has well defined semantics and can be mapped to ANSI C and LISP efficiently. Various optimizations are implemented as FOAM-to-FOAM transformations.

Each Aldor source program is compiled into a FOAM unit. A FOAM unit represents a complete module. A unit lists its imports and exports. Units share global variables, imports and exports. Figure 2.3 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 thesis.

Below is an trivial example which demonstrates an Aldor source program and its FOAM code. The example does not do anything.

```
Aldor source program ``foam.as'':

#include "aldor"
import from MachineInteger;
1;

Its FOAM code ``foam.fm'':

(Unit
  (DFmt
    (DDecl
      Globals
      (GDecl Clos "foam" -1 4 0 Init)
      (GDecl Clos "noOperation" -1 4 1 Foam)
      (GDecl Clos "runtime" -1 4 1 Init))
    (DDecl Consts (Decl Prog "foam" -1 4))
    (DDecl LocalEnv)
```

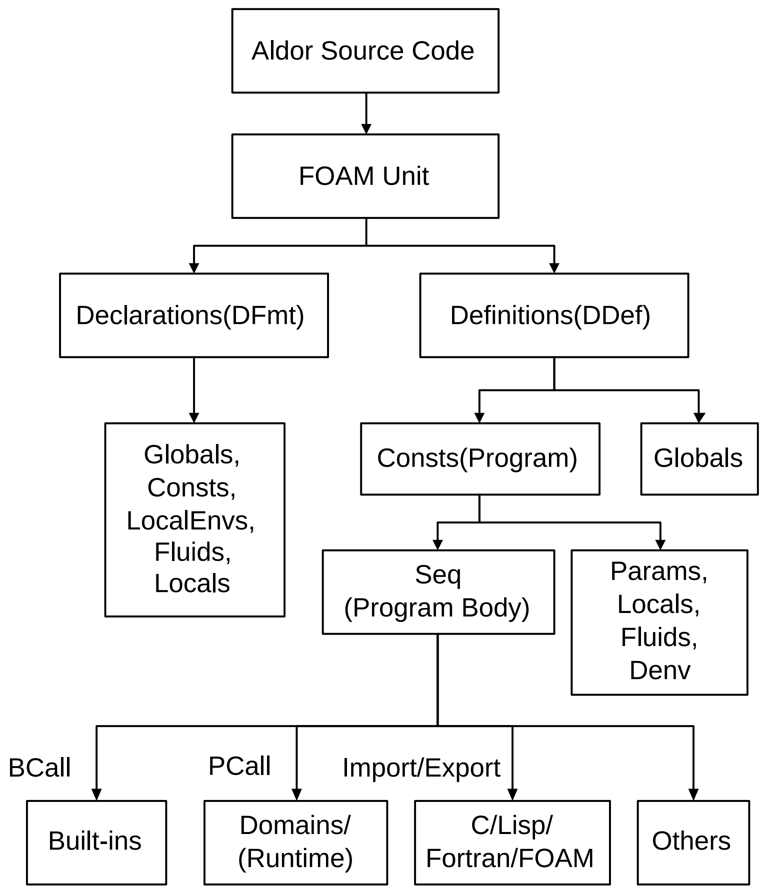


Figure 2.3: High-level structure of a FOAM unit

```
   (DDecl Fluids)
   (DDecl Locals)
   (DDecl LocalEnv))
 (DDef
   (Def
     (Const 0 foam)
     (Prog
       0
       1
       NOp
```

```
    0
    8194
    0
    0
    0
    (DDecl Params)
    (DDecl Locals)
    (DFluid)
    (DEnv 5)
    (Seq
      (CCall NOp (Glo 2 runtime))
      (Set (Glo 0 foam) (Glo 1 noOperation))
      (Return (Values)))))
(Def (Glo 0 foam) (Clos (Env 0) (Const 0 foam)))))
```

The remaining part of this sub-section is a brief overview of a FOAM unit. This material is quite technical and you may refer to [16] for a more expanded presentation.

Each Aldor source program is compiled to a FOAM unit. A FOAM unit consists of a list of declarations (DFmt) and definitions (DDef). Typical declarations include global variables, constants (programs) and local environments (localEnv). The definitions consist of global variables, programs and initializations. A localEnv declaration lists the lexical variables of a lexical environment. A FOAM unit contains several programs. The body of a FOAM program is made up of a sequence of commands. 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 5.1, p. 37) 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 5.1) returns

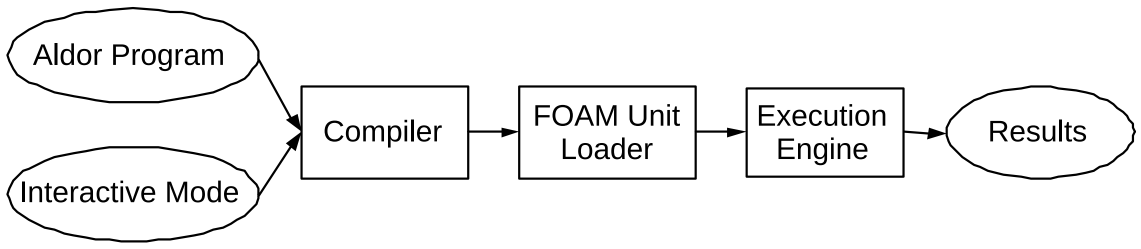


Figure 2.4: High-level structure of the Aldor interpreter

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.

2.3.2 How the Aldor 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 of the called libraries and the runtime system. 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.

2.3.3 FOAM Unit Loader

The main unit is loaded into the instances of two data structures: `fintUnit` and `progInfo` shown below.

```
typedef struct fintUnit {
       unsigned     unitId;      /*Unique for every unit */
       UByte        * tape;
       Buffer       buf;
       String       name;
       Fmt          fmtGlobs;
       ShDataObj    * globValues;
       int          globsCount;
       Fmt          fmtConsts;
       DataObj      constValues;
       int          constsCount;
       Fmt          fmtFluids;
       int          fluidsCount;
       LexLevels    lexLevels;
       int          lexLevelsCount;
}FintUnit;

typedef struct progInfo {
       FintUnit     unit;       /* unit in which is contained*/
       String       name;       /* const name that defines prog */
       FiProgPos    fiProgPos;  /* pos. of Seq command */
       int          size;       /* size */
       FiProgPos    * labels;   /* labels */
       int          nLabels;
       int          labelFmt;
       AInt         retType;    /* return type */
       int          mValFmt;
       UByte        bMask;
       Fmt          fmtLoc;     /* locals */
```

```
        int          locsCount;
        Fmt          fmtPar;     /* parameters */
        int          parsCount;
        UByte        * dfluid;   /* DFluid  */
        UByte        dfluidsCount;
        UByte        * denv;     /* DEnv  */
        UByte        denvsCount;
        FiWord       _progInfo;
}ProgInfo;
```

FintUnit stores the declarations of a FOAM unit including global variables, constants, fluids and lexical environments. ProgInfo stores the definitions of programs of a FOAM unit including local variables, parameters and lexical environments. These two data structures have cross references to each other. The constValues element of FintUnit references all programs in a unit. The unit element of progInfo refers to the fintUnit which it belongs to. The tape parameter of FintUnit holds the FOAM byte code associated with that unit. The fiProgPos of ProgInfo holds the starting position of FOAM_Seq command in a program.

```
typedef struct fmt {
      int     type;
      String  id;
      int     protocol;
      int     format;
      int     offset;              /* used for lex vars */
}*Fmt;
```

The above data structure Fmt (format) is to store the declaration of a global, constant, fluid and local, parameter. The following is an example of a global variable which feeds each elements of Fmt by ”string” assigned to id and so on:

(GDecl Clos “string” -1 4 1 Init)

The definitions (formats) of global variables are stored in the `fmtGlobs` element of the `FintUnit` structure. There are `globsCount` global formats indexed from 0. The values are stored in the `globValues` element. Each of the global values is initialized with a call to the function `shDataObjAdd()` which initializes and resolves the global references of global variables.

2.3.4 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.
- Start interpreting by calling the `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 the 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.

In order to provide a virtually infinite stack, the interpreter stack is organized as a list of stacks. Every element of the list has size `STACK_SIZE`. The starting stack is $< headStack >$. If a `stackFrameAlloc()` or `stackAlloc()` operation needs X bytes and the amount is not available in the current stack, then a new stack of constant size STACK_SIZE is dynamically allocated and is chained to the previous stack. A procedure stack frame consists of a header, parameters, locals and fluids which stores the lexical environment and the current unit of current executing program. Once the program has a call to an other program, the frame will be saved in the stack and a new frame for the call will be allocated.

2.3.5 A Session with the Aldor Interpreter

As shown in Figure 2.4, the Aldor interpreter can be used in two different ways:

- running Aldor programs without compiling them to executable files,
- 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. Note that the user inputs start with the percentage sign, and system outputs include the compiling and interpreting time of user inputs.

```
%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
```

Chapter 3

The Aldor Debugger

3.1 The Aldor Debugger Architecture

3.1.1 Previous Aldor Debugger Architecture

Our debugger relies on a preliminary prototype by Martin Dunstan [10] who developed the Aldor debugger v0.6. The Aldor debugger v0.6 has the debugging functionalities including breakpoint, step, next, cont, off, where, and so on (refer to Appendix A for the definitions of these commands). However, it does not have query commands such as print and update.

As shown in Figure 3.1, the architecture of the Aldor debugger v0.6 can be separated into three 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 like breakpoints,
- the debugger state which represents the internal state (breakpoint list, current function and line, ...) of the debugger.

3.1.2 Extended Aldor Debugger Architecture

In our new combined compiled/interpreted environments, each line of the source program to be debugged is run as compiled code, whereas each query command

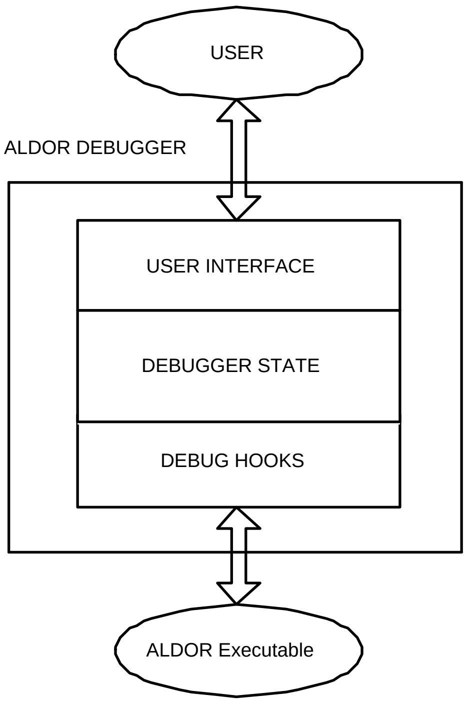


Figure 3.1: Architecture of the Aldor debugger v0.6

(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 3.2, the architecture of the Aldor debugger v0.7 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,

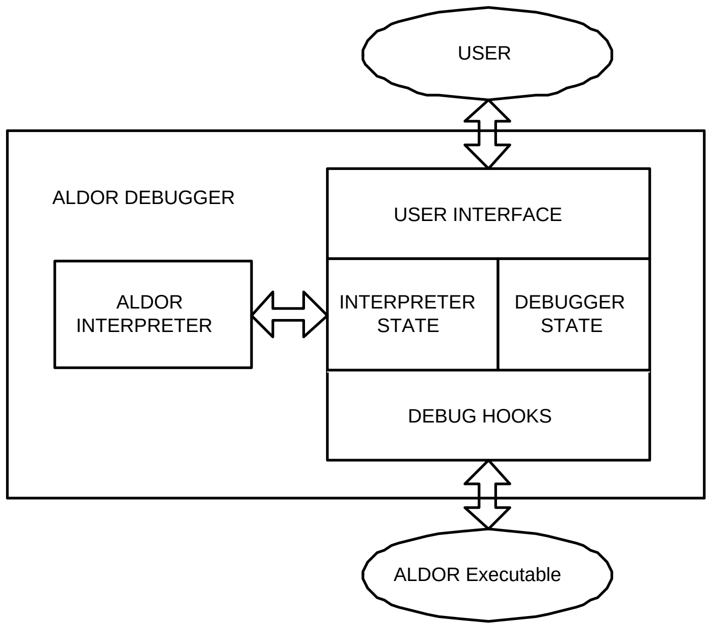


Figure 3.2: Architecture of the Aldor debugger v0.7

- the debug hooks which are the statements to be added into the Aldor source program in order to control its execution like breakpoints,
- 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.

Compared with Dunstan's architecture, the new architecture has two new modules: the interpreter state and the Aldor interpreter. Moreover, the other three modules have been enhanced to support the query commands. The user interface accepts the query commands. Some new debug hooks have been created to update the interpreter state.

3.2 The Debug Library

The debug library is written in the Aldor programming language providing support for debugging. The debug library implements functions, domains and macros that are useful for logging and debugging.

3.2.1 The User Interface

The Aldor debugger has a friendly user interface. New users could easily start by reading the help page of the debugger. Please refer to Appendix A for the details.

3.2.2 The Debug Hooks

The library exports functions which are registered by the Aldor runtime system. These functions are the debug hooks. 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 the generated code. Then it links them against the debug library to generate the target executable. When the executable is running, these debug hooks will save debugging information such as which function is executing or which line of Aldor source code is reached and check if the breakpoints that user has set are hit. 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 this debug information 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 (comments begin with "--"):

```
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);
   }
```

3.2.3 The Debugger State

The debugger state is an Aldor domain used to represent the internal state of the debugger. It includes:

- the configuration of the user such as breakpoints, last command from one of step, next and cont,
- the call stack of the process, which records information about specific function call instances and the variables/constants declared.

The following debug hooks are used to update the status of the debugger state.

```
DbgEnterSIG  - Handler for function-entry events.
DbgInsideSIG - Handler for function-inside events.
DbgReturnSIG - Handler for function-return events.
DbgExitSIG   - Handler for function-exit events.
DbgStepSIG   - Handler for single-stepping events
```

3.2.4 The Interpreter State

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 (see the above `IntState`) 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.

Chapter 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.

4.1 The Aldor Runtime Environment

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 the functional closure and the lexical environment. Before discussing the unification of the runtime environments, let us recall the notion of a lexical environment and of a functional closure [6] and explain their implementation in the Aldor runtime environments.

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; //lexical level
  struct _FiEnv *next;
  FiWord info;
}*FiEnv;
```

The above struct is common to both compiled code and interpreted code. However, the field `level` in this struct is a pointer to an array in the Aldor interpreter and a pointer to a struct in the Aldor executable.

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 (i.e. lambda-expression) 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 data structure implementing a functional closure in Aldor is:

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

The above struct is common to both compiled code and interpreted code. However, the field `prog` in this struct is a pointer to interpreted code in the Aldor interpreter and a pointer to compiled code in the Aldor executable.

4.2 Unification of the Aldor Compiler and Interpreter Runtime Environments

The runtime environments of the Aldor interpreter and Aldor executables are quite different by the nature of execution. As discussed in the previous section, the fields of struct _FiClos and _FiEnv are mapped in a different manner.

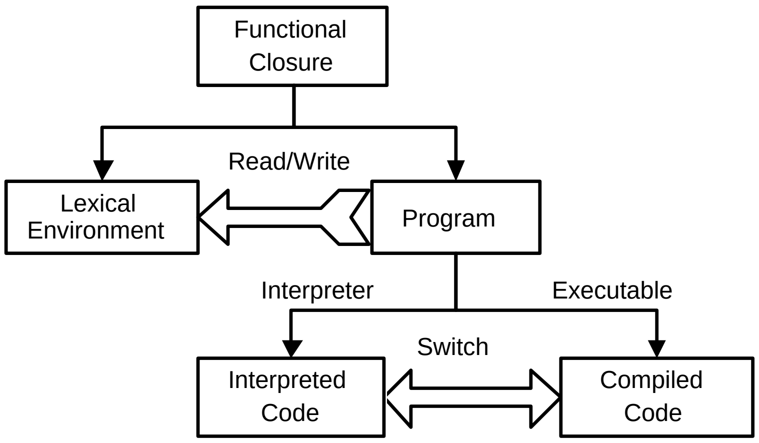


Figure 4.1: Mixed-mode interpreted/compiled system

As shown in Figure 4.1, 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 the 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 chapter. Indeed, the interpreter state keeps track of the program name of the functional closure being currently executed such that the Aldor interpreter can initialize and invoke the FOAM version of this program. We will discuss more details on the switch between interpreted code and compiled code in Section 5.3.

4.3 Constructing C Structs On The Fly with Memory Alignment

We return now to the problem of having the field `level` in the struct `_FiEnv` pointing to different data structures. Here is an example of a data pointed to by a lexical level (i.e. the field `level`) in an Aldor executable (total size = 12 bytes):

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

However, the corresponding pointed to data 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 FOAM code corresponding to the 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 levels, the data structures representing the lexical levels should be unified.

Since a C struct is more efficient than an array of `union dataObj`, we align the latter data structure to the former.

Other data-structures, like records, require this unification between compiled code and interpreted code. The following sub-section explains how to construct in the Aldor interpreter C structs representing lexical levels. This unification applies also to records.

4.3.1 Memory Alignment

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 a such that a is divisible by 4. This requirement is called "memory alignment". Thus, a 4-byte `int` 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 C++ classes occupy is often larger than the sum of their members' sizes.

As shown in Figure 4.2, apparently, the 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, the struct member occupies 8 bytes rather than 5. We can examine the actual size of an aggregate by using the expression sizeof(struct member).

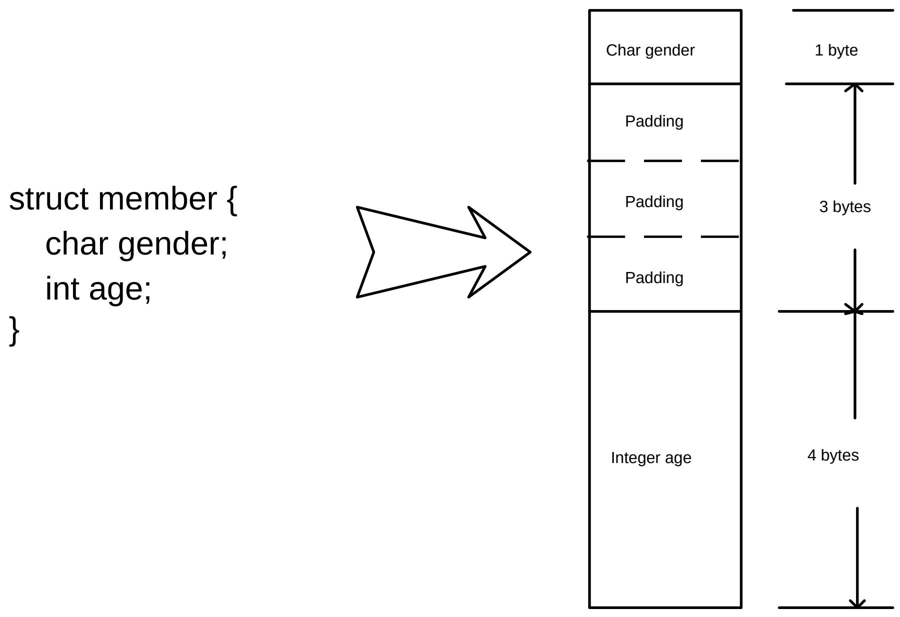


Figure 4.2: Memory alignment of C struct

4.3.2 Creating, Reading and Writing of C Structs

To construct this lexical level in the Aldor interpreter, a block of memory whose size is the total number of bytes of all its fields and the unused padding bytes for memory alignment, is requested from the heap. Then the memory is initialized with null characters. The following explains how to read and write the fields of the lexical environment.

Pointer fields and non-pointer fields are read from (or written to) the block of memory by different processes. The value of a field that is not a pointer is written to memory directly in the corresponding offset. For example:

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

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

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

To read a non-pointer field from memory, the value is 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 field in the record. If the field is a pointer, then it will be cast to proper pointer from long integer after it is read from 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 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.

Chapter 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 added to our debugger:

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

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

5.1 FOAM Code Generation of Debug Info

In the Aldor debugger, the compiler compiles the Aldor source code in a different way than usual. Firstly, the compiler does not optimize the programs. (We aim to relax this limitation in the future.) To utilize all the capabilities of the Aldor debugger, there must be an accurate correspondence between object code and source line numbers, and optimizations can alter this correspondence.

Since it is possible to query any variable from different lexical scopes in the debugger, all debuggable variables and function parameters ought to be put into the lexical environments. However, 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 are generated as local variables or parameters, instead of being placed into a lexical environment. (Variables and function parameters that appear in at least two lexical scopes are placed into a lexical environment so that they can be passed into different lexical scopes.) This choice is motivated by efficiency.

5.2 FOAM Code Generation of Query Commands

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 5.4.2 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 are 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 in the Aldor Interpreter

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 the 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 the query program in the current FOAM unit,
- return to the Aldor debugger.

As discussed in Section 2.3.2, 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 variable prog which is the 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 the 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

```
1:    (Def
2:     (Const 2 query)
3:      (Prog ...
4:       (DDecl Params)
5:        (DDecl Locals)
6:        (DFluid)
7:        (DEnv 4 6 7)
8:        (Seq
9:         (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 5.1: The FOAM code of a query command

the user command "print x" issued between the line 7 and 8 of the source program "deb40.as" in Section 5.4.

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 should be switched to interpreted code 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 the 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 the 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"

```
1:  #include "aldor"
2:  #include "debuglib"
3:
4:  import from MachineInteger, String, Character;
5:  import from Array MachineInteger, TextWriter;
6:
7:  start!()$NewDebugPackage;
8:
9:  main(p: MachineInteger):  MachineInteger == {
10:     local x:  MachineInteger : = p*p;
11:     y:  MachineInteger : = foo(x);
12:
13:     z:  String : = "ORCCA @ UWO";
14:     w:  Array MachineInteger : = new(2, x);
15:
    hin(p1: MachineInteger):  MachineInteger == {
        stdout << z << newline;
        stdout << w << newline;
        foo(p1);
    }

    if (x = 1) then
        return foo(x);

    hin(y);
}

foo(f: MachineInteger):  MachineInteger == {
    f : = f * f;
    f;
}

main(2);
```

5.4.2 A Debug Session

```
bash$ aldor -wdebugger -debuglib -laldor -grun deb40.as
--------------------------------------------------------
Aldor Runtime Debugger
v0.60 (22-May-2000), (c) NAG Ltd 1999-2000.
v0.70 (05-Dec-2003), ORCCA @ UWO.

Type "help" for more information.
--------------------------------------------------------

main(p:MachineInteger == {2}) ["deb40.as" at line 10]
10  local x: MachineInteger := p*p;

(debug) help
```

```
Available commands:
  break: display or insert breakpoints.
   cont: continue execution of program.
 delete: delete breakpoints.
disable: disable breakpoints.
 enable: enable breakpoints.
   halt: abort the program.
   help: access this help system.
  hints: hints and tips for using the debugger.
    int: interactive or non-interactive debugging.
   next: move to next statement (stepping over calls).
    off: turn off the debugging system and continue.
   quit: abort the program.
   step: move to next statement (stepping into calls).
   tips: hints and tips for using the debugger.
verbose: display of event details.
  where: display a stack trace.
  print: print out the value of a variable.
update: update the value of a varibale.
  shell: execute a shell command.

Use "help <command>" for more help on <command>.
(debug) where
1:
main(p:MachineInteger == {2}) ["deb40.as" at line 10]
(debug) step

main(p:MachineInteger == {2}) ["deb40.as" at line 11]
11  y: MachineInteger := foo(x);

(debug) step

foo(f:MachineInteger == {4}) ["deb40.as" at line 29]
```

```
47: 29         f := f * f;
48:
49: (debug) step
50:
51: foo(f:MachineInteger == {4}) ["deb40.as" at line 30]
52: 30  f;
53:
54: (debug) print f
55: 4
56: (debug) where
57: 2:
58: main(p:MachineInteger == {2}) ["deb40.as" at line 11]
59: 1:
60: foo(f:MachineInteger == {4}) ["deb40.as" at line 30]
61: (debug) step
62:
63: main(p:MachineInteger == {2}) ["deb40.as" at line 13]
64: 13  z: String := "ORCCA @ UWO";
65:
66: (debug) print x
67: 4
68: (debug) print y
69: 16
70: (debug) print gcd(x, y)
71: 4
72: (debug) break deb40.as 17
73: Breakpoint set: [*] "deb40.as" at line 17 (hit 0 times).
74: (debug) break
75: ) [*] "deb40.as" at line 17 (hit 0 times).
76: (debug) cont
77: Hit breakpoint #1
78: hin(p1:MachineInteger == {16}) ["deb40.as" at line 17]
79:
```

```
hin(p1:MachineInteger == {16}) ["deb40.as" at line 17]
17      stdout << z << newline;

(debug) print z
ORCCA @ UWO
(debug) update z:="debugger"
(debug) print z
debugger
(debug) step
debugger

hin(p1:MachineInteger == {16}) ["deb40.as" at line 18]
18      stdout << w << newline;

(debug)
[4,4]

hin(p1:MachineInteger == {16}) ["deb40.as" at line 19]
19      foo(p1);

(debug)

foo(f:MachineInteger == {16}) ["deb40.as" at line 29]
29         f := f * f;

(debug)

foo(f:MachineInteger == {16}) ["deb40.as" at line 30]
30     f;

(debug)
```

Chapter 6

Related Work

Debuggers are an essential development aid for the discovery and removal of bugs in software programs [9]. This section discusses a number of papers and products presenting other approaches of debuggers or mixed-mode compiled/interpreted systems.

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.

In [2], B. B. Chase and R. T. Hood have built a facility for program execution that solves some of the problems inherent in debugging large, computationally intensive FORTRAN programs.

To handle the execution of compiled code, their debugger is implemented in the style of other Unix debuggers such as DBX and GDB. That is, the debugger resides in a process that has the program being debugged as a child process. This approach is probably more powerful than the one of our Aldor debugger but would require a multiple year development.

To handle interpretation during program execution, the debugger process includes the code that implements the interpreter in the debuggee. There is no evidence that this FORTRAN debugger could allow the user to evaluate an expression (not included in the debuggee) during the debugging process. Our Aldor debugger offers this feature, which seems to us as a strong requirement for debugging mathematical software.

In [11], Norman Ramsey and David R. Hanson present LDB, a retargetable debugger for cross-debugging through a network. It can use a network to connect to faulty processes and can do cross-architecture debugging. The debugger embeds a machine-independent PostScript interpreter to interpret the symbol table of the debuggee in PostScript format. It is among our future work to extend our Aldor debugger for cross-architecture debugging.

In [5], Gilles Muller, Barbara Moura, Fabrice Bellard and Charles Consel discuss an approach which reconciles portability and efficiency, and preserves the ability to dynamically load Java bytecode. The well-known tradeoff of Java's portability is the inefficiency of its basic execution model, which relies on the interpretation of an object-based virtual machine. Many solutions have been proposed to overcome this problem, such as just-in-time (JIT) and off-line bytecode compilers. However, most compilers trade efficiency for either portability or the ability to dynamically load bytecode. The authors designed and implemented an efficient environment for the execution of Java programs, named Harissa. Harissa's compiler translates Java bytecode to C. The authors claim that the C code produced by Harissa's compiler is more efficient than all other alternative ways of executing Java programs.

JDB [7] and GDB [4] are probably the most popular debuggers today. GDB's architecture is close to that of [2]. Although, they do not mix interpreted code and compiled code, we believe it is important to give a brief comparison between them and our debugger.

- 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.

For example, the following C++ program computes LCM of two integers. When debugging with GDB, we can not print the GCD of these two integers because the GCD function was not presented in this program. However, in 5.4.2, we can call GCD for two integers in the Aldor debugger although the GCD function was not in the debuggee.

```
#include <iostream>
#include ‘‘integer.h’’
int main(){
        Integer x(2), y(8);
        cout << lcm(x,y) << endl;
        return 0;
}
```

- 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.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Let us review our initial objectives stated in the Introduction (Chapter 1), p.2.

Objective 1 Our implementation allows interpreted code to masquerade as compiled code. Thus, supporting our original objective of compiled code and interpreted code being freely mixed.

Objective 2 To make sure that we have achieved the objective that compiled code remains the efficiency in a purely compiled environment, the following experiment was carried out by checking the running times of the interpreted code, compiled code, compiled code for debugging and query commands. The timing of interpreted code was the result of interpreting the FOAM object file. The compiling time was not included. The timing of compiled code for debugging is obtained by piping the "off" command to run the debuggee automatically.

Our experiment is conducted on a PC running at 800 MHZ equipped with 256 MBytes of main memory. The unit of running time is milliseconds. Each running time is the average time of 5 repetitions. The Aldor source files of the experiment are included in Appendix C.

In the examples reported in Table 7.1, the number of elementary arithmetic operations is in $\Theta(2^n)$, whereas the number of debugger prompts is in $\Theta(n)$.

n	100	300	500
Pure compiled code	60	680	2380
Compiled code for debugging	150	950	3280
Query commands	925	925	825
Pure interpreted code	1660	10730	81580

Table 7.1: Benchmarks for compiled, interpreted and mixed code in milliseconds

This shows that the overhead of debugging support for compiled code becomes negligible for large computations comparing to the running time of pure interpreted code. Thus, it proves that the compiled code remains the efficiency in a purely compiled environment.

Objective 3 Since our interpreted code is run by the Aldor interpreter and since the user can evaluate an expression (not present in the debuggee) during the debugging process, our third objective is clearly reached.

Objective 4 From the sample debug session p.40 we can observe that the values of variables in the environment of either compiled or interpreted programs can be displayed using their own high-level methods. Indeed, the value of variable "z" can be displayed by both the interpreted code and the compiled code. Therefore, our fourth objective is reached too.

Objective 5 The Aldor debugger consists of 6,000 lines of source code, including 2,000 lines of ANSI C source code and 4,000 lines of Aldor source code. It is highly portable for Unix/Linux and Windows platforms because ANSI C and Aldor can be run in these platforms.

7.2 Future Work

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. The improvement can be achieved by applying the approach of compiling user inputs in the Aldor interpreter. It requires a lot of work which I cannot finish in the thesis. In the benchmarks reported in Table 7.1, the running time of query commands is a little high but acceptable.

- Beside the functionalities such as step, next, breakpoint, where, print and update, some additional features will be added later including:
  - Setting conditional break point as in GDB
  - watch-points as in [6]
  - Moving up and down the call stack as in GDB
  - reversible execution as in [2]
  - pause command as in [2]

- The Aldor debugger should be implemented as a two process model rather than the single process model. A two process model means that the debugger and the debuggee are in different processes. Currently, the Aldor debugger is compiled with the debuggee in one process. It is easier to communicate between the debugger and the debuggee because they are in the same memory space. However, the single process architecture suffers from some limitations:

  - The debugger debugs itself. The debugger may crash with the debuggee, thus it can not display the stack frames when it crashes.
  - It is unable to perform multi-process, distributed debugging.

- Some limitations of the current Aldor debugger are presented in the Aldor debugger manual in Appendix A. These limitations may be improved later.

Appendix A

Aldor Debugger v0.70 Manual

A.1 Preparing a Program for Debugging

- Include debuglib in the Aldor source program
- Insert start!()$NewDebugPackage into where the Aldor debugger will be invoked
- The Aldor source program must meet the following requirements:
  - Take every `import from` expression to the top level scope (outside of any functions).
  - Include every other statement in a function.

The following Aldor source program is runnable but not suitable for debugging because it does not meet the above requirements.

```
#include "aldor"
#include "debuglib"
start!()$NewDebugPackage;

import from MachineInteger;
x: MachineInteger := 1;

main(): () == {
        import from TextWriter, Character;
        stdout << x << newline;
}
```

```
main();
```

The following Aldor source program is well prepared for debugging:

```
#include "aldor"
#include "debuglib"
start!()$NewDebugPackage;

import from MachineInteger;
import from TextWriter, Character;

main(): () == {
        x: MachineInteger := 1;
        stdout << x << newline;
}

main();
```

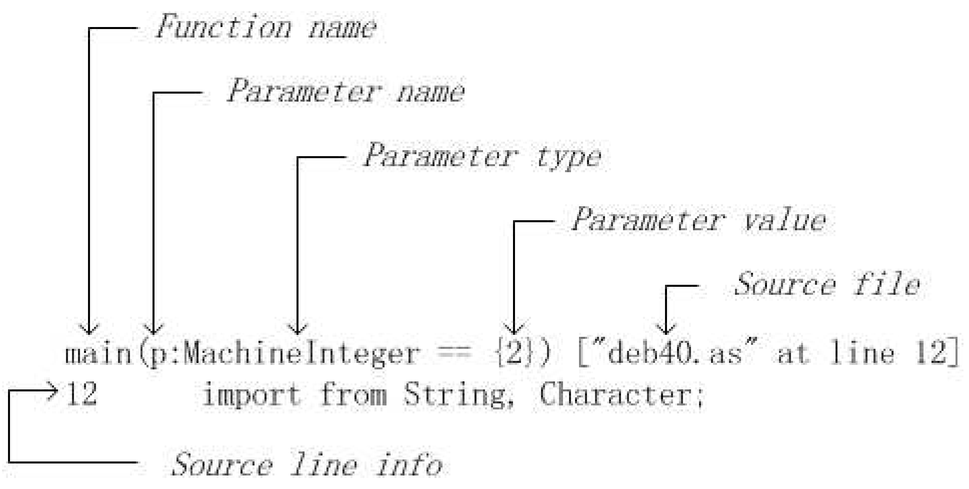
A.2 Starting the Debugger

Compile the program and run the Aldor debugger with debugger option.

Note that the debugger cannot run in interpreter mode (-Gloop and -GInterp).

$ ***aldor -wdebugger -ldebuglib -laldor -grun deb40.as***

```
--------------------------------------------------------
Aldor Runtime Debugger
v0.60 (22-May-2000), (c) NAG Ltd 1999-2000.
v0.70 (05-Dec-2003), ORCCA @ UWO.

Type "help" for more information.
--------------------------------------------------------

main(p:MachineInteger == {2}) ["deb40.as" at line 12]
12      import from String, Character;

(debug)
```



```
main(p:MachineInteger == {2}) ["deb40.as" at line 12]
12      import from String, Character;
```

Figure A.1: The meaning of the debugging prompt message

A.3 Entering Debugger Commands

The debugger shows a prompt when it is ready for the user inputs from the terminal:

(debug) ***you type here***

When you enter commands, you use the Backspace key to delete the wrong characters. When you finish entering a command, press the Enter key to submit the completed line to the debugger for processing. On a blank line, press the Enter key to re-execute the default command. The system default command is step. If you used next command after step command, then the default command is next. Help command is a very useful command:

(debug) ***help***

```
Available commands:
  break: display or insert breakpoints.
   cont: continue execution of program.
 delete: delete breakpoints.
disable: disable breakpoints.
```

```
 enable: enable breakpoints.
   halt: abort the program.
   help: access this help system.
  hints: hints and tips for using the debugger.
    int: interactive or non-interactive debugging.
   next: move to next statement (stepping over calls).
    off: turn off the debugging system and continue.
   quit: abort the program.
   step: move to next statement (stepping into calls).
   tips: hints and tips for using the debugger.
verbose: display of event details.
  where: display a stack trace.
  print: print out the value of a variable.
 update: update the value of a variable.
  shell: execute a shell command.

Use "help <command>" for more help on <command>.
(debug) help print
Syntax: print x

where x is a variable or operation whose return value is printable.
```

A.4 Continuing Execution of the Process

After you are satisfied that you understand what is going on, you can move the process forward and see what happens. The following commands you can use to do this.

next: Single step by instruction, over calls

step: Single step by instruction, into calls

The following example demonstrates stepping through lines of source code.

```
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 12]
12      import from String, Character;
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 13]
13      import from Array SI, TextWriter;
```

```
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 15]
15      local x: SI := p*p;
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 16]
16      y: SI := foo(x);
(debug) next
main(p:MachineInteger == {2}) ["deb40.as" at line 17]
17      z: STR := "ORCCA @ UWO";
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 18]
18      w: Array SI := new(2, x);
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 20]
20      hin(p1:SI): SI == {
(debug) step
hin(p1:MachineInteger == {8}) ["deb40.as" at line 21]
21              stdout << z << newline;
(debug)
ORCCA @ UWO
hin(p1:MachineInteger == {8}) ["deb40.as" at line 22]
22              stdout << w << newline;
(debug)
[4,4]
foo(f:MachineInteger == {8}) ["deb40.as" at line 29]
29 foo(f:SI): SI == {
(debug)
hin(p1:MachineInteger == {8}) ["deb40.as" at line 23]
23              foo(p1);
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 26]
26      hin(y);
(debug)
```

A.5 Setting breakpoints

The following commands are to set breakpoints, delete breakpoints, disable breakpoints and enable breakpoints:

```
(debug)break [<file>] [<line>]
```

With no arguments, this command lists the current set of breakpoints. If one argument is specified then it must be a valid line number in the current file. If two arguments are given then the first is a filename and the second a line number within that file. For example:

```
(debug) break
No breakpoints defined.
(debug) break deb40.as 23
Breakpoint set: [*] "deb40.as" at line 23 (hit 0 times).
(debug) cont
Hit breakpoint #1 hin(p1:SingleInteger == 8) ["deb40.as" at line 23]
hin(p1:SingleInteger == 8) ["deb40.as" at line 23]
23                      print << i << newline;

(debug) print i
4
(debug) break
  1) [*] "deb40.as" at line 23 (hit 1 times).
```

```
(debug)delete [<breakpoint list>]
```

This command can be used to remove a breakpoint from the set of active and inactive breakpoints. If no arguments are given, all breakpoints are deleted. For example:

```
(debug) break
  1) [*] "deb40.as" at line 10 (hit 0 times).
  2) [*] "deb40.as" at line 23 (hit 0 times).
```

```
(debug) delete 1
Deleted 1 breakpoint.
(debug) break
  2) [*] "deb40.as" at line 23 (hit 0 times).
```

```
(debug)disable [<breakpoint list>]
```

This command can be used to disable breakpoints. They will remain in place but will not trigger the debugger if hit. If no arguments are given, all breakpoints are disabled. For example:

```
(debug) break
  1) [*] "deb40.as" at line 23 (hit 0 times).
  2) [*] "deb40.as" at line 10 (hit 0 times).
(debug) disable 1
Disabled 1 breakpoint.
(debug) break
  1) [ ] "deb40.as" at line 23 (hit 0 times).
  2) [*] "deb40.as" at line 10 (hit 0 times).
```

```
(debug)enable [<breakpoint list>]
```

This command can be used to enable breakpoints that were previously disabled. If no arguments are given, all breakpoints are enabled. For example:

```
(debug) break
  1) [ ] "deb40.as" at line 23 (hit 0 times).
  2) [*] "deb40.as" at line 10 (hit 0 times).
(debug) enable 1
Enabled 1 breakpoint.
(debug) break
  1) [*] "deb40.as" at line 23 (hit 0 times).
  2) [*] "deb40.as" at line 10 (hit 0 times).
```

A.6 Examining the Paused Process

This section describes how to examine runtime stack of the paused process.

A.6.1 Looking at the Source Files

You can perform the following shell operations to look at source files:

(debug)***shell more deb40.as***

```
#include "aldor"
#include "debuglib"

start!()$NewDebugPackage;

main(): () == {
        import from SingleInteger;
        x: SingleInteger := 1;

        print << x << newline;
}

main();
(debug)
```

A.6.2 Looking at the Call Stack

You can examine the call stack of the process using the "where" command. The command will display the functions entered. For example:

(debug) ***where***

```
main(p:MachineInteger == {2}) ["deb40.as" at line 12]
12      import from String, Character;
(debug) where
1: main(p:MachineInteger == {2}) ["deb40.as" at line 12]
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 13]
13      import from Array SI, TextWriter;
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 15]
15      local x: SI := p*p;
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 16]
16      y: SI := foo(x);
```

```
(debug)
foo(f:MachineInteger == {4}) ["deb40.as" at line 29]
29 foo(f:SI): SI == {
(debug) where
2: main(p:MachineInteger == {2}) ["deb40.as" at line 16]
1: foo(f:MachineInteger == {4}) ["deb40.as" at line 29]
```

A.6.3 Looking at the Data

You can look at variables and evaluate expressions involving them. For example:

```
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 16]
16      y: SI := foo(x);
```

(debug) ***print x***

```
4
(debug)
foo(f:MachineInteger == {4}) ["deb40.as" at line 29]
29 foo(f:SI): SI == {
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 17]
17      z: STR := "ORCCA @ UWO";
(debug) print y
8
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 18]
18      w: Array SI := new(2, x);
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 20]
20      hin(p1:SI): SI == {
(debug) print z
ORCCA @ UWO
(debug) print w
[4,4]
(debug)print gcd(x,y)
4
```

```
(debug)print foo(1)
2
(debug)
```

A.6.4 Updating the Data

You can update the variables using the "update" command. The following example demonstrates updating the values of "x" and "z":

```
(debug) print x
4
```

(debug) ***update x:=100***

```
(debug) print x
100
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 20]
20      hin(p1:SI): SI == {
```

```
(debug) print z
ORCCA @ UWO
```

(debug) ***update z:="ISSAC"***

```
(debug) print z
ISSAC
(debug)
```

A.7 Exiting from the debugger

You can disable debugging by the "off" command. Then the debuggee should run to the end automatically. For example:

```
main(p:MachineInteger == {2}) ["deb40.as" at line 10] 10 local x:
MachineInteger := p*p;
```

```
(debug) off
```

```
ORCCA @ UWO
[4,4]
-bash$
```

A.8 Limitations

In addition to the limitations introduced in A.1, querying variables may lead to compilation errors in some special cases. However, the user can work around this difficulty by transforming the source code.

In the example below, querying the value of “p” in line 11 will cause a compilation error. This is because the debugger inserts the query command to the source program between line 10 and line 11, leading here to a syntax error. The solution is to modify the source program by using curly brackets. Please refer to the following source code for detail.

```
1: #include "aldor"
2: #include "debuglib"
3:
4: MI ==> MachineInteger;
5:
6: import from MI;
7: start!()$NewDebugPackage;
8:
9: main(p:MI): MI == {
10:     if (p = 2) then
11:             return 1;
12:     else
13:             return 0;
14:}
15:
16:main(2);
```

```
Solution of the limitation:
```

```
main(p:MI): MI == {
     if (p = 2) then {
             return 1;
      }
     else {
             return 0;
      }
}
```

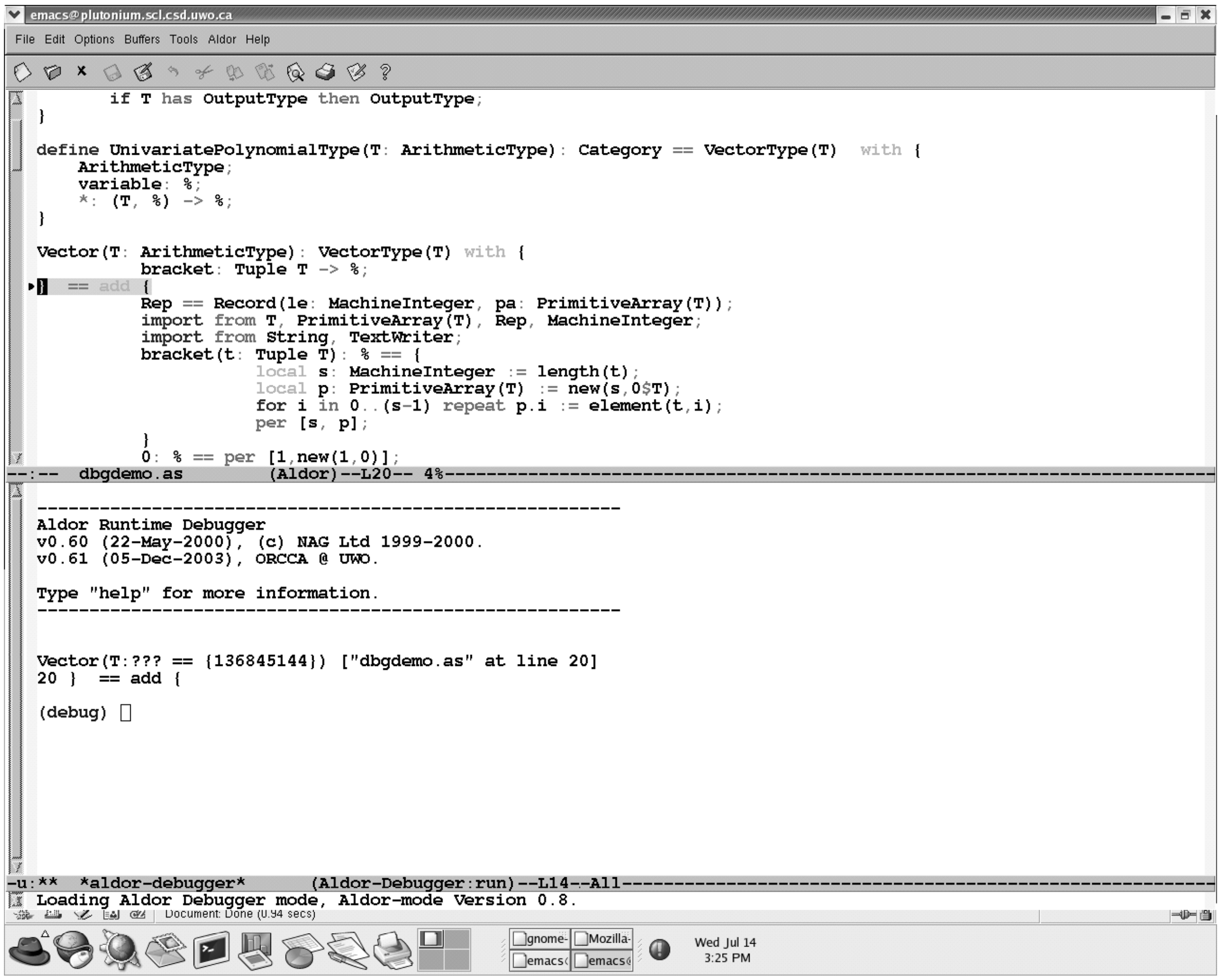
Appendix B

Aldor Emacs Mode

B.1 Demo of Aldor Emacs Mode

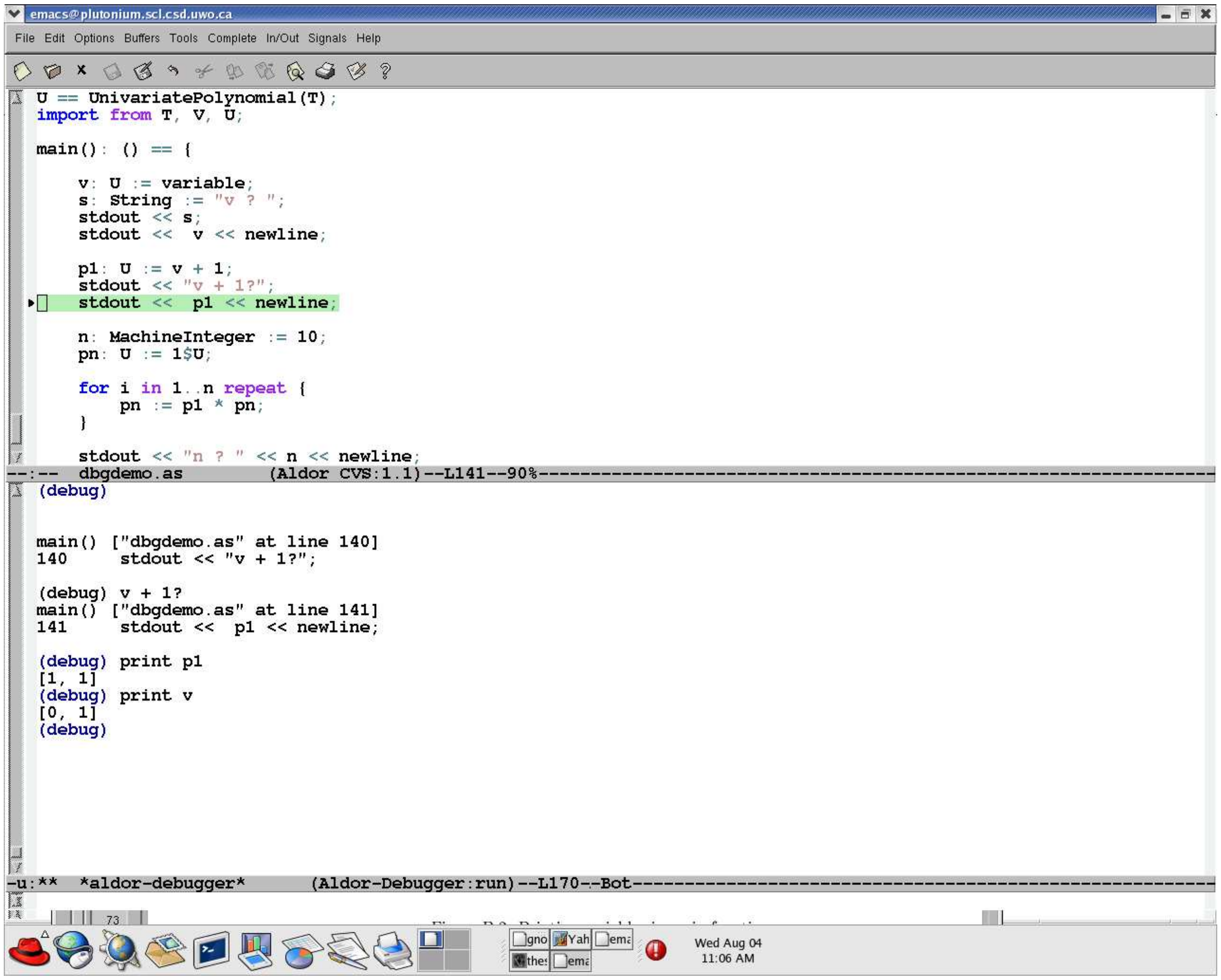
The easiest way to run the Aldor debugger is from inside Emacs. The reason for this is quite simple. Emacs will synchronize a display of the source code with the current debugger state, so that as you use the debugger to move through the code, Emacs will show you just where in the code you are.

Steve Wilson developed Aldor Emacs Mode for the Aldor source editing, the Aldor interpreter and the Aldor debugger. The following figures show how the debugger runs in Aldor Emacs Mode.


emacs@plutonium.scl.csd.uwo.ca
File Edit Options Buffers Tools Aldor Help

```
        if T has OutputType then OutputType;
}

define UnivariatePolynomialType(T: ArithmeticType): Category == VectorType(T)  with {
    ArithmeticType;
    variable: %;
    *: (T, %) -> %;
}

Vector(T: ArithmeticType): VectorType(T) with {
          bracket: Tuple T -> %;
}  == add {
          Rep == Record(le: MachineInteger, pa: PrimitiveArray(T));
          import from T, PrimitiveArray(T), Rep, MachineInteger;
          import from String, TextWriter;
          bracket(t: Tuple T): % == {
                      local s: MachineInteger := length(t);
                      local p: PrimitiveArray(T) := new(s,0$T);
                      for i in 0..(s-1) repeat p.i := element(t,i);
                      per [s, p];
          }
          0: % == per [1,new(1,0)];
--:--  dbgdemo.as        (Aldor)--L20-- 4%-------------------------------------------------------------------

---------------------------------------------------------
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.
---------------------------------------------------------


Vector(T:??? == {136845144}) ["dbgdemo.as" at line 20]
20 }  == add {

(debug) 
-u:**  *aldor-debugger*      (Aldor-Debugger:run)--L14--All-------------------------------------------------------
Loading Aldor Debugger mode, Aldor-mode Version 0.8.
```

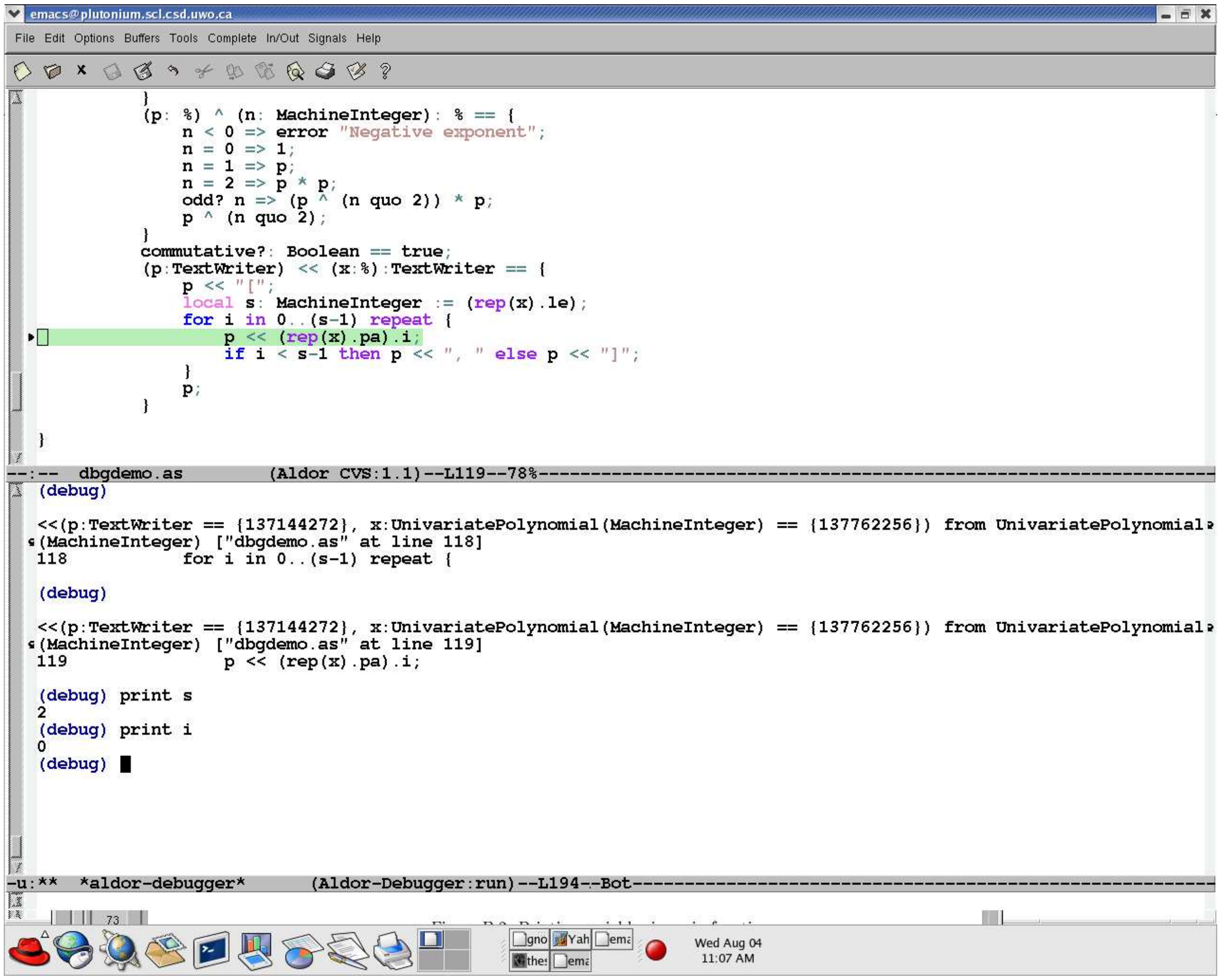
Figure B.1: Starting the Aldor Debugger in Emacs



emacs@plutonium.scl.csd.uwo.ca
File Edit Options Buffers Tools Complete In/Out Signals Help

```
U == UnivariatePolynomial(T);
import from T, V, U;

main(): () == {

    v: U := variable;
    s: String := "v ? ";
    stdout << s;
    stdout <<  v << newline;

    p1: U := v + 1;
    stdout << "v + 1?";
    stdout <<  p1 << newline;

    n: MachineInteger := 10;
    pn: U := 1$U;

    for i in 1..n repeat {
        pn := p1 * pn;
    }

    stdout << "n ? " << n << newline;
--:--  dbgdemo.as          (Aldor CVS:1.1)--L141--90%----------------------------------------------------------
(debug)


main() ["dbgdemo.as" at line 140]
140     stdout << "v + 1?";

(debug) v + 1?
main() ["dbgdemo.as" at line 141]
141     stdout <<  p1 << newline;

(debug) print p1
[1, 1]
(debug) print v
[0, 1]
(debug)
-u:**  *aldor-debugger*     (Aldor-Debugger:run)--L170--Bot------------------------------------------------
```

Figure B.2: Printing variables in main function



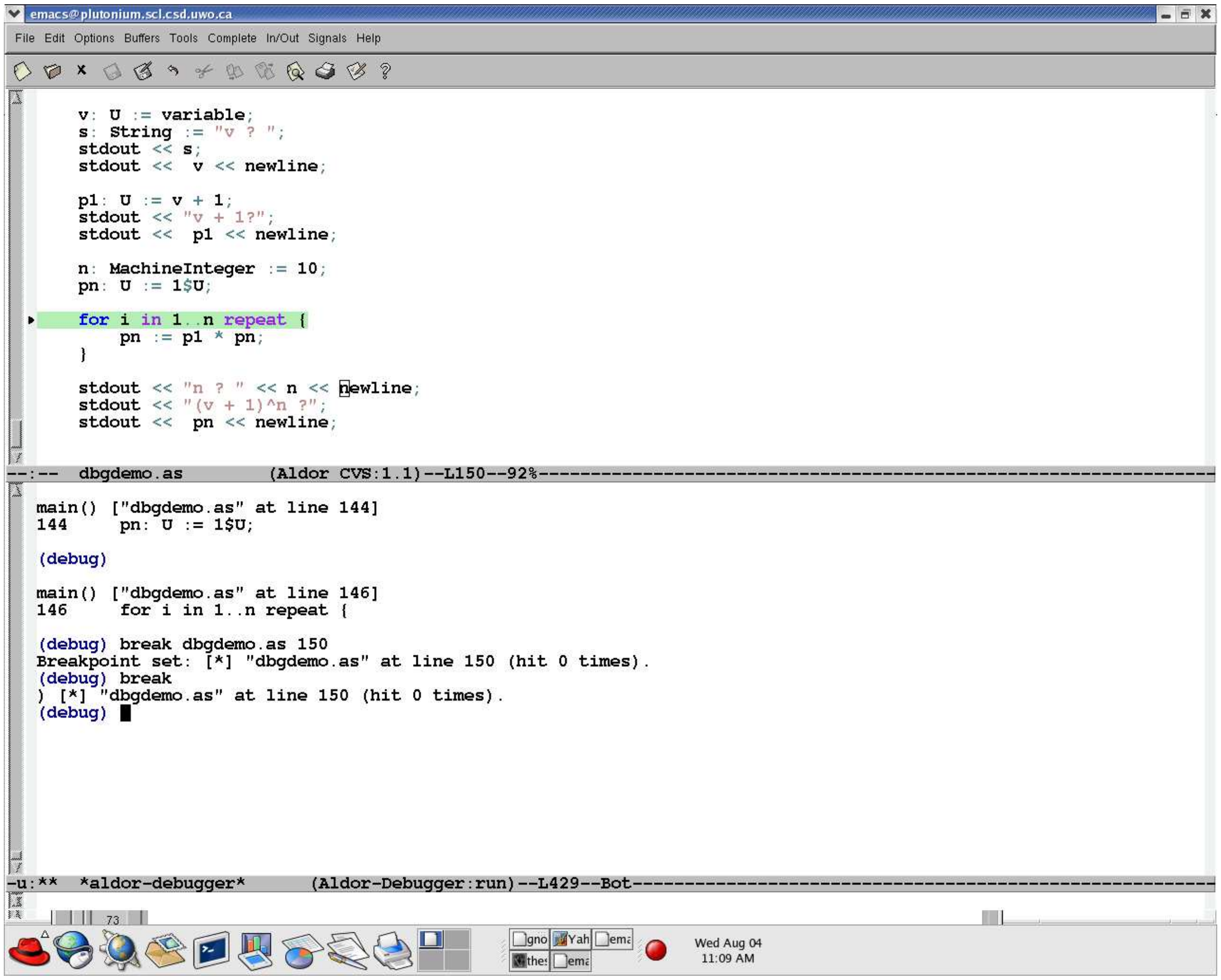
```
            }
            (p: %) ^ (n: MachineInteger): % == {
                n < 0 => error "Negative exponent";
                n = 0 => 1;
                n = 1 => p;
                n = 2 => p * p;
                odd? n => (p ^ (n quo 2)) * p;
                p ^ (n quo 2);
            }
            commutative?: Boolean == true;
            (p:TextWriter) << (x:%):TextWriter == {
                p << "[";
                local s: MachineInteger := (rep(x).le);
                for i in 0..(s-1) repeat {
                    p << (rep(x).pa).i;
                    if i < s-1 then p << ", " else p << "]";
                }
                p;
            }

}
--:--  dbgdemo.as          (Aldor CVS:1.1)--L119--78%------------------------------------
```

```
(debug)

<<(p:TextWriter == {137144272}, x:UnivariatePolynomial(MachineInteger) == {137762256}) from UnivariatePolynomial
(MachineInteger) ["dbgdemo.as" at line 118]
118           for i in 0..(s-1) repeat {

(debug)

<<(p:TextWriter == {137144272}, x:UnivariatePolynomial(MachineInteger) == {137762256}) from UnivariatePolynomial
(MachineInteger) ["dbgdemo.as" at line 119]
119               p << (rep(x).pa).i;

(debug) print s
2
(debug) print i
0
(debug)
```

```
-u:**  *aldor-debugger*       (Aldor-Debugger:run)--L194--Bot---------------------------------
```

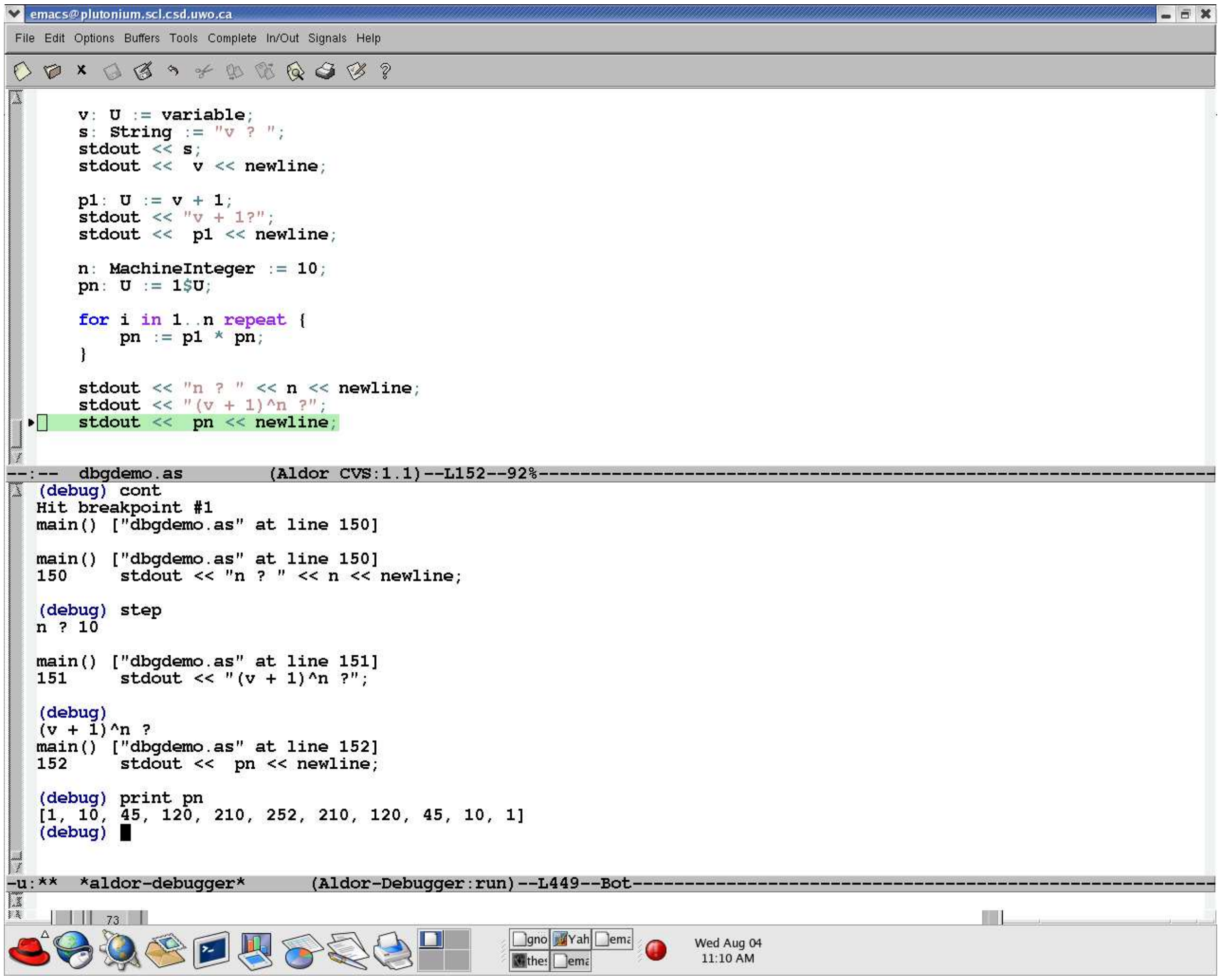
emacs@plutonium.scl.csd.uwo.ca
File Edit Options Buffers Tools Complete In/Out Signals Help
Wed Aug 04
11:07 AM

Figure B.3: Printing variables in the called domain


emacs@plutonium.scl.csd.uwo.ca
File Edit Options Buffers Tools Complete In/Out Signals Help

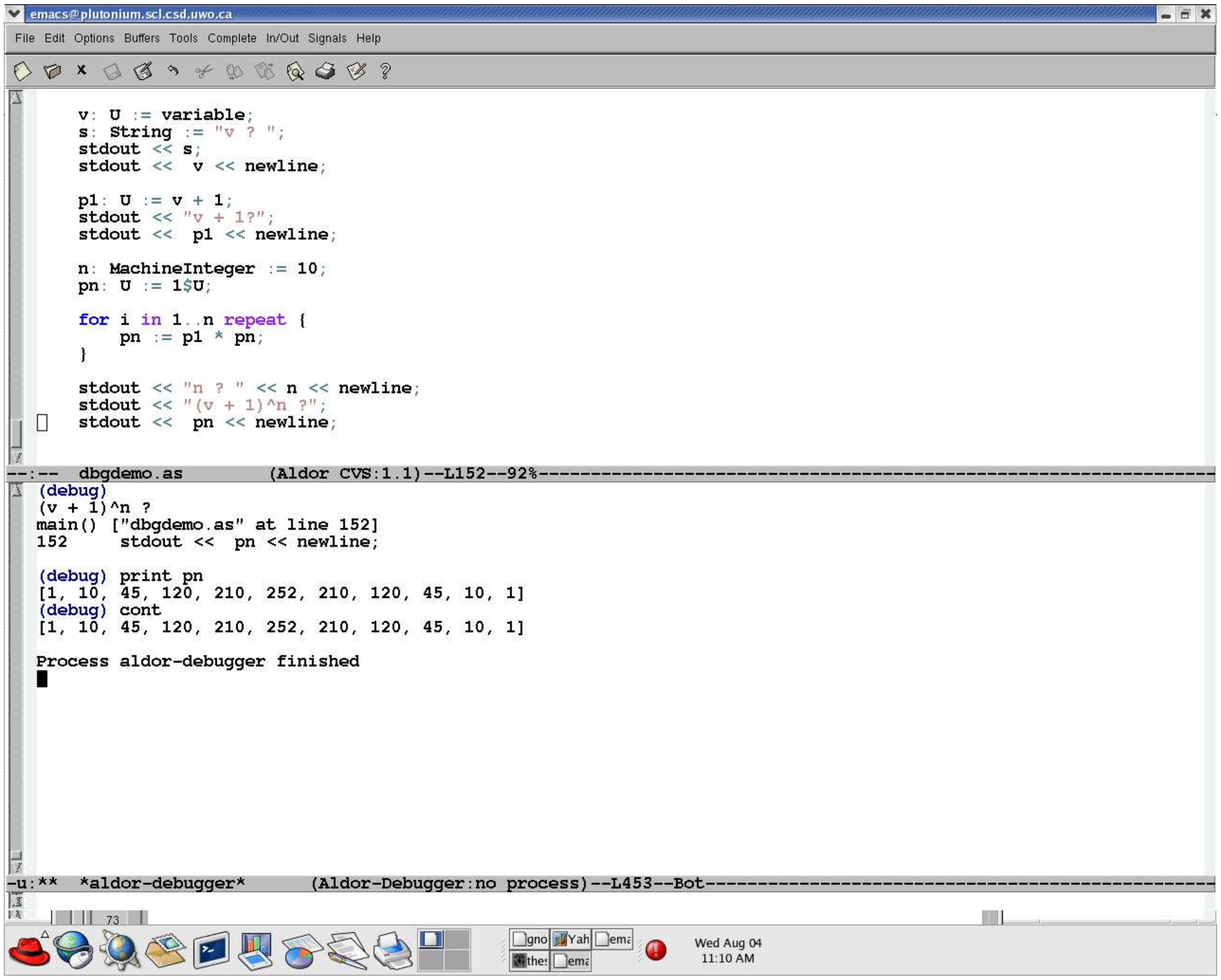
```
    v: U := variable;
    s: String := "v ? ";
    stdout << s;
    stdout <<  v << newline;

    p1: U := v + 1;
    stdout << "v + 1?";
    stdout <<  p1 << newline;

    n: MachineInteger := 10;
    pn: U := 1$U;

    for i in 1..n repeat {
        pn := p1 * pn;
    }

    stdout << "n ? " << n << newline;
    stdout << "(v + 1)^n ?";
    stdout <<  pn << newline;

--:--  dbgdemo.as          (Aldor CVS:1.1)--L150--92%----------------------------
main() ["dbgdemo.as" at line 144]
144     pn: U := 1$U;

(debug)

main() ["dbgdemo.as" at line 146]
146     for i in 1..n repeat {

(debug) break dbgdemo.as 150
Breakpoint set: [*] "dbgdemo.as" at line 150 (hit 0 times).
(debug) break
) [*] "dbgdemo.as" at line 150 (hit 0 times).
(debug)

-u:**  *aldor-debugger*      (Aldor-Debugger:run)--L429--Bot----------------------
```

Figure B.4: Setting a breakpoint


emacs@plutonium.scl.csd.uwo.ca
File Edit Options Buffers Tools Complete In/Out Signals Help

```
    v: U := variable;
    s: String := "v ? ";
    stdout << s;
    stdout <<  v << newline;

    p1: U := v + 1;
    stdout << "v + 1?";
    stdout <<  p1 << newline;

    n: MachineInteger := 10;
    pn: U := 1$U;

    for i in 1..n repeat {
        pn := p1 * pn;
    }

    stdout << "n ? " << n << newline;
    stdout << "(v + 1)^n ?";
    stdout <<  pn << newline;
--:--  dbgdemo.as        (Aldor CVS:1.1)--L152--92%---
(debug) cont
Hit breakpoint #1
main() ["dbgdemo.as" at line 150]

main() ["dbgdemo.as" at line 150]
150     stdout << "n ? " << n << newline;

(debug) step
n ? 10

main() ["dbgdemo.as" at line 151]
151     stdout << "(v + 1)^n ?";

(debug)
(v + 1)^n ?
main() ["dbgdemo.as" at line 152]
152     stdout <<  pn << newline;

(debug) print pn
[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]
(debug)
-u:**  *aldor-debugger*     (Aldor-Debugger:run)--L449--Bot---
```

Figure B.5: Hitting the breakpoint


emacs@plutonium.scl.csd.uwo.ca
File Edit Options Buffers Tools Complete In/Out Signals Help

```
    v: U := variable;
    s: String := "v ? ";
    stdout << s;
    stdout <<  v << newline;

    p1: U := v + 1;
    stdout << "v + 1?";
    stdout <<  p1 << newline;

    n: MachineInteger := 10;
    pn: U := 1$U;

    for i in 1..n repeat {
        pn := p1 * pn;
    }

    stdout << "n ? " << n << newline;
    stdout << "(v + 1)^n ?";
    stdout <<  pn << newline;
--:--  dbgdemo.as          (Aldor CVS:1.1)--L152--92%-----------
```

```
(debug)
(v + 1)^n ?
main() ["dbgdemo.as" at line 152]
152     stdout <<  pn << newline;

(debug) print pn
[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]
(debug) cont
[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]

Process aldor-debugger finished

-u:**  *aldor-debugger*      (Aldor-Debugger:no process)--L453--Bot-----
```

Figure B.6: The Aldor debugger stopped

Appendix C

Source Files for Benchmarks

The following files are source files for benchmarks in Chapter 7. These files are typical computer algebra programs. To run the benchmark, firstly modify the “n” in the main file “dbgdemo.as” to the desired value, then type “make test” in Unix shell.

C.1 Makefile

```
AOLIB=libdbg.al
OLIB=libdbg.a
AONOLIB=libnodbg.al
ONOLIB=libnodbg.a

OBJS=dbg_vectortype.o dbg_vector.o dbg_uniptype.o \
    dbg_unip.o dbgdemo.o
AOBJS=dbg_vectortype.ao dbg_vector.ao dbg_uniptype.ao \
    dbg_unip.ao dbgdemo.ao

all: nodbgdemo dbgdemo

nodbgdemo: dbg_unip.o
    @(aldor -ldbg -laldor -Fx dbgdemo.as;\
      mv dbgdemo nodbgdemo)

dbgdemo: dbg_unip.o
    aldor -wdebugger -DDEBUGGER -ldbg -ldebuglib -laldor -Fx dbgdemo.as;

dbg_vectortype.o:
    @(aldor -wdebugger -DDEBUGGER -fao -fo -laldor dbg_vectortype.as; \
      ar r $(AOLIB) dbg_vectortype.ao; \
      ar r $(OLIB) dbg_vectortype.o; \
      aldor -fao -fo -laldor dbg_vectortype.as; \
      ar r $(AONOLIB) dbg_vectortype.ao; \
      ar r $(ONOLIB) dbg_vectortype.o)
dbg_vector.o: dbg_vectortype.o
    @(aldor -wdebugger -DDEBUGGER -fao -fo -ldbg -laldor dbg_vector.as; \
      ar r $(AOLIB) dbg_vector.ao; \
      ar r $(OLIB) dbg_vector.o; \
      aldor -fao -fo -ldbg -laldor dbg_vector.as; \
      ar r $(AONOLIB) dbg_vector.ao; \
      ar r $(ONOLIB) dbg_vector.o)
dbg_uniptype.o: dbg_vector.o
    @(aldor -wdebugger -DDEBUGGER -fao -fo -ldbg -laldor dbg_uniptype.as; \
      ar r $(AOLIB) dbg_uniptype.ao; \
      ar r $(OLIB) dbg_uniptype.o; \
      aldor -fao -fo -ldbg -laldor dbg_uniptype.as; \
      ar r $(AONOLIB) dbg_uniptype.ao; \
      ar r $(ONOLIB) dbg_uniptype.o)
dbg_unip.o: dbg_uniptype.o
```

```
    @(aldor -wdebugger -DDEBUGGER -fao -fo -ldbg -laldor dbg_unip.as; \
      ar r $(AOLIB) dbg_unip.ao; \
      ar r $(OLIB) dbg_unip.o; \
      aldor -fao -fo -ldbg -laldor dbg_unip.as; \
      ar r $(AONOLIB) dbg_unip.ao; \
      ar r $(ONOLIB) dbg_unip.o)

.PHONY clean:

clean:
    rm -f *.ao *.o dbgdemo *.a *.al

test: clean install

    @(time echo "off" | dbgdemo;\
      time nodbgdemo;\
      aldor -ldbg -laldor -fao dbgdemo.as;\
      time aldor -ldbg -laldor -ginterp dbgdemo.ao)

install: all
    @(cp *.al *.a $(ALDORROOT)/lib; \
      cp dbg.as $(ALDORROOT)/include)
```

C.2 dbg.as

```
#if "DEBUGGER"
#library dbglib "libdbg.al"
#else
#library dbglib
"libnodbg.al"
#endif

import from dbglib;
inline from dbglib;
```

C.3 dbgdemo.as

```
#include "aldor"
#include "aldorio"
#include "dbg"

#if DEBUGGER
#include "debuglib"
start!()$NewDebugPackage;
#endif

T == MachineInteger;
Z == Integer;
V == Vector(Z);
U == UnivariatePolynomial(Z);
import from Z, V, U;
```

```
main(): () == {

    v: U := variable;
    s: String := "v ? ";
    stdout << s;
    stdout <<  v << newline;

    p1: U := v + 1 + 1 + 1;
    stdout << "v + 1?";
    stdout <<  p1 << newline;

    n: MachineInteger := 500; -- Modify the value of n here
    pn: U := 1$U;

    for i in 1..n repeat {
    pn := p1 * pn;
    }

    stdout << "n ? " << n << newline;
    stdout << "(v + 1)^n ?";
    stdout <<  pn << newline;
}

main();
```

C.4 dbg_vector.as

```
#include "aldor"
#include "aldorio"
#include "dbg"


Vector(T: ArithmeticType): VectorType(T) with {
      bracket: Tuple T -> %;
}  == add {
      Rep == Record(le: MachineInteger, pa: PrimitiveArray(T));
      import from T, PrimitiveArray(T), Rep, MachineInteger;
      import from String, TextWriter;
      bracket(t: Tuple T): % == {
             local s: MachineInteger := length(t);
             local p: PrimitiveArray(T) := new(s,0$T);
             for i in 0..(s-1) repeat p.i := element(t,i);
             per [s, p];
          }
      0: % == per [1,new(1,0)];
      zero?(x: %): Boolean == {
           for i in 0..(rep(x).le) repeat {
               not zero? ((rep(x).pa).i) => return false;
           }
           true;
      }
      (x:%) + (y:%):% == {
          local s: MachineInteger := (rep(x).le);
```

```
    s ~= (rep(y).le) => error "incompatible vectors";
    local p: PrimitiveArray(T) := new((rep(x).le),0$T);
    for i in 0..(s-1) repeat p.i := (rep(x).pa).i + (rep(y).pa).i;
    per [s, p];
    }
- (x: %): % == {
    local s: MachineInteger := (rep(x).le);
    local p: PrimitiveArray(T) := new((rep(x).le),0$T);
    for i in 0..(s-1) repeat p.i := - (rep(x).pa).i;
    per [s, p];
    }
(t:T) * (x:%): % == {
    local s: MachineInteger := (rep(x).le);
    local p: PrimitiveArray(T) := new(s,0$T);
    for i in 0..(s-1) repeat {
    p.i := t * (rep(x).pa).i;
    }
    per [s, p];
    }
(x: %) = (y:%): Boolean == {
    local s: MachineInteger := (rep(x).le);
    s ~= (rep(y).le) => false;
    for i in 0..(s-1) repeat {
    (rep(y).pa).i ~= (rep(x).pa).i => return false;
    }
    true;
}
```

```
      if T has OutputType then {
          (p:TextWriter) << (x:%):TextWriter == {
          p << "[";
          local s: MachineInteger := (rep(x).le);
          for i in 0..(s-1) repeat {
              p <<$(T pretend OutputType) (rep(x).pa).i;
              if i < s-1 then p << ", " else p << "]";
          }
          p;
          }
      }
}
```

C.5 dbg_unip.as

```
#include "aldor"
#include "aldorio"
#include "dbg"

UnivariatePolynomial(T: ArithmeticType): UnivariatePolynomialType(T)
== Vector(T) add {
          Rep == Record(le: MachineInteger, pa: PrimitiveArray(T));
      import from T, PrimitiveArray(T), Rep, MachineInteger;

      variable: % == {
          local p: PrimitiveArray (T) := new (2,0$T);
```

```
    p.1 := 1;
    per [2,p];
}
1: % == per [1,new(1,1$T)];
(x:%) + (y:%):% == {
    local s__x: MachineInteger := (rep(x).le);
    local s__y: MachineInteger := (rep(y).le);
    local d1: MachineInteger := min(s__x - 1, s__y - 1);
    local d2: MachineInteger := max(s__x - 1, s__y - 1);
    local p: PrimitiveArray(T) := new(d2+1,0$T);
    for i in 0..d1 repeat p.i := (rep(x).pa).i + (rep(y).pa).i;
    if (s__x < s__y) then {
    for i in (d1+1)..d2 repeat p.i := (rep(y).pa).i;
    } else {
    for i in (d1+1)..d2 repeat p.i := (rep(x).pa).i;
    }
    per [d2+1, p];
    }
(x:%) * (y:%): % == {
    local d__x: MachineInteger := (rep(x).le) - 1;
    local d__y: MachineInteger := (rep(y).le) - 1;
    local d: MachineInteger := d__x + d__y;
    local p__x: PrimitiveArray(T) := (rep(x).pa);
    local p__y: PrimitiveArray(T) := (rep(y).pa);
    local p: PrimitiveArray (T) := new (d+1,0$T);
    for i in 0..d__x repeat {
    for j in 0..d__y repeat {
```

```
                p.(i+j) := p.(i+j) + p__x.i * p__y.j;
            }
            }
            per [d+1, p];
        }
        (p: %) ^ (n: MachineInteger): % == {
            n < 0 => error "Negative exponent";
            n = 0 => 1;
            n = 1 => p;
            n = 2 => p * p;
            odd? n => (p ^ (n quo 2)) * p;
            p ^ (n quo 2);
        }
        commutative?: Boolean == true;

}
```

C.6 dbg_vectortype

```
#include "aldor"
#include "aldorio"

define VectorType(T: ArithmeticType): Category == AdditiveType with {
       *: (T, %) -> %;
       if T has OutputType then OutputType;
}
```

C.7 dbg_uniptype

```
#include "aldor"

#include "aldorio"

#include "dbg"


define UnivariatePolynomialType(T: ArithmeticType): Category ==

VectorType(T)  with {

    ArithmeticType;

    variable: %;

    *: (T, %) -> %;

}
```

REFERENCES

[1] Alfred V.Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison Wesley Longman, 1986.

[2] B. B. Chase and R. T. Hood, Selective interpretation as a technique for debugging computationally intensive programs, *ACM SIGPLAN Notices , Papers of the Symposium on Interpreters and interpretive techniques, Volume 22 Issue 7, pp. 113-124*, July 1987.

[3] D. Kevin Layer and Chris Richardson, Lisp systems in the 1990s, *Communications of the ACM, Volume 34 Issue 9, pp. 48-57*, September 1991.

[4] GDB: The GNU Project Debugger
URL: http://www.gnu.org/software/gdb/gdb.html, 2003.

[5] Gilles Muller, Barbara Moura, Fabrice Bellard and Charles Consel, Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code,*the Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS), pp. 1-20*, June 16-20, 1997, Portland, Oregon, USA.

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

[7] JDB - The Java Debugger
URL: http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html, 2003.

[8] Jinlong Cai, Marc Moreno Maza, Stephen Watt and Martin Dunstan. Debugging a High Level Language via a Unified Interpreter and Compiler Runtime Environment. *In the Proceedings of the 10th international conference on Applications of Computer Algebra (ACA'04), pp. 125-138*, July 21-23, 2004, Beaumont, Texas, U.S.A.

[9] Jonathan B. Rosenberg. *How Debuggers Work.* John Wiley & Sons, INC, 1996.

[10] Martin Dunstan, *Private Communication*, Department of Applied Computing, University of Dundee, UK, mdunstan@computing.dundee.ac.uk.

[11] Norman Ramsey and David R. Hanson, A retargetable debugger, *ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation, Volume 27 Issue 7, pp. 22-31*, July 1992.

[12] 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), pp. 6-14*, June 2003.

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

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

[15] Stephan Diehl and 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),* pp. 31-34, 2003.

[16] 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.

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

[18] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach and Robert S. Sutor. A first report on the A# compiler. *In Proceedings of the international symposium on Symbolic and algebraic computation (ISSAC 1994), pp. 25-31*, August 1994.

[19] 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.)

VITA

NAME:	Jinlong Cai
PLACE OF BIRTH:	Xiamen, Fujian, China
YEAR OF BIRTH:	1971
POST-SECONDARY EDUCATION AND DEGREES:	University of Western Ontario London, Ontario, Canada 2001-2003 B.Sc. Southwest Jiaotong University Chengdu, Sichuan, China 1989-1993 B.Eng.
HONOURS AND AWARDS:	Special University Scholarship, 2003
RELATED WORK EXPERIENCE:	Research Assistant and Teaching Assistant, 2003-2004 Ontario Research Center of Computer Algebra TA of CS026, CS350
PUBLICATIONS:	Jinlong Cai, Marc Moreno Maza, Stephen Watt and Martin Dunstan. Debugging a High Level Language via a Unified Interpreter and Compiler Runtime Environment. *In the Proceedings of the 10th international conference on Applications of Computer Algebra (ACA'04), pp. 125-138,* July 21-23, 2004, Beaumont, Texas, U.S.A.

Jinlong Cai, Marc Moreno Maza, Stephen Watt and Martin Dunstan. Debugging a High Level Language via a Unified Interpreter and Compiler Runtime Environment. *In the Proceedings of Encounters of Computer Algebra and Applications 2004 (EACA'04), pp. 119-124,* July 1-3, 2004, Santander, Cantabria, Spain