Bugloo: A Source Level Debugger for Scheme Programs Compiled into JVM Bytecode

Damien Ciabrini
INRIA Sophia Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
Damien.Ciabrini@sophia.inria.fr

Manuel Serrano
INRIA Sophia Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
Manuel.Serrano@sophia.inria.fr

ABSTRACT

This paper presents Bugloo, a source level debugger for Scheme programs. It enables debugging of programs compiled into JVM bytecode by the Bigloo compiler. It aims at being easy to use because it provides a user interface. It aims at being practical to use because it is easy to deploy and because it is efficient enough to debug large programs such as the Bigloo compiler itself.

The JVM platform provides two standard APIs for implementing debuggers and profilers: JVMDI and JVMPI. One of the motivations for providing the Bigloo compiler with a JVM back-end was to benefit from these two APIs to implement a debugger for Scheme. Bugloo is this debugger. This paper presents the pros and cons of the JVM debugging infrastructure for implementing a debugger for a functional language such as Scheme. It also presents the implementation of the main features of Bugloo.

1. INTRODUCTION

The process of debugging is the action of detecting, locating, and correcting errors - or *bugs* - in programs. A debugger is a tool which assists the programmer in this task.

There are two kinds of debuggers:

- *Static debuggers* whose goal is to determine before the execution if some operations of a program can cause errors at runtime. These debuggers rely on static analysis [5, 16].
- *Dynamic debuggers* which allow users to instrument program executions and to obtain exact information about the state and the value of the variables during the executions.

These two models are complementary because a dynamic debugger could take benefit of properties statically demonstrated by a static debugger. Our work focuses on the dynamic debugging approach and on the Scheme [20] programming language.

1.1 Motivation of Our Work

In practice, programmers hardly use debuggers. Obviously, they prefer *ad hoc* debugging techniques such as inserting `print`s in the source code. We think that the reason of this disinterest is twofold: *(i)* many debuggers are not efficient enough to debug large programs, *(ii)*, features offered by many debuggers are unsufficient for the type of bug programmers have to correct. In consequence, they often conclude that "`print`s are simpler and quicker". In other words, the overall benefits of debuggers might not be worth the time and effort programmers must supply to use them.

We believe debuggers can be made more convenient and thus more attractive by following some rules:

1. They must be easily accessible in order to ease, as much as possible, the debugging process. One way of making them accessible is to embed debuggers inside integrated programming environment.
2. They must keep performance slowdown reasonable, so that real size programs can still be debugged.
3. They have to match the specificities of the language of debugged programs. For instance for the Scheme programming language, debuggers have to deal with the automatic memory management, higher order, polymorphism, etc.

These ideas have driven the design and implementation of Bugloo, a debugger for Scheme programs written for the Bigloo compiler [13, 11].

1.2 Context of Development

Bigloo is a Scheme compiler that produces C code for efficiency, or JVM bytecode [23] for high portability. A debugger has already been designed for the C backend of Bigloo, but it has been abandoned. We think the JVM is an appealing platform that can help to make a better debugger that meets the requirements previously exposed.

Since JDK 1.3, the JVM platform provides JPDA (See `http://java.sun.com/products/jpda`), a set of standard API for debugging and profiling. In particular, it provides two API named JVMDI and JVMPI, that allow to instrument the executions of the JVMs. These API are used by common JVM debuggers such as Jdb [14] or JSWAT [15].

Our interest in JPDA is threefold:

- JPDA is standardized and portable. So, a debugger based on JPDA is portable and its behavior does not depend on JVM implementation details.
- A program debugged with JDPA can mix interpreted and compiled functions. Contrarily to native platforms where compilation takes place before the executions, with the JVM, functions are compiled at run-time by a Just In Time (JIT) compiler. Basically, the interpreter is used for functions containing breakpoints or for stepping. Other functions are compiled and optimized as in the "normal" execution mode.
- There is no need to compile source code in special debugging modes. This advantage is particularly important for libraries: it is not needed anymore to provide one version of the libraries for debugging and another one for performance. With the JVM and the JIT technology, the same version of the library serves these two roles. This makes the debugger more accessible and this reduces efforts programmers must provide to use debuggers.

We don't want to use existing JVM debuggers to debug Bigloo programs for two reasons. First, the compilation of Scheme to JVM bytecode introduces internal structures and name mangling that must be hidden to the user. Second, we want to provide custom features such as memory debugging, and features specific to the Scheme programming language.

1.3 The Source Level Debugging Model

Finding a bug with a dynamic debugger generally involves common features like controlling the execution of the debuggee (the program being debugged) and obtaining information about the execution state. Besides these features, debuggers vary from one to another. In particular, the facilities provided by the debuggers depend on their execution environment.

In interpreted environments, executing a program consists in evaluating its source code. A debugger is generally implemented as a library of user-level functions embedded into the run-time interpreter. These functions often rely on source code instrumentation to control the debuggee. Such a debugger provides breakpoints that suspend the execution when a function is entered. It uses the structure of the source program to provide single stepping on every sub-expression of a function. The user can inspect the value of the lexical environment for each step that occurred in the function.

When programs are compiled, the source code becomes unavailable. The debugger is a stand-alone program that instruments the binary program representing the debuggee. Compilation does not preserve the structure of the source code, so the debugger no longer relies on it to locate the progression of the execution. During compilation, programs are annotated with line-based or expression-based position information. This information is used at debug time to set breakpoints or to do single stepping.

Debuggers that rely on line information are called *source level debuggers*. This is the debugging support provided by the JVM platform. BUGLOO uses it in conjunction with custom features such as embeddable interpreters to limit the loss of interactivity caused by the compilation of programs.

1.4 Overview

Section 2 introduces BUGLOO and the major features it provides. Section 3 describes the JVM debugging architecture and how we implemented the debugging facilities. Section 4 exposes the mechanisms we provide to conveniently debug Scheme programs inside the JVM platform. Section 5 gives details about performances of the debugger. Section 6 compares BUGLOO with other debuggers for functional languages. Section 7 gives some perspectives of future work. Section 8 concludes the paper.

2. THE DEBUGGER BUGLOO

This section presents BUGLOO from a user perspective. It presents the user environment developed for the Emacs editor. At last, it describes advanced features such as traces, debug sessions, memory debugging capabilities and embeddable interpreters.

2.1 Core Debugging Mechanism

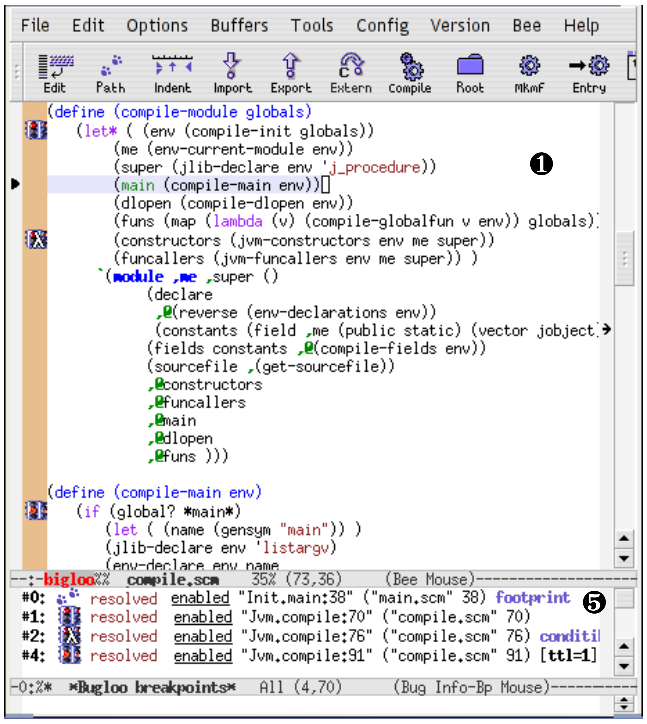
BUGLOO is a source level debugger that can debug Scheme and Java programs. In this paper, we only focus on Scheme programs, although features like memory debugging (see section 2.4) can be applied to Java programs too. BUGLOO is controled by mean of a command language with a syntax close to Scheme. It provides two basic sets of features to find a bug in a program:

- **Controlling the execution**. The user can set *breakpoints* in the source code that suspend the execution when reached. The execution is also suspended when Scheme runtime errors (such as type errors) occur. At last, he can force the suspension by sending a break signal (*i.e.* `CTRL+C`). To control more precisely the progression of the execution, he can do *single stepping*. That is, he can either trace a function "step by step", he can step inside inner function calls, or he can simply execute the whole function in one step.
- **Inspecting the state of the debuggee** when the debuggee is suspended. The user can see the function into which the execution stopped and all other pending function calls in the stack (henceforth the stack frame). He can get or set the values of the variables and introspect Scheme objects. He can also query the debugger to evaluate S-expressions defined at debug time.

Control features of BUGLOO are line based. That is, breakpoints can only be associated with *lines* in the source code (they cannot be associated with expressions). One execution step switches to the next line of the program (not to the next expression). This granularity is generally considered to be well adapted to statement oriented languages but unadapted to expression based languages such as Scheme. In practice, we have found it convenient for Scheme too, because users rarely want to step into every sub-expressions. In addition, if absolutely needed, users can still split complex expressions on multiples lines and recompile the program. To our experience, this hardly happens.

2.2 A Graphical Debugging Environment

To make BUGLOO practical to use, a complete user interface embedded in the Bee [10] Emacs programming environment has been designed. Figures 1, 2 and 3 illustrate a typical debugging session inside the environment. Every major feature of the debugger is accessible through mini-buffers, tool-bars or pop-up menus and is bound to individual views:



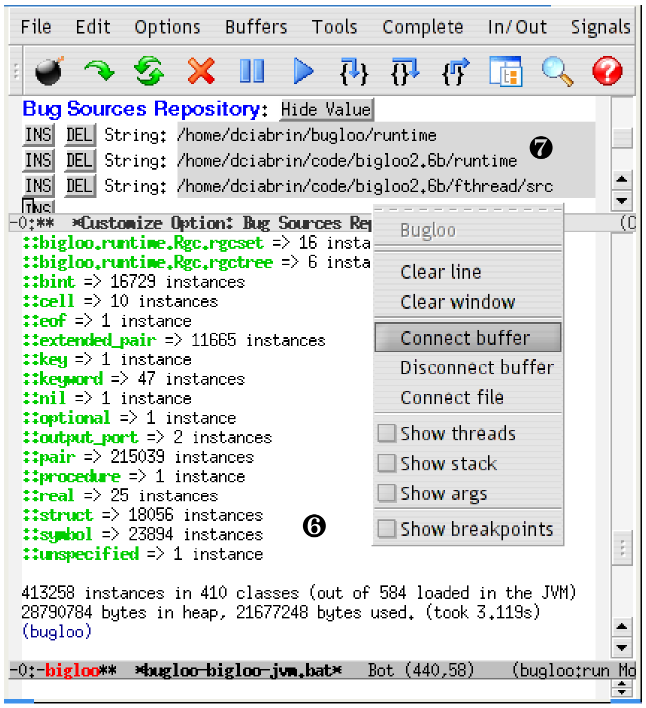
```
(define (compile-module globals)
   (let* ( (env (compile-init globals))
           (me (env-current-module env))
           (super (jlib-declare env 'j_procedure))
           (main (compile-main env))
           (dlopen (compile-dlopen env))
           (funs (map (lambda (v) (compile-globalfun v env)) globals)
           (constructors (jvm-constructors env me super))
           (funcallers (jvm-funcallers env me super)) )
        `(module ,me ,super ()
              (declare
               ,@(reverse (env-declarations env))
               (constants (field ,me (public static) (vector jobject
              (fields constants ,@(compile-fields env))
              (sourcefile ,(get-sourcefile))
              ,@constructors
              ,@funcallers
              ,@main
              ,@dlopen
              ,@funs )))

(define (compile-main env)
   (if (global? *main*)
       (let ( (name (gensym "main")) )
       (jlib-declare env 'listargv)
       (env-declare env name
--:-bigloo%%  compile.scm    35% (73,36)    (Bee Mouse)-------------------
 #0:   resolved  enabled "Init.main:38" ("main.scm" 38) footprint
 #1:   resolved  enabled "Jvm.compile:70" ("compile.scm" 70)
 #2:   resolved  enabled "Jvm.compile:76" ("compile.scm" 76) conditi
 #4:   resolved  enabled "Jvm.compile:91" ("compile.scm" 91) [ttl=1]
-0:%*  *Bugloo breakpoints*   All (4,70)     (Bug Info-Bp Mouse)----------
```

Figure 1: Source and breakpoint windows

❶ **Source windows** interact with BUGLOO when they are *connected*. A colored left margin indicates the connection. It used to set, enable, disable or kill breakpoints in the code. When the debuggee is suspended, the source file is automatically displayed and the location of the suspension is highlighted.

❷ The **threads list** displays the JVM's currently running threads, and their state at the time the debuggee was suspended. The active thread is highlighted. Selecting a thread automatically displays its execution stack frame in ❸.

❸ The **Stack frame** view displays the pending function calls of the current thread. Clicking on a frame highlights in ❶ the location where the control-flow stopped inside the associated function. It also displays the parameters and the local variables of that function invocation into another window (see ❹).



```
Bug Sources Repository: Hide Value
INS DEL String: /home/dciabrin/bugloo/runtime
INS DEL String: /home/dciabrin/code/bigloo2.6b/runtime
INS DEL String: /home/dciabrin/code/bigloo2.6b/fthread/src
INS
-0:**  *Customize Option: Bug Sources Re
::bigloo.runtime.Rgc.rgcset => 16 insta
::bigloo.runtime.Rgc.rgctree => 6 insta
::bint => 16729 instances
::cell => 10 instances
::eof => 1 instance
::extended_pair => 11665 instances
::key => 1 instance
::keyword => 47 instances
::nil => 1 instance
::optional => 1 instance
::output_port => 2 instances
::pair => 215039 instances
::procedure => 1 instance
::real => 25 instances
::struct => 18056 instances
::symbol => 23894 instances
::unspecified => 1 instance

413258 instances in 410 classes (out of 584 loaded in the JVM)
28790784 bytes in heap, 21677248 bytes used. (took 3.119s)
(bugloo)

-0:-bigloo**  *bugloo-bigloo-jvm.bat*   Bot (440,58)     (bugloo:run Mo
```

Figure 2: BUGLOO **command-line inside the GUI**

❹ **Local variables** of the current function invocation and their value. In conjunction with Emacs *tags* feature, clicking on a variable pops up its definition in a source window.

❺ **Breakpoint window** shows information for each breakpoint: set/unset, enabled/disabled, its location, its type and its time to live for temporary breakpoints.

❻ The **command-line** window allows the user to interact with BUGLOO manually and to use features that are not accessible from the UI such as the heap inspector (see section 2.4).

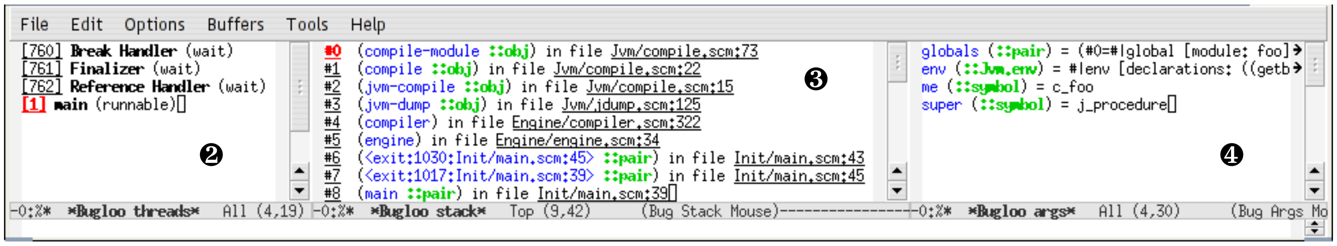
❼ A **source path repository** is used to manage a set of locations where the front-end looks for source files. This is necessary because the location of a source file is stored into the classfile as a relative pathname. The repository is customizable in the standard Emacs way.

The UI front-end makes the debugger accessible because programs can be edited and debugged in the same environment and in a simple way: the source-code window allows the user to instrument the debuggee and also to visualize the debugger's outputs.

2.3 Recording Events During the Execution

2.3.1 *Debug Session*

Throughout the execution, BUGLOO maintains the list of commands emitted during the debug session. By preserving this history of commands it is possible to save in a rudimentary way both the state of the debuggee and the current configuration of the debugger. This allows the user to stop a debug session and continue it later, or simply to replay the session many times without having to rewrite everything



```
[760] Break Handler (wait)
[761] Finalizer (wait)
[762] Reference Handler (wait)
[1] main (runnable)
-0:%*  *Bugloo threads*   All (4,19)
```

```
#0  (compile-module ::obj) in file Jvm/compile.scm:73
#1  (compile ::obj) in file Jvm/compile.scm:22
#2  (jvm-compile ::obj) in file Jvm/compile.scm:15
#3  (jvm-dump ::obj) in file Jvm/jdump.scm:125
#4  (compiler) in file Engine/compiler.scm:322
#5  (engine) in file Engine/engine.scm:34
#6  (<exit:1030:Init/main.scm:45> ::pair) in file Init/main.scm:43
#7  (<exit:1017:Init/main.scm:39> ::pair) in file Init/main.scm:45
#8  (main ::pair) in file Init/main.scm:39
-0:%*  *Bugloo stack*  Top (9,42)     (Bug Stack Mouse)----------------
```

```
globals (::pair) = (#0=#|global [module: foo]
env (::Jvm.env) = #|env [declarations: ((getb
me (::symbol) = c_foo
super (::symbol) = j_procedure
-0:%*  *Bugloo args*  All (4,30)      (Bug Args Mo
```

Figure 3: Information about a JVM thread's continuation

manually. Of course, this feature does not correctly restore the state of programs that are non deterministic, but it provides a simple mechanism that turns out to be convenient in many cases.

History management is also used as a configuration mechanism. When BUGLOO starts, it loads a command file that defines the default behavior of the debugger, such as function filtering applied when stepping (see section 4.3) or how to display Scheme objects. By customizing this file, the user can add default actions like custom filters for libraries he commonly uses.

2.3.2 *Tracing Debuggee Events*

When a bug occurred in the program, information about the current state of the debuggee may not be sufficient to understand the cause of the bug. For instance, a stack frame only shows functions that are still waiting for a result. Like debuggers for interpreted environments, BUGLOO provides a `trace` facility that can record the name of each function that was entered during a period of time defined by the user. Obviously this execution mode slows down the execution because it can exhibit a full execution path. Let us consider the following program:

```
1: (module trace (main go))
2:
3: (define (my-map f l)
4:    (if (null? l)
5:        '()
6:        (cons (f (car l)) (my-map f (cdr l)))))
7:
8: (define (go args)
9:    (my-map (lambda (x) (+ x 1)) '(1 2)))
```

When the execution reaches line 5, the stack is:

```
(bugloo) (info stack)
#0  (my-map ::procedure ::obj) in file trace.scm:5
#1  (my-map ::procedure ::obj) in file trace.scm:6
#2  (my-map ::procedure ::obj) in file trace.scm:6
#3  (go ::pair) in file trace.scm:9
```

the trace feature shows all functions that were executed since the beginning of the execution:

```
(bugloo) (trace list)
. (go ::pair) in file trace.scm:9
. (my-map ::procedure ::obj) in file trace.scm:4
. (<anonymous:1024:trace.scm:9> ::procedure ::obj) ...
. (my-map ::procedure ::obj) in file trace.scm:4
. (<anonymous:1024:trace.scm:9> ::procedure ::obj) ...
. (my-map ::procedure ::obj) in file trace.scm:4
```

Likewise, BUGLOO can watch a set of variables to obtain some data-flow information. As soon as a watched variable is accessed (read or written), the debugger adds into a trace the location in the source code where the event occurred and the value of the variable at that time. There is only one trace history for all watched variables. When the trace is dumped, events are displayed in the order into which they occurred.

2.4 Debugging of Memory Allocation

In languages with explicit memory deallocation such as C, when the programmer releases a pointer to a memory area, it may happen that this memory cannot be deallocated anymore, because it is no longer referenced anywhere in the program. This is called a *memory leak*. The principle of a garbage collector is to eliminate such leaks by automatically deallocating memory when it becomes inaccessible.

Garbage collected environments are vulnerable to another kind of memory leak, that occurs when memory still appears to be accessible for the collector while it is no longer needed by the program. BUGLOO provides a heap inspector and analysis of objects references in order to detect and to understand the causes of these memory retentions.

2.4.1 *The Heap Inspector*

When the debuggee is suspended, the debugger can give statistics about memory consumption. The user can query the number of *live* objects (*i.e.* those still accessible by the program) for each type of class loaded in the virtual machine. The result of the query can be filtered according to the type of objects by means of regular expressions. For example, the user can filter the dump to show Bigloo classes only by typing [1]:

```
(bugloo) (info heap "::")
```

[1] All Bigloo types begin with characters `::`, hence the regexp.

The heap inspector allows to roughly verify that the GC freed all the memory allocated by the program during the execution. In conjunction with the back-references and the incoming references commands described in sections 2.4.2 and 2.4.3, it helps to understand why an object is still alive.

2.4.2 *Querying a Back-References Path*

An object is *alive* for a garbage collector if it is a *root* (*i.e.* if it is contained in a class variable, in a local variable inside the stackframe, or in the JVM operand stack), or if it is referenced to by other live objects. The fact that an object remains alive after a garbage collection when it was supposed to be collected indicates that it is still accessible from at least one GC roots.

Bugloo can unveil one of these GC root which is responsible for the memory leak leak and by showing a complete chain of back-references from the target object up to this GC root. To compute a back-references path, Bugloo starts from every GC root R_n, and follows every object it can reach until it finds the target object T. The algorithm is a simple depth first search that marks an object as seen (to avoid cycles) and that recursively searches into all its references. When T is found, the computations that remain in the search stack represent the nodes of the back-references path. This feature is deterministic: the same root is discovered each time the command is called. Even if O is reachable from many GC roots, only one root can be exhibited. This limitation simplifies the implementation of this debugging feature and improves its speed. In practice, it is rare that many roots are responsible of the same leak. If needed, it is possible to discover more roots by querying a back-references path for every incoming references of O (see section 2.4.3).

To understand the process of finding a memory leak with the back-references paths, let us consider the following program, that symbolizes a mini-language compiler.

```
 1: (module leak2
 2:    (export (class ast-node
 3:               type::symbol
 4:               value::obj))
 5:    (main compile))
 6:
 7: (define *nodes-cache* (make-hashtable))
 8:
 9: (define (compile args)
10:    (let ((obj (file->ast (car args))))
11:       (set! obj (ast->il obj))
12:       (set! obj (il->bytecode obj))
13:       (bytecode->file obj (cadr args))))
```

When running the program, the mini compiler loads a file and stores it into an AST at line 10. It compiles the AST into an intermediate language at line 11. It then runs out of memory at line 12, failing to produce bytecode. It is likely that some computation done in `file->ast` or `ast->il` is responsible of the memory leak. To verify this assertion, we run the program again and set a breakpoint at line 12. At this point, we can trigger a GC and then query a heap dump to see live objects that remain in the heap.

```
(bugloo) (gc)
(bugloo) (info heap "::")
::ast-node => 29988 instances ::bint => 25982 instances
::leak2 => 11 instances       ::nil => 1 instance
::pair => 91109 instances     ::procedure => 1 instance
::struct => 1 instance        ::symbol => 800 instances
5137224 bytes used. (took 0.929s)
```

The output of the dump clearly shows that instances of `::ast-node` still resides in the heap while they are no longer used. To find out the object which is responsible for the retention, we select the first `::ast-node` object returned by the dump, and then we query a back-references path for this object:

```
(bugloo) (heap get "::ast-node" 0)
(bugloo) (backref %obj%)
#0 ::ast-node
     | field car
#1 ::pair
     | field car
#2 ::pair
     | at index 4082
#3 ::vector
     | at index 2
#4 ::vector
     | field values
#5 ::struct  ====>  module leak2 : *nodes-cache*
command took 0.743s.
```

The returned path starts from the instance and goes down to the GC root `*nodes-cache*`. It is now obvious that the problem comes from `file->ast`: it uses the hashtable defined at line 7 to cache the AST nodes, and does not clear it on exit.

2.4.3 *Querying Incoming References*

In addition to the back-references paths, Bugloo can show all the *incoming references* of a particular object O, *i.e.* all objects and roots that directly point to O. This is useful to debug programs where computations involve the sharing of many data structures. The following example illustrates how to use the feature:

```
1: (define *foo* #unspecified)
2:
3: (define (fun1 x)
4:    (set! *foo* x)
5:    (print x))
6:
7: (define (main args)
8:    (let ((dummy (cons 1 args)))
9:       (fun1 args)))
```

Suppose execution is suspended at line 5. The user can see all incoming references of the object contained in variable `x` by typing:

```
(bugloo) (incoming x)
#0  frame 0, (fun1 ::pair) in thread main => x
#1  frame 1, (main ::pair) in thread main => args
#3  object ::pair => cdr
#4  module incoming3 => *foo*
#5  class bigloo.foreign => command_line
```

The user can verify that variables `x`, `args` and `*foo*` are equal (according to the Scheme definition of equality) because they reference the same object. This feature reveals that the object is also referenced by the Bigloo runtime, in the static variable `command_line` that resides in Bigloo module `bigloo.foreign`. At last, reference `#3` seems to be variable `dummy`. Its name is not dumped, because it is not considered as a direct reference. The user has to query incoming references of object `#3`:

```
(bugloo) (incoming get 3)
(bugloo) (incoming %obj%)
#0  frame 1, (main ::pair) in thread main => dummy
```

2.5 Evaluating Code at Run-Time

Although BUGLOO is a debugger for compiled programs, it can evaluate arbitrary S-expressions added at run-time in the debuggee. Commands that use this feature are described below.

2.5.1 Embedded Scheme Interpreter

Anytime the execution of the debuggee is suspended, BUGLOO also has the ability to evaluate any S-expression written by the user. This facility is used to emulate the `print` command found in GDB, and also to spawn an interpreter in the debuggee's JVM. The environment of this interpreter is automatically extended by the environment of the debuggee at the place where it was suspended [2]. To do so, it suffices to evaluate the folowing S-expression:

```
(let <environment of the debuggee> (repl))
```

The user can then dialogue with the debuggee by using all the expressiveness of Scheme. In particular, this feature is more powerful than GDB's `print` feature because the user can define new functions on-the-fly.

2.5.2 Programmable Breakpoints

BUGLOO provides breakpoints that suspend the debuggee only "at a certain condition". In BUGLOO, a *condition* is a Scheme closure. When a conditional breakpoint is reached, the closure is evaluated, and the execution of the debuggee is suspended if the result of the evaluation is the boolean `#t`.

Conditional breakpoints can be used to emulate *ad hoc* debugging with `prints` inside the source code. Their advantage is that they can be added at run-time without needing to recompile the program and to restart the debugger.

[2]It is not exactly a *lexical* environment, because local functions might have been inlined.

In traditional debuggers, the conditional language is usually a subset (which is not clearly defined) of the source code programming language as in GDB with C. Sometimes only one global condition is allowed in the program, as in the Emacs-Lisp debugger EDebug. In BUGLOO, conditions can be as expressive as the source code of the program, since the same language is used in both cases. In particular, the user can benefit from the properties of closures to add a "memory" to a condition. The following program illustrates how one can use the *memo-conditions*. This fragment of code is an event handler listening to mouse click events:

```
1:(define (mouse-click-handler e::int)
2:   (cond
3:      ((= e 1) (print "Button 1 pressed"))
4:      ((= e 2) (print "Button 2 pressed"))
5:      (else (print "never mind"))))
```

Let us suppose the user wants to track down a bug that occurs only when a button1 event comes after a button2 event. Instead of suspending the execution on every button event, he can set a breakpoint at line 2, with the following memo-condition:

```
(let ((but2-ok #f))
   (lambda (env)
      (cond
         ((and (= (env-ref env 'e) 1) but2-ok)
           (set! but2-ok #f) #t)
         ((= (env-ref env 'e) 2) (set! but2-ok #t)))))
```

The environment of the condition is extended with the environment of the debuggee at the place where the breakpoint is located. Unlike the embedded interpreter, this cannot be done transparently. Indeed, an up-to-date environment must be passed to the condition each time the breakpoint is reached, but the condition must be evaluated only once, to always use the same closure. If the environment was hardcoded into the condition (as for the interpreter), it would be necessary to create and execute a new closure each time the breakpoint is reached. Unfortunately, this would break the "memory" property.

It is mandatory to pass the environment as a parameter to the closure. BUGLOO extends the closure by defining a local function `env-ref` that is used to access the debuggee environment through the opaque variable `env`. This approach is constraining for the user but it has a clear semantic: in order to keep the memory property of the closure, it suffices to create only one closure extended with the local function `env-ref`.

3. IMPLEMENTATION OF BUGLOO

This section describes how we used JPDA to implement BUGLOO. We present the JVM debugging architecture and the connection between the debugger and the debuggee. We describe the core mechanisms designed to instrument the debuggee, and how we used them to implement advanced debugging features.

3.1 The Debugging Model of the JVM

The core debugging architecture of the JVM is composed of two virtual machines that communicate through an abstract channel, as shown in figure 4. The debugger is run in the first JVM. The second hosts the program to debug. It is a classical JVM started in a special mode so that it can be instrumented in real-time by the debugger.

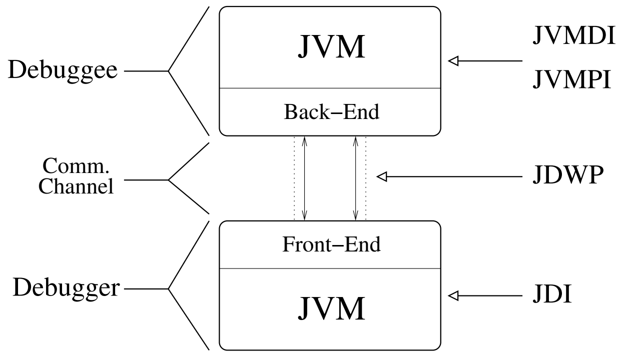


Figure 4: The JVM Debugging Architecture

The JVM Debugging Architecture is split into many layers that are accessible through specific APIs. The low-level instrumentation of the JVM is achieved through JVMDI and JVMPI, respectively the JVM Debugging Interface and the Profiling Interface [3]. These native interfaces are only accessible within the debuggee's JVM. They reify the primitive concepts of the execution (such as a type, a stack-frame or a breakpoint) and actions that occurs at run-time (class loading, memory allocation...). JDI is the high level, Java based API that allows accessing JVMDI services from the debugger JVM.

The transmission of information between the debugger and the debuggee is managed by JDWP (Java Debug Wire Protocol). This protocol is used internally by JDI to transfer data through shared memory or sockets, which allows transparent debugging of a program located on a remote site.

3.2 Managing Debuggee Events

The debugger can monitor changes in the state of the debuggee by listening to *events* that are emitted throughout the execution. Typical events include:

- Execution goes one step ahead.
- Execution stops on a breakpoint.
- A thread is created or killed.
- A thread enters or exits a function.

Some types of events can be flagged so that their emission automatically suspends the debuggee (*e.g.* the *entering function* event), while other events are only useful for information purpose. (*e.g.* the *JVM Start* event). Figure 5 describes the way BUGLOO processes these events.

[3]See `http://java.sun.com/products/jpda`

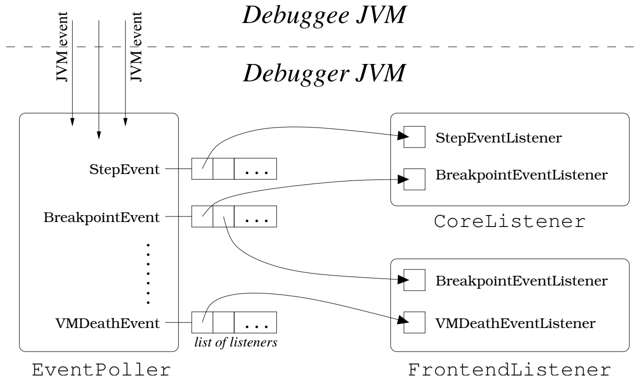


Figure 5: the event processing mechanism

When the debuggee is running, the debugger polls events emitted by the target JVM and dispatches them according to their type to registered listeners. Two listeners are used to process events. `CoreListener` receives events first. It is used to implement internals of the debugger. For example, when it receives an *entering function* event, it adds the event into the trace of functions (see section 2.3.2). `FrontendListener` is the second listener. It is roughly responsible for the user interface, *e.g.* displaying a message on program termination or when a breakpoint is reached. Managing two listeners for processing events allows to short-circuit the propagation of certain events. For instance, if a conditional breakpoint is reached but its condition is not satisfied, `CoreListener` tells the debugger to drop the event and to silently continue the execution, so that `FrontendListener` does not display "breakpoint reached" messages.

3.3 Special Instrumentation of the Debuggee

As shown in section 3.1, the debuggee is instrumented from the debugger JVM. However, some debugging features like conditional breakpoints or memory debugging need to operate directly inside the debuggee JVM. The code of these features is embedded into a *Control Class* that is loaded into the debuggee's JVM just before the debuggee program starts. Later, the debugger can query JDI to execute functions of the *Control Class*. The result of the function call is automatically transmitted back to the debugger thanks to JDWP. The usage of the *Control Class* is explained in sections 3.4 and 3.5.

3.4 Function Calls Inside the Debuggee JVM

During the debug session, BUGLOO have to call arbitrary functions inside the debuggee's JVM, for example to `write` the value of a Scheme list, or to compute a back-references path. When the debuggee is suspended, JVMDI can order a debuggee's thread to execute an arbitrary function. We describe the mechanisms implemented in BUGLOO in order to use this JVMDI facility.

3.4.1 Asynchronous Invocation

JVMDI function invocations are synchronous, which means that the debugger thread that does the invocations is blocked until the function terminates. During this period, the debugger must continue to poll events. In particular, if an event suspended the debuggee during an invocation, the debugger must not wait for the end of the invocation because this would cause a deadlock.

BUGLOO uses a dedicated thread to simulate asynchronous function invocation, *i.e.* to be able to poll events while the function is being executed. In order to invoke a function in the debuggee's JVM, the debugger performs the following actions:

- It sets the invocation flags into the dedicated thread (function to call, arguments, debuggee's thread to use) and order it to invoke the function in background.
- It starts a new polling loop to listen to events emitted during the invocation.

When the target function returns, the debugger must be notified that it can stop polling events. JVMDI doesn't provide support to interrupt polling that is in-progress. The trick is to force the debuggee to pass into a function that will emit an event to cause the polling termination.

When the function invocation returns, the dedicated thread invokes a second function named `terminatePolling()`, which is defined in the *Control Class*. BUGLOO automatically sets an invisible breakpoint in this function at debuggee load-time. When the function is entered, an event is generated. `CoreListener` recognizes this event and notifies the debugger that it can stop the polling loop. At this time, the debugger can query the dedicated thread for the result of the first call.

The debuggee could be suspended during an JVMDI invocation. In this case, the user cannot use a debugger command that would require another JVMDI invocation while the first is not terminated yet.

3.4.2 Valid Thread for Function Invocation

When the debuggee is suspended, the only thread that can be used for a JVMDI function invocation is the one that triggered the event that caused the suspension of the debuggee (*e.g.* the thread that hit a breakpoint or that raised an exception).

BUGLOO allows to suspend the debuggee when the user sends an *Interrupt Signal*[4]. It cannot suspend the debuggee manually, otherwise no thread would be eligible for function invocations. The invisible breakpoint trick presented in section 3.4.1 is used to generate an event that will cause a valid suspension.

The *Control Class* contains an empty function named `breakIntoDebugger()` that is used to suspend the debuggee. BUGLOO automatically sets an invisible breakpoint in this function at debuggee load-time. When the execution begins, the *Control Class* starts a *dedicated thread* that connects to the debugger through a socket and waits for a break signal. When the user sends an *Interrupt Signal*, the following actions occur:

[4]By typing `CRTL+C` in a conventional terminal.

- The debugger is preempted and enters its signal handler function. It notifies the debuggee of the break request by sending a message into the socket, and immediately returns. Then it resumes its job which was to poll events coming from the debuggee.
- The debuggee's dedicated thread is awaken by the data available in the socket. It jumps into the function `breakIntoDebugger()`. The invisible breakpoint causes the emission of an event that suspends the debuggee. The debugger correctly interpretes the event as a user break request.

When the user takes back control, the debuggee program is stopped inside the *Control Class* special function, and the dedicated thread that caused the suspension is a valid candidate for JVMDI function invocations. Running a dedicated thread doesn't affect the scheduling of the debuggee, since this thread is only runnable when there is data in the socket, *i.e.* when there's a break request.

3.5 Implementation of Advanced Features

The heap inspector feature described in section 2.4.1 must use JVMPI to collect statistics about the heap of the debuggee JVM. However, the debugger cannot access JVMPI directly, because JDI doesn't provide binding for this API. The *Control Class* is used to query the JVMPI heap dump and to return the result to the debugger through JDWP.

The back-references and the incoming references commands described in respectively sections 2.4.2 and 2.4.3 are executed in debuggee's JVM for speed reasons: we found that manipulating objects from the debugger through JDI mirrors leads to unacceptable performance penalties (up to 50 times slower than our current implementation). Currently, the back-references feature uses JVMPI to retrieve all the GC roots, and custom native code to follow references of JVM objects. Likewise, the incoming references feature queries a JVMPI heap dump, and scan the returned object to find out those that own a reference on a target.

Adding the read-eval-print-loop and conditional breakpoints features requires to embed in the debuggee's JVM the part of the Bigloo runtime that contains the Scheme interpreter. Indeed, the Bigloo runtime is split into many modules, to avoid loading code that is not used by programs. By default, the Scheme interpreter in not loaded. *Control Class* contains the code that orders the JVM to load and initialize the necessary Bigloo classes (in case they were not already loaded) as soon as the former features are used. Conditional breakpoints requires additional care: a closure object is created in the debuggee JVM for each conditional breakpoint, and the same object must be called each time the conditional breakpoint is reached (as explained in section 2.5.2). The *Control Class* keeps a reference on the closure object in order to prevent the GC from collecting it. When a conditional breakpoint is deleted, its closure is unreferenced so that it becomes collectable again.

4. MAPPING SCHEME TO JVM BYTECODE

Bigloo generates various intermediate structures to compile Scheme into JVM bytecode. At debug-time, the debugger must correctly interpret the generated JVM structures with regards to the original Scheme program, while hiding as much as possible the encodings introduced by the compiler. This section describes the mechanisms that we have implemented so far in both the compiler and the debugger to conveniently debug Scheme programs.

4.1 Classfile Debug Information

A Bigloo program is compiled into a JVM classfile. Each function of the source program is compiled into a JVM method, and two kinds of debugging information are generated for the methods:

- *Lines information.* Each executable line of the function corresponds to a range of bytecodes (*i.e.* a start index plus a length) in the method. The correspondance is stored in the method descriptor.
- *Local variables.* The name of local variables and their life range (*i.e.* a start line and an end line) is stored in the method descriptor. Bigloo introduces many intermediate local variables during the compilation stages, but takes care of not including them in the debug information.

At last, the name of the source file is stored in the classfile. Prior to JDK 1.4, there was no way to store the source path name inside the classfile. Current classfile specifications provide extended debugging information. The classfile can embed multiple debugging views called *strata*. Each stratum provides its own file name, its own path name and its own line information. Moreover, the `Java` stratum always exists and correspond to the original debugging information.

Bigloo uses the extended format to store debugging information into the classfile. It generates a `Bigloo` stratum that contains the source filename and the correct path name. Lines and local variables information is stored into the `Java` stratum. This extension only adds 50 bytes of data (plus source path) to a classfile and is compatible with older JVMs.

4.2 Custom Displaying Functions

Scheme identifiers can contain characters that are not allowed in JVM identifiers. The compiler must mangle such identifiers so that they become valid JVM identifiers. BUGLOO provides Java and Bigloo *demanglers* to display names, types and signatures. The name of a variable is automatically demangled if it comes from Scheme code. Likewise, the signature of functions are automatically displayed in a Scheme or Java fashion. Each demangler provides various *displayers* to format the value of an object.

```
(bugloo) (displayer list)
bigloo:
    closure, write-circle, write, display, default
java:
    tostring, default
```

When the user queries information about a variable, the right demangler automatically prints the name and the type of the variable. The current displayer for that demangler formats the value of the variable. This mechanism allows the user to print its objects with standard Scheme functions such as `display` or `write`.

Displayers can execute custom code to format objects. For instance, `closure` is a displayer that can display the name of the function associated with a closure. Let us consider the following example:

```
1:(define (foo f)
2:   (print (procedure? f)))
3:
4:(define (main args)
5:   (foo main))
```

Suppose that the execution stops line 2. Querying the value of variable `f` with displayer `write` returns the opaque value:

```
f (::procedure) = #<procedure:>
```

while displayer `closure` returns a more useful output:

```
f (::procedure) = procedure (main ::pair)
                  in file closure2.scm:5
```

JVM methods generated by Bigloo are produced in a specific order in the classfile, which allows the displayer `closure` to associate a function name to a closure without using any debugging information. This means that this displayer also works on classes that do not include debugging symbols (for example the Bigloo runtime).

4.3 Filtering Single Stepping

During single stepping, the user may encounter internal functions that reveal the implementation of the Scheme-to-JVM mapping. In order to prevent the user from stopping into such functions, BUGLOO provides *filters* that change the behavior of single steps that occur in a certain context. A context is a POSIX regular expression representing a fully qualified location in the program (*i.e.* class plus method). When a thread is stepping into a function that matches a context, two kinds of actions can occur: the execution can continue until the function is popped from the stack frame, or until another function is pushed into the stack frame. The following filters uses both possibilities:

```
(filter ext add ("bigloo\\..*\\.<clinit>" . out))
(filter ext add ("bigloo\\..*\\.<init>" . out))
(filter ext add ("\\.funcall[0-4]\\(" . next))
```

The first two filters illustrate the first kind of action. They prevent the user from stepping inside JVM constructors defined in the Bigloo runtime, as they only represent implementation internals. For example, when the user steps into a line that allocates a pair, its constructor is entirely executed and the step terminates one line forward.

The last filter it is a bit more complex. It uses the second kind of action to mask the implementation of higher-order call. As stated in [11], a `procedure` object is allocated for each closure defined in a Bigloo module. A higher-order call is implemented as a call to a dispatcher function defined in the `procedure` class, and chosen with respect to the callee arity (hence the names `funcall0` to `funcall4`). The dispatcher function uses an index contained in the `procedure` object to call the correct Scheme function. When the user steps into such a dispatcher, the last filter forces the execution to continue until the beginning of the real Scheme function.

This filtering mechanism works well to prevent steps from entering internal functions generated by the compiler. However, it is not flexible enough to mask undesirable line information generated by the macro expansion of some Bigloo primitives. The problem is illustrated below:

```
1:(try
2: (begin
3:    (print (%reverse (list 1 2 3 4 5 6)))
4:    (print (fac 5)))
5: (lambda (escape proc mes obj) (escape #f)))
```

The `try` primitive is a macro that allows to execute the error handler at line 5 if an error occurs inside the `begin`. When the user single steps this program, the control flow strangely jumps from line 1 to line 5, and then jumps back to line 2. There is no simple way to avoid this behavior except, maybe, by modifying the compiler itself.

5. PERFORMANCE OF THE DEBUGGER

Many source level debuggers for Scheme become impractical for programs of many thousands of lines. We think that it is important that programmers can use BUGLOO to debug large programs such as the Bigloo compiler itself (about 130000 lines).

We have run a serie of tests without using any debugging features, in order to measure the performance penalties imposed by BUGLOO. FIB is a very small program that computes Fibonacci numbers. QSORT is a 100 lines program that tests array and fixnum arithmetic. PEVAL is a partial evaluator that uses a lot of nested function. It is allocation intensive. CGC is a compiler for a C like language that produces MIPS assembly code. The benchmarks were run on an Athlon XP 1900+, 256Mb RAM with Linux 2.4.19 and Sun JDK 1.4.1 HotSpot client VM. Results are reported in Table 1. They show that memory consumption is not affected by the instrumentation, and that execution time is always slightly higher when using BUGLOO. In *empty* test, we measured that the debugger's JVM takes $\mathcal{T}_0$ =0.98s to load and exit immediately. We consider that

$$\frac{\mathsf{TIME}_{\textsc{Bugloo}} - \mathcal{T}_0}{\mathsf{TIME}_{\textsc{Jvm}}}$$

is a reasonable approximation of the real execution overhead caused by BUGLOO. This reveals programs are from 1.5% to 6% slower when debugged. The same tests compiled into C and debugged with GDB leads to comparable slowdowns (between 0.4% and 3%).

Benchmarks confirm that the JIT stays enabled when debugging. This allows JVM debuggers to compete with native debuggers like GDB in term of performance. Few features like single stepping are still executed in interpreter mode by current JVMs, which can locally leads to slowdowns (*e.g.* stepping into a Swing program that builds a GUI).

Performance of advanced features are quite good. For instance, we debugged the Bigloo compiler and suspended its execution in a pass that contained 396011 live objects in the heap (more than 20 Mb). It took 4.516s to compute the back-references path of an arbitrary object in the heap (the path was 546 objects long).

In conclusion, we found that a large program such as the Bigloo compiler can be debugged in practice, so we consider that the overhead caused by BUGLOO is acceptable.

6. RELATED WORK

KBDB [12] is a debugger for programs compiled with the C backend of Bigloo. It relies on the C debugger GDB [21] for instrumenting the execution, and provides features for memory debugging, which inspired our back references command. Compared to BUGLOO, KBDB can reveal the function where an objects have been allocated, and cannot exhibit sharing properties presented in section 2.4.3. Unlike BUGLOO, memory features of KBDB don't do computations *on-demand*, thus they enlarges heap size and slow down executions. Experience has shown that KBDB suffers two problems. First, it is painful to maintain because the communication with GDB is done via a specific language that changes across GDB versions. Second, to use memory debugging features, code must be compiled in a special debug mode (which adds two fields to Bigloo objects and uses a special debug GC) which is incompatible with normal compiled code. This make these features constraining to use.

JSWAT [15] is a free graphical Java debugger that also uses the JPDA framework to instrument programs. It provides an embedded source file viewer for Java programs, and a command line language to instrument programs manually. It uses more JVMDI features than BUGLOO. In particular it provides debugging support for monitors in concurrent programs. However, it does not use custom instrumentation as shown in section 2.4, so it cannot provide features like memory debugging.

Scheme implementations such as Chez Scheme [18], Scheme 48 [19] and MIT Scheme [3] provide common and simple debugging features inside their interpreter. They can trace function entry and suspend the execution when a function is entered. Stack frame inspection does not exhibit the runtime stack (as in GDB) but sub-expressions that have been evaluated so far, and the value of the environment in each sub-expression. When a function is single stepped, each of its sub-expressions is stepped, which tends to be too verbose to be really meaningful.

PSD [17] and EDebug [6] provide debugging features close to GDB (source level debugging, line based operations) by doing instrumentation at source code level. In general, this

	(empty)		FIB		QSORT		PEVAL		CGC	
	TIME	_	TIME	MEM	TIME	MEM	TIME	MEM	TIME	MEM
JVM	_	_	6.52s	322 Kb	9.95s	36.35 Mb	17.49s	1344.90 Mb	27.33s	30.42 Mb
BUGLOO	$\mathcal{T}_0$ =0.98s	_	7.68s	322 Kb	11.08s	36.59 Mb	19.56s	1345.19 Mb	30.01s	31.05 Mb
NATIVE	_	_	5.85s	_	7.60s	_	16.88s	_	12.82s	_
GDB	$\mathcal{T}_1 \approx$0s	_	6.01s	_	7.63s	_	17.28s	_	12.91s	_

Table 1: Performance hits caused by Bugloo without using any debugging features

transformational approach induces problems which do not occur in BUGLOO. First, code whose source is not available (*e.g.* a byte compiled library such as the Scheme runtime) cannot be debugged. In comparison, A JVM class only need to embed line and variable information to be debugged correctly. Even without debugging symbols, it is still possible to introspect a class or to see function names inside a stack frame. Second, the debuggers must instrument all forms of the language while preserving the original semantics of the program, which can be a tremendous task. For example, tail recursive functions become deep-recursive when single stepping with PSD. BUGLOO does not have such problems because line information is generated at compile time by Bigloo itself.

DrScheme [22] is a complete programming environment for Scheme that provides two kinds of debugging tools. The first one, Mr Spidey [4], is a static debugger that relies on a set-based analysis [16] to detect at compile-time type and arity errors. The second is an algebraic stepper [9] that evaluates functions by successive reductions of the code. The latter tool is only useful to understand the behavior of small programs. On the other hand, Mr Spidey can obtain information or properties on programs that cannot be exhibited with BUGLOO.

Allegro CL is a Common Lisp [2] implementation that includes a complete source level debugger. This debugger does complex stack frame inspection. It is able to infer *ghost frames* which are functions that disappeared from the stack-frame because they were exited by a tail-call. the JVM bytecode does not provide tail calls, so such information would be hard to implement in BUGLOO.

Objective Caml [7] provides a source level debugger for programs compiled into its bytecode format. It has the ability to do *time travel* by stepping back through the execution history. This feature is implemented by saving the state of the VM from time to time with a `fork()` system call. Clearly, we cannot imitate such a very low level instrumentation technique, because the JVM platform don't provide any support for controlling its internals.

7. FUTURE WORK

We plan to improve features described in section 4.2. For example, we could add a specific displayer for continuations, to exhibit information such as the captured runtime stack.

So far, BUGLOO cannot hide some implementation details. This is typically the case with anonymous functions or macros expanded forms. For example, the Bigloo `bind-exit`/`unwind-protect` macro expands into an anonymous function that executes user code inside a JVM `try-catch` block. The line information generated by Bigloo is correct, but the anonymous function is visible inside the stack frame, which introduces noise in the user vision of the execution. We should provide a way to filter functions in the stack frame.

The JVM operates on bytecode level to provide breakpoints and single stepping. To interpret S-expressions, the `eval` function uses its own intermediate bytecode format. Thus, It cannot be debugged directly because it is not JVM bytecode. We should modify the Bigloo interpreter and provide additional filters (like those in section 4.3) so that the user can debug compiled or interpreted code in a transparent way.

The extended classfile debugging information presented in section 4.1 allows strata to embed their own line information. We could add a stratum with S-expressions position information, *e.g.* a character position of the beginning and the end of the S-expression. This would allow both line oriented and S-expression oriented single stepping.

Some features like trace recording or object inspection are only accessible through the command-line, which is not intuitive enough. A real graphical environment *à la* DDD [1] could improve the effectiveness of the debugger.

In a near future, we plan to provide debugging support for Bigloo FairThreads [8], which are cooperative threads that communicate through broadcast events and run into a deterministic scheduler.

8. CONCLUSION

We have developed BUGLOO, a source level debugger for Scheme programs compiled with Bigloo into JVM bytecode. It provides basic features found in classical C or Java debugger, and also extended features like traces, debug sessions and memory debugging. It limits the loss of interactivity due to the compilation of programs by providing programmable breakpoints and interpreters embeddable into the debuggee. To make BUGLOO easily accessible, we have developed a complete user interface embedded into the Bee Emacs programming environment.

We found that the JVM platform offers advantages over existing debugging platforms. It offers clean APIs to instrument programs and to easily implement custom debugging features. Above all, thanks to the JIT compiler, the same version of a library can be used for both debugging and performance. This eases the use of the debugger for programmers. Tests have shown that BUGLOO is usable in practice to debug large Scheme programs such as the Bigloo compiler itself, because the JIT allows to keep good performance when programs are debugged. BUGLOO is available at `http://www-sop.inria.fr/mimosa/fp/Bugloo`.

ACKNOWLEDGMENTS

Many thanks to Bernard Serpette, Frédéric Boussinot, Erick Gallesio and Stef Epardaud for their helpful feedbacks on this work.

9. REFERENCES

[1] Andreas Zeller and Dorothea Lutkehaus – **DDD - A Free Graphical Front-End for UNIX Debuggers** – *SIGPLAN Notices*, 31(1), 1996, pp. 22-27.

[2] Bobrow, D. and DeMichiel, L. and Gabriel, R. and Keene, S. and Kiczales, G. and Moon, D. – **Common lisp object system specification** – *special issue*, *Sigplan Notices* (*23*), Sep, 1988.

[3] Chris Hanson – **MIT Scheme Reference Manual** – AITR-1281, 1991, pp. 248.

[4] Cormac Flanagan and Matthew Flatt and Shriram Krishnamurthi and Stephanie Weirich and Matthias Felleisen – **Catching Bugs in the Web of Program Invariants** – *ACM SIG Notices*, 31(5), 1996, pp. 23–32.

[5] Cousot, P. and Cousot, R. – **Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints** – *Symposium on Principles of Programming Languages*, Los Angeles, CA, USA, Jan, 1977, pp. 238–252.

[6] Daniel LaLiberte – **Edebug, Emacs Lisp debugger** – `included in the GNU Emacs Lisp Reference Manual`1994.

[7] Emmanuel Chailloux and Pascal Manoury and Bruno Pagano – **Developpement d'applications avec Objective CAML** – *O'Reilly France*, 2000, pp. 700.

[8] Frédéric Boussinot – **Java Fair Threads** – RR-4139, *INRIA*, 2001.

[9] John Clements and Matthew Flatt and Matthias Felleisen – **Modeling an Algebraic Stepper** – *Lecture Notes in Computer Science*, 20282001, pp. 320.

[10] Manuel Serrano – **Bee: an integrated development environment for the Scheme programming language** – *Journal of Functional Programming*, 10(4), 2000, pp. 353-395.

[11] Manuel Serrano and Bernard Serpette – **Compiling Scheme to JVM bytecode: a Performance Study** – *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, October, 2002, pp. 259–270.

[12] Manuel Serrano and Hans-J. Boehm – **Understanding memory allocation of scheme programs** – *ACM SIG Notices*, 35(9), 2000, pp. 245–256.

[13] Manuel Serrano and Pierre Weis – **Bigloo: A Portable and Optimizing Compiler for Strict Functional Languages** – *Static Analysis Symposium*, 1995, pp. 366-381.

[14] Mike Cohn and Bryan Morgan and Michael Morrison and Michael T. Nygard and Dan Joshi and Tom Trinko – **Java Developer's Reference** – *Sams*, 1997, pp. 1258pp.

[15] Nathan Fiedler – **JSWAT, The Java Debugger** – `http://www.bluemarsh.com/java/jswat/index.html`1999.

[16] Nevin Heintze and Joxan Jaffar – **Set Constraints and Set-Based Analysis** – *Principles and Practice of Constraint Programming*, 1994, pp. 281-298.

[17] P. Kellomaki – **Psd — a portable scheme debugger** – `citeseer.nj.nec.com/kellomaki94psd.html`1995.

[18] R. Kent Dybvig – **Chez Scheme User's Guide** – *Cadence Research Systems*, 1998.

[19] Richard A. Kelsey and Jonathan A. Rees – **A Tractable Scheme Implementation** – *Lisp and Symbolic Computation*, 7(4), 1994, pp. 315–335.

[20] Richard Kelsey and William Clinger and Jonathan Rees (Editors) – **Revised5 Report on the Algorithmic Language Scheme** – *ACM SIGPLAN Notices*, 33(9), 1998, pp. 26–76.

[21] Richard Stallman and Roland H. Pesch – **Debugging with GDB: the GNU source-level debugger** – 1993.

[22] Robert Bruce Findler and John Clements and Cormac Flanagan, Matthew Flatt and Shriram Krishnamurthi and Paul Steckler and Matthias Felleisen – **DrScheme: A Progamming Environment for Scheme** – *Journal of Functional Programming*, 12(2), March, 2002, pp. 159–182.

[23] Tim Lindholm and Frank Yellin – **The Java Virtual Machine Specification** – *Ad*, Reading, MA, USA, 1997, pp. xvi + 475.