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

Damien Ciabrini  Manuel Serrano
firstname.lastname@sophia.inria.fr

INRIA Sophia Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
Outline

- Introduction
- The Bugloo debugger
- Custom debugging features
- JVM debugging architecture
- Mapping Scheme to JVM
- Performances
- Conclusion
Introduction

• Debugging programs:
  • to detect, to locate and to correct errors

• Two kinds of debuggers:
  • static debuggers
  • dynamic debuggers
Motivation of our Work

- Programmers hardly use debuggers:
  - sometimes not efficient enough
  - not adapted to correct certain bugs
  - "prints are simpler and quicker"

- How to make debuggers more attractive?
Motivation of our Work

- Programmers hardly use debuggers:
  - sometimes not efficient enough
  - not adapted to correct certain bugs
  - "prints are simpler and quicker"

- How to make debuggers more attractive?
  - easily accessible from the IDE
  - acceptable performance slowdown
  - to deal with the language specificities
Work with the Bigloo Scheme compiler:

- Scheme $\Rightarrow$ C. Already has a debugger.
- Scheme $\Rightarrow$ JVM bytecode. JVM provides JPDA.

JVM Platform Debugging Architecture (JPDA):

- A set of APIs to make debugger and profilers
- Standardized $\Rightarrow$ portability across JVMs
- JIT can limit performances slowdown
- Same classfile for normal or debug executions
The Bugloo Debugger

• Bugloo is a *source level debugger*:  
  • Operates on compiled Bigloo programs
• Basic instrumentation of the *debuggee*:  
  • breakpoints, stack and variables inspection
• Advanced debugging features  
  • Traces, memo-conditions, memory debugging
• Controlled by a command language
• Integrated into the Emacs editor
Emacs Environment (1/2)

- Source buffers:
Emacs Environment (1/2)

- Source buffers:
- Breakpoints list
Emacs Environment (1/2)

- Source buffers:
  - Breakpoints list

- Execution State:
  - Debuggee inspection
Emacs Environment (1/2)

- Source buffers:
  - Breakpoints list
- Execution State:
  - Debuggee inspection
Emacs Environment (1/2)

- Source buffers:
  - Breakpoints list

- Execution State:
  - Debuggee inspection
Emacs Environment (1/2)

- Source buffers:
  - Breakpoints list
- Execution State:
  - Debuggee inspection
- Hyperlink configuration
Source path repository:
Emacs Environment (2/2)

Source path repository:
- Where to look for sources
- Customization à la Emacs

[Image of Bugloo debugger window with source code and heap statistics]
Source path repository:

- Where to look for sources
- Customization \(à \) la Emacs

Command line buffer:
Source path repository:
- Where to look for sources
- Customization à la Emacs

Command line buffer:
- Manual interactions
- For advanced features
Custom Debug Features

- Events recording
  - Trace of function calls
- Eval code at run-time
  - Memo-breakpoints
- Memory Debugging
  - Back references paths
Event Recording

- All events that occur during a debug session:
  - History of user commands
    - Simple *replay* mechanism
  - Traces of debuggee events
    - Variable accesses, functions calls
Event Recording - example

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

Bugloo: A Source Level Debugger for Scheme Programs Compiled into JVM Bytecode – p.10/20
Event Recording - example

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

(bugloo) (info stack)
#0 (my-map ::procedure ::obj) in file trace.scm:6
#1 (my-map ::procedure ::obj) in file trace.scm:7
#2 (my-map ::procedure ::obj) in file trace.scm:7
#3 (go ::pair) in file trace.scm:2
Event Recording - example

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

(bugloo) (trace list)
. (go ::pair) in file trace.scm:2
. (my-map ::procedure ::obj) in file trace.scm:4
. (<lambda:2> ::obj) in file trace.scm:2
. (my-map ::procedure ::obj) in file trace.scm:4
. (<lambda:2> ::obj) in file trace.scm:2
. (my-map ::procedure ::obj) in file trace.scm:4

Bugloo: A Source Level Debugger for Scheme Programs Compiled into JVM Bytecode – p.10/20
Eval code at runtime

- Bugloo uses the built-in Scheme interpreter
  - Debugger: eval arbitrary S-exp
  - Debuggee: conditional breakpoints

- In Bugloo, a condition is a lambda
  - Various usage:
    - insert code without recompiling
    - a closure is a memo-condition
Eval code at runtime - example

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"))))
Eval code at runtime - example

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 ((but2-ok #f))
  (lambda (env)
    (cond
      ((and (= (dbg env 'e) 1) but2-ok)
       (set! but2-ok #f) #t)
      ((= (dbg env 'e) 2)
       (set! but2-ok #t)))))
Memory Debugging

• Scheme provides a GC
  • Not sufficient to avoid memory leaks!

• Services provided by Bugloo:
  • Heap inspector
    • To monitor memory consumption
  • Incoming references
    • To exhibit sharing properties
  • Back references path
    • Which GC root is responsible of a leak
Memory Debugging - example

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))))

Bugloo: A Source Level Debugger for Scheme Programs Compiled into JVM Bytecode – p.14/20
Memory Debugging - example

(bugloo) (gc)
(bugloo) (gc)

(bugloo) (info heap "::")
::ast-node => 29988 instances
::leak2 => 11 instances
::pair => 91109 instances
::struct => 1 instance
::bint => 25982 instances
::nil => 1 instance
::procedure => 1 instance
::symbol => 800 instances
::cell => 3 instances
::eof => 1 instance
::key => 1 instance
::nil => 1 instance
::unspecified => 1 instance
5137224 bytes used. (0.929s)
Memory Debugging - example

(bugloo) (gc)
(bugloo) (info heap "::")
::ast-node => 29988 instances
::leak2 => 11 instances
::pair => 91109 instances
::struct => 1 instance
::bint => 25982 instances
::nil => 1 instance
::procedure => 1 instance
::symbol => 800 instances
::cell => 3 instances
::eof => 1 instance
::key => 1 instance
::nil => 1 instance
::unspecified => 1 instance
5137224 bytes used. (0.929s)
Memory Debugging - example

(bugloo) (gc)

(bugloo) (info heap "::")
::ast-node => 29988 instances
::leak2 => 11 instances
::pair => 91109 instances
::struct => 1 instance
::bint => 25982 instances
::nil => 1 instance
::procedure => 1 instance
::symbol => 800 instances
::cell => 3 instances
::eof => 1 instance
::key => 1 instance
::nil => 1 instance
::unspecified => 1 instance
5137224 bytes used. (0.929s)

(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 ===> *nodes-cache*
command took 0.743s.
JVM Debugging Architecture

- Debugging with two JVMs
  - JVMDI: instrumentation
  - JDI: control (Java)
- Event-driven Communication
  - Manipulation of stubs
  - JDWP abstract channel
- Embed code in the debuggee
  - conditional breakpoints
  - memory debugging
Mapping Scheme To JVM (1/2)

- Bigloo directly maps into JVM:
  - modules $\Rightarrow$ classes
  - functions $\Rightarrow$ methods
Bigloo directly maps into JVM:

- modules ⇒ classes
- functions ⇒ methods

Proper display of Bigloo objects:
Mapping Scheme To JVM (1/2)

- Bigloo directly maps into JVM:
  - modules $\Rightarrow$ classes
  - functions $\Rightarrow$ methods

- Proper display of Bigloo objects:

  Normal display:
  
  $f (\text{::procedure}) = \#<\text{procedure:1}>$
Mapping Scheme To JVM (1/2)

- Bigloo directly maps into JVM:
  - modules $\Rightarrow$ classes
  - functions $\Rightarrow$ methods

- Proper display of Bigloo objects:

  Bugloo display:

  \[
  f \ (\mathbf{::procedure}) = \mathbf{procedure} \ (\mathbf{foo} \ \mathbf{::obj}) \text{ in file } \mathbf{foo.scm:2}
  \]
Mapping Scheme To JVM (1/2)

- Bigloo directly maps into JVM:
  - modules $\Rightarrow$ classes
  - functions $\Rightarrow$ methods

- Proper display of Bigloo objects:

  Bugloo display:
  \[
  f \ (::\text{procedure}) = \text{procedure} \ (\text{foo} :: \text{obj}) \ \text{in file foo.scm:2}
  \]

- Some construction are emulated:
  - closure, higher order functions
Mapping Scheme To JVM (2/2)

Hide internals of Bigloo compilation

- Filtering Single Stepping:
  - Step out of JVM constructors:
    ```
    (filter ext add ("bigloo\..*\.<clinit>" . out))
    ```
  - Don’t stop in higher-order call dispatcher:
    ```
    (filter ext add ("\.funcall[0-4]\(" . next))
    ```
Mapping Scheme To JVM (2/2)

Hide internals of Bigloo compilation

- Filtering Single Stepping:
  - Step out of JVM constructors:
    \[
    \text{(filter ext add ("bigloo\..*\.<clinit>"). out)}
    \]

  - Don’t stop in higher-order call dispatcher:
    \[
    \text{(filter ext add ("\\.funcall[0-4]\(" . next))}
    \]

- Limitations
  - Does not filter steps inside a function
  - Functions still visible in the stack frame
Performances

- Performance penalties are limited:
  - 1.5% to 6% slower than normal execution
  - No impact on memory consumption
  - JIT stays enabled during debug
  - Good performances for memory debugging: back-reference path (546 links): 4.5s on a 20 Mb heap (>396000 objects) (Athlon XP 1900+)
  - Can debug the Bigloo compiler (¼ 130000 lines)
Performances

• Performance penalties are limited:
  • 1.5% to 6% slower than normal execution
Performances

- Performance penalties are limited:
  - 1.5% to 6% slower than normal execution
  - No impact on memory consumption
Performances

• Performance penalties are limited:
  • 1.5% to 6% slower than normal execution
  • No impact on memory consumption

• JIT stays enabled during debug
Performances

- Performance penalties are limited:
  - 1.5% to 6% slower than normal execution
  - No impact on memory consumption
- JIT stays enabled during debug
- Good performances for memory debugging:
  - back-reference path (546 links): 4.5s on a 20 Mb heap (> 396000 objects) (Athlon XP 1900+)
Performances

- Performance penalties are limited:
  - 1.5% to 6% slower than normal execution
  - No impact on memory consumption

- JIT stays enabled during debug

- Good performances for memory debugging:
  - back-reference path (546 links): 4.5s on a 20 Mb heap (> 396000 objects) (Athlon XP 1900+)

- Can debug the Bigloo compiler (∼ 130000 lines)
Conclusion

- We have developed Bugloo:
  - Can debug Bigloo programs compiled for JVM
  - Source level debugger + custom features
  - Integration into the (X)Emacs editor
Conclusion

- We have developed Bugloo:
  - Can debug Bigloo programs compiled for JVM
  - Source level debugger + custom features
  - Integration into the (X)Emacs editor

- Advantages of the JVM as a debug platform:
  - Clean API to instrument the debuggee
  - Same classfile for debug and for performances
  - Usable for debugging large programs
Conclusion

- We have developed Bugloo:
  - Can debug Bigloo programs compiled for JVM
  - Source level debugger + custom features
  - Integration into the (X)Emacs editor

- Advantages of the JVM as a debug platform:
  - Clean API to instrument the debuggee
  - Same classfile for debug and for performances
  - Usable for debugging large programs

http://www-sop.inria.fr/mimosa/fp/Bugloo