I Stepped Too Far

Bdbj: Reverse Execution for Debugging Java

Jon Cook

Abstract: We modify the Kaffe implementation of the Java Virtual Machine, to enable Java bytecode to be run backwards, which is useful in debugging. Also, the operational semantics of reversing stack-based code are investigated.

Introduction

Most commercially available debuggers are only capable of stepping forwards through the source code. Because the location of the bug is usually unknown, we may find after having stepped through hundreds of lines of code that we have stepped well past the bug and have to start again. In such instances it is helpful to be able to step back through the source code to the location of the bug.

Kaffe is written in C, but we wrote the user interface in Java, using Swing. The user interface runs Kaffe and communicates with it via a socket.

Two Circular Buffers

As the program is run forwards, data needed for reverse execution is stored on two circular buffers. These buffers employ a stack-like first-in-last-out discipline, which makes them similar to the kill ring of emacs. The first circular buffer, the logging circular buffer, stores data lost from the operand stack, local variables and fields. The other buffer, the program counter circular buffer, stores data on the flow of the program by noting the program counter before each instruction.

The buffers are bounded, and if more data than they can hold is pushed to them, they wrap around and data already on the buffers is overwritten. Thus, how far we can step back is determined by the sizes the user chooses for the buffers.

Handling Most Java Instructions

We have to tailor the reverse execution scheme to a stack based virtual machine. Almost all the Java bytecode instructions can be characterised by two numbers, m and n, where the instruction can be thought of as popping m items from the operand stack, possibly operating on them and pushing n new items onto the operand stack.

We reverse most instructions as follows. Before the instruction is executed in the forwards direction we push the top m items of the operand stack onto the logging circular buffer. When the instruction is reversed, we pop (discard) the top n items of the operand stack and then push the top m items of the logging circular buffer onto the operand stack.

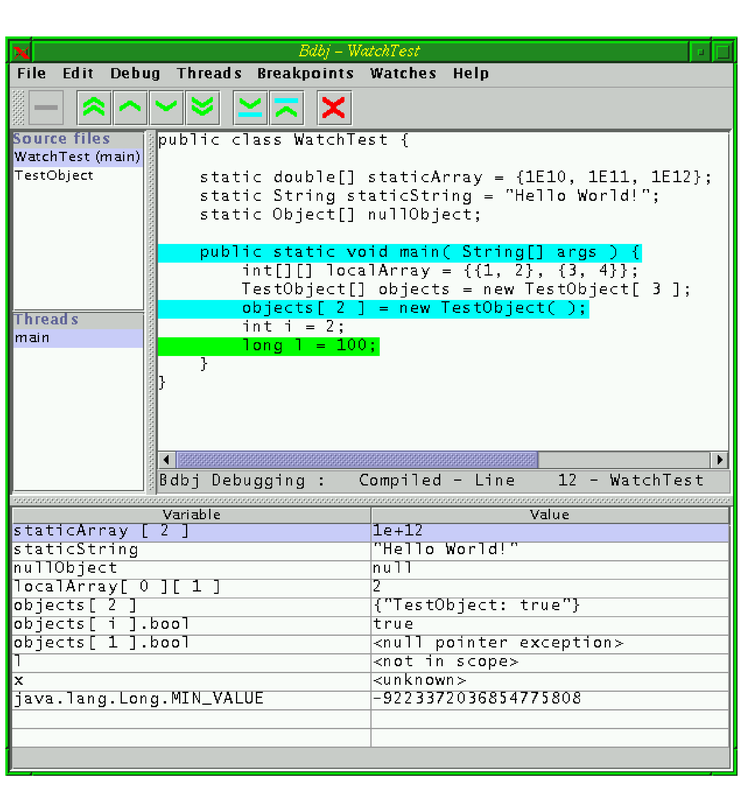
Operational Semantics

Let **STANDARD** represent the set of Java bytecode instructions which can be reversed by this scheme. We denote the state with program counter pc, operand stack S, local variables L, fields F, logging circular buffer D and program counter circular buffer C by $\langle pc, S, L, F; D, C\rangle$. For the forwards direction, the function *trans* produces the new items to push to the operand stack, the new program counter value and any new fields.

The result of pushing the value x onto the stack or buffer, S, is denoted by $x \cdot S$.

Then $P \vdash \langle pc, S, L, F; D, C\rangle \longleftrightarrow \langle pc', S', L', F'; D', C'\rangle$ indicates that in one step the program running in the forwards direction can make the transition indicated by going to the right, and the program running in the backwards direction can make the transition indicated by going to the left. The premise enclosed in square brackets is only required in the forwards direction.

$$\frac{P[pc] \in \textbf{STANDARD} \quad pops[P[pc]] = m \quad pushes[P[pc]] = n \quad [trans(P[pc], L, x_1 \cdot x_2 \cdots x_m) = (y_1 \cdot y_2 \cdots y_n, pc', F')]}{P \vdash \langle pc, x_1 \cdot x_2 \cdots x_m \cdot S, L, F; D, C\rangle \longleftrightarrow \langle pc', y_1 \cdot y_2 \cdots y_n \cdot S, L, F \cup F'; x_1 \cdot x_2 \cdots x_m \cdot D, pc \cdot C\rangle}$$



Screen-shot (the green line is the current line of execution and the cyan lines are breakpoints)

Other Instructions

Some instructions store items on the operand stack into local variables or fields. In these cases we must push the overwritten variable's value to the logging circular stack as well.

Method Calls and Exceptions

Kaffe is recursive, in that it simulates a method call in the Java code by a function call in its code. To achieve reverse execution we pass extra data (saying which direction to go in) to and return extra data (saying which end we came out of) from the function in Kaffe which runs the bytecode of a method.

Before leaving a method call we must push the local variables and operand stack to the logging circular buffer so that they can be restored on stepping back into the end of the method.

When an exception is thrown we store the local variables and operand stack for each method we jump back through. On reversal these method calls are recreated.

Garbage Collection

We prevent objects which might be accessed again on reverse execution from being garbage collected by adding the objects in the logging circular buffer to a special array of objects we create in the virtual machine.

To do this we must keep track of which operand stack slots contain objects, and which local variables currently contain valid objects (from within the virtual machine we can see local variables before they are given values).

Multi-threading

Each thread in the program being debugged is assigned a queue in the debugger. When the user issues a command, the user interface adds it to the relevant queue. Thus the debugger has control over which direction every thread in the program being debugged is executing in.

Each thread has its own circular buffers.

We ensure that on reverse execution the critical regions protected by a monitor are the same. For example, a method synchronized in the forwards directions is also synchronized in the reverse direction.

Further Information

The reader may wish to obtain our debugger which is available from

`http://www.dcs.ed.ac.uk/home/jjc/`

Division of informatics