A Tool debugger in the browser

Compiler Construction 2014 Final Report

Hadrien Milano Christophe Tafani-Dereeper
EPFL
{hadrien.milano, christophe.tafani-dereeper}@epfl.ch

1. Introduction

During the semester, we have implemented a full compiler for Tool written in Scala. It outputs Java byte code which can consequently run on the JVM.

Throughout this period, we had the occasion to write quite a number of Tool programs, which allowed us to notice that it could be tedious to debug Tool code. That's why we chose to develop a debugger for Tool.

1.1 Features

1. Load Tool source files, or develop directly in the web IDE
2. Set breakpoints in the code
3. Step-by-step execution: step over lines, step into / out function calls
4. See the value of every variable
5. See the current call stack
6. Integrated code editor and compiler

We chose to implement our debugger as a single web page such that it is cross platform, easy to use (no binary to download) and can still be used offline. Moreover, it allows us to leverage the browser's garbage collector to implement easily the debugging VM (see "Virtual machine implementation")

1.2 Running the debugger

An online version of the debugger is available at the following address : bit.ly/tool-debugger. It can also be ran locally by opening the INDEX.HTML in the TOOL_DBG directory of the Git repository.

2. Implementation

2.1 Modifying the compiler backend

First, we had to make a new version of the code generator. The compiler now outputs a JSON file, containing the elements described below.

- The list of the classes that the program contains
- For each class, the list of its methods
- For each method
  - the variables declared in it
  - the arguments list, along with their type
  - the code

The code is a simplified bytecode-like language (in its textual representation), whose full specification can be found in the file ASM_DEFINITION.MD (Github link at the end of the document). The opcodes are similar to the ones found in Java's bytecode, with several notable differences.

- variables, methods and classes are referred to by their name, and not a numeric identifier
- an instruction STAT is used to map instructions to source code lines
- locales are resolved at runtime

This bytecode-like language seemed to be the best solution. First, we could have transpiled directly to JavaScript and use some JS magic to allow the step-by-step execution. While it could seem like a feasible solution at first, due to the highly dynamic nature of JS, it is in practice not possible to break a function call to allow for instance *step into* and *step out* operations. This means that we need to create a machine able to execute the code with a clean debugging interface. We could also have interpreted the tool code directly, but since we already have a compiler for tool, it allows us to implement a very simple stack machine — which is way easier than a complete tool interpreter. This justifies our choice regarding this low-level code representation.

2.2 Porting the compiler in the browser

To make the compiler run in the browser, we used SCALAJS, an EPFL-made library that transpiles Scala to JavaScript. It needs however some bindings to properly interact with our JavaScript code. We wanted to make as few modifications as possible to the compiler's CLI interface. To this end, we used BrowserFS, a library that creates a NodeJS style file system interface, and created a few simple nodejs-scala bindings so that this interface could be used instead of java's file system. The compiler code can then be kept intact and will work as expected when run in a browser. It worked well for input files but got a little dirty as far as output was concerned. We therefore opted for a more invasive technique. A JS function in charge of receiving the compilation output is called directly in the code generation stage of the compiler which makes this specific version of the compiler unusable outside the web IDE.

This work of porting the compiler to the browser makes a huge difference in terms of usability of our debugger. It becomes almost a minimalist IDE for tool. We can compile as the code is typed and report errors in the editor, thanks to the positioned error reporting previously implemented in the compiler.

2.3 Implementation of the virtual machine

At this stage, we had a compiler able to run in the browser and output a custom representation of any Tool program, but nothing to interpret it. We have implemented a JavaScript virtual machine able to execute instructions that we have defined for our custom bytecode-like language.

2.3.1 Architecture

The execution is handled by the Engine class. It contains a StateMachine and methods to handle the debugging control flow (breakpoints, run, step into, etc.). The state machine contains information about the current state of the program (stack, program counter, current scope...) as well as methods to change it.

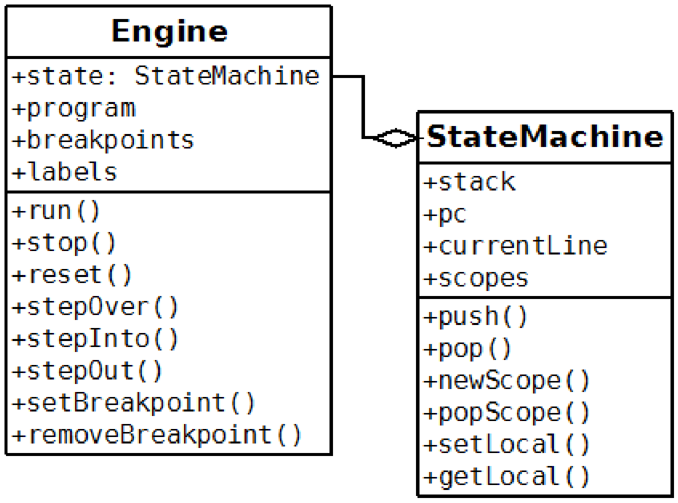

Figure 1. (Non-exhaustive) class structure

2.4 Developing the web interface

After that, we had to implement the web interface of the debugger. It is divided into four main parts.

- The code editor, where you can type code and set breakpoints
- The toolbar, which allows you to load a Tool source file, compile it, and control the debugging process (run / stop / step over / step into / step out)
- The console, where you can see messages from the debugger, compilation errors, and the program output
- The right panel, that allows you to see the values of the variables in the current scope, the call stack and the breakpoints you have set. Clicking on an element of the call stack will bring you at the location the call was made. Breakpoints can be disabled by clicking on the green dot icon, and removed with a right click.

The interface was made in HTML and JavaScript, with the use of Dijit, a popular library to create web user interfaces.

A special tab entitled *ASM* allows you to see what *bytecode-like* code was generated. It is not relevant when debugging a tool program, but can be useful to see what's happening behind the scenes. It will have at least allowed us to debug the debugger.

2.4.1 Control flow

Once the tool code has been typed, it can be debugged using the control buttons at the top of the interface.

The run action runs the program. As everything (debugger and virtual machine) is single threaded, there is no way to stop a program that loops forever if no breakpoints have previously been set.
The stop action resets the program. One must click on the stop button to reset the program once it has run before running it again.
The left arrow — *step over* action — steps over the current line.
The down arrow — *step into* action — steps into the current function call. If the current instruction is not a function call, it goes to the next instruction.
The up arrow — *step out* action — steps out of the current function call, returning to the line where the function has been invoked. If there are multiple function calls in one instruction, one can use a combination of step into/step out to visit each call.

Breakpoints The last control flow feature is breakpoints. To set a breakpoint, one can click on the left gutter, next to the line number corresponding to the line you want to break on. Note that not every line is breakable. You can only break on actual statements. Once a breakpoint has been set, it can be removed by clicking again on the same spot. A breakpoint can also be disabled by clicking the green icon in the breakpoints view on the right, making it ineffective but still set for later use. The debugger will always stop at an active breakpoint when its line is reached.
Breakpoint rules takes precedence over action rules, i.e. if a *step out* action is requested but there's a breakpoint before the end of the function, the execution will pause at the line where the breakpoint is set.

3. Possible Extensions

We will use this section to discuss another implementation possibility as well as how we could improve our solution.

One of the things we were told is that we could have transpiled tool to javascript, generated a source map file and then used a native debugger such as the Chrome developper tools. While this approach may seem beneficial at first (we use a robust and full featured piece of GUI), it is limited and actually not suitable for our purpose. A source map is only a way to debug something that is meant to be run in a javascript environment. It depends on javascript's underlying types and does not allow a great deal of language-specific customization. On the other hand, we want the tool program to behave identically as in the JVM or anywhere else — and this even when types are pushed to their limits. Also, our approach allows to integrate the compiler and the debugger in a coherent environment which is much more comfortable to use.

We are quite satisfied with the solution we came up with. We believe that with this tool (no pun intended), we would have gained massive time when writing test programs. The feedback loop of writing code, finding bugs, killing them and writing code again would have been much tighter with everything contained in one single interface. We acknowledge that it is far from being perfect and we have a few ideas of things that could have been further improved.

- The VM should run in a separate thread than the UI. This allows the interface to stay responsive even when the program is stucked. Currently, if a program loops indefinitely, the web page has to be closed. It is not a trivial modification and involves some work with HTML5 web workers and messaging — we believe this was a little out of the scope of this extension.
- Although the core software involved here works well, the UI is still pretty rough and could have been improved a bit. For instance, we could make the variables tree not to collapse each time we step over an instruction.
- Some debugging related features could be added: watch the value of expressions, change the value of variables from the interface, set conditional breakpoints...

4. Resources

- Dojo toolkit website
- BrowserFS on Github
- ScalaJS website
- Online version of the debugger
- Specification of our custom bytecode-like language