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, and 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. Real-time compilation

To avoid the need of downloading extra software, we chose to make our debugger run in the browser.

2. Examples

Give code examples where your extension is useful, and describe how they work with it. Make sure you include examples where the most intricate features of your extension are used, so that we have an immediate understanding of what the challenges are.

You can pretty-print tool code like this:

```
object {
  def main() : Unit = { println(new A().foo(−41)); }
}

class A {
  def foo(i : Int) : Int = {
    var j : Int;
    if(i < 0) { j = 0 − i; } else { j = i; }
    return j + 1;
  }
}
```

This section should convince us that you understand how your extension can be useful and that you thought about the corner cases.

3. Implementation

This is a very important section, you explain to us how you made it work.

3.1 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 takes the form of a custom simplified bytecode-like language, whose full specification can be found in the Annex. It is similar to Java's, 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 bytecode instructions to source code lines

3.2 Porting the compiler in the browser

To make the compiler run in the browser, we used SCALAJS, an EPFL-made library that transcompiles

Scala to JavaScript. **TODO[Hadrien] Talk about browserFS** This allows us to conveniently output a JSON file (described in section **??**) containing the program structure and code.

3.3 Implementation of the virtual machine

At this stage, we have 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.

3.3.1 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 them.

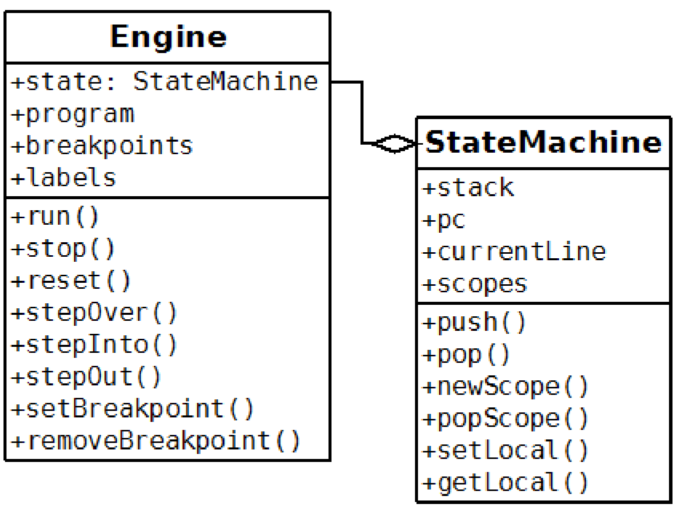

Figure 1. (Non-exhaustive) class structure

Each instruction is represented by a JavaScript function that can change the program's state by using the state machine's methods.

```
function fn_mul(sm) {
  var b = sm.pop();
  var a = sm.pop();
  sm.push(a * b);
}
```

Code example 1. Example implementation the multiplication instruction

When a (bytecode level) instruction has to be executed, the JavaScript function associated is executed, and the program counter incremented.

3.3.2 Control flow

TODO : talk about labels, step in, step out, etc

3.4 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 and the callstack .

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

3.5 Implementation Details

Describe all non-obvious tricks you used. Tell us what you thought was hard and why. If it took you time to figure out the solution to a problem, it probably means it wasn't easy and you should definitely describe the solution in details here. If you used what you think is a cool algorithm for some problem, tell us. Do not however spend time describing trivial things (we what a tree traversal is, for instance).

After reading this section, we should be convinced that you knew what you were doing when you wrote your extension, and that you put some extra consideration for the harder parts.

4. Possible Extensions

If you did not finish what you had planned, explain here what's missing.

In any case, describe how you could further extend your compiler in the direction you chose. This section should convince us that you understand the challenges of writing a good compiler for high-level programming languages.

References