Documentation for SMTInterpol Webinterface

Andrea Qosja

November 9, 2020

Abstract

This is a short document to describe the experience of using TeaVM to create a Web-Interface in JavaScript for SMTInterpol. It contains a brief explanation of my first try to convert Java into WebAssembly, the concrete usage of TeaVM libraries, and the third part will describe my experience of using TeaVM to convert SMTInterpol including the issues encountered, the solution and the notices for further examination.

1 WebAssembly

The initial goal of the project was to have our Java code run as WebAssembly code in the browser, since from our search it indicated to have a better performance than JavaScript and to be user friendlier. There is a list[1] of the converters into WebAssembly, from which we tested all three converter, that convert from the Java language:

- TeaVM,
- JWebAssembly,
- Bytecoder.

The three converter did work on ground level(simple examples), but I was unable to test bigger Java-Projects, out of following main reasons. Inexperience in Java, JavaScript, WebAssembly, Maven and Gradle, combined with short documentation (maybe written to be understood by an experienced user) made it too complex for me to understand the basic principle

of all the combined usage of the above mentioned languages/tools. I mentioned JavaScript because at the time this document is written, it was not possible to access WebAssembly code in browser without JavaScript. Adding to that WebAssembly is at this time a fairly new, not fully integrated web-programming-language with some missed features, where also different kinds of browsers may or may not process WebAssembly Code at all, or specific libraries/functionalities. That is where we decided to bypass WebAssembly and focus on JavaScript only.

2 TeaVM Usage

In this section I will go through the steps on how I used TeaVM and explain the key components that made the project whole and working, in a way it can be useful for every reader.

2.1 First Implementation

The TeaVM usage for converting into JavaScript was good documented and strait forward for our project focus, which was to fetch the input, feed it to SMTInterpol for processing, and give the output back. I had to write a Java main file, where the connection with the JavaScript code was done in two parts, which is partially already prepared for you from the TeaVM project build[2]. Part one, in the `index.html` file of the webpage, you should already have this line: `<body onload="main()">`. It starts the connection between your `Main.java` file and the html file, through the `classes.js` file that was generated by TeaVM containing the translated Java code of your Java main file.

Secondly the connection also works the other way around, meaning from you Java main file, one has access to the elements of the webpage/html document, e.g. the input and the output field. It works through the HTML libraries provided by TeaVM. A short syntax example, to access a html element:

```
private static HTMLDocument document =
    Window.current().getDocument();
private static HTMLElement responsePanel =
    document.getElementById("response-panel");
```

The "response-panel" is the name of the output panel on the webpage, which you now can modify and rewrite from your Java main file.

2.2 Alternative Implementation, Web-Worker Functionality

The SMTInterpol can, depending on the complexity of the input take a certain amount of time to process, this would freeze the webpage and it was unclear if the program is still processing or not. A solution therefore was the use of web-worker, a JavaScript solution to run the thread on a parallel script. But web-worker is only able to send and receive simple string messages with the main thread, and did not recognise any of the elements of the main thread, therefore the first usage of TeaVM was not compatible with the web-worker's goal. To handle this issue, we reshaped the task for each script and Java file as following, where the usage of TeaVM libraries was also changed.

2.2.1 Main Script

The main script written inside `index.html` is responsible for all the elements of the webpage, for fetching the input, starting the worker script, sending the input to it, waiting for an answer and write it at the webpage when it has the answer.

2.2.2 Web-Worker Script

The web-worker script, `webworker.js`, will run in the background only when the first script sends the input. It will pause working when it has send back the output and can be completely closed only from the main script. Here is were the main Java file will load and run at command, so the heavy processing will happen in the background and not effect the webpage. To run our SMTInterpol program we declared a variable `var runWebInterface;` and call `runWebInterface.runSMTInterpol(input);` inside the worker script when the input is there. Why this works it will be explained in the next sub-section.

2.2.3 Main Java File

In contrary to the first implementation, we will no longer have access to the elements of the webpage from the `Main.java` file, as we want to load it to

the web-worker script, and as explained above, the web-worker script has no access to the elements of the webpage. The syntax therefore is simple, you load the main file with `importScripts("classes.js");`, and initialize it with `main();`. In this case TeaVM with the adequate syntax offers usage of the Java classes and single methods from your JavaScript file. TeaVM uses JSObjects within a specific work frame to make this available [3]. To ensure the functionality we had to make the following changes, the first is that the main class now extends an interface named SolverInterface from `SolverInterface.java` (our choice of name), which itself extends the JSObject interface provided by the TeaVM libraries. In the main class we used the provided syntax to write the next two lines of code:

```
@JSBody(params = { "handler" },
    script = "runWebInterface = handler;")
public static native void
    setSolverInterface(SolverInterface handler);
```

The native method `setSolverInterface` is used just to assign an initialization, here called handler, of the class SolverInterface to a variable, not to a Java variable but to a JavaScript one, specifically named `runWebInterface`, hence the declaration of the variable name in the worker script. This means that the Java main class is initialized as variable in the worker script and that we can use all the methods and the functionality of that class in this script. E.g. `runWebInterface.runSMTInterpol(input);` calls the method `runSMTInterpol();` from the Main Java class. Similar coded lines as explained above allowed the use of postMessage-Method of another Java class named WebEnvironment, proving the point that this functionality is not restricted to one single usage.

3 Experience, Adaptation, and Bugs

This section will contain the rest of my experience with TeaVM, including all the implemented adaptations necessary for the interface to run, and the encountered bugs.

3.1 Otimization Level in TeaVM

There is an option from the compiler of the TeaVM, to be found in the pom.xml file on line 150, `<optimizationLevel>SIMPLE</optimizationLevel>`,

that gives the possibility to optimize the code translation from Java to JavaScript, in three increasingly different level SIMPLE, ADVANCED, FULL.

We did not try to test the optimization quotas, because of the errors, and were able to deploy only level SIMPLE, to ensure a stable version. Since every higher level would result in errors, which even after closer inspection were not to be explained. The code is wrongly translated from TeaVM when using this higher levels. You can duplicate the error specifically when using the proof check mode of SMTInterpol.

3.2 Supported Java Library/Classes from TeaVM.

There is only partial implementation from TeaVM for the Java Formatter Class, and it is not planned to be completed in the future[4]. This results in less developer feedback from the Web-Interface regarding the measurement e.g. of the execute speed, since no `%.f` numbers can be shown. The only fix possible was commenting the responsible lines out. Following files contain instances of the described issue: `QuantifierTheory.java`, `ArrayTheory.java`, `LinArSolve.java`.

3.3 Adaptation

SMTIntrpol used `java.lang.System.exit()` as a way to close the program, that was not compatible with JavaScript and needed to be changed, specifically in the parseEnvironment we overwrited the exitWithStatus-Method.

The output of SMTInterpol was also very simplified, so that we needed to overwrite just the one function printResponse.

TeaVM is also not compatible with unmodifiable sorted set that is why SMTInterpol uses only modifiable one, but only on the TeaVM branch.

3.4 Bugs

TeaVM implementation of the `TreeMap<type, type>.subMap(index, index)` is broken, which result in wrongly outcomes and/or errors in the our Web-Interface. Issued open at the GitHub page[5] of TeaVM.

HashCode of TeaVM gave an other element as expected, that is why we changed from using `Integer.hashCode();` to using just HashCode[6].

References

[1] Steve Akinyemi: Awesome WebAssembly Languages.
https://github.com/appcypher/awesome-wasm-langs#java

[2] TeaVM 2019: Getting started.
http://teavm.org/docs/intro/getting-started.html.

[3] TeaVM 2019: Interacting with JavaScript.
http://teavm.org/docs/runtime/jso.html

[4] MeFisto94: Float/Double Support for Formatter #472.
https://github.com/konsoletyper/teavm/issues/472.

TeaVM: Jcl-support.
http://teavm.org/jcl-support/0.6.x/jcl.html

[5] And-rea-Q: subMap does not return empty maps correctly #519.
https://github.com/konsoletyper//issues/519

[6] jhoenicke: Don't rely on `Integer.hashCode()`.
https://github.com/ultimate-pa/smtinterpol/commit/2260635b54c0af738551849c99d2e2a179499da5