JavaScript Sealed Classes (Artifact)

Manuel Serrano
Inria/UCA, Inria Sophia Méditerranée, 2004 route des Lucioles, Sophia Antipolis. France

Abstract

In this work, we study the JavaScript *Sealed Classes*, which differ from regular classes in a few ways that allow ahead-of-time (AoT) compilers to implement them more efficiently. Sealed classes are compatible with the rest of the language so that they can be combined with all other structures, including regular classes, and can be gradually integrated into existing code bases.

We present the design of the sealed classes and study their implementation in the hopc AoT compiler. We present an in-depth analysis of the speed of sealed classes compared to regular classes. To do so, we assembled a new suite of benchmarks that focuses on the efficiency of the `class` implementations. On this suite, we found that sealed classes provide an average speedup of 19%. The more classes and methods programs use, the greater the speedup. For the most favorable test that uses them intensively, we measured a speedup of 56%.

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers; Software and its engineering → Source code generation; Software and its engineering → Object oriented languages; Software and its engineering → Functional languages

Keywords and phrases JavaScript, Compiler, Dynamic Languages, Classes, Inline caches

Digital Object Identifier 10.4230/DARTS.8.2.23

Related Article Manuel Serrano, "JavaScript Sealed Classes", in 36th European Conference on Object-Oriented Programming (ECOOP 2022), LIPIcs, Vol. 222, pp. 24:1–24:27, 2022.
`https://doi.org/10.4230/LIPIcs.ECOOP.2022.24`

Related Conference 36th European Conference on Object-Oriented Programming (ECOOP 2022), June 6–10, 2022, Berlin, Germany

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2022 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

This artifact enables scientists to re-build and re-run the experiments of the associated paper. The artifact enables scientists to compare the performance of the proposed JavaScript Sealed Classes to that of regular classes. The artifact contains all the necessary material to confirm the speedup of sealed classes claimed in the paper.

2 Content

The artifact package includes:

- The new benchmark suite created for evaluating the performance of JavaScript class implementations;
- The benchmarking harness used to run all the tests;
- The source code of the modified hopc compiler that natively supports sealed classes.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at: `ftp://ftp-sop.inria.fr/indes/fp/Hop/artifact/ecoop2022-artifact.tgz`

4 Tested platforms

This artifact is about performance. Then, the results presented in the paper can only be reproduced using realistically performant platforms. That is, it is strongly advised to run the artifact on an actual multi-core platform. If an emulator and virtualization are used pay attention to:

- Enable hardware virtualization, otherwise performance will be that of the emulator instead of the performance of the hopc compiler.
- Enable multi-core emulation as `hopc`'s garbage collector is multi-threaded and running on a single core platform degrades performance significantly.

Starting from a vanilla Linux box, the libraries and compilers that are required to run the tests can be automatically installed using the `install.sh` shell script. If anything goes wrong, please check that file. The script installs all the tools and compilers needed to run the artifact. The longest part of the installation is the compilation and installation of `nodejs-16.13.0.tar.gz`. This version is needed for building the figures of the paper that compare Nodejs performance with the performance obtained using the material presented in the paper. Installing Nodejs requires root privileges. The shell script `install.sh` uses `sudo`. Hence, the user running `./install.sh` must be in the sudoers list.

In the shipped image, these required compilers are already installed and don't need to be re-installed.

The artifact proceeds to many compilations and executions. This takes time. Using the Docker version, running the full artifact, takes about 80 minutes on an 2017 mid-range desktop computer.

Preparing the artifact using the `install.sh` script for a native execution last about 60 minutes.

5 License

GNU General Public License version 2.

6 MD5 sum of the artifact

8a9f211b93872e7a89f974603c7037c8

7 Size of the artifact

1.21 GiB

A Getting Started

This artifact compiles and runs *all* the benchmarks described in the paper. However, to limit the execution time of the execution of this artifact, instead of running each test 30 times, as we do for the paper, they are only executed 3 times.

The compilers needed for running the test are all compiled inside a docker image. The docker image is `docker-image-jssealedclasses-ecoop22.tgz`. To load it use:

```
(host) docker load < docker-image-jssealedclasses-ecoop22.tgz
```

To run and generate all the figure of the paper, simply run:

```
(host) docker run -t -i --entrypoint=/bin/bash jssealedclasses
(docker) make
```

This will produce in the current directory all the PDF files corresponding to the figures presented in the paper but figure 13. This command internally uses the 'Makefile' located in the bench directory described earlier. Each figure can be generated separately:

```
(docker) make figure2-lhs.pdf
(docker) make figure2-rhs.pdf
(docker) make figure3-lhs.pdf
(docker) make figure3-rhs.pdf
(docker) make figure5-lhs.pdf
(docker) make figure5-rhs.pdf
(docker) make figure7.pdf
(docker) make figure9-lhs.pdf
(docker) make figure9-rhs.pdf
(docker) make figure10.pdf
(docker) make figure11.pdf
(docker) make figure12.pdf
(docker) make figure13.pdf
```

The test executions generate json files stored in the `bench/LOG.xxx` directory, where *xxx* is the name of the test (*e.g.*, `"class"`, `"ipoly"`, `"size"`, ...). Each of these files contains information about the platform executing the test, the compiler used, and the execution times. The format of these files is designed to make them easy to process. All the graphs and tables of the paper are mere visualizations of these files.

WARNING: Figure 13 is based on the results of the Linux `perf` command. Running `perf` under Docker requires a special configuration that if not meet, prevents Figure 13 to be be regenerated.

1. as Docker uses the host kernel, `perf` can only be used under docker, if the host kernel version is the one expected by the docker `perf` command. The docker image is based on Debian bullseye, which assumed Linux 5.10. Hence, 'perf' under docker *can only be executed if the host also uses Linux kernel 5.10.*
2. Monitoring the host kernel requires special authorization. These authorization can be granted permantently by adding the following line `kernel.perf_event_paranoid = -1` to `/etc/sysctl.conf`. This action requires root privilege on the host. This is mandatory for generating Figure 13.
3. Finally, Linux `perf` needs special authorization for being executed under Docker (see `https://newbedev.com/use-perf-inside-a-docker-container-without-privileged`).

To enable Linux `perf` add the following option to docker:

```
--security-opt seccomp=docker-ecoop-seccomp.json
```

where `ecoop22-seccomp.json` is a file shipped with this artifact. This merely white-lists the system Linux syscall `perf_event_open`.

Provides with this extra option, All figures can be generated:

```
(host) docker run --security-opt seccomp=ecoop22-seccomp.json \
  -t -i --entrypoint=/bin/bash jsrecords
(docker) make artifact     # generate all the figures
```

or, to generate only the `figure9.pdf` file:

```
(host) docker run --security-opt seccomp=ecoop22-seccomp.json \
  -t -i --entrypoint=/bin/bash jsrecords
(docker) make figure13.pdf  # generate only figure13.pdf
```

Bear in mind that hopc uses multi-threaded executions. If your docker daemon is not able to use the physical parallelism of its host, you will observe significant differences with the values reported in the paper. This is why we *strongly* recommand, when possible, to run the test natively. For that, you will have to extract the `bench` directory from the docker image and execute the following:

```
$ mkdir -p /home/jsrecords/.config/hop && \
    echo "(add-user! \"anonymous\" :directories '* :services '*)" \
      > /home/jsrecords/.config/hop/hoprc.hop
$ make -C /home/jsrecords/bench bigloo # install in /tmp/JSRECORDS/local
$ make -C /home/jsrecords/bench hop
$ make -C /home/jsrecords/bench hop.icfp21
```

If Node.js (at least node-v16) is not already installed on you machine you can install it with:

```
$ cd /tmp
$ wget https://nodejs.org/download/release/v16.13.0/node-v16.13.0.tar.gz
$ tar xvfz node-v16.13.0.tar.gz
$ cd node-v16.13.0 && ./configure && make && make install
```

Then, you can run the test

```
make -C bench
```

The compilation of some tests triggers warning messages during the compilation of the generated Scheme file. These warning messages can be safely ignored because they are simply due to the weakness of the Scheme occurrence typing that is not always able to track JavaScript types precisely.

The hopc compiler is installed in the `$PWD/JSRECORDS/local/bin` directory. The following shows its options:

```
hopc --help
```

As an example, to compile a JavaScript file proceed as follows:

```
$ cat > ex.js << EOF
"use string";

console.log( "hello world" );
EOF
$ hopc -v2 -Ox foo.js -t foo.scm
$ ./a.out
```

The command "`hopc -v2 -Ox foo.js -t foo.scm`" builds a binary file `./a.out` and it generates the intermediate Scheme file `foo.scm`.

All the JavaScript test source files are located in the `bench/jsbench` directory. They can all be compiled using similar invocation. Example:

```
cd bench/jsbench/octane; hopc -Ox richards.js && ./a.out
```