Scope States (Artifact)

Hendrik van Antwerpen
Delft University of Technology, The Netherlands

Eelco Visser
Delft University of Technology, The Netherlands

Abstract

Compilers that can type check compilation units in parallel can make more efficient use of multi-core architectures, which are nowadays widespread. Developing parallel type checker implementations is complicated by the need to handle concurrency and synchronization of parallel compilation units. This artifact contains benchmarks and sources for a new framework for implementing hierarchical type checkers that provides implicit parallel execution in the presence of dynamic and mutual dependencies between compilation units. The resulting type checkers can be written without explicit handling of communication or synchronization between different compilation units. We achieve this by providing type checkers with an API for name resolution based on scope graphs, a language-independent formalism that supports a wide range of binding patterns. Our framework is implemented in Java using the actor paradigm. We evaluated our approach by parallelizing the solver for Statix, a meta-language for type checkers based on scope graphs, using our framework. Benchmarks show that the approach results in speedups for the parallel Statix solver of up to 5.0x on 8 cores for real-world code bases.

2012 ACM Subject Classification Software and its engineering → Compilers; Theory of computation → Parallel algorithms

Keywords and phrases type checking, name resolution, parallel algorithms

Digital Object Identifier 10.4230/DARTS.7.2.1

Funding NWO VICI Language Designer's Workbench project (639.023.206)

Acknowledgements We thank the anonymous reviewers for their helpful comments.

Related Article Hendrik van Antwerpen and Eelco Visser, "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers", in 35th European Conference on Object-Oriented Programming (ECOOP 2021), LIPIcs, Vol. 194, pp. 1:1–1:29, 2021.
`https://doi.org/10.4230/LIPIcs.ECOOP.2021.1`

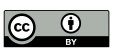
Related Conference 35th European Conference on Object-Oriented Programming (ECOOP 2021), July 12–16, 2021, Aarhus, Denmark (Virtual Conference)

1 Scope

The artifact supports the following contributions from the paper:

- We present a scope graph-based name resolution API for use by type checker implementations (Section 4.3).
- We present an actor-based algorithm that implements the hierarchical compilation unit model and the name resolution API, and provides implicit parallel execution of the compilation units (Section 5).
- We present a fine-grained deadlock handling approach to ensure termination that is well-suited for interactive applications of the type checkers (Section 5).

The parallel type checker framework is implemented as a library, and its source is included in the artifact (see section 10). This library is independent of the Statix type checker implementation, and reusable for other type checkers.



- We show that our framework captures the scheduling behavior of Rouvoet et al. [13], by porting the Statix solver to our framework. We discuss inference support and the need for a specification style that splits a declaration and its type into two connected scopes (Section 6). We parallelize all Statix type checkers, provided they follow this specification style.

A Statix solver is implemented on top of the parallel framework, and its source is included in the artifact (see section 10). This solver implementation is not tied to a specific Statix specification, and the included integration test is an example of another specification. The integration test runs the same specification against the original *and* the parallel Statix solver to ensure that the behavior of the original is preserved.

- We benchmark the parallelized Statix solver using a specification for a subset of Java on a few real world projects, showing speedups up to approximately 3x on 6 cores for larger projects.

The benchmark is included in the artifact, both as source and as executable (see section 9). The sources of the Statix specification, as well as the sources of the benchmark Java projects, are included in the artifact (see section 10).

2 Content

The artifact consists of the following:

- An executable JAR of the benchmark.
- Sources of the parallel type checker framework and the scope graph resolution algorithm.
- Sources of the Statix implementation based on the parallel type checker framework.
- Sources of the Statix specification, of a subset of Java, that is used in the benchmark.
- Sources of the Java projects that are used in the benchmark.

The artifact archive contains two directories:

Description	Directory
Self-contained benchmark JAR and sources	`scope-states-artifact`
Virtual machine self-contained benchmark JAR, sources, and pre-installed dependencies	`scope-states-artifact-vm`

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).

4 Tested platforms

Running the virtual machine requires an x86-64 system with preferably 8GiB of RAM (4GiB is an absolute minimum), and virtualization software. An OVF compatible application such as VirtualBox[1], or QEMU[2] are both supported virtualization software.

[1] `https://www.virtualbox.org/`
[2] `https://www.qemu.org/`

Running the benchmark locally requires a x86-64 system running Linux or macOS and at least 4GiB of RAM. The following software needs to be installed:

- Java[3] (versions 8–11 have been tested), to run the benchmark.
- Maven[4], to build the sources.
- R or Rstudio[5], for plotting benchmark results. Ensure the following R packages are installed: `tidyverse`, `whereami`, and `knitr`.

5 License

6 MD5 sum of the artifact

`14d77d071cfa046e3259465de789e8b9`

7 Size of the artifact

`1.5 GiB`

8 Getting Started

This section describes how to get the artifact running.

8.1 Running the Virtual Machine

To run the artifact virtual machine, follow the steps in this section.

> While this is the easiest method to run the artifact, benchmark results may be unreliable, depending on the specifics of virtualization. When in doubt, consider running benchmarks directly on the host.

Unpack the ZIP archive `scope-states-artifact.zip`, and change to the directory `scope-states-artifact-vm`. On Linux/macOS this can be done using

```
unzip scope-states-artifact-vm.zip
cd scope-states-artifact-vm
```

To run the virtual machine with QEMU, simply execute:

[3] `https://adoptopenjdk.net/`
[4] `https://maven.apache.org/`
[5] `https://www.r-project.org/` or `http://www.rstudio.com/`

```
./run
```

Otherwise, import `artifact.ovf` in VirtualBox and start the machine.

> The virtual machine is started with 4 virtual cores and 8G of RAM by default. To change these defaults, set the `CORES` and/or `MAXMEM` environment variable before invoking `run`, or change their values in VirtualBox. Running with less than 4G is not recommended. For example run with 10GB and 8 cores by invoking:
>
> ```
> CORES=8 MAXMEM=10G ./run
> ```
>
> Details on expected memory usage of the different benchmark projects can be found in section 9.

Once the virtual machine has started, you are automatically logged in, and the artifact files are found in the user's home directory.

Shutdown can be triggered from the virtualization application, or by executing the following command:

```
sudo shutdown -h now
```

Starting a graphic environment.

It is possible to start a graphic environment inside the virtual machine. Once the machine is started, execute:

```
startx
```

The graphic environment can be particularly helpful to view the plots generated from the benchmark results.

8.2 Running Locally

Unpack the ZIP archive `scope-states-artifact.zip`, and change to the directory `scope-states-artifact`. On Linux/macOS this can be done using

```
unzip scope-states-artifact.zip
cd scope-states-artifact
```

8.3 Running the Benchmark

First, execute the type checker on one of the benchmark projects using:

```
./benchmark/java-run review $(nproc) commons-csv
```

This starts a single run of the type checker, using all cores, on the smallest benchmark project: `commons-csv`.

> Benchmarks get 6G of RAM by default. Set `MAXMEM` to specify a different amount. For this initial run, using `MAXMEM=2G` should be enough. Details on expected memory usage of the different projects can be found in section 9.

This produces output similar to:

```
Running concurrent solver...
Reported errors: 367+1
Finished after 10.361 s.
```

The type checker reports a number of errors, because the Java specification does not support all the features that are used in the project code yet. The number is very high because Statix also reports many cascading errors that come from a single root cause.

Run statistics are written to a file, with a name determined by the benchmark parameters: `java-run-PROJECT-DATETIME-review-HOSTNAME-CORES.csv`. Plot the distribution of estimated run times per compilation unit and the top 10% longest-running units, by running:

```
./Rscripts/plot-java-run.R java-run-*.csv
```

The resulting plots are stored in `java-run-*.runtime_{histogram,top10}.pdf`. Details on the resulting plots are found in section 9.

A benchmark consists of running multiple iterations of the type checker with different parameters for projects and parallelism. Run a simple benchmark with:

```
./benchmark/java-benchmark 1 3 review 1,2 commons-csv
```

This runs the type checker on the `commons-csv` project with 1 warmup and 3 sample iterations, using 1 and 2 cores. The benchmark tool reports progress and ends with a summary of the results, similar to:

```
Benchmark          (parallelism)    (project)  (specVersion)  Mode  Cnt  Score   Error  Units
JavaBenchmark.run              1  commons-csv  ...              ss    3  9.194 ± 3.773   s/op
JavaBenchmark.run              2  commons-csv  ...              ss    3  5.314 ± 1.587   s/op
JavaBenchmark.run              4  commons-csv  ...              ss    3  4.404 ± 9.747   s/op
JavaBenchmark.run              8  commons-csv  ...              ss    3  4.346 ± 3.568   s/op

Benchmark result is saved to java-benchmark-*.csv
```

Benchmark statistics are written to a file, with a name determined by the benchmark parameters: `java-benchmark-DATETIME-review-HOSTNAME-WARMUP+SAMPLES.csv`. Plot runtime and speedup versus the number of cores, by running:

```
./Rscripts/plot-java-benchmark.R java-benchmark-*.csv
```

The resulting plots are stored in `java-benchmark-*.{runtime,speedup}{summary,details}.pdf`. Details on the resulting plots are found in section 9.

Inside the graphic environment, plots can be displayed by executing the following inside a terminal:

```
xpdf FILENAME.pdf
```

This is everything necessary to run benchmarks and reproduce the result plots. Read on for more details on the benchmark and the sources.

9 Benchmark: In Detail

This section describes the setup and parameters of the benchmark, and explains how to run the benchmark and how to interpret the produced results.

There are many factors that may influence and invalidate the results of a benchmark. Here are a few to consider when using this artifact:

- Other active programs. If other programs are active during the benchmark, performance of the benchmark may decrease.
- CPU scaling. If the OS does automatic CPU scaling (e.g., based on core temperature or net/battery power), the performance may decrease, as well as vary during the benchmark.

- Reported CPU's versus hardware cores. CPU's with hyper threading or similar techniques may report more CPU's than there are physical cores. Since type checking is a CPU bound problem, the scaling may taper of quickly above the number of physical cores.
- Not enough memory. If not enough memory is allocated to the benchmark, the JVM will spend more time garbage collecting, which is detrimental to performance. See below for approximate memory usage of the different benchmark projects.
- Running inside a virtual machine. The effect of the virtual machine on performance can be unpredictable. For the most accurate results, we recommend running the benchmark directly on the host.

9.1 Benchmark Parameters

The benchmark is controlled by four parameters:

- The *project* to analyze: `commons-csv`, `commons-io`, `commons-lang3`, `single-unit-clusters-call`.
- The *parallelism* to use for analysis, that is, the number of cores.
- The number of *warmup* iterations.
- The number of *sample* iterations.

The following table gives approximate running times and memory usage of individual runs for the different projects, when using 8 cores, on a Apple MacBook Pro with 2.8 GHz Intel Core i7 and 16 GB RAM:

Project	Runtime (s)	Max. Memory (GB)
`commons-csv`	26	2
`commons-io`	36	3
`commons-lang3`	86	3.5
`single-unit-clusters-call`	110	4

9.2 Paper Benchmark

The results from the paper were obtained by running the following command:

```
./java-benchmark 5 15 review \
  1,2,4,6,8,12,16,24 \
  commons-csv,commons-io,commons-lang3,single-unit-clusters-call
```

The benchmark was executed on a Linux system with 128 AMD EPYC 7502 32-Core Processors 1.5GHz and 256GB RAM. The benchmark ran with 5 warmup and 15 sample iterations for each of the projects with increasing number of cores from 1 to 24. Total runtime was approximately 6 hours. We also ran each project individually using 8 cores to gather per-unit statistics. The resulting files for all these can be found in `statix-benchmark/results/20210415`.

9.3 Results

The main results are the `*.speedup_summary.pdf` and `*.speedup_details.pdf` plots, which show the speedup achieved by using more cores. The speedup values are also saved in `*.speedup_data.csv` files.

Use the runtime plots to understand the scaling profile in more detail. The `*.runtime_summary.pdf` and `*.runtime_details.pdf` plots show the absolute runtime measured during the benchmark. The `*.runtime_top10.pdf` and `*.runtime_histogram.pdf` plots

from individual runs show the runtime of individual compilation units. The runtime of the whole problem is necessarily determined by the longest runtime of any individual unit. We found that the point where the scaling curve flattens, the total benchmark is close to the runtime of the longest running unit.

10 Sources: In Detail

This section gives pointers to the most interesting parts of the sources.

The implementation of the parallel framework can be found in `sources/statix/p_raffrayi`. The following files may be of interest:

- `src/main/java/mb/p_raffrayi/ITypeChecker.java` implements the *TypeChecker* interface from Algorithm 1.
- `src/main/java/mb/p_raffrayi/ITypeCheckerContext.java` implements the *CompilationUnit* interface from Algorithm 1.
- `src/main/java/mb/p_raffrayi/impl/TypeCheckerUnit.java` implements most of Algorithms 4–6.

The implementation of parallel Statix can be found in `sources/statix/statix.solver`. The following files may be of interest:

- `src/main/java/mb/statix/concurrent/UnitTypeChecker.java` implements the type checker for files, which in turn calls `src/main/java/mb/statix/concurrent/StatixSolver.java` which actually uses the provided API.

The Statix specification for the subset of Java can be found in `sources/java-front/lang.java.statics/`. It is located in the `trans` folder and follows roughly the structure of the Java Language Specification.

The sources of the benchmark projects can be found in `sources/java-evaluation`. For technical reasons the Java files have the file extension `.jav`.

Finally, the benchmark runner, based on JMH, can be found in `sources/statix-benchmark`. The Java code is located in the directory `statix.benchmark`. The directory `results` contains the result files, including for the last experiment of *20210430*, which is reported in the paper.