Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity (Artifact README)

1 Introduction

This artifact is provided to enable the results of all four research questions (RQ1 – RQ4) in our companion paper, i.e., the results in Table 1, Table 2, Figure 7 and Figure 8, to be reproduced. The artifact contains Scaler and Doop (a state-of-the-art whole-program pointer analysis framework for Java) to reproduce the results. This document describes how to use this artifact. For installing and setting up the artifact, please refer to `INSTALL.pdf`.

Note that in our paper, most of our experiments are carried out on a machine with 48GB of RAM. Using a machine with a smaller RAM may cause some analyses to run more slowly or even to the point of being unscalable. In addition, the results concerning analysis times need to be checked with caution due to the differences in the running environments used.

2 Content of Executable

The executable of this artifact is in the `artifact` folder in the directory where you unpacked the artifact package. Specifically, the `artifact` folder includes:

- `run.py`: A Python script for driving all the provided analyses.
- `scaler`: The folder containing our implementation of Scaler.
- `doop`: The folder containing Doop framework. It also contains the Java programs and the library to be analyzed in the evaluation.

3 Running Experiments

To run the experiments, please firstly follow the instructions in `INSTALL.pdf` to setup, and then change your current directory to the `artifact` folder.

3.1 Running Pointer Analysis

Running pointer analysis produces the results in Table 1 and Figure 8. The command to run pointer analysis is:

```
$ ./run.py   <ANALYSIS>|-all   <PROGRAM>   [-tst <TST>]
```

The `<ANALYSIS>` can be one of the following pointer analyses evaluated in our experiments:

`ci`, `2obj`, `2type`, `1type`, `scaler-pa`, `introA`, `introB`

The first four analyses are the baseline analyses provided by Doop. `scaler-pa` is the Scaler-guided pointer analysis. `introA` and `introB` are the two introspective pointer analyses.

The `<PROGRAM>` can be one of the following Java programs analyzed in our experiments:

jython, soot, pmd, briss, jedit, eclipse, findbugs, chart, luindex, lusearch.

For example, to use 2type to analyze eclipse, type:

```
$ ./run.py 2type eclipse
```

For SCALER-guided pointer analysis, i.e., scaler-pa, we provide the -tst option, to enable the user to run scaler-pa with different TST values and reproduce the results in Figure 8. Note that this argument is **optional** as shown in Figure 5 of our companion paper. When it is not specified, SCALER uses a default TST value of 30M (million), as described in Section 6 of the paper.

For example, to use scaler-pa to analyze eclipse with TST value of 60M (the user can specify the TST value directly, e.g., -tst 60000000, or by abbreviation for million, e.g., -tst 60M), just type:

```
$ ./run.py scaler-pa eclipse -tst 60M
```

When running scaler-pa, SCALER will be invoked, which produces the results in Table 2 and Figure 7. For examining the detailed results of SCALER, please refer to the "**Results of Running SCALER**" paragraph in Section 3.2 of this document.

For convenience, we provide the -all option to run all pointer analyses in Table 1 for a certain program. For example, to run all pointer analyses for eclipes, type:

```
$ ./run.py -all eclipse
```

To save user's time, this option will skip the cases when the analyses that are not scalable for the program (which costs $>$3 hours under our experimental setting as shown in Table 1). Besides, for the conventional context-sensitive analyses (i.e., 2obj, 2type and 1type), this option only runs the most precise one which is scalable (e.g., 2type for eclipse).

When running pointer analysis, the data in Table 1, i.e., the analysis time and the results of the clients used (#may-fail casts, #poly calls, #reach methods and #call graph edges) will be printed on the screen. In addition, all the detailed results of the clients (e.g., which casts may fail or which call sites are polymorphic, etc.) will be written to the directory artifact/output-client. For convenience, the results for different programs with different clients and analyses will be organized and outputted to different files. For example, eclipse-2type.mayfailcasts stores the detailed results for client may-fail casts, under analysis 2type for program eclipse.

3.2 Running SCALER

We also provide the command for the user to run SCALER solely (i.e., scaler, not SCALER-guided pointer analysis scaler-pa). This command will produce the results in Table 2 and Figure 7. The command to run SCALER solely is:

```
$ ./run.py   scaler   <PROGRAM>|-all   [-tst <TST>]
```

The <PROGRAM> option is the same as the ones in Section 3.1.

The <TST> is similar as in Section 3.1. It is the given TST value for SCALER. This argument is also **optional** as shown in Figure 5 of our companion paper. When it is not specified, SCALER uses a default TST value of 30M (million), as described in Section 6 of the paper.

For example, to use SCALER to analyze findbugs, type:

```
$ ./run.py scaler findbugs
```

For convenience, we provide the command to run SCALER for all the evaluated programs:

```
$ ./run.py scaler -all
```

To use SCALER to analyze `findbugs` with TST value of 60M:

```
$ ./run.py scaler findbugs -tst 60M
```

SCALER requires the points-to information produced by a pre-analysis. Thus when running SCALER, a context-insensitive points-to analysis will be automatically invoked at first as the pre-analysis as described in Section 3 of our companion paper. After the pre-analysis, the points-to information will be dumped for later use (by SCALER). For large programs such as `soot`, this would take several minutes.

Results of Running SCALER When running SCALER, the analysis time of SCALER (in Table 2), the given TST value (30M by default), and the selected ST_p value for each program (the numbers above the bars in Figure 7) will be printed on the screen. In addition, all the corresponding detailed results will be written to this directory:

```
artifact/output-scaler
```

In the above directory, for each program (say `<PROGRAM>`) and a specified TST value (say `<TST>`), SCALER will produce a file, `<PROGRAM>-ScalerMethodContext-TST<TST>.facts`, which stores the facts of the context-sensitivity variants selected by SCALER for the methods of `<PROGRAM>`. This file enables the user to inspect the results (bars) in Figure 7, and it will also be used by Doop to perform SCALER-guided pointer analysis.

3.3 Running All Experiments

We provide an option to run SCALER and all pointer analyses for all programs (except the unscalable cases as described in Section 3.2), to produce all data in Table 1, Table 2 and Figure 7 in one shot:

```
$ ./run.py -all
```

Although it is convenient, we recommend the user to try the commands in Sections 3.1 and 3.2 first, since this option is both time- and space-consuming by running a large amount of sets of experiments. In our experimental settings, this option spends about 17 hours and consumes about 190GB of disk space as DOOP keeps flushing all its results to the disk.