Getting Started

We show here the simplest way to (dry) run the artifact, which is via Docker. The alternatives are explained in the comments to setting up the environment. Every step shouldn't take longer than 5 minutes.

1. Extract the artifact tarball somewhere on the disk, and open a shell session in the root directory. The `ls` (accordingly, `dir` on Windows) should show you:

   ```
   Stability    pkgs    start    Overview.pdf    shell.nix    top-1000-pkgs.txt
   ```

2. From here, run Docker as follows:

   ```
   docker run --rm -it -v "$PWD":/artifact nixos/nix sh -c "cd artifact; nix-shell --pure"
   ```

   Note: running Docker may require root privileges. Other notes from the Steps To Setup section also apply.

3. Make sure (e.g. with `ls`) that you are in the artifact root (e.g. `ls` shows the `Stability` directory among others). Pull in dependencies of our code:

   ```
   JULIA_PROJECT=Stability julia -e 'using Pkg; Pkg.instantiate()'
   ```

4. To run our type stability analysis on one simple Julia package `Multisets`, do the following:

   a. Enter `start` directory and process the package:

      ```
      cd start
      ../Stability/scripts/proc_package.sh "Multisets 0.4.12"
      ```

      There should be a number of new files with raw data in `start/Multisets`, e.g.

      ```
      stability-stats-per-instance.csv
      ```

   b. Convert the raw data into a CSV file:

      ```
      ../Stability/scripts/pkgs-report.sh
      ```

      There should be a new file `start/report.csv`.

   c. Generate tables:

      ```
      JULIA_PROJECT=../Stability ../Stability/scripts/tables.jl
      ```

      You should see stability data about the Multisets package as an output:

      ```
      Table 1
      3×3 DataFrame
       Row │ Stats      Stable   Grounded
           │ String     Float64  Float64
      ─────┼──────────────────────────────
         1 │ Mean          73.0      55.0
         2 │ Median        73.0      55.0
         3 │ Std. Dev.      0.0       0.0

      Table 2
      1×7 DataFrame
       Row │ package    Methods  Instances  Inst/Meth  Varargs (%)  Stable (%)  Grounded (%)
           │ String     Int64    Int64      Float64    Int64        Int64       Int64
      ─────┼──────────────────────────────────────────────────────────────────────────────────
         1 │ Multisets        7         11        1.6            0          73            55
      ```

      Note that the format is the same as Tables 1 and 2 of the paper, only the numbers are computed over one small package.

d. Generate plots:

```
JULIA_PROJECT=../Stability julia -L ../Stability/scripts/plot.jl -e 'plot_pkg("Multisets")'
```

There should be a group of figures created under `Multisets/figs`. The ones that correspond to Figure 11 of the paper are

```
Multisets/figs/Multisets-size-vs-stable.pdf
Multisets/figs/Multisets-size-vs-grounded.pdf
```

You should be able to browse the figures using your host system's PDF viewer (all files created inside Docker container under the `/artifact` directory will be visible in your host file system).
The figures should be similar to ones in `start/Multisets/figs-ref` (provided for reference).
Note: some PDF viewers render the graphs very blurry. We know that Adobe Acrobat should work fine.

5. To shut down the Docker session, run `exit`.

List of Claims From Paper Supported by Artifact

This artifact aims to give exact directions on how to reproduce experiments reported in Section 5 (Empirical Study) of the Type Stability in Julia paper submitted to OOPSLA '21; in particular:

- Table 1,
- Table 2,
- Figure 11.

Contents of Artifact

Figure 1 describes the contents of our artifact. Some of the scripts and source files have instructional comments inside. We submit the Git history of the project: the submitted version lives on the `artifact` branch. The same repository is available on Github (prl-julia/julia-type-stability) but without the `pkgs` directory: this directory contains data about the exact Julia package versions we used to compute statistics, and also the full raw data produced by our analysis, which you can use to build the final tables without having to run the experiment on all 1000 packages. This directory takes 1.6 Gb space when uncompressed, the rest of the artifact fits into several megabytes including the Git history.

Step By Step Instructions

There are three parts to the rest of this document:

1. Metadata about our artifact and experiment as suggested by AEC Guidelines.
2. Discussion of environment setup.
3. Reproducing results from Section 5 of the paper.

Metadata

Hardware and Timings AEC guidelines suggest adding approximate timings for all operations.

First, our hardware specs. We run experiments in a virtualized environment on server hardware:

- 32 Intel (Skylake) CPUs — speed up our experiments quite a bit (nearly linear thanks to GNU parallel)
- 64 Gb RAM — shouldn't be critical to the experiment, but the more CPUs you have the more memory you need (our 32 CPUs consumed 15 Gb at some point)

```
├── shell.nix - system dependencies via Nix
├── Stability - our Julia package to analyze type stability
│   ├── Manifest.toml - machine description of our Julia package (1)
│   ├── Project.toml  - machine description of our Juila package (2)
│   ├── scripts
│   │   ├── proc_package.sh - run stability analysis on one package (produce raw analysis data)
│   │   ├── proc_package_parallel.sh - run stability analysis on several packages (batch mode)
│   │   ├── plot.jl - using raw data generate figures like Figure 11
│   │   ├── pkgs-report.sh - using raw data generate aggregate report (report.csv)
│   │   └── tables.jl - using report.csv render Tables 1 and 2 of the paper
│   ├── src
│   │   ├── pkg-test-override.jl - override Julia's test suite functionality to hook in our analysis
│   │   └── Stability.jl - code analyzing type stability
│   ├── startup.jl - convenience script to start Julia with our package loaded (1)
│   └── startup.sh - convenience script to start Julia with our package loaded (2)
├── pkgs
│   ├── ... - 1000 packages with descriptions ({Project, Manifest}.toml) and raw data precomputed
│   └── report.csv - report on 1000 packages used to generate tables and figures in the paper
├── start
│   └── Multisets - example package for the Getting Started phase
│       ├── figs-ref - reference figures for the Getting Started phase
│       ├── Manifest.toml - machine description of the package (1)
│       └── Project.toml - machine description of the package (2)
├── top-1000-pkgs.txt - list of 1000 most-starred Julia project
├── Overview.pdf - this file
├── .git - Git history of this project
└── .gitignore - we ignore the pkgs directory (see above)
```

Figure 1: Contents of Artifact

- 1.5 Tb disk space — same as RAM, shouldn't be critical but be sure to provision several Gb per processor.

Using that hardware we get:

- Reproducing Table 1 (1000 packages): 7 hrs
- Reproducing Table 2 (10 packages): 20 mins
- Reproducing Fig. 11 (heat maps): < 5 minutes

These numbers, again, are heavily dependent on the number of CPUs available on the system.

Expected Warnings AEC guidelines suggest mentioning expected warnings in the README.

1. During package analysis (e.g. reproducing Tables 1 and 2) Julia warns about us overriding its test-suite functionality. This is fine because that appeared to be the most convenient way of gathering statistics we need.

   ```
   WARNING: Method definition gen_test_code##kw(Any, typeof(Pkg.Operations.gen_test_code), String)
     in module Operations at /buildworker/.../julia/stdlib/v1.5/Pkg/src/Operations.jl:1316
     overwritten in module Stability at /artifact/Stability/src/pkg-test-override.jl:6.
     ** incremental compilation may be fatally broken for this module **
   ```

2. GNU parallel wants to be cited and prints the following message unless you run it dry with the `--citation` flag first. It is safe to ignore.

   ```
   Academic tradition requires you to cite works you base your article on.
   If you use programs that use GNU Parallel to process data for an article
   ...
   Come on: You have run parallel 13 times. Isn't it about time
   you run 'parallel --citation' once to silence the citation notice?
   ```

3. The main body of the experiment for Tables 1 and 2 uses Julia Package manager to get and run tests of various Julia packages. The package manager as well as test suites are rather verbose, so expect a lot of text on the screen. Some tests will fail, but this is expected: the idea is that no matter what the outcome of the test is, some Julia code will be compiled during the test, and that is the code that we analyze.

   Some test suites among the 1000 packages in our data set will fail fatally for various reasons (usually, dependencies-related). As noted in the paper, Section 5.1 (Methodology), out of the 1000, over seven hundred test suites do produce results that we can analyze.

Comments on Setting Up Environment

Our artifact has modest dependencies, and it should be possible to manually install them on a different machine; however, it is not necessary to do so because we provide two other essential pieces of infrastructure: a) automatic dependency management (via the Nix package manager), b) isolation from the host system (via Docker). Still, both of these pieces are optional.

Note: if the Docker approach from Getting Started section works good for you, it's not necessary to follow this section.

We first expand a little more on the setup and then provide the sequence of steps that get you in the right environment.

Dependencies

- Bash shell
- Julia v1.5.4
- GNU parallel
- `timeout` from GNU coreutils

Dependency Management via Nix and Reproducibility To make sure the versions of dependencies are the same as we used, and also to automate the process of getting them, we use the Nix package manager – a general package manager available on Linux, MacOS and Windows (via WSL).

The `shell.nix` file in the root of the artifact describes the dependencies we have. If the user has Nix installed and doesn't need isolation (e.g. believes that we don't `rm -rf` their home directory), they can use it directly without Docker and skip the section on Docker.

There is another kind of dependencies, though: the analyzed Julia packages, which are not covered by Nix or any other (known to us) general package manager. To get those, we use the Julia package manager `Pkg` (part of the Julia dependency). To generate figures from the paper as close as possible during the experiment, we supply the metadata about versions of the packages we analyzed. These metadata are stored in `pkgs` and `start` directories of the artifact. They have a set of subdirectories named after particular packages we analyzed, and every package subdirectory contains two files related to `Pkg`: `Project.toml` and `Manifest.toml`. Supplying these ensures that you will run the analysis on the same versions of the packages as we did, but it does not guarantee 100% reproducibility of the reported numbers because test suites of some analyzed packages contain non-determinism (naturally).

Isolation Plus Nix Dependency via Docker We suggest using Docker to avoid unexpected interactions with the host system. This also gets you Nix. In theory, you could use a Docker image without Nix and get all the dependencies there manually, or we could supply a custom Docker image with them (this would cost several hundreds megabytes in the artifact's real estate). But we don't discuss these configurations because Docker+Nix seems like a strictly better option.

Note that insider Docker, the commands are run under the root user. This makes life easier in one aspect: Julia package manager stores a lot of metadata and makes some of it inaccessible for cleanup in the end. This is no problem under reboot but if running outside Docker and without root privileges, this will eat up some disk space, proportional to the number of packages analyzed. For 1000 packages this can add up to hundreds of gigabytes. If you don't have that much disc space, run either in Docker or under root.

Steps To Setup We mark the first two steps as optional, but the simplest and most-likely-to-succeed path is to apply both of them. Alternatively, you can manually install the dependencies listed at the beginning of this section, open artifact root in the shell, and go to step 3.

1. [Optional, apply if Isolation is desired]
   Start a Docker container with Nix installed by executing the following in the root directory of the artifact:

   ```
   docker run --rm -it -v "$PWD":/artifact nixos/nix sh -c "cd artifact"
   ```

   Note 1: running the `docker` command may require root privileges on your system. We assume that you know how to use Docker on your system.

   Note 2: `"$PWD"` expands to the path of the current directory in most shells. You can check it with `echo "$PWD"`. If not, please use equivalent command for your shell or type in the path manually.

   Note 3 for Windows and MacOS users: note that Docker is sometimes reported to be slow with file operations concerning bind mounts (that's the `-v` flag) on these platforms. If you happen to use them and want to reproduce the experiment on 1000 packages, you may be better off with applying this manually inside a virtual machine with some Linux installation. Getting a Linux VM is not covered here because it's assumed to be a common knowledge.

2. [Optional, apply if Dependency Management is desired]
   Make sure the artifact files are in the current directory in your shell session (e.g. by executing `ls` in the shell). From there, enter the Nix shell by executing `nix-shell --pure`. Nix shell is a Bash shell that has all the dependencies specified in `shell.nix` visible (i.e. in `$PATH`).

3. [Sanity Checking] Make sure you have Julia and GNU parallel up: `julia --version && parallel --version`. Current directory should be artifact's root (e.g. `ls` should show Stability directory, shell.nix file, etc.).

4. Pull in dependencies of our Julia code:

   ```
   JULIA_PROJECT=Stability julia -e 'using Pkg; Pkg.instantiate()'
   ```

Reproducing Results from Section 5

As noted in the Timings section, reproducing Table 1 requires a lot of computational resources to analyze 1000 Julia packages. We suggest starting from only top 10 packages, which also provide data for Table 2. In the end of this section, we show how to process all 1000 packages (or less).

Instructions below assume the environment is set up, e.g. by applying the first 3 steps from the Getting Started section or following the discussion in the Comments on Setup section above.

1. Enter the `pkgs` directory in the root of the artifact and run the analysis on 10 packages:

```
cd pkgs
../Stability/scripts/proc_package_parallel.sh ../top-1000-pkgs.txt 10
```

This takes 20 minutes in our fully parallel setup (# CPUs > 10). This script (1) runs the analysis producing the raw data per package, and (2) runs the reporting script; this is equivalent is steps 4.a and 4.b from the Getting Started section. The resulting `report.csv` in the current directory should get a new time stamp.

2. Generate both tables as before:

   ```
   JULIA_PROJECT=../Stability ../Stability/scripts/tables.jl
   ```

   Table 1 will have significantly different numbers than Table 1 in the paper because we analyzed only 10 packages, not 1000.

   Table 2 should have mostly the same numbers as in the paper. Slight differencies in several lines are due to inherent non-determinism in the test suites of respective packages.

3. Reproducing Figure 11 of the paper:

   ```
   JULIA_PROJECT=../Stability julia -L ../Stability/scripts/plot.jl -e 'plot_pkg("Pluto")'
   ```

   There should be a group of figures created under `Pluto/figs`. The ones that we have on Figure 11 are

   ```
   Pluto/figs/Pluto-size-vs-stable.pdf
   Pluto/figs/Pluto-size-vs-grounded.pdf
   ```

Step 1 can be adjusted to run either on the whole 1000-packages set by omitting the 10 argument to `proc_package_parallel.sh` or to run on first n packages by replacing 10 with n.