Why We Eval in the Shadows

OOPSLA21 Artifact

Aviral Goel Pierre Donat-Bouillud Filip Krikava Christoph Kirsch
Jan Vitek

9.7.2021

Contents

Introduction

This is the artifact for the paper *Why We Eval in the Shadows* by Aviral Goel, Pierre Donat-Bouillud, Filip Krikava, Christoph Kirsch and Jan Vitek submitted to OOPSLA 2021.

In short, this paper analyzes the use of the `eval` function in R code. Using `eval` hinders static analysis and prevents compilers from performing optimizations. This paper aims to provide a better sense of how much and why programmers use `eval` in R. Understanding why `eval` is used in practice is key to finding ways to mitigate its negative impact. It is a large-scale study of more than 4.5M lines of R code.

Methodology

To better understand this artifact it is good to get familiar with the methodology that is presented in paper's Section 3. The following summarizes the main points about the corpus, the pipeline, and teh dynamic analysis that is used.

Corpus We look at three different corpora: the packages distributed with the R language (*base*), packages from the CRAN repository which contains the largest collection of R packages (*CRAN*), and scripts from the Kaggle website, an online platform for data science. Details about the corpus can be found in the paper in Section 3.1.

Pipeline This artifact presents the data analysis pipeline that we have build for this paper and used to gather the presented data. Conceptually, it consists of four main steps:

1. Download and installs the corpora.
2. Extracts and instruments all runnable code snippets from the installed packages and Kaggle scripts.
3. Run the instrumented code capturing all calls to the `eval` function.
4. Analyze the results.

The pipeline is implemented in a makefile with a few auxiliary scripts in R and bash. Details about the pipeline are presented in Section 3.2 of the paper.

Dynamic analysis The main component of the pipeline is an `eval` tracer. It is an R package (called `evil`) that uses our dynamic analysis framework to record all calls to `eval` at runtime. The framework is built on top of a modified GNU R 4.0.2 virtual machine (R-dyntrace) that exposes low-level callbacks for a variety of runtime events. Details about the dynamic analysis are in Section 3.3 of the paper.

Prerequisites

The artifact is packed in a docker image with a makefile that orchestrates various tasks. The following are the requirements for getting the image ready (the versions are just indicative - these are the ones that we worked with):

- GNU bash (5.0.17)
- Docker (20.10.2)
- git (2.24.1)
- GNU Make (4.2.1)

These instructions were tested on a fresh minimal installation of Ubuntu 20.04 LTS and OSX (11.4).

For the fresh Ubuntu installation we installed the requirements using the following:

```
sudo apt install git make docker.io
sudo usermod -a -G docker <username>
```

NOTE

- There seems to be a problem running the artifact on Apple M1 hardware, concretely `pandoc` which is used for rendering the analysis notebooks does not work (cf. #6960). The error is only manifested at the end when running the analysis:

```
------------------------------------------------------------------------
=> RENDERING corpus.Rmd into corpus.html
------------------------------------------------------------------------
/R/R-dyntrace/bin/R --quiet --no-save -e 'rmarkdown::render("corpus.Rmd", output_file="/Users/ck/ev
> rmarkdown::render("corpus.Rmd", output_file="/Users/ck/evalr-experiment//run/analysis/corpus.html
Killed
Error in strsplit(info, "\n")[[1]] : subscript out of bounds
Calls: <Anonymous> ... pandoc_available -> find_pandoc -> lapply -> FUN -> get_pandoc_version
In addition: Warning message:
In system(paste(shQuote(path), "--version"), intern = TRUE) :
  running command ''/usr/bin/pandoc' --version' had status 137
Execution halted
Makefile:29: recipe for target 'corpus' failed
make[1]: *** [corpus] Error 1
make[1]: Leaving directory '/Users/ck/evalr-experiment/analysis'
Makefile:661: recipe for target 'package-analysis' failed
make: *** [package-analysis] Error 2
```

Not in the artifact

Because of the licensing issues, we can redistribute neither the code nor the data from the Kaggle website. The rest of this document will therefore only focus only on base and CRAN corpora.

Organization

This document consists of two major sections:

1. *Getting started guide* which will guide you through the process of setting the environment and making sure the artifact is functional, and
2. *Step-by-Step instructions* which will guide you through the process of reproducing the data presented in the paper.

Reported times were measured on Linux 5.12 laptop with Intel i7-7560U @ 2.40GHz and 16GB RAM.

Evaluating the artifact

The quantitative results in the paper that are the subject of this artifact are presented in Sections 4 and 5. They are in form of tables and figures as well as numbers embedded in the text. All are generated from R markdown notebooks. Concretely, the figures are saved into PDF files, tables into TEX files included in the paper, and the rest of the numbers also in TEX files that use `\newcommand` macros (e.g., `\newcommand{\CranAvailablePackages}{15,962\xspace}`). In the relevant sections, we provide concrete links to all the figures and tables.

The paper presents a large-scale study using over 15K CRAN packages. Redoing the very same experiment is a resource-intensive task. It took over 60 hours on a cluster of 3 servers (cf. Figure 2 in the paper). While you are free to do the same, we do not expect you to redo the entire experiment. Instead, we provide a preprocessed data so you could run just the data analysis. However, the artifact should work on any number of CRAN packages, so feel free to experiment as you find fit.

Getting started guide

In this section, we will go through the steps of getting the artifact up and running. Before you begin, make sure you have a functional docker on your system by running:

```
docker run --rm hello-world
```

It should eventually print `Hello from Docker!`. If it does not, your docker environment is not properly configured and the following instructions will not work.

A common problem with a newly installed docker is missing permission. If you got the following error:

```
Got permission denied while trying to connect to the Docker daemon socket at
unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/json:
dial unix /var/run/docker.sock: connect: permission denied
```

Simply add your username into the `docker` group (e.g. `sudo usermod -a -G docker <username>`)

1. Get a copy of the repository

```
git clone -b oopsla21 ssh://git@github.com/PRL-PRG/evalr-experiment
cd evalr-experiment
```

IMPORTANT

- **Please make sure you use the `oopsla21` branch.**
- **From now on, all of the commands should be run inside the cloned repository.**

The artifact repository should look like this

```
.
  analysis             # R markdown notebooks used for analysis
  docker-image         # docker image source code
  Makefile             # data analysis pipeline
  Makevars             # basic pipeline configuration
  packages-core.txt    # list of R core packages (used only internally)
  README.md            # this readme
  scripts              # utility scripts for the pipeline
```

2. Get the docker image

For ease of use, we pack all the dependencies in a docker image. We also use the very same image to run the experiment in a cluster to make sure each node has the same environment.

There are two options to get the image: pull it from the docker hub or build it locally.

1. Pulling from docker hub:

   ```
   docker pull prlprg/project-evalr:oopsla21
   ```

   This might take a few minutes, the image has ~4GB.

2. Building the image locally:

   ```
   make docker-image
   ```

   It takes about an hour as it needs to install several R packages.

3. Run the docker container

The pipeline will run inside the docker container. The following will spawn a new docker container giving you a bash prompt in the directory that contains the repository:

```
make shell
```

On Apple M1 you might need to run:

```
make shell SHELL_EXTRA='--platform linux/amd64'
```

A few details about how the container is run:

- It sets the internal docker container user (called `r`) to have the same UID and GID as your username. This is to prevent any permission problems.
- It mounts the artifact repository folder to the very same path as it is on your machine (i.e. if you cloned the repository to `/home/alicia/Temp/evalr-experiment`, the current working directory in the container will be the same).

IMPORTANT

From now on, all of the commands should be run inside this docker container!

Issuing `ls` should show the same structure as above with two additional directories:

```
...
  CRAN                # R package sources
  library             # R package binaries
...
```

Right now they are empty. They will be filled as we install packages for the experiment.

NOTE

- The container comes with a very limited set of tools, if you find you are missing something, you can install anything from Ubuntu repository using the usual: `sudo apt install <package>`. However, keep in mind that the container will be removed the moment you exit from the shell prompt.

---

4. Create a sample corpus

The pipeline works with a corpus of R packages. This corpus is defined in a file `packages.txt` with one package name per line. To run the pipeline, we first need to create such a file. You could try any R packages that are compatible with R 4.0.2 and are available in CRAN. The more packages, the longer will it take.

We recommend (to meet the 30min deadline) starting with the following single package:

```
echo withr > packages.txt
```

Next, we need to install the packages:

```
make package-install
```

This will install the package including all of their dependencies. By the end you should see something like (the versions might differ if a package has been updated in the meantime):

```
...
------------------------------------------------------------------------
=> Extracting package source
------------------------------------------------------------------------
- withr_2.4.2.tar.gz
```

The installed packages will be placed in the `library` and their sources under `CRAN/extracted`.

NOTE

- The `packages.txt` can be edited in both the container (using `vim`) or any editor on your local machine as the repository is mounted inside the container.

---

5. Run the eval tracer

The dynamic analysis that traces the eval calls is run using:

```
make package-trace-eval
```

Among others, it will do the following:

- Extracts package metadata (*package-metadata*)
- Extracts and instruments runnable code from packages (*package-runnable-code-eval*)
- Finds `eval` call sites from package source code (*package-evals-static*)
- Runs the package code while tracing the calls to `eval` (*package-trace-eval*)

For each of the long running task, there should be a progress bar which shows an estimate remaining time. Allow at least 15 minutes. By the end, it should finish with something like:

```
...
------------------------------------------------------------------------
=> MERGING calls.fst
------------------------------------------------------------------------
...
make[1]: Leaving directory '/home/krikava/Research/Projects/evalR/artifact'
```

After it finishes, there should a new `run` directory with the following content:

```
run
  package-evals-static
  package-metadata
  package-runnable-code-eval
  package-trace-eval
  corpus.fst
  corpus.txt
  package-evals-to-trace.txt
  package-scripts-to-run.txt
```

Each directory contains results for each of the tasks that were run. To quickly check the results of the tracer, you can run:

```
./scripts/parallel-log.R run/package-trace-eval
```

which should print out something like:

```
Duration: 183.94 (sec)
Average job time: 18.15 (sec)
Number of hosts: 1
Number of jobs: 39
Number of success jobs: 39 (100.00%)
Number of failed jobs: 0 (0.00%)

Exit codes:
  0:    39 (100.00%)
```

This says that it successfully ran all 39 extracted R programs in about 3 minutes (18 seconds average is OK as jobs run in parallel, cf. below). We will go over the details in the next section.

NOTE

- We use GNU parallel to run certain tasks in parallel. By default, the number of jobs will equal the number of available cores. If you want to throttle it down, set the `JOBS` variable to a lower number (e.g. `make package-trace-eval JOBS=2`). You can see the current value by running `make info`.
- Next time you run `make package-trace-eval` it will only run the tracer unless `packages.txt` was changed in the meantime which will trigger regeneration of the auxiliary data.
- If anything goes wrong, you can always start from scratch by removing the `run` folder.
- The results are either plain text files, CSV files, or for the larger output we use fst format which provides a fast data frame compression based on Facebook's zstd library. To view the content of an fst file, you could use the `scripts/cat.R` utility (e.g. `./scripts/cat.R run/package-trace-eval/writes.fst`)

---

6. Run the analysis

Right now you should have the raw data. Next, we need to preprocess them (mostly a data cleanup) and run the actual analysis. The analysis is done in R using R markdown notebooks.

First, we run the data preprocessing

```
make package-preprocess
```

This will re-extract runnable code from packages (*package-runnable-code*), this time without any instrumentation, and run it (*package-run*) so it can compute the tracer failure rate.

It should take about 2 minutes and the results will be in `run/preprocess/package`. The content should like like this:

```
run/preprocess/package
   corpus.fst
   corpus.txt
   evals-static.csv
   normalized-expressions.csv
   run-log.csv
   runnable-code.csv
   side-effects.fst
   summarized-externals.fst
   summarized.fst
   trace-log.csv
   undefined.fst
```

This is the source for the next step, the analysis. We will run (*knit*) six analysis notebooks (from the `analysis` folder):

```
make package-analysis
```

It should take a couple of minutes and the results will be in `run/analysis`:

```
run/analysis/
   paper
       img
           package_calls_per_run_per_call_site.pdf  # Figure 7b
           package_combination_minimized.pdf
           package_eval_calls_per_packages.pdf
           package_events_per_pack_large.pdf        # Figure 9b
           package_events_per_pack_small.pdf        # Figure 9a
           package_size_loaded_distribution.pdf     # Figure 8
           pkgs-eval-callsites-hist.pdf             # Figure 3
           se-types.pdf
           traced-eval-callsites.pdf                # Figure 6
       tag
            corpus.tex
            package_normalized_expr.tex              # Table 1
|           package_environments.tex                # Tables 2, 3, 4, 5
|           package_provenance.tex                  # Table 6
           package_usage_metrics.tex                 # Figures 4, 5, 7a
           side-effects.tex
           table-se-target-envs.tex                  # Table 7
           table-se-types.tex                        # Table 8
   corpus.html
   normalized.html
```

```
package-usage.html
side-effects.html
environments.html
provenance.html
```

There are two results:

1. The HTML files that contain the actual analysis
    - `corpus.html` is mostly used for Section 3.1.
    - `package-usage.html` contains data for the CRAN dataset and is used for Section 4.
    - `normalized.html` is used for Section 5.1.
    - `environments.html` is used for Section 5.2.
    - `provenance.html` is used for Section 5.3.
    - `side-effects.html` provides data for Section 5.4.
2. The files generated in the `paper` directory are used for typesetting the paper.
    - The `img` sub-directory contains all the plots included in the paper.
    - The `tag` sub-directory contains latex tables and *tag* files - latex macros for each of the numbers that is used in the paper.

You can view the files from your machine. Please note that all the data are base on just a single-package corpus and thus some metrics are not relevant.

The following is the list of the results that we include in the paper (the figure/table headings should be clickable links):

FIGURES

- Figure 3: CRAN `eval` call sites
- Figure 4: CRAN `eval` call frequency
- Figure 5: CRAN `eval` variants
- Figure 6: `eval` call sites coverage
- Figure 7a: Normalized calls - all
- Figure 7b: Normalized calls - small
- Figure 8: Loaded code
- Figure 9a: Instructions per call - small
- Figure 9b: Instructions per call - large

TABLES

- Table 1: minimized expressions
- Table 2: kinds of environments per site
- Table 3: function offset for function environments
- Table 4: wrapper environments (synthetic)
- Table 5: multiplicities, i.e. how many kinds of environments there are per site
- Table 6: provenance of the expression argument of `eval`
- Table 7: Target environments for side-effects
- Table 8: Types of `eval` side-effects

Congratulations! If you managed to get this far, you essentially analyzed the use of `eval` for a single CRAN package.

Thank you for evaluating the artifact!

Step-by-Step instructions

In this section, we provide additional details about how to trace `eval` calls for the R base libraries and how to reproduce the findings presented in the paper.

For the steps in this section, we assume that you have completed all the steps from the getting started guide and are in the bash prompt in the docker images (i.e. executed `make shell`).

Tracing eval calls in base

Next to CRAN, we also report on the use of `eval` in the core packages that are part of the R language. The reason why we treat them separately is that they are relatively stable, part of the language itself, written by R core maintainers, and finally, there are relatively few `eval` call sites. Nevertheless, they are also heavily exercised as there is hardly any R code that would execute without calling `eval` from core libraries.

To collect information about the base usage of `eval` we do an isolated run of a subset of the extracted programs from CRAN packages while tracing only the `eval` call sites presented in core packages.

Reusing the extracted programs from the getting started guide, we can trace base using the following:

1. Run the tracer with base evals only
   ```
   make base-trace-eval
   ```
2. Preprocess base
   ```
   make base-preprocess
   ```
3. Run the analysis
   ```
   make base-analysis
   ```

The results are in `base-usage.html`.

NOTE

- The number of programs it will run is controlled by the `BASE_SCRIPTS_TO_RUN_SIZE` environment variable, which is by default 25K. It will therefore run-up to that number of programs.

Reproducing paper findings

To redo the same experiment as we report in the submitted paper, one only needs to get all CRAN packages and put them in the `packages.txt` file.

```
R -q --slave -e \
  'cat(available.packages(repos="https://cloud.r-project.org")[, 1], sep="\n")'
```

However, the experiment is rather lengthy, in our cluster of three servers, each with 2.3GHz Intel Xeon 6140 processor with 72 cores and 256GB of RAM, it took over 60 hours. You can also rerun the experiment on a subset of CRAN. For example:

1. Clean the run folder
   ```
   rm -fr run
   ```
2. Create a corpus of 10 randomly selected CRAN packages
   ```
   R -q --slave -e \
     'cat(available.packages(repos="https://cloud.r-project.org")[, 1], sep="\n")' | \
     shuf -n 10 > packages.txt
   ```

3. Install the packages
   ```
   make package-install
   ```
4. Run the tracer
   ```
   make package-trace-eval base-trace-eval
   ```
5. Run the preprocessing
   ```
   make package-preprocess base-preprocess
   ```
6. Run the analysis
   ```
   make package-analysis base-analysis
   ```

The results will be in the same files as indicated above.

NOTE

- It might take up to a few hours depending on the selected packages.
- The final package count might be smaller as not all packages use evals. Some packages could also be filtered out because they cannot be installed (missing some native dependencies) or they do not contain R code.
- By default, there is 35 minutes timeout for all the tasks that run in parallel (e.g., extracting package metadata, running R programs). It can be adjusted by the `TIMEOUT` environment variable.
- R does not provide any mechanism for pinning package versions. This means that even if you try all the CRAN packages, the results could be slightly different from ours as the package evolves. However, the general shape should be the same.
- Clean the `run` folder every time you experiment with a new corpus.

As an alternative, we provide the preprocessed data on which you can run the analysis. This is the result of running:

`make package-preprocess base-preprocess kaggle-preprocess`

on the entire corpus.

1. Download the data (~180MB)
   ```
   wget -O run-submission.tar.xz https://owncloud.cesnet.cz/index.php/s/O2ntsqPufhKRObv/download
   ```
2. Extract the archive (~1.5GB)
   ```
   tar xfvJ run-submission.tar.xz
   ```
3. Run the analysis using the new full corpus (~40 minutes)

   This is again done by make. The only thing that we need to do is to tell make to use a different `run` directory, i.e. `run-submission`:
   ```
   make analysis RUN_DIR=$PWD/run-submission
   ```

The results will be generated in `run-submission/analysis` and they follow the very same structure as before. The following are links for convenience.

NOTEBOOKS

- `corpus.html` is mostly used for Section 3.1.
- `package-usage.html` contains data for the CRAN dataset and is used for Section 4.

- `normalized.html` is used for Section 5.1.
- `base-usage.html` contains data for the base dataset and is used primarily for Section 4.
- `kaggle-usage.html` contains data for the Kaggle dataset and is used primarily for Section 4.
- `environments.html` provides data for Section 5.2.
- `provenance.html` provides data for Section 5.3.
- `side-effects.html` provides data for Section 5.4.

FIGURES

TABLES
