OOPSLA20 Artifact - Designing Types for R, Empirically

Introduction

This is the artifact for the OOPSLA 2020 paper *Designing Types for R, Empirically*. The aim is to show the tools that were developed to infer and assert types for R functions and R packages.

The artifact is composed of two parts:

1. a *getting started guide* that contains the setup instructions and a small experiment to verify that the artifact is usable, and
2. a *step-by-step instructions* how to run the tools developed for this paper and how to reproduce the data reported in the paper.

Note: This document contains output from executing R code whose line width spans over what can comfortably fit on an A4 format in the PDF. Because of that, the evaluation of some of the code blocks is disabled in the PDF. It is therefore much nicer to **work with the HTML version or directly have it opened in RStudio** (cf. below).

Requirements

The pipeline depends on a number of tools and R packages:

- bash
- git
- GNU parallel >= 20190322
- GNU bison >= 3.5
- GNU make >= 4.1
- R (https://cran.r-project.org) == 3.5.0
- R-dyntrace (https://github.com/PRL-PRG/R-dyntrace/tree/r-3.5.0) 3.5.0
- a number of our R packages with their dependencies
    - contractr (https://github.com/PRL-PRG/contractr/tree/typer-oopsla20) - runtime type assertions
    - injectr (https://github.com/PRL-PRG/injectr/tree/typer-oopsla20) - for injecting code into R functions
    - propagatr (https://github.com/PRL-PRG/propagatr/tree/typer-oopsla20) - tracing type usage
    - runr (https://github.com/PRL-PRG/runr/tree/typer-oopsla20) - running the experiments
    - tastr (https://github.com/PRL-PRG/tastr/tree/typer-oopsla20) - grammar and parser for our R types

and of course a corpus of R packages to be run.

To make it more convenient we have build a docker image that has all these dependencies installed. The image can be pulled directly from Docker HUB (https://hub.docker.com/r/prlprg/oopsla20-typer) or built locally. To use the image, you will need:

- git
- bash, and
- docker community edition (https://docs.docker.com/install/) version 18+.

This artifact requires about ~20GB of free space, depending on the number of packages that should be analyzed. It has been tested on Linux (Manjaro 19 and Ubuntu 18.04).

Getting Started Guide

For the initial kick-the-tires phase, please go over the following steps to determine if the artifact is usable in your environment.

Clone the artifact git repository

In a terminal, run the following to get a copy of the artifact repository:

```
git clone https://github.com/PRL-PRG/OOPSLA20-typer-artifact
cd OOPSLA20-typer-artifact
```

We will refer to this directory as `$REPO` .

The artifact has the following structure:

```
$REPO
├── docker-image                     -- files needed to build the docker image
├── README.Rmd                       -- this readme in R markdown format
├── README.html                      -- this readme in rendered into HTML
├── README.md                        -- this readme in rendered into markdown
├── run.sh                           -- script to run Rstudio
└── typeR                            -- directory in which the type inference pipeline runs
    ├── data-corpus.tar.xz           -- archived data from analyzing the full 412 package corpu
s
    ├── in-docker.sh                 -- a helper script executinig given command in a container
    ├── Makefile                     -- makefile orchestrating the type inference pipeline
    ├── notebooks
    │   ├── corpus-analysis.Rmd      -- data points for paper's Chapter 5
    │   ├── evaluation-asserts.Rmd   -- data points for paper's Chapter 6.3
    │   ├── evaluation.Rmd           -- data points for paper's Chapter 6.{1,2}
    │   └── inc                      -- auxiliary R files for data analysis
    ├── packages-corpus.txt          -- list of the 412 packages
    ├── packages-small-corpus.txt    -- a smaller corpus to be run during the artifact evaluati
on
    ├── packages-tiny-corpus.txt     -- a tiny corpus to be run during the kcik the tires phase
    ├── scripts                      -- auxiliary scripts
    └── type-analyzer                -- type analyzer source code
```

Run RStudio from the artifact docker container

To make the artifact evaluation a bit more convenient, the docker image includes RStudio (https://rstudio.com), a popular R IDE. The following command will pull the docker image from Docker HUB and start an instance of RStudio on port 8787. If you need to use an alternative port, you can specify it using `-p PORT` argument to the `run.sh` script.

```
./run.sh
```

Once you see an output like:

```
[services.d] starting services
[services.d] done.
```

you should be able to access RStudio in your browser at http://localhost:8787 (http://localhost:8787).

To terminate it, simply interrupt the process by pressing `Ctrl-C` / `Command-C`.

Open this readme (`README.Rmd`) in RStudio

The rest of the steps in this guide can be done directly in the RStudio. To do that, open this readme file `README.Rmd` file by either navigating to `File -> Open File...` menu item

The file is written in Rmarkdown (https://rmarkdown.rstudio.com/) (or Rmd), also referred to as notebooks. It is essentially a markdown document with code snippets. These snippets can be run directly from RStudio. Rmd files can be also rendered (*knitted*) to various output file formats running all code snippets and embedding their outputs directly to the resulting file. Next to this readme file, we use several of the Rmd files to analyze the data for the paper.

The rest of the R code snippets can be run from within R by either clicking the play icon next to the snippet or by placing cursor somewhere inside the snippet and pressing `Ctrl+Enter` / `Command+Enter`.

Check that RStudio can load contractr package.

First, we need to make sure RStudio can load the `contractr` package.

```
library(contractr)
```

You should see an output similar to this:

```
Loading required package: roxygen2
Added contract to 118 roxygen2 function(s)
No type declarations found for package stats
No type declarations found for package graphics
No type declarations found for package grDevices
No type declarations found for package utils
No type declarations found for package datasets
No type declarations found for package methods
No type declarations found for package base
No type declarations found for package contractr
```

What is means is discussed in a later section.

Infer types for one package

For the last test, we will try to run the type inferring pipeline for a single package: stringr (https://cran.r-project.org/web/packages/stringr/index.html).

This has to be done in a terminal. Navigate to the `$REPO/typeR` and run the following:

```
./in-docker.sh make clean all PACKAGES_FILE=packages-tiny-corpus.txt
```

Note: Even though it just one package, the pipeline runs all its reverse dependencies to capture type check assertions which might take some time. On a Linux laptop (i7-7560U, 16GB RAM) it takes ~10 minutes to run.

The `make` command must be prefixed with `in-docker.sh` as we want to run it inside a docker container. The `$REPO/typeR` directory is mounted into `/home/rstudio/typeR` inside the container. All files stored there will be persisted.

The `PACKAGES_FILE` parameter specify which package corpus we want to use. A package corpus is a simple text file with one package name at a time. In general, you could specify any package, but to keep the docker image size in a reasonable bounds, we only include a small subset of the 15K CRAN packages.

The result should be `$REPO/typeR/data` with the following files:

```
data
├── assertions-all.csv
├── assertions-failed.csv
├── type-analysis
│   ├── merged.csv
│   └── packages
│       └── stringr.csv
├── TYPEDECLARATION
│   └── stringr            -- the actual inferred types for contractr
└── type-traces.csv
```

- `data/assertions-all.csv` : file contains all the assertions.
- `data/assertions-failed.csv` : file contains only failed assertions with additional details for debugging.
- `data/type-analysis/packages/<package>.csv` : file contains information about type signatures of `<package>` functions.
- `data/type-analysis/merged.csv` : file contains information about type signatures of functions from all packages.
- `data/TYPEDECLARATION/<package>` : files contain contractr type declarations for all recorded functions for a given package.
- `data/type-traces.csv` : file contains type traces obtained from dynamic analysis of package code.

Once completed, you can see the inferred types for the stringr package:

```
navigateToFile("~/typeR/data/TYPEDECLARATION/stringr")
```

This concludes the getting started guide and the kick the tires part.

Step-by-step Instructions

In this section we will provide more details about the tooling and the type inference pipeline. We start with a quick tutorial about how the contractr package work. Next, we will infer types for a small corpus. Finally, using the original data, we will reproduce the data points provided in the paper.

Quick Tutorial

The best is to interactively follow the tutorial in RStudio, running the code snippets one by one.

contractr

The `contractr` library helps generates runtime contracts from function type signatures. It modifies the function body to check for function argument and return value types. Let us look at how this library works.

First we load `contractr` . It depends on the R package `roxygen2` , which gets loaded as well. `contractr` comes equipped with type declarations (https://github.com/PRL-PRG/contractr/tree/typer-oopsla20/inst/TYPEDECLARATION) for 395 packages, `roxygen2` being one of them. As shown in the console output below, when `roxygen2` is loaded, `contractr` automatically inserts contracts to it's 118 functions. In general, `contractr` sets up package load hooks. So if any of the 395 packages are loaded, `contractr` automatically adds contracts to their functions. If no type declarations are available for a package, `contractr` reports that as well.

```
library(contractr)
```

Now, we load the `stringr` library, for which `contractr` has type declarations.

```
library(stringr)
```

The aim of `contractr` is to insert runtime type checks and provide meaningful error messages when the checks are violated. Let us call one of the `stringr` functions and observe `contractr` in action. For this, we choose the `str_count` function from `stringr`. This function counts the number of matches of a pattern in each element of a string vector (character vector in R speak). Here is an example of a type-correct interaction withe the `str_count` function.

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_count(fruit, "a")
```

Now, we will pass a pattern value with incorrect type, a double instead of a character. Firstly, as expected the function terminates with an error because the pattern argument is not supposed to be a numeric value. Secondly, we observe two warning messages. These warning messages are provided by `contractr`. The first message originates from the type checking of the `pattern` argument of `str_count`. The error message contains the following bits of information:

- Parameter name and position
- Function name
- Expected type
- Actual type
- Stack trace

Since the `str_count` function is called from the top-level, the stack trace only has a single frame, the `str_count(fruit, 1)` call.

The second warning message originates from a call to the `stringr::type` function from within the definition of `str_count` function. It turns out that passing a numeric argument to `str_count` also violates the type declaration of `type` function. As can be seen from the warning message, the `trace` has size 2 with `type(pattern)` being called from `str_count(fruit, 1)`.

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_count(fruit, 1)
```

An important point to note here is that `contractr` reports type mismatches as warnings. Unfortunately, if there are too many warnings, R collects them together and reports them at the end of the program. It may be desirable to report the mismatch when it happens and halt the program at that point. `contractr` allows this behavior to be configured using the `set_severity` function. The default severity is set to `"warning"`. It can be set to `"error"` to report type mismatch as an error and stop the program. With this setting, execution halts with the type mismatch error, so only the first type mismatch is reported.

```
set_severity("error")
str_count(fruit, 1)
```

Severity can also be set to `"silence"` to ignore all type mismatches. With this setting, the execution halts with the error message from the function execution.

```
set_severity("silence")
str_count(fruit, 1)
```

We set the severity back to `"warning"` for the rest of the session.

```
set_severity("warning")
```

Let’s look at how the function definition is modified by `contractr` to enable the type mismatch checks. If we type the function name on the R console, it prints out the definition of the function as it parsed from the package code.

```
str_count
```

However, `contractr` modifies this definition internally, without affecting the printed representation of the function definition. The modified function body can be obtained using the `body` function. As shown below, `contractr` adds two code blocks at the top of the body. The `on.exit` block attaches a function exit handler to check the type of the return value. The next block checks the type of all the arguments. The real heavy-lifting happens inside native code. The blocks immediately call C functions (prefixed with `C_`) using the `.Call` function. The reason is that `contractr` has to get around R's laziness semantics for argument type checking. Due to laziness, the arguments to a function call are unevaluated code thunks (promises). To maintain non-strict evaluation semantics, `contractr` inserts type-checking and error-reporting logic inside these thunks. It would have been easier to evaluate thunks at this point and check the type of arguments values but that would modify the semantics and break code by prematurely evaluating arguments which may not even be evaluated or perhaps evaluated in a specific order.

```
body(str_count)
```

`contractr` comes equipped with type declarations for 395 packages. Furthermore, type declarations can be provided during package development as part of function documentation as shown below in the definition of `add3` function. All type declarations from the `@type` tag are exported to `TYPEDECLARATION` file inside the package that during the package's installation. These files are automatically read by `contractr` when the package is loaded.

```
#' NOTE: running this block has no effect, it has be part of package definition
#' @type <int> => int
add3 <- function(x) {
  x + 3
}
```

`contractr` also provides API to inject type annotations for functions. This is useful for adding contracts to functions defined interactively on the console. Note how `contractr` provides type errors for return types and all parameter types.

```
f <- function(x, y) {
    x + y - 1L
}
insert_contract(f, "<int, int> => int")

f(1L, 2L)

f(1, 2)

f(1L, 2)

f(1, 2L)
```

A user may wish to get all the contract assertions at the end of the sessions. `contractr` provides an API, `get_contracts`, to query all contract assertions. The following interaction outputs a large data frame. Each row is a contract assertion (failed or succeeded). The columns contain information about the assertions.

```
get_contracts() %>% str()
```

`clear_contracts` API clears the internal record of contract assertions.

```
clear_contracts()
get_contracts()
```

For large programs, a user might be interested in selectively capturing and ignoring contract assertions. For this, `contractr` provides two functions, `capture_contracts` and `ignore_contracts` to execute code blocks by enabling and disabling contract assertion respectively. The two functions can be arbitrarily nested. `capture_contracts` returns an R list with two fields, `result` field contains the result of evaluating the code block and `contracts` field contains the data frame of all contract assertions from the code block. `ignore_contracts` returns just the result of evaluating the code block as it does not perform any contract assertions.

In this code block, type checking of argument and return values for `f(1, 2)` will not happen. Only `f(1L, 2)` will be type checked.

```
result <- ignore_contracts({
    capture_contracts({
        f(1L, 2)
    })
    f(1, 2)
})
result
```

In this code block, the behavior is reversed.

```
result <- capture_contracts({
    ignore_contracts({
        f(1L, 2)
    })
    f(1, 2)
})
result
```

In summary, this section has described the design and use of `contractr` library and discussed the following salient aspects of its design.

- Error messages
- Contract violation severity (silent, warning and error)
- Addition of type annotations as `roxygen2` tags
- Injection of type declarations for custom user-defined functions
- API to get and clear contract assertions
- API to enable and disable contract assertions for code blocks

Inferring Types from R Code

In this section we run the type inference pipeline on a small corpus of 5 packages:

- assertthat
- dbplyr
- glue
- lubridate
- stringr

The reason why we run only 5 packages is that (1) the dynamic tracing is quite resource intensive as all invocations are recorded and (2) running 12M of R lines (runnable code of the corpus + reverse dependencies) takes a lot of time.

Running the small corpus is similar to the tiny one in the above getting started guide:

```
./in-docker.sh make clean all
```

This time, we do not need to not need to specify the `PACKAGES_FILE` as the `packages-small-corpus.txt` is the default.

The type inference pipeline does the following:

1. `package-functions` : get information about all exported functions from a package (`runs/package-functions/functions.csv`),
2. `propagatr-code` : extract the runnable code from a package and wrap it inside a tracing block,
3. `propagatr-run` : run the code using R-dyntrace (https://github.com/PRL-PRG/R-dyntrace), a modified R VM with a support for dynamic tracing (cf. more information (https://zenodo.org/record/3625397#.Xy-9pWNfgUE)) recording all types seen at function invocations,
4. `consolidate-types` : consolidate recorded types,
5. `package-revdeps` : find packages' reverse dependencies,

6. `revdeps-code` : extract runnable code from these dependencies, and finally
7. `revdeps-run` : run this code with contractr enabled using type definitions inferred in the step 4.

Each of this task has a target in the makefile (`~/typeR/Makefile`). The intermediate results of these steps are stored in `~/typeR/run` directory. We use GNU parallel (https://www.gnu.org/software/parallel/) and runr (https://github.com/PRL-PRG/runr/tree/typer-oopsla20).

The makefile has a few arguments: - `TIMEOUT` set to 30 minutes by default - if any of the above tasks per package takes longer, it will be terminated - `JOBS` the number of jobs to be run in parallel. The default for this artifact is 1.

Redoing the Paper Experiment

In this section, we present an R Notebook capable of reproducing the figures and key numbers seen in the paper.

To obtain the data for the paper, we conducted an experiment wherein we chose a subset of CRAN packages with > 5 reverse dependencies and > 65% code coverage. We ran our type tracing tool on the runnable test, example, and vignette code for these packages, and obtained types for exported package functions. Then, we loaded those types into the aforementioned `contractr` tool, and ran the test, example, and vignette code of the clients of our core corpus. We recorded contract assertion successes and failures during this stage of the evaluation.

As you may guess, the data we worked with is too large to be reasonably processed on standard computers. Our data collection and processing pipeline ran on the order of days on a server with 72 cores and 256GB of RAM. To that end, we packaged our data and include it with this artifact. The data itself is liable to be useful to any future type-based research of R.

To rerun the analysis, first we have to extract the data for the entire corpus:

```
./in-docker.sh make data-corpus
```

These should gives us similar files as in the previous section where we only analyzed a few packages. The additional `cran` directory and `corpus-details.csv` include package metadata obtained for the entire CRAN and the corpus of 412 packages respectively. This is used in the next two sections. The reason why we include them here is that these were the most resource intensive to generate. This is because we needed to get the information about package code coverage and package reverse dependencies code coverage. It on itself took about 2 days of run on 72 cores machine.

All results are store in `~/typeR/evaluation` directory.

Expressiveness & Robustness

In this section we show how to generate data presented in *Section 6.1 Expressiveness* and *Section 6.2 Robustness*. We do this in a separate notebook: `~/typeR/notebooks/evaluation.Rmd` .

```
navigateToFile("~/typeR/notebooks/evaluation.Rmd")
```

Note: As you conduct your evaluation, you will find a number of small discrepancies between the numbers seen here and those in the paper. Running large experiments involves random failures, non-deterministic function behavior, and weird environment configurations leading to sporadic issues. These discrepancies are very minor, amounting to differences on the order of fractions of a percent, and we don't believe this issue to be a cause for major concern.

Rendering the `evaluation.Rmd` can be done by running:

```
./in-docker.sh make evaluation
```

Note: the evaluation takes a lot of time, it can easily be over an hour on a regular laptop.

If you successfully infered types for the small corpus (cf. above), you can run the evaluation using these data:

```
./in-docker.sh make evaluation DATA_CORPUS_DIR=data
```

However, the obtained numbers will be quite different.

The result can be viewed using:

```
viewer("~/typeR/evaluation/evaluation.html")
```

Assertions

In this section we show how to generate data presented in *Section 6.3 Usefulness of the Type Checking Framework*. We do this in a separate notebook: `~/typeR/notebooks/evaluation-asserts.Rmd` .

This notebook uses two inputs:

- `data-corpus/corpus-details.cvs` - metadata of the corpus 412 packages
- `data-corpus/cran/cran-asserts.csv` - information about the use of runtime assertions.

The latter has been generated by a makefile target (`package-asserts`). This task runs an R script which loads an AST of each function in a given package and look for calls to one of the R runtime assertion functions (e.g. stopifnot (https://stat.ethz.ch/R-manual/R-devel/library/base/html/stopifnot.html) or assertthat (https://cran.r-project.org/web/packages/assertthat/index.html)).

To render this notebook, in `$REPO/typeR` run:

```
./in-docker.sh make evaluation-asserts
```

Once finished, the resulting HTML file can be viewed by:

```
viewer("~/typeR/evaluation/evaluation-asserts.html")
```

This also produce a corresponding Latex tags to be included in the paper:

```
viewer("~/typeR/evaluation/package-asserts.tex")
```

Corpus

In this section we show how to generate data presented in *Chapter 5 - Project Corpus*. We do this in a separate notebook: `~/typeR/notebooks/corpus-analysis.Rmd` .

It loads the metadata collected for the whole of CRAN, namely: - `cran-details.csv` - CRAN packages metadata including code coverage and reverse dependencies code coverage - `cran-revdeps.csv` - reverse dependencies of R packages

These CSV files were obtained by running the corresponding tasks from rapr (https://github.com/PRL-PRG/runr/tree/typer-oopsla20/inst/tasks).

To knit the notebook, run:

```
./in-docker.sh make corpus-analysis
```

This will generate the following files:

- `corpus-analysis.html` : an HTML file rendering the notebook

```
viewer("~/typeR/evaluation/corpus-analysis.html")
```

- `corpus.tex` : a Latex tags to be included in the paper

```
viewer("~/typeR/evaluation/corpus.tex")
```

- `corpus.pdf` : a plot of the corpus

```
viewer("~/typeR/evaluation/corpus.pdf")
```

Building Image Locally

To build image locally, run the following in `$REPO`:

```
make -C docker-image
```

If you do not have GNU make, you can run:

```
docker build --rm -t prlprg/oopsla20-typer docker-image
```

Troubleshooting

File Permissions

The problem with docker is that it runs as root and so do its containers. A common trick is to setup a user in an entry-point script which will have the same UID and GID as the current user on the host system and then run the rest of the script as that user. Even though we have tried that, it does not work since GNU parallel cannot then reliably set working directory using the `--workdir` flag. In the current version therefore, all the generated files will be owned by root user.

To mitigate this problem, at least partially, the `in-docker.sh` script at the end runs:

```
docker run \
    --rm \
    -v "$base_dir/typeR:/home/rstudio/typeR" \
    prlprg/oopsla20-typer \
    chown -R $(id -u):$(id -g) /home/rstudio/typeR
```

text to r ### Docker on OSX

It is better to use homebrew cask to install docker:

```
brew cask install docker
```

in case you see `docker: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?.` error message cf: https://stackoverflow.com/a/44719239 (https://stackoverflow.com/a/44719239)

Docker on Linux

In some distribution the package does not add the current user to `docker` group. In this case, either add yourself to `docker` group or run all docker-related command with `sudo`.