Final Report CS 686

NIKITA TKACHENKO

This report presents a program analysis developed for identifying defined and available variables in R scripts. The motivation for this work arises from the frequent issues with undefined variables and the inadequacy of IntelliSense for R. The developed analysis, a points-to-definition approach, highlights variables used but not defined at specific program points. The system parses R scripts into an abstract syntax tree (AST), manages variable definitions, and identifies undefined variables. Despite certain limitations due to R's dynamic nature, this analysis effectively fills significant gaps left by existing tools, enhancing tooling for R programmers and improving development practices.

ACM Reference Format:
Nikita Tkachenko. . Final Report CS 686. In *Proceedings of* . ACM, New York, NY, USA, 4 pages.

This is the final report for 686 Program Analysis. In this report, I (Nikita Tkachenko) will explain the program analysis focusing on identifying defined/available variables in R scripts. The motivation for this project largely stemmed from my frustration with finding undefined variables, typos, forgetting to load a function, and many other issues that are generally addressed by proper IntelliSense. However, for some reason, IntelliSense never worked with R or did so unreliably due to a lack of proper code analysis tools and other issues with the existing solutions. Consequently, I would spend minutes figuring out what function I forgot, what I export and from where, and what I forgot to export. So, I decided to spend days writing a half-working solution.

Thus, the goal of the analysis is to create a points-to-definition analysis that would highlight variables that are used but are not defined at the specific program point.

The assignment in R is <-. It is possible to assign with =, but it is stylistically incorrect and may cause some issues in very specific circumstances[1].

Handling control flow is straightforward as I am building a may analysis, so if any branches define a variable I consider it defined for later use.

1 OTHER SOLUTIONS

To begin with the available solutions. There are no program analyzers for R I could find. Perhaps, it comes from me not being savvy in this area and just not knowing how to search for those solutions and papers. There are analyzers implemented using other languages such as a slicer using JavaScript.

The closest to something similar is a language server, but it works half-half in my NeoVim setup. There is also a linter for R, but that one uses xpath and something like regex, so it is far from perfect.

[1]It did happen to me 4 years ago when I was just starting and it was very frustrating.

Author's Contact Information: Nikita Tkachenko.

Manuscript submitted to ACM

There is little to no information on R AST and the only source is a chapter `Expressions` in `Advanced R` by Hadley Wickham, who had to do a lot of research himself to write the book. That was a good starting point with a few examples, including simple recursive functions to find all assignments, and everything I needed to get started.

2 A FEW CHALLENGES

One challenge is functions as they may contain variables that are undefined at the point of definition of the function, but the analysis still needs to account for those variables at function call. Additionally, R analyses depend on external packages, so attaching a package with `library()` needs to put all of the functions into a defined store. Also, special notation for using functions without attaching the namespace for the whole library `package_name::function_name` needs to work.

One of the issues with loading a package is that there are collections of packages such as `library('tidyverse')` that use a custom loading function `.onAttach`, which makes it challenging to load all the packages that they attach. So, this isn't implemented. But anyway, this is an issue with other alternatives, so it is not a big deal.

R also works with data frames, named lists, vectors, etc. There are a number of built-in datasets. A lot of common functions use quoting and unquoting of function arguments to manipulate columns, which would require creating stubs for all of those functions as well as implementing pointer analysis. As a band-aid solution, I just say whenever I call a built-in dataframe I define all of its columns as available variables. In other words, data analysis projects, won't be able to benefit much from it as this is a static analysis and doesn't have access to the current environment.

Additionally, in R you often `source()` other R files that are evaluated line by line. This is not captured by currently available solutions, so the analysis evaluates the whole file that is being sourced and includes all the definitions from there as well as highlights issues in that file.

3 HOW IT ROUGHLY WORKS

So, the way the analysis works. It takes a `file.r` and parses it into expressions, which later converts into AST. There is a single store with all available variables, an error store that keeps track of all errors with the usage of undefined variables including their location in the tree. The analysis walks the entire AST tree, looking for `<-` functions and looks for symbols for the undefined variables, which position it records. Note that if in `x <- y` y is undefined, we still add `x` to the store. When we encounter a function assignment, we record its globals used in a special function store, and when we call that function we check whether all of those globals are defined. If we encounter a loop, we add the variables only available inside of the loop meaning `i` in `for (i in 1:10) {...}` to the store, remember its location and once we check the inside of the loop and return back to the top of the loop delete that variable at the noted location. It works, because we don't remove duplicates, and walk graphs sequentially. At the end, we have the solution for our analysis in the error store, which contains the location of the undefined

variable in the AST and variable name. The location may contain `source()` file name in that case we use the last sourced file AST to display the error. The AST is modified with the undefined variable being wrapped in `%`[2] for instance `%undefined_var%`. Then I look at the expression that had the error and parse the new AST into text and display it alongside the path and the undefined variable. Additionally, for convenience, I wrote a NeoVim Lua script that would take the current buffer run the analysis and return it in a new tab, which had its own challenges, but this is out of scope.

[2]No reason behind this symbol, just something that looks unusual.

4 EXAMPLE

Notice how the second call to baz doesn't produce any errors and that errors in the function body just show function definition and the undefined variable.

4.1 R Script:

```
foo <- function(a, b) {
    return(a + b * bop())
}

bar <- function(a, b) {
    sum <- foo(a, b)
    return(sum * constant)
}

baz <- function(a, b) {
    bar(a, b) * bar(a, b)
}

var <- baz(a = 2, b = b)

constant <- 5

bop <- function() {
    return(5)
}

baz(1, var)
```

4.2 Output:

```
3 : constant
------------------------------------------------------------
%baz <- function(a, b) {%

3 : bop
------------------------------------------------------------
%baz <- function(a, b) {%

4,3,3 : b
------------------------------------------------------------
var <- baz(a = 2, b = `%b%`)
```

5 CONCLUSION

I won't talk much about challenges as I have overcome most of them in my previous assignments, but the main one being is R is not a programming language that should be used for this kind of tasks. It is made for data analysis, where its quirks are features not bugs. This experience definitely made me 100% times better at R, its list manipulations, debugging, shell scripting, and many other things. My tears and sweat were not for nothing. I think they should teach program analysis at monasteries as to test their patience and belief.