A large-scale study of polymorphism in R

ANONYMOUS AUTHOR(S)

The R programming language is widely used in a variety of scientfic domains for tasks related to data science. The language was designed to favor an interactive style of programming with minimal syntactic and conceptual overhead. This design is well suited to support interactive data analysis, but is not well suited to generating performant code or catching programming errors. In particular, R has no type annotations and all operations are dynamically checked at runtime. The starting point for our work is the question: *what could a static type system for R look like?* To answer that question we study the polymorphism that is present in over X millions of lines of R code, written over a period of Y years by Z programmers.

1 INTRODUCTION

2 BACKGROUND

2.1 The R Programming Language

R is a lazy, side-effecting, dynamically-typed, ...

Typical usage of the language is ... . **AT** Do we have something to cite for this? **AT** Should we motivate why we only look at R packages? Some people might wonder why we don't look at non-library code.

Most R libraries (called *packages*) are stored in the Comprehensive R Archive Network (CRAN) and Bioconductor package repositories. **AT** Talk about important packages? tidyverse?

AT Kind of ... want to combine the above two paragraphs somehow? Together, they lead in nicely to the genthat talk.

2.2 The genthat Package

Unfortunately, getting our hands on "front-line" R code is tricky: R is not like many other languages, where large systems are built up and stored in a repository such as Github. Instead, R programmers are liable to leverage a number of packages and work them into their data analysis pipelines.

Interestingly, we are somewhat able to replicate this using only package code. CRAN, for instance, has a policy that packages made available on the CRAN platform should be accompanied by a series of examples, tests, and/or vignettes showing off how the code is intended to be used. Nominally unrelated, the `genthat` project aims to trace this swath of example code to generate new examples which package designers may not have thought of. Crucially, `genthat` outputs complete trace information for each of the functions that it runs so that they can be run again in a fresh R instance (e.g., by capturing the random seed, environment variables, etc.). We can leverage this trace information to glean type and attribute patterns of functions, and since these traces correspond to examples of the packages being used (rather than just arbitrarily running package code), we have a reasonable approximation of how R programmers would use the packages.

AT We use/modify the `genthat` tracer, probably worth going in to more detail here.

3 MOTIVATION AND HIGH-LEVEL OVERVIEW

Our community is centred on building, improving, and reasoning about programming languages. A key component in all this is to truly *understand* the languages we're working on, so that we may make the best decisions for the benefit of all users of a programming language. In many cases, we as researchers can appeal to our own intuition to at least build a sense for how a language is

2019. XXXX-XXXX/2019/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

used, as language users typically have some training from an experienced programmer or software engineer (e.g., computer science studies, online tutorials written by language experts, etc.). **AT** We claim that this is emphatically not the case in R.

R is a languages written by and for *statisticians*. It traces its roots to the S language, developed by ... . R programmers don't write big servers or software systems; instead, they write and modify small data processing pipelines as a means to facilitate interaction with digital data. This leads to "front-line" R code (i.e., code written for users, not programmers) not being made available on typical platforms like Github. **AT** Maybe front-line code isn't a good word, is there a good way to describe non-library code? Put differently, R is essentially a really big domain-specific language.

The design of R is also rife with questionable decisions. For example, the language is lazy *and* side-effecting, which one could describe as an antagonistic pair of features. In addition, data structures must be entirely copied when being modified, which appears to be far from ideal in a language meant to process (thus, modify) big data.

In spite of these flaws, the language is incredibly popular. **AT** We'll want to cite something here.

One way or another, R is here to stay, so we may as well figure out how to deal with it. One aspect of this is understanding R programmers and how they interact with the language, and a step towards this goal is to understand **AT** how types arise in R programs. We will shed some light on this by analyzing the usage of R functions, and build a picture of how polymorphic R code is.

- Understanding R can help us improve the language, as well as better equip us to build languages which have an appeal outside of the realm of computer science.
- We're gonna tackle this by looking at how R programmers use types and type information in their programs.
- We will be mindful of the runtime representations of values to give more context to the information we glean from programmer-visible type information.

4 THE METHOD

In this section, we will detail our methodology for collecting data.

First, we leverage `genthat`'s tracer to trace function executions. The idea here is that we don't really have access to non-library code written in R, as general use patterns are (possibly?) to write small scripts which analyze some bit of data and possibly visualize results. The goal is likely not to build big working systems, instead to explore data with by writing and rewriting small scripts, ad infinitum. In looking at package tests, vignettes, and examples, we are painting a picture of how the package designers intended their packages to be used, which we believe is a close approximation of what R users would do.

`genthat` generates a trace file which can be loaded into R and contains all of the necessary information to rerun the tested function. This includes the random seed and all relevant environment data. We utilize this trace data to collect type information: we run each argument in the specified environment, and collect type information through R's `typeof` function, and attribute information through R's `attributes` function.

Then, we aggregate the information in all traces for a particular function. If there was only one function trace, we discard the result as the function is trivially monomorphic. We can see which arguments had which types over how many traces, and from here we can build a *signature* for each function argument, and consider them together to create a function signature. A **polymorphic argument** is one which has been inhabited by values of at least two different types, and a **polymorphic function** is a function with at least one polymorphic argument or a polymorphic return.

We repeat the above for each package on the Comprehensive R Archive Network (CRAN), the premier source for R libraries (called packages), and the Bioconductor package repository. Now, depending on the data point we're after, our methods from here differ slightly.

- for **counting signatures**: for each signature, we count how many functions or arguments have the requisite signature;
- for **finding the most common signatures**: we enumerate as before, and sort the signatures based on how often they appear;

4.1 Attributes

Types (in the sense of the result of a call to `typeof`) are easy to analyze as values can only have one type. Attributes, however, are a different story, as values can have an arbitrary number of attributes. Essentially, we're not interesting by function arguments which have had multiple attributes, we're interested in function arguments which have had multiple attribute *patterns*. **AT** is this defined earlier? Arguments are said to be polymorphic in attribute if they have been inhabited with values which have had different attribute patterns. **AT** We have not investigated whether functions which are polymorphic in attribute use attributes extensively inside the function code.

AT Paragraph justifying our construction of an attribute pattern. Either just name, or (name, type).

AT Paragraph about so-called "naturally-occurring" attribute patterns.

4.2 `genthat` Tracer Quirks

First and foremost, `genthat` captures arguments only on function exit, so as to not force promises during function execution. **AT** Uh oh: One clear limitation here is that if an argument *gains* attributes over the functione execution, then they will appear in our signatures. As R is side-effecting, this is an entirely valid practice (i.e., passing objects to functions to modify them), though **AT** it's unclear if people actually do this.

AT Maybe talk about the quirks of genthat in here? For example, how it deals with default arguments, and stuff like that.

4.3 Goals

Ultimately, our goal is to collect data and report on patterns which emerge organically. That said, the comprehensive picture of type usage we are developing can also be used to inform the design of a set of *type annotations* for the language, and this goal informs our analysis. As we collect our data, we keep in mind the idea that the data we produce should clearly suggest what sorts of type annotations would reflect language usage patterns.

4.4 Back-End Data Collection

AT Do we want to go down this route with this paper? This would be Konrad's bit, if we want to try to stick our data together.

5 RESULTS

AT Tables, graphs, code examples should go in here. In this section, we ... .

5.1 Usage Patterns

In this section, we will discuss function usage patterns which arose in our analysis. We will start by looking at the morphicity of functions.

5.1.1 Function Argument and Return Morphicity. First and foremost, we would like to know how often R programmers create polymorphic functions. Recall that we define a polymorphic function to be a function with at least one polymorphic argument, or a polymorphic return. We will turn our attention now to the data in **TODO** Figure, and go through each entry in the table.

Here, we see that the vast majority of function arguments are indeed monomorphic. Monomorphic arguments are easy to annotate, as the `typeof` the arguments is exactly the most precise annotation we could give, at least in terms of *type*. That said, types alone don't always paint the whole picture: recall that *attributes* are an R language feature which allows programmers to stick metadata onto values. So in which circumstances *do* types paint the whole picture?

The second data point in **TODO FIGURE** indicates that a **Check** majority of function arguments are monomorphic in type *and* have no attributes. These represent arguments which are truly trivial to annotate, as the `typeof` an argument perfectly describes the usage of that argument. That said, some attributes arise naturally in R: For instance, names in a named list (e.g. `x` and `y` in `list(x=0, y=0)`) appear as an attribute on the value. In Section 5.1, we outlined these naturally-occurring attribute patterns, and the third data point in **TODO FIGURE** shows that a nontrivial amount of functions are monomorphic in type with said natural attribute patterns.

Another facet of type information in R is in the `class` attribute. Recall that values have a *class* in addition to a type: For example, a `data.frame` has type `list` and class `data.frame`. **TODO** Plug classes earlier R has a number of built-in classes, such as **TODO** X, Y, and Z, but users are free to redefine the class of any value at runtime, and easily define new classes. The next data point in **TODO** Figure shows that user-defined classes don't appear altogether often, though they do indeed feature. **AT** In a following section, we will discuss how the usage of these classes manifests itself.

AT Separate section for functions? Right now, this is at argument granularity.

5.1.2 Type Signatures. In the last section, we presented a high-level overview of the morphicity ... .

5.1.3 Attribute Signatures. Recall that attributes are a way for programmers to store metadata on values in R. What are the common attribute patterns? And how often is the attribute pattern polymorphic?

5.1.4 Takeaways.

- the vast majority of arguments are monomorphic in type;
- of those, over 60% have no attribute information;
- of the 40% with attribute information, roughly 1/2 have fairly simple attributes corresponding to base R constructs (named lists and vectors, matrices, and data frames);
- now, of those arguments which are polymorphic, a sizable chunk (well over half) have defensible signatures (e.g., double and character for named list indexing, double and integer for obvious reasons, etc.).

In short, it looks like R (package) programmers are reasonable. I'd conjecture that a lot of the polymorphism (e.g., double and list) is coming from how easy it is to use either type in a given situation (e.g., converting from vector of doubles to list or vice versa is simple).

6 SYNTHESIS

In this section, we will discuss the conclusions that we draw from our data.

6.1 Suggested Annotations

AT How can we capture these patterns with annotations?

Some possible annotations:

- *real*: for *double* and *integer* values

- *function*: for *closure*, *special*, and *builtin* values
- *vector*: to indicate that something should be vectorized
- *scalar*: to indicate that something should **not** be vectorized
- *index*: for *real* and *character* values

6.1.1 Struct-Like Attribute Declarations. **AT** What's a convenient way to annotate attributes? Should investigate how often attributes are consistent.

6.1.2 Coverage. **AT** What is the coverage of these new annotations?

7 CONCLUSIONS AND FUTURE WORK