Towards a Type System for R

Anonymous Author(s)

Abstract

R is a crazy language. **AT** Briefly describe why R is crazy. In spite of this, R is widely used. We would like to build a full-fledged type system for the language *that programmers will want to use*. Not only would this would serve to increase the assurance of R code, it would also open the door to type-based program optimizations.

1 Introduction

2 The R Programming Language

R [1] is ... .

3 Why Types?

We want to add types to R for two main reasons.

First, types increase the assurance that programmers have in their code. If using a static type system, type-related errors can be caught statically, which communicates potential bugs to programmers at compile time. If using a dynamic type system, type-related errors are caught at runtime, which may well abort the program, but at least does not allow for values to be quietly misused.

When we increase assurance in a programming language, we thereby increase assurance in the products of that language. For instance, one might have more faith in a web server written in TypeScript than one written in JavaScript, since TypeScript at least catches static type errors in annotated code. So what are the products of R? R is a language primarily used by statisticians and data scientists to perform data analysis. The conclusions drawn from these analyses go on to inform decisions in broader organizations: For instance, census data analysis results go on to inform policymaking decisions. It is critical that we can trust the software used to draw these conclusions.

Second, type information can be used by just-in-time compilers (JIT compilers, or JITs) to perform program optimizations at runtime. **AT** This one goes without saying, not sure how much to elaborate here.

4 Concerns

Designing a type system for R is no small task.

4.1 Type System Granularity and Complexity

When designing a type system for an existing, dynamic language, we have the freedom to decide what granularity of type information we want to express. In data-oriented languages, such as R, choosing the level of granularity is not so straightforward.

For instance, consider vectors in R. Primitive types in R are vectorized, meaning that even scalars are actually vectors (e.g., the scalar 1 is implemented as a unit-length vector). That said, in many other languages, a distinction is made between scalars and vectors, and this distinction is coupled with some performance benefit. When designing a type system for R, we might be interested in distinguishing vectors and scalars.

Going a step further, perhaps it would be valuable to include the dimensions of data in a vector type. R already implements different functionality for operators acting on vectors depending on the lengths of the vectors. For example, consider the following code snippet:

```
c(1, 2) + c(1, 2) # => c(2, 4)
c(1, 2, 3) + c(1, 2) # => c(2, 4, 4)
c(1, 2, 3, 4) + c(1, 2). # => c(2, 4, 4, 6)
```

On vectorized primitives, the + operator will duplicate the second argument until it matches the length of the first. The 3rd line above illustrates this clearly: the second argument to + there becomes `c(1, 2, 1, 2)`. This is useful functionality, and indeed users of R are familiar with it, but it does introduce possible sources of error if used incorrectly. For instance, imagine if we were working with some data, and we wanted to normalize the data w.r.t. a known vector of values. Consider:

```
norm <- function(data) {
  data / c(1, 2, 4)
}

norm( c( 5, 4, 8)) # => c(5, 2, 2)
norm( c( 5, 2)) # => c(5, 1, 0.25)
```

In the first call to `norm`, we see that the argument to `norm` was divided element-wise by the vector defined in the function itself. In the second call, we see how R automatically pads vectors when the lengths don't quite match: here, `c(5, 2)` essentially becomes `c(5, 2, 1)`. In general, this is rather harmless, but when we consider R's use case as a data analysis tool, this represents *generating arbitrary data*: The value inserted by the division operator is really a data point fabricated by the implementation of the function. In a sense, off-by-one errors can easily go unnoticed in R programs.

All that said, adding lengths to types may not be desirable. The complexity of the type system and associated annotations would be far greater, and there's no telling if users would subscribe to the restrictions imposed by e.g. statically encoding lengths. Many of these errors could be caught by

Conference'17, July 2017, Washington, DC, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

runtime checks, for instance with an explicit length check in the `norm` function above, but it is as easy to write those checks as it is to forget to put them. Further, designing annotations to allow users to express types in the language is complicated by the lexical complexity of R itself; in short, there is not a lot of wiggle room in the parser. It is critical that we balance complexity and granularity with usability. This is discussed next.

4.2 Ensuring User Engagement

If one of our goals is to increase our confidence in R programs, we need to ensure that language users will want to use our types.

R users represent an interesting point in this design space. R is an old language, and R users are not your average programmers in the sense that they are likely not software engineers, developers, or computer scientists. The best way to understand how they interact with the language is through observation and analysis of the programs that they have written. But even that is no small feat, as R programs are not published the same way that the products of many other languages are on e.g. GitHub. This last point will be discussed further in Section 5.1.

Ultimately, we need to balance the complexity of the type system with needs of R programmers. Perhaps a dependent type system would be useful and capable of capturing large swaths of existing R paradigms, but these types are difficult to express and may be too burdensome for R users. On the other hand, differentiating scalars and vectors may not reflect existing R implementations, but perhaps R users would be able to quickly adapt to such a distinction. The key to adoption is to strike this balance, and conceive of a type system which captures everything that R programmers would want to capture. We have made some progress towards this, discussed further in Section 5.

4.3 Types for Big Data Objects

A unique aspect of R is the data frame, which serves as the cornerstone of nearly all analyses written in R. As we mentioned, the process of a typical R program is to load data, modify and/or transform it, and perform an analysis. Typically, imported data is a *data frame*: an R data type which is effectively a list of lists, where each row represents an observation, and each column represented a thing that was observed. These objects underpin data analysis in R, and understanding the shape of your data is critical to successful data analysis.

As data frames are of utmost importance, it stands to reason that we might like to create some type for them. How specific that type should be, however, is unclear: data frames are often quite large, and writing a type for such a thing would be tedious and perhaps not altogether informative if the data frame is used in simple ways. If the only operation performed on the data frame is to normalize some column and plot it, then types are not supremely helpful. But what if, during the modification and transformation phase of a data pipeline, the analyst is working with several data frames and joining and collating them to create new data frames? Many of these operations are dependent on the types of some columns, and could fail in various ways if the column types are not compatible. These errors would be mitigated by (relatively) sophisticated data frame types.

4.4 The Burden of Annotations

A nontrivial barrier to building types into R is the need for annotations. For one, the syntax of the language is tricky to extend, as many characters already serve a purpose. This makes designing a satisfying syntax for types nontrivial. Further, as the type system becomes more complex, so too do the annotations: R is rich in runtime type information already, and statically encoding that via annotations can quickly become burdensome. For instance, if we would like to express that an argument to a function is a list, with some class, and certain attributes, we have over 3 things to annotate onto a single function argument. Ultimately, friendly-looking annotations will be attractive to users, and making the annotations easy to write and minimizing the burden of annotating is critical to adoption.

5 Our Plan

5.1 A Practical Type System

End-user R code is typically some small script (called a *notebook*) which imports some data and modifies it in some way before performing some analysis. In many cases, the data being fed to the script is proprietary (e.g. customer data) or confidential (e.g. census data), which results in an inability to publish the script, even if the analyst were so inclined. In sum, getting our hands on R user code can be tricky.

Thankfully, all is not lost: the Comprehensive R Archive Network (CRAN) is a repository of publicly available R package (i.e. library) code. Packages made available through CRAN must adhere to some criteria; among them, a guarantee that the package must be accompanied by some examples, tests, and/or vignettes to showcase package functionality. These examples are about as close as we can come to end-user R code, and serve as a great starting point for any analysis we undertake.

6 Related Work

There is a wealth of literature on analyzing language usage patterns. For instance, **??**

References

[1] R Core Team. 2018. R Language Manual. https://stat.ethz.ch/R-manual/R-devel/doc/manual/R-lang.html.