Towards a Type System for R

Alexi Turcotte
Northeastern University
alexi@ccs.neu.edu

Jan Vitek
Northeastern University & Czech Technical University in Prague
j.vitek@northeastern.edu

Abstract

R is a fascinating language: It is dynamically typed, vectorized, both lazy and side-effecting, and it fosters an interactive style of programming. This unique combination of features makes it easy to use, but prone to errors and strange behaviour. R is the tool of choice for many data analysts, and our aim is to empower them with a language that is not simply easy to use, but easy to use *well*, so as to increase their confidence in the data analyses they undertake. To that end, we are developing a type system for R that is simple enough to be attractive to programmers while being expressive enough to capture existing programming paradigms. In this paper, we outline past, present, and future work as we build up to a type system for R.

1 Introduction

Data collection and analysis occurs at an incredible scale, and this process underpins modern decision-making. For instance, government data analysts process census data to inform policymaking decisions. Given that we are so impacted by this process, we should have the utmost faith in the tools which are used to perform this data analysis. The R programming language is one such tool.

R is an interesting language: It is a dynamically typed, vectorized, lazy, side-effecting language designed by and for statisticians and data scientists. As a programming language, R is quite fascinating, and the language sees widespread use for data analysis tasks.

To increase our assurance in the products of R, we want to design a type system for the language. Types are a great way to make programming languages less bug prone: Catching potentially insidious type errors ahead of time eliminates a significant source of run-time errors in programs. Good statistical analysis are difficult enough as it is, so we should empower data scientists with tools that they can rely on, and in which we have confidence.

We want R programmers to want to use our type system, so we aim to propose a system that is informed by existing programming paradigms. We will achieve this by using a large-scale analysis of R programs as a sounding board, to check and measure the effectiveness of possible type system designs. When we finalize a design, we will likely implement it as a *gradual type system*: this will allow programmers to opt-in to the type system where they deem appropriate, and will allow typed and untyped R code to interoperate seamlessly.

In this position paper, we discuss some of the challenges of designing a type system for R, and outline next steps in this journey towards building better R programs.

2 The R Programming Language

R [3] is a programming language developed by statisticians for the purpose of data analysis. R is a successor to the S programming language, and was originally designed to process data into vectors to be used by Fortran data analysis programs. It has since evolved into a full-fledged, general purpose programming language with an estimated 2 million users as of 2011 [6]. R is built to foster an interactive model of programming, where the user works closely with the data: a typical R program will read in some data, mutate and/or transform it, and finally perform an analysis on the modified data. Depending on the results of the analysis, the program is then itself modified and rerun by the analyst.

When considering what types would be relevant to R users, we should be cognizant of the notions of "type" already present in the language. These are discussed next.

2.1 Run-Time Type Information in R

R is highly reflective, and a wealth of type-related information is available to programmers at run-time. Some of R's reflection functions are mentioned below, and examples of each of them can be found in Figure 1.

First, the `typeof` function yields the run-time type tag associated with a value—this tag represents the type of the value according to the R internals. In a sense, `typeof` is the programmer's window into the R implementation.

Another source of run-time type information (RTTI) in R are a value's *attributes*: In R, values have an arbitrary amount of (named) metadata, called attributes, which can be easily modified at run-time using the `attributes` and `attr`

ICOOOLPs, July 2019, London, United Kingdom

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/3340670.3342426

```
typeof(5L) == class(5L) # integer

# list() creates an empty list
typeof(list()) == class(list()) # list

# data.frame() creates an empty data frame
typeof(data.frame()) # list
class(data.frame()) # data.frame
attr(data.frame(), "class") # data.frame

x <- 5 # initialize x to 5
class(x) # numeric
class(x) <- "character"
class(x) # character
```

Figure 1. Examples of reflective functions in R.

functions. Some R values, such as data frames, named lists, and matrices, have default attributes which may be used by R's builtin functions.

Another reflective function is `class`, which yields the class of a value. In R, values have a class according to their `class` attribute, and the class of a value can be easily (re)defined by programmers at run-time. Importantly, dynamic dispatch is done based on the class of function arguments.

In sum: R has an eclectic mix of RTTI, all of which is available to the programmer. The ease with which programmers can modify RTTI makes for some interesting usage patterns, and gives us a lot of interesting space to explore when designing a type system for the language. R programmers creatively use this RTTI, and capturing the paradigms of these creative programmers is an exciting undertaking.

3 Why Types?

We want to develop a type system for R for two main reasons.

First, types increase the assurance that programmers have in their code. For example, a static type system allows type-related errors to be caught statically, communicating bugs to programmers early. A type system may also help IDEs or analysis tools to identify issues while code is being written.

When we increase assurance in a programming language, we are thereby increasing 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 catches static type errors in annotated code. So what are the products of R? R is a language primarily used to perform data analysis, and 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.

Besides giving us more faith in the language, a type system can be used by just-in-time compilers (JIT compilers, or JITs) to perform program optimizations at run-time. As an example, statically-available type information could be leveraged by a JIT to better speculate on call targets for generic functions. Since R is absurdly dynamic, and arbitrary changes to the run-time environment can be enacted during execution, we cannot do away with speculation and profiling entirely, but perhaps the types could be used as an initial guess for the JIT compiler. Concretely, a call such as `x + y` could be initially targeted at the implementation of + for whatever types `x` and `y` are said to have ahead-of-time.

4 Concerns

Thus far, we have established that designing and implementing a type system for R may prove useful. That said, developing such a type system is no simple task, and there is a huge design space to consider. In this section, we explore some of the design concerns we are likely to face.

4.1 Type System Granularity and Complexity

When designing a type system for an existing, dynamic language, we have freedom to decide on the granularity of expressible type information. In a language such as R, choosing the right level of granularity is not so straightforward.

As an example, consider vectors in R. Primitive types in R are vectorized, meaning that primitives are always vectors (e.g., the scalar 1 is implemented as a unit-length vector). This is not the case in many other languages, where a distinction is made between scalars and vectors; a distinction that carries some performance benefit. Thus, it may be worthwhile to consider distinguishing scalars and vectors in R.

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. That functionality is illustrated in the following code snippet, where `c(...)` denotes R's builtin vector constructor:

```
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)
```

Above, `c(1, 2)` denotes a vector with elements 1 and 2. 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 + is treated as `c(1, 2, 1, 2)`. This functionality can be useful, 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 with respect to 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 the division operator in R automatically pads vectors when the lengths do not quite match: here, `c(5, 2)` is treated as `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 a data point fabricated by the implementation.

All that said, adding lengths to types may not prove to be all that desirable. For one, the complexity of the type system and associated annotations would be greater, and there is no telling if users would subscribe to the restrictions imposed by e.g. statically encoding vector lengths. As it happens, many of the errors caused by vector lengths could be caught by run-time 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.

No matter the complexity, we need to ensure that the type system will be used by programmers. 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 R users will want to use our types.

R users represent an interesting point in this design space. R is not designed by computer scientists, and R users are not classically-trained programmers. 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 this 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. The key to adoption is to strike a balance, and conceive of a type system which captures everything that R programmers would want to capture. We have made some progress on this, discussed further in Section 5.

4.3 Types for Data Frames

An integral component 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, with a row for each observation and a column for each thing that was observed. These objects underpin data analysis in R, and understanding the shape of the 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, but how specific that type should be is unclear. Data frames can have many columns, 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 the number of rows, and could fail if the column types or row counts are not compatible; failures that would be caught by 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, which 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 run-time 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 Outlook

Our aim is to design and implement a type system for R which will make correct R programs easier to write, and be widely adopted by R users. Thus the type system must be able to capture established programming paradigms, and to ensure this we will ground our design on a large-scale analysis of existing R programs.

In this section, we will describe our plan to build this type system. First, we discuss how we will ensure that the type system we design is practical for R users. Then, we describe a metric that we have established to measure how well a type system captures language usage in order to compare candidate type systems.

5.1 A Practical Type System

As we endeavour to design a type system which appeals to R programmers, we should take steps to ensure that it is grounded in the paradigms that are familiar to said programmers. To this end, we leverage some forthcoming work which performed a large-scale analysis of existing R code.

End-user R code is typically some small script which imports some data and modifies it in some way before performing an 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 short, end-user R programs are rarely published.

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 be accompanied by some examples, tests, and/or vignettes to showcase package functionality. These examples are about as close as one can get to end-user R code, and serve as a starting point for any analysis of R programming patterns.

We will test candidate type systems against a corpus of code consisting of millions of lines of R code across over 10,000 packages. The full treatment of this corpus will be made available in a forthcoming paper.

5.2 Measuring Effectiveness

One metric for measuring the effectiveness of a particular type system is to consider the percentage of functions which are observed to be *polymorphic* when viewed through the lens of that system. The polymorphism we are concerned with is *ad hoc polymorphism*, where a function is polymorphic if it can be applied to arguments of different types. This may need to be balanced against a (perhaps subjective) dimension of "usefulness", as e.g. ascribing the "Any" type to all values would see 100% of functions being monomorphic. To quickly illustrate, consider the following snippet:

```
addVec <- function(a, b) {
  if (length(a) == length(b)) { a + b }
  else NULL
}

addVec(c(1L, 3L), c(1L, 3L))
addVec(c(1.5), c(1.5))
```

Note that the type of `c(1L, 3L)` (according to the `typeof` function) is `integer`, and the type of `c(1.5)` is `double`.

Considering only these two calls, the `addVec` function is polymorphic, as it is called with two integer and separately two double arguments. This polymorphic behaviour is prevalent throughout R, and we claim that we can capture this polymorphic behaviour by defining the type "integer" to be a subtype of "double". With integer <: double, then `addVec` would be monomorphic (accepting two double arguments).

Now, can we do more? We *could* encode more information in the types, say the lengths of the vectors, and then the function would once again be polymorphic, accepting vectors of varying lengths of type double. We could just as well parameterize over the length, in which case `addVec` would be monomorphic, accepting double vectors of length n. A polymorphism-based metric captures how effective a particular type system is at capturing function usage patterns.

In sum, even for this very simple function, we might well consider the following types:

1. {dbl, dbl} → dbl, {int, int} → int
2. {dbl, dbl} → dbl, with int <: dbl
3. {dbl[2], dbl[2]} → dbl[2], {dbl[1], dbl[1]} → dbl[1], where dbl[x] denotes a double vector of length x
4. {dbl[n], dbl[n]} → dbl[n]

Ultimately, we want our design to have the broadest appeal, and we will endeavour to balance the richness of our annotations with the needs of everyday R programmers.

To give a concrete example from forthcoming work, we have found that nearly 20% of functions were polymorphic with respect to R's `typeof` function, meaning that nearly 20% of functions were called with at least one polymorphic argument. Here, a polymorphic argument is one that was inhabited by at least two values which would return different results if passed to `typeof`. The `addVec` function is an example. We can reduce the proportion of polymorphic functions by explicitly defining subtypes (e.g. taking integer <: double), and exploring this space is a part of said forthcoming work.

6 Related Work

Part of our process for ensuring the practicality of our type system is to use a large-scale language analysis as a sounding board, and there is a wealth of literature on analyzing language usage patterns. For instance, some work [4] explored the dynamic behaviour of JavaScript programs, testing assumptions that appeared frequently in the literature. Other, similar work [1] explored the dynamic behaviour of Python programs. Our aim is to use a similar analysis of R programs to check our type system, and keep our design grounded in the day-to-day usage of the language.

When we finalize a type system, our implementation will likely be a gradual type system [8]. In gradually typed languages, type annotations are optional, and typed and untyped code is allowed to interact seamlessly within the same program. In this design space, there are even more considerations: for instance, should we implement a sound gradual type system, where checks are inserted at the boundary of typed and untyped code? Or should we leave that boundary unchecked, as is the case in languages like TypeScript? If we insert checks at the boundary, we catch dynamic type errors at run-time, but a possibly steep performance cost may be incurred [7].

Python, the other eminent data processing language, has been the object of similar work. Mypy [2] is an optional static type checker for Python, with rather rich type system which includes generics and inheritance. Reticulated Python [5] is full-fledged gradual type system for Python. While Python is heavily utilized for data analysis, it was not designed by data

scientists and feels a lot more like other general-purpose programming languages than R does.

7 Conclusion

With an unprecedented wealth of data at our fingertips, we should ensure that we have the utmost confidence in the tools that we use to analyze that data. To this end, we aim to build a type system for the R programming language, the tool of choice for millions of data analysts worldwide, and have outlined our plans to do so in this short paper.

Acknowledgments

This work received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (awards 1544542 and 1518844), the ONR (grant 503353), the Czech Ministry of Education, Youth, and Sports (grant agreement CZ.02.1.01/0.0/0.0/15__003/0000421), and NSERC.

References

[1] Alex Holkner and James Harland. 2009. Evaluating the Dynamic Behaviour of Python Applications. In *Australasian Computer Science Conference (ACSC)*. 19–28.

[2] The mypy Project. 2014. mypy. http://www.mypy-lang.org.

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

[4] Gregor Richards, Sylvain Lesbrene, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*.

[5] Jeremy Siek, Michael Vitousek, Andrew Kent, and Jim Baker. 2014. *Design and Evaluation of Gradual Typing for Python*. Technical Report. Indiana University.

[6] David Smith. 2011. The R Ecosystem. In *The R User Conference 2011*.

[7] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max New, Jan Vitek, and Matthias Felleisen. 2016. Is sound gradual typing dead?. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2837614.2837630

[8] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Dynamic Language Symposium (DLS)*.