Designing Types for R, Empirically

ALEXI TURCOTTE, Northeastern University
AVIRAL GOEL, Northeastern University
FILIP KRIKAVA, Czech Technical University in Prague
JAN VITEK, Northeastern / Czech Technical University

The R programming language is widely used in a variety of domains. It was designed to favor an interactive style of programming with minimal syntactic and conceptual overhead. This design is well suited to interactive data analysis, but a bad fit for tools such as compilers or program analyzers which must generate native code or catch programming errors. In particular, R has no type annotations, and all operations are dynamically checked at run-time. The starting point for our work are the twin questions: *what expressive power is needed to accurately type R code?* and *which type system is the R community willing to adopt?* Both questions are difficult to answer without actually experimenting with a type system. The goal of this paper is to provide data that can feed into that design process. To this end, we perform a large corpus analysis to gain insights in the degree of polymorphism exhibited by idiomatic R code and explore potential benefits that the R community could accrue from a simple type system. As a starting point, we infer type signatures for 25,215 functions from 412 packages among the most widely used open source R libraries. We then conduct an evaluation on 8,694 clients of these packages, as well as on end-user code found on the Kaggle competition website.

CCS Concepts: • **Software and its engineering** → **Language features**; *General programming languages.*

Additional Key Words and Phrases: type declarations, dynamic languages, R

ACM Reference Format:
Alexi Turcotte, Aviral Goel, Filip Krikava, and Jan Vitek. 2020. Designing Types for R, Empirically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 153 (October 2020), 23 pages. https://doi.org/10.1145/3360579

1 INTRODUCTION

Our community builds, improves, and reasons about programming languages. To make design decisions that benefit users, we need to understand our target language as well as the real-world needs it answers. Often, we can appeal to our intuition, as many languages are intended for general purpose programming tasks. Unfortunately, intuition may fail when looking at domain-specific languages designed for a specific group of users to solve very specific needs. This is the case of the data science language R.

R and its ancestor S were designed, implemented, and maintained by statisticians. Originally they aimed to be glue languages for reading data and calling statistical routines written in Fortran. Over three decades they became widely used across many fields for data analysis and visualization. Modern R, as an object of study, is fascinating. It is a vectorized, dynamically typed, lazy functional language with limited side-effects, extensive reflective facilities and retrofitted object-oriented programming support.

Authors' addresses: Alexi Turcotte, Northeastern University; Aviral Goel, Northeastern University; Filip Krikava, Czech Technical University in Prague; Jan Vitek, Northeastern / Czech Technical University.

2475-1421/2020/10-ART153
https://doi.org/10.1145/3360579

Many of the design decisions that gave us R were intended to foster an interactive and exploratory programming style. These include, to name a few, the lack of type annotations, the ability to use syntactic shortcuts, and the automatic conversion between data types. While these choices have led to a language with a low barrier to entry—many data science educational programs do not teach R itself but simply introduce some of its key libraries—they have also created a language where errors can easily go undetected.

Retrofitting a type system to the R language would increase our assurance in the result of data analysis, but we are faced with two challenges. First, it is unclear what would be the *right* type system for a language as baroque as R. For example, one of the most popular data type, the `data.frame`, is manipulated through reflective operations—a data frame is a table whose columns can be added or removed on the fly. Second, but just as crucially, designing a type system that will be adopted would require overcoming some prejudices and educating large numbers of users.

This paper is a data-driven study of what a type system for the R language could look like. Our intention is to eventually propose changes to the language, but we are aware that for any changes to be accepted by the user community they must clearly benefit the language without endangering backwards compatibility. Our goal is thus to find a compromise between simplicity and usefulness; any proposal should cover common programming idioms while remaining easy to learn and to use.

This paper focuses on a simpler problem than an entire type system, instead, we limit our scope to giving types to function signatures. For this, we design a simple type language, one that matches the R data types but omits features such as parametric polymorphism and subtyping between user-defined data types. We then extract type signature from execution traces of a corpus of widely used libraries. This allows us to see how far one can get with the simple type language and identify limitations of the design. We validate the robustness of the extracted type signature by implementing a contract checker that weaves the types around their respective functions, and use a both a large number of clients of the target packages as well as end-user code for validation. The contract system can be used by the R community to experiment with type signatures and replace ad hoc error checking code.

To sum up, our paper makes the following contributions:

- We implemented tooling to automatically extract type signatures from R functions and to instrument R functions with checks based on their declared types. The tooling is robust and scalable to the entire R language.
- We carried out a large-scale analysis of corpus of 412 widely used and actively maintained libraries to extract function type signatures and validated the robustness of the inferred type signatures against 160,379 programs that use those functions.
- We report on the appropriateness and usefulness of a simple type language for the R programming language.

Our tools are open source and publicly available on GitHub[1], all results in this paper are reproducible and were submitted for artifact evaluation.

2 BACKGROUND

In this section, we introduce related work on extending dynamic languages with static type systems and we give a short primer on the R programming language.

2.1 Related Work

Dynamic programming languages such as Racket, JavaScript and PHP have been extended post-facto with various static type systems. In each case, the type system was carefully engineered to

[1] Typetracer: https://github.com/PRL-PRG/propagatr, ContractR: https://github.com/PRL-PRG/contractr

match the salient characteristics of the host languages and to foster a particular programming style. For example, Typed Racket emphasizes functional programming and supports the migration from untyped to fully typed code [Tobin-Hochstadt and Felleisen 2008], Hack [Verlaguet 2013] and TypeScript [Bierman et al. 2014] focus on typing object-oriented features of PHP and JavaScript, respectively. They allow users to intersperse typed and untyped code in a fine-grained manner. Other work looks at retrofitting types onto Lua [Maidl et al. 2014] which account for the myriad ways Lua programmers use tables.

But what if the design of the type system is unclear? Andreasen et al. [2016] propose an intriguing approach called trace typing. With trace typing, a new type system can be prototyped and evaluated by applying the type rule to execution traces of programs. While the approach has the limitation of dynamic analysis techniques, namely that the results are only as good at the coverage of the source code, it allows one to quickly test new design and quantify how much of a code base can be type-checked. Other approaches that infer types for dynamic analysis include the work of Furr et al. [2009] and An et al. [2011] for Ruby.

Recent work has gone into adding types to Python, the other eminent data science language. Typilus is an interesting piece of recent work which explores using neural networks to infer types for Python programs [Allamanis et al. 2020], and Python itself added support for type hints in Python 3.5 [Python Team 2020]. There is no previous work on types for R, besides a short sketch of issues the type system would need to tackle [Turcotte and Vitek 2019]. We take inspiration in the aforementioned works but focus on adapting them to our target language.

2.2 The R Programming Language

The R Project is a key tool for data analysis. At the heart of the R project is a *vectorized, dynamic, lazy, functional, object-oriented* programming language with a rather unusual combination of features [Morandat et al. 2012] designed to ease learning by non-programmers and enable rapid development of new statistical methods. The language was designed Ihaka and Gentleman [1996] as a successor to S [Becker et al. 1988].

The basic building blocks of R are vectors of primitives (primitives in R are characters (strings), integers, logicals (booleans), doubles, complex numbers, and raw bytes). When we say that R is *vectorized*, we mean that primitive values are *always* vectors. For example, in R one might see the literal `2.5` and think it to be a scalar double, but it is in actuality a unit-length vector containing a single double. Vectors can also be explicitly constructed using the vector constructor `c(...)`, For example, `c(1L, 2L, 3L)` creates a vector of three integers (in R, integer literals are denoted by an `L`). To illustrate that scalars are vectors, in R `(c(2.5) == 2.5) == TRUE`.

The main object of our work is R functions, and functions in R have a number of quirks. Arguments can be assigned default values (which can be arbitrary expressions!), and functions can support variable argument lists, and functions can be called by specifying argument names explicitly in the call. To illustrate all of this in a single example, consider:

```
f <- function(x, ..., y=if(z==0) 1, z=0) {
  x + y + if (missing(...)) 0 else c(...)
}
```

This function has four formal parameters, `x`, `...`, `y` and `z`. Argument `x` can be bound positionally or passed by name. The vararg argument, `...`, is always passed positionally. The remaining two arguments must be passed by name as they are preceded by the `...` parameter. Arguments `y` and `z` have default values, in the case of `z` this is a constant, but `y`'s default value is an expression that depends on the value of `z` (if `z` is not zero, `y` will default to `NULL`). The body of the function will add parameters `x` and `y` with either the scalar 0 or the result of concatenating the varargs into a primitive vector. The function `missing` tests if an argument was explicitly passed. The following are some valid invocations of `f`:

```
> f(1)
  [1] 2          # a double vector, y is 0, ... is missing
> f(1, 2)
  [1] 4          # a double vector, y is 0, ... is 2
> f(1, 2, 3)
  [1] 4 5        # a double vector, y is 0, ... is 2, 3
> f(2, 3, x=1)
  [1] 4 5        # a double vector, y is 0, ... is 2, 3
> f(x=1, y=1)
  [1] 2          # a double vector, y is 1, ... is missing
> f(x=1, z=1)
  numeric(0)     # a double vector of length 0, y is NULL
> f(1L, 2L, y=1L)
  [1] 4          # an integer vector, y is integer 1, ... is integer 2
> f(1, y=c(1,2))
  [1] 1 2        # a double vetor, y is 1, 2, ... is missing
```

The above hints at the polymorphism of the language: `f` may return a vector of integers or of doubles, the length of the vector depends on the length of the varargs, and of `x` and `y`. As we mentioned, the language does not really differentiate between scalar and vectors. Some more exotic types that can be encountered include vectors of complex number and list of arbitrary types. Function `f` can be invoked with those as well.

```
> c1 <- complex(re=1, im=2)
> c2 <- complex(re=2, im=1)
> f(c1)
  [1] 1          # a double vector, x is complex, y is double
> f(c1, y=c2)
  [1] 2+1i       # a complex vector, x and y are complex
> l1 <- list(1, 2)
> l2 <- list(c1, c2)
> f(l1)
  [1] 1          # a double vector, x is a list of doubles
> f(l1, y=l2)
  [[1]]          # a list of complex, x is a list of doubles
  [1] 1+2i       #                    y is a list of complex

  [[2]]
  [1] 2+1i
```

R has a builtin notion of type that can be queried by the `typeof` function. Figure 1 lists all of the builtin types that are provided by the language; these are the possible return values of `typeof`. There is no intrinsic notion of subtyping in R, but in many contexts a `logical` will be coerced to `integer`, and an `integer` will be coerced to `double`. Some odd conversions can occur in corner cases, such as `1<"2"` holds and `c(1,2)[1.6]` returns the first element of the vector, as the double is converted to an integer. R does not distinguish between scalars and vectors (they are all vectors), so `typeof(5) == typeof(c(5)) == typeof(c(5,5)) == "double"`.

All vectorized data types have a distinguished missing value denoted by `NA` (for “not available”). The default type of `NA` is `logical`. We can see that `typeof(NA)=="logical"`, but `NA` inhabits every type: `typeof(c(1,NA)[2]) == "double"`. In addition to `NA`, R has a `NULL` value, and to understand why R has both a `NULL` value and `NA` values it is useful to remember that R is primarily designed for statistics and data science. In experimental data science, it is useful to have a notion of a "missing observation", which is often NaN (not a number, or similar) in, e.g., MATLAB and Python. Since vectors are unityped in R, it *must* have a "NaN" value for *each* primitive type. There single `NULL` null-typed value in the language to encompass all other cases. In a sense, you can think of `NA` as representing a missing data point, and `NULL` as a missing data *set*.

With one exception, all vectorized data types are monomorphic. The exception is the `list` type which can hold values of any other type including `list`. For all monomorphic data types, attempting to store a value of a different type will cause a conversion. Either the value is converted to the type of the vector, or the vector is converted to the type of the value.

Vectors:	
`logical`	vector of boolean values
`integer`	vector of 32 bit integer values
`double`	vector of 64 bit floating points
`complex`	vector of complex values
`character`	vector of strings values
`raw`	vector of bytes
`list`	vector of values of any type
Scalars:	
`NULL`	singleton null value
`S4`	instance of a S4 class
`closure`	function with its environment
`environment`	mapping from symbol to value
Implementation:	
`special`, `builtin`, `language`, `char`, . . . , `externalprt`,	`symbol`, `pairlist`, `promise` `any`, `expression`, `bytecode`, `weakref`

Fig. 1. Builtin Types

Over the years, programmers have found the need for a richer type structure and have added *attributes*. The best way to think of attributes is as an optional map from names to values that can be attached to any object. Attributes are used to encode various type structures. They can be queried with functions such as `attributes` and `class`. The addition of attributes lets programmers extend the set of types by tagging data with user-defined attributes. For example, one could define a vector of four values, `x<-c(1,2,3,4)` and then attach the attribute `dim` with a pair of numbers as value: `attr(x,"dim")<-c(2,2)`. From that point, arithmetic functions will treat `x` as a 2x2 matrix.

Another attribute that can be set is the `class`. This attribute can be bound to a list of class names. For instance, `class(x)<-"human"` will set the class of `x` to be `human`. The main usage of attributes is for object-oriented programming. There are three object-orientation frameworks in R: S3, S4, and R5. The S3 object system support single dispatch on the class of the first argument of a function, whereas the S4 object system allows multiple dispatch (on all arguments). R5 allows for users to define objects in a more imperative style. Some of the most widely used data types leverage attributes, e.g., data frames and matrices. A data frame, for instance, is a list of vectors with a class and a column name attribute, and matrices are vectors with a `dims` attribute.

Scalar data types include the distinguished `NULL` value, instances of classes written using the S4 object system, closures, and environments. The implementation of R has a number of other types listed in Figure 1 for reference.

3 DESIGNING A TYPE LANGUAGE FOR R

In this section, we set out to propose a candidate design for a type language to describe the arguments and return values of R functions. The goal is not to propose a final design, but rather to propose a starting point for an iterative design process. Fig. 2 presents our type language. An early design choice was to stay close to the R language and only depart in small and hopefully non-controversial ways. Functions types have the form $\langle A_1, \ldots, A_n \rangle \to T$ where each A_i argument is either a type T or `...`, a variable length argument list. The rest of this section details and motivates our design.

T	::=	**any**	*top type*
	|	**null**	*null type*
	|	**env**	*environment type*
	|	S	*scalar type*
	|	V	*vector type*
	|	$T \mid T$	*union type*
	|	$?\,T$	*nullable type*
	|	$\langle A_1, ..., A_n \rangle \rightarrow T$	*function type*
	|	**list**$\langle T \rangle$	*list type*
	|	**class**$\langle ID_1, ..., ID_n \rangle$	*class type*

A	::=	T	*arguments*
	|	...	
V	::=	S[]	
	|	^S[]	*na vector types*
S	::=	**int**	
	|	**chr**	
	|	**dbl**	
	|	**lgl**	
	|	**clx**	
	|	**raw**	

Fig. 2. The R type language

Scalar vs. Vectors. We distinguish between vectors of length 1, and vectors of any dimension. The base types, **int**, **dbl**, **clx**, **lgl** (booleans), **chr**, and **raw** (bytes), can be either vectors (e.g., **int**[]) or scalars (e.g., **int**). A vector can happen to be of length 1, and thus a scalar value is also a vector. Vectors are homogeneous, in that a vector of doubles contains only doubles.

NA values. In R, a value can be *not available*, written NA. Each of basic types has its specific NA. Thus there is an **int** NA as well as a **dbl** NA, and they are different. The type system allows one to distinguish between vectors that may contain NAs (written ^**int**[]) and those who are guarantee to be NA-free (**int**[]). A NA-free vector can be treated as a vector that may NAs. We do not allow scalar values to be NA and the type **raw** does not allow NAs.

Unions. We support untagged unions of types written $T_1 \mid ... \mid T_n$.

Nullables. The **null** type is inhabited by a singleton NULL value often used as a sentinel or a stand in for arguments with no value. To capture this behavior, we introduce a nullable type $?\,T$. The functional difference between the NA and NULL values is that the latter cannot be stored in vectors.

Lists. Heterogeneous collections are implemented using lists. Lists and vectors are closely related: a vector converts to a list with `as.list`, and lists to vectors with `unlist` (coercions may ensue). The list type is parameterize: **list**$\langle T \rangle$.

Classes. R has more than one notion of type. As we mentioned, values can be attributed, and a very important attribute is the class of a value. A class is a list of names that are used to mimic object-oriented programming. The type system includes class types written **class**$\langle ID_1, \ldots, ID_n \rangle$.

Environments. Environments are lists with reference semantics: mutating a value in an environment is performed in-place. They are used to store variables and to escape from the copy-on-write semantics of other data types.

4 EXTRACTING AND CHECKING SIGNATURES

For this paper, we have built tooling to (*a*) automate the extraction of raw type signatures from execution traces, (*b*) infer type signatures from a set of raw types, and (*c*) validate the inferred signatures by the means of contracts. Figure 3 shows an overview of this pipeline. This section gives details of the main steps. The pipeline is run using GNU parallel [Tange et al. 2011] on Intel Xeon 6140, 2.30GHz with 72 cores and 256GB of RAM.

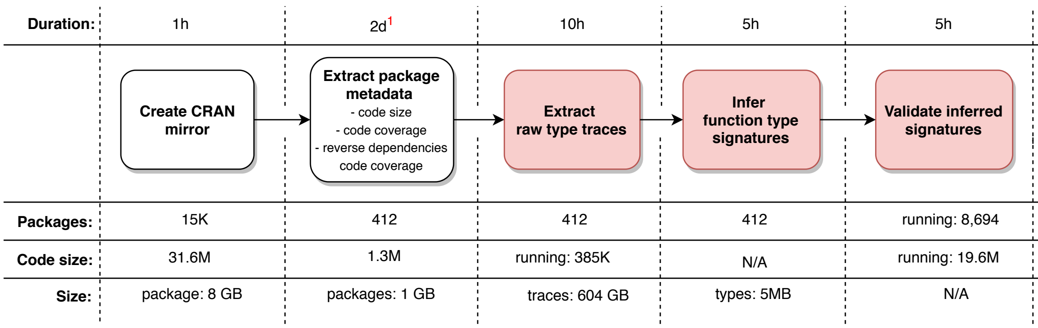

Fig. 3. The Analysis Pipeline. (1) the rather long time is because we had to extract metadata from all CRAN packages before we could compose our 412 package corpus. Here CRAN refers to the Comprehensive R Archive Network, R's repository of packages and libraries.

4.1 Extracting raw type signature from traces

We implemented Typetracer, an automated tool for extracting raw type signatures from execution traces of R programs. The goal of this tool is to output a tuple $\langle f, t_1, \ldots, t_n, t\rangle$ for each function call during the execution of a program, where f is an identifier for a function, t_i are type-level summaries of the arguments and t is a summary of the return value.

While this task is seemingly simple, the details and their proverbial devil are surprisingly tricky to get right and to scale to large, long running programs. To avoid starting from scratch, our implementation reuses R-dyntrace, an open source dynamic analysis framework for R [Goel and Vitek 2019] which consists of an instrumented R Virtual Machine based on GNU-R version 3.5.0. The framework exposes hooks in the interpreter to which user defined callbacks can be attached. These hooks include function entry and exit, method dispatch for the S3 and S4 object systems, the longjumps used by the interpreter to implement non-local exit, creation and forcing of promises, variable definition, value creation, mutation and garbage collection.

Raw types. The type information output by the tool includes the *type tag* of each value. Internal types are translated to names in the proposed type system. The next bit of information is the *class*, an optional list of names that may be absent, and, in some cases, is implicit (i.e. the interpreter blesses some values with the `matrix` and `array` classes even without attributes). Depending on a value's type, the tool collects further information:

- For vectors, the presence of `NA` values.
- For lists, element types by a recursive traversal.
- For promises, an approximation of the expression type.

To obtain these raw types, we make use of R's C FFI and use low-level machinery to collect information from the R language run-time. Types are completed during post-processing, and rely on the detailed information made available by these low-level reflection mechanisms.

Promises. The fact that arguments are lazy (expression are packed into promises and only evaluated on first access) complicates information gathering. For example, some promises may remain un-evaluated, and it would be erroneous to force them as they may side-effect and change program behavior. To deal with unevaluated arguments, we make an initial guess for each argument at function entry. If the promise is later forced, we simply update the recorded type for the argument.

Missing arguments. Parameters which receive no values when the function is called are termed missing. This occurs when a function was called with too few arguments and no default values were specified for missing arguments. We record a `missing` type for such argument. There are two obvious ways to deal with missing arguments: type them as **`any`** or type them as some unit type. As we are performing a dynamic analysis, we conservatively type them as **`any`**.

Non-local returns. When a function exits with a longjump, there is no return value to speak of. To ensure call traces are valid when a longjump occurs, we intercept the unwinding process and record a special `jumped` return type for function returns that are skipped. As we cannot be sure of the intended return value, these `jumped` values become **`any`** types.

Varargs. Arguments that are part of a function's varargs (denoted `...`) are ignored. We do not attempt to give varags a type in this type system.

Implementation details. We primarily rely on eight callbacks: `closure_entry`, `closure_exit`, `builtin_entry`, `builtin_exit`, `special_entry`, `special_exit`, `promise_force_entry`, and finally `promise_force_exit`. The function-related callbacks are used mainly for bookkeeping: the analysis is notified that a construct has been entered by pushing the call onto a stack. The calls themselves store a trace object that holds the type information. As R can perform single or multiple dispatch on function arguments depending on their class, the relevant information is kept by the `_entry` variants.

4.2 Inferring type signatures from raw types

The output of the Typetracer tool consists of a set of tuples of raw types, each representing a function call in some program's execution. The inference step consolidates the different tuples corresponding to a particular function definition across multiple programs and distills them into a single type signature. The shape of the inferred function signatures is:

$\langle T_{1,1} \mid T_{1,i}, \ldots, T_{n,1} \mid T_{n,j} \rangle \rightarrow T_1 \mid ... \mid T_k$

In other words, we take the union of the types occurring at individual argument positions rather than an union of function types. Furthermore, we apply some transformation on the types to keep the size of types in check. Figure 4 overviews the main simplification rules as well as the simple notion of subtyping that we have adopted here. Our subtyping rules are both born out of necessity (e.g., scalars are subtypes of vectors since a. that is the case in R and b. we want to allow types like **`dbl`** | **`dbl`**`[]` to simplify), as well as from prevalent data type conversion with no ambiguity (e.g., **`int`** to **`dbl`**).

Assuming that type sequences can be reordered freely, we rewrite types to minimize their size by removing redundant types, types that are subsumed by subtyping, immutable lists, and remove **`null`** type to replace them with nullables. Functions are limited to twenty arguments as a simplifying assumption. Higher-order functions are conservatively inferred as **`any`** $\rightarrow$ **`any`** by Typetracer.

4.3 Checking types signatures with contracts

One way to validate an inferred function type is to check that it is respected in a different program which uses the function. For this purpose, we developed ContractR, an R package that allows one to decorate a function with assertions. We use it to insert type checking of arguments and return values.

ContractR's primary logic has been implemented in C++ to reduce its runtime overhead, and we have not observed a single segmentation fault during our use of the package. It has been tested with GNU R-3.5.0 and hardened with a battery of 400 test cases. It works by modifying function definitions to insert a call to a type-checking function.

$$
\begin{array}{rcll}
T \mid T & \Rightarrow & T & \\
T \mid T' & \Rightarrow & T & \textit{iff } T' <: T \\
\texttt{list}\langle T\rangle \mid \texttt{list}\langle T'\rangle & \Rightarrow & \texttt{list}\langle T\rangle & \textit{iff } T' <: T \\
\texttt{null} \mid S_1[\,] \mid \ldots S_n[\,] & \Rightarrow & \hat{}S_1[\,] \mid \ldots \hat{}S_n[\,] & \\
\\
\hat{}S[\,] & <: & S[\,] & \\
T & <: & ?\,T & \\
\texttt{list}\langle T\rangle & <: & \texttt{list}\langle T'\rangle & \textit{iff } T <: T' \\
S & <: & S[\,] & \\
\texttt{lgl} & <: & \texttt{int} & \\
\texttt{int} & <: & \texttt{dbl} & \\
\texttt{dbl} & <: & \texttt{clx} &
\end{array}
$$

Fig. 4. Simplification rules and subtyping

In terms of usability, our checker is enabled automatically whenever a package is loaded in R. So, a simple invocation of `library(contractr)` causes contracts to be injected in all packages. On loading, ContractR scans all packages in the user's workspace and inserts contracts in functions for which type signatures are available. Then ContractR sets up package load hooks which are executed when new packages are loaded. Furthermore, ContractR automatically removes contracts from all functions and restores them to their original state when it is unloaded. The type signatures can be provided in an external file, thus avoiding the need to change the source code of checked packages. Type declarations can also be written directly in the adjunct function comments using Roxygen2 [2] annotation tags. ContractR introduces a new use `@type` tag. When a package these type signatures are extracted and stored in the appropriate location. The following is an example of a type annotation using Roxygen2:

```
#' @type <chr> => int
#' @export
file_size <- function(f) { ... }
```

ContractR provides an extensive API that allows users to explicitly insert contracts during interactive development by supplying the type signature as a string, selectively disable contract checking, and inspecting type assertions.

Checking the values. A number of properties can be checked by a simple tag check, namely whether a value is a null, environment, vector or scalar. Other properties require an inspection of the contents of a value, such as checking for the absence of `NA` or the type of the elements of list. Union types require checking for each of the members of the union. As the usage of higher-order functions remains limited in R, we do not check higher-order functions in this version of the tool. Checking them could be done by dynamically wrapping functions with checks. When argument values are wrapped in promises (this is not always the case due to compiler optimizations), in order to retain the non-strict semantics of R, the expression held in the promise is wrapped in a call to the type checker, and type checking is delayed until the promise is forced. This leads to corner cases such that the type checking of a function may happen after that function has returned.

Return values require care as well. Functions return the last expression they evaluate, thus to avoid having to analyze the code of the called function the checker will register a callback on the exit hook. The hook is executed in the function call's environment. Another wrinkle is due to

[2] https://cran.r-project.org/web/packages/roxygen2/

longjumps which causes active function calls on the stack to be discarded. When they are discarded, their exit hooks are called but they do not have a return value to type-check. ContractR deals with this problem by allocating a unique sentinel object which serves as the return value for calls that are discarded. The exit hook does not call the type-checker if it see the sentinel.

5 PROJECT CORPUS

For this paper we have selected 412 packages consisting of 760.6K lines of R code and 534.4K lines of native code (C/Fortran). Figure 5 shows these packages: the size of the dots reflects the project's size in lines of code including both R and native code[3], the x-axis indicates the expression code coverage as a percentage and the y-axis gives the number of reverse dependencies in on log scale. Dotted lines indicate means. Packages with over 5K lines of code are annotated.

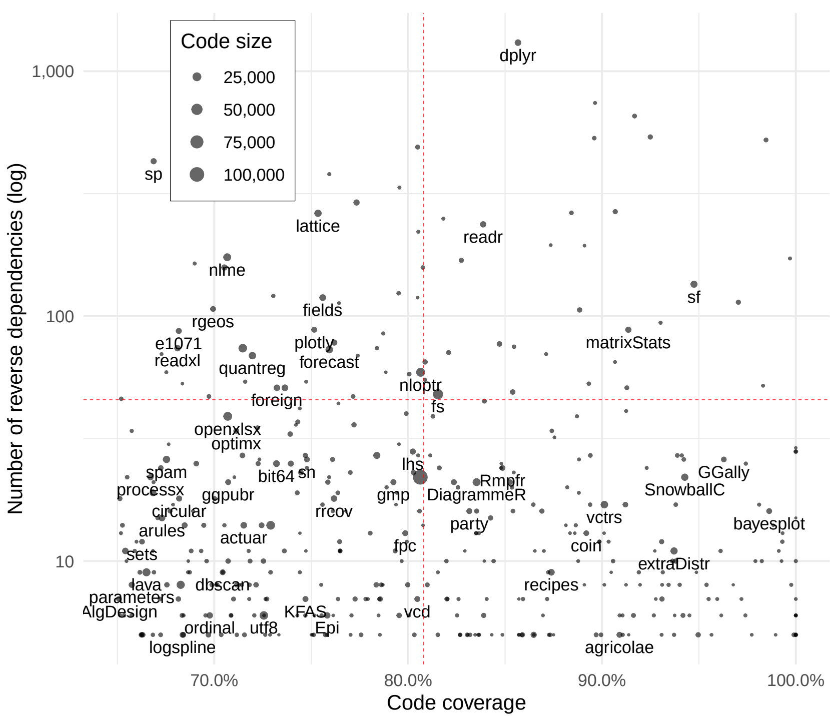

Fig. 5. Corpus

These packages come from the Comprehensive R Archive Network (CRAN[4]), the largest repository of R code with over 15.4K packages[5] containing over 19.6M and 12.2M lines of R and native code respectively. Unlike other open code repositories such as GitHub, CRAN is a curated repository where each submitted package must abide by a number of well-formedness rules that are automatically checked to assess package quality. Notably, CRAN packages must have a set of *runnable* example, test, and vignette code which showcase package functionality. The code is run by CRAN, and only a successfully running package is admitted to the archive.

[3]Lines of source code reported excludes comments and blank lines, counted by *cloc*, *cf.* https://github.com/AlDanial/cloc
[4]http://cran.r-project.org
[5]CRAN receives about 6 new package submissions a day [Ligges [n. d.]]

We have downloaded and installed all available CRAN packages. Out of the 15.4K packages, we managed to install 13.5K. The main reason for this is that R-dyntrace (and, by extension, Typetracer) is based on GNU-R 3.5.0 and some of the packages are not compatible with this version. Some packages also require extra native dependencies which were not present on our servers. We defined two criteria for including a package into the corpus: (1) the package must have a runnable code that covers a significant part of the package source code from which type signatures could be inferred, and (2) the package must have some reverse dependencies that will allows us to evaluate the inferred types using the runnable code from these dependencies. The concrete thresholds used were: at least 65% of expression coverage and at minimum 5 reverse dependencies. The code coverage was computed for each package using Covr[6], the R code coverage R tool. The reverse package dependencies were extracted from the package metadata using built-in functions.

The 412 selected packages contain 385.8K lines of runnable code in examples (98.9K), tests (258K) and vignettes (28.9K). Running this code results on average in 80.8% package code coverage (the average for all of CRAN is 65.6%). Together, there is 18.8K (on average 45.5, median 12; CRAN average is 12.8, median 2) reverse dependencies with 11.2M runnable lines of code resulting in 45.9% coverage (on average) of the corpus packages. Together there are 38.2K defined R functions (17.4K are from the packages' public APIs). 11.8K are S3 functions, either S3 generics or S3 methods. Packages in the corpus define 81 S3 classes.

User code. To represent end-user code in the corpus, we turned to Kaggle, an online platform for data-science and machine-learning. The website allows people to share data-science competitions and data-analysis problems together with data for which users try to find the best solution (something like a repository of hackathon or datathon code). The solutions, called *kernels*, are then posted on Kaggle either as plain scripts or as notebooks. One of the most popular competitions is about predicting passenger survival on Titanic[7] with 2,890 kernels in R (over 1/4 of all available R kernels) which we used for our corpus.

Unlike CRAN, Kaggle is not a curated repository and therefore there are no guarantees about the quality of the code. After downloading all of the 2,890 kernels and extracting the R code from the various formats,[8] we found that 1,079 were whole-file duplicates (37.3%). From the resulting 1,811 kernels, 1,019 failed to execute. Next to various runtime exceptions, common problems were missing libraries (no longer available for R 3.5), parse errors and misspelled package names. The final set contains 792 kernels with 33.7K lines of R code. The Kaggle kernels are used for additional validation of the inferred types.

Type usage. During execution 3,147 different types were observed. Classes are the most common types, accounting for roughly 31% of types of arguments. The most common classes are `matrices` (12%), `data.frames` (7.5%), `formulas` (2%), `factors` (2%), and `tibbles` (2%). Roughly 25% of classes are part of R's base libraries, the others are user-defined. Scalars and vectors are the next most common kind, making up 41% of remaining types. with scalars making up 28% of types and vectors 12%. Nulls and lists follow at 8% and 7% respectively, and the vararg type makes up 6% of arguments. This all totals up to over 90% of types. Table 1 reports on the 10 most frequent types occurring in the corpus. The first row of the table reads: the **`dbl`** type occurs in 12,298 (11.24%) argument types, and accounts for over 12 million (20.3%) of the types observed by Typetracer's dynamic analysis.

[6]https://github.com/r-lib/covr
[7]https://www.kaggle.com/c/titanic
[8]We use rmarkdown to convert from notebooks to R.

Type	Args	% of Args	Observations	% of Obs.
dbl	12,298	11.24	12,152,787	20.3
lgl	9,366	8.7	6,650,294	11.1
null	8,799	8.0	2,187,611	3.7
chr	8,727	8.0	2,564,207	4.3
dbl[]	7,190	6.6	4,934,773	8.2
...	6,611	6.0	6,075,874	10.1
any	6,120	5.6	339,299	0.6
chr[]	4,325	4.0	1,060,466	1.8
class⟨matrix⟩	4,152	3.8	2,805,718	4.7
class⟨data.frame⟩	2,608	2.4	352,655	0.6

Table 1. Top types of arguments in R

6 EVALUATION

We ran Typetracer on the test, example, and vignette code of the aforementioned corpus of 412 packages and successfully inferred types for 25,215 functions. Table 2 illustrates the process with ten representative signatures. Many of the features of our type language are represented here, and some signatures are telling of the function's behaviour. For example, consider `decrypt_envelope`: the first three parameters of the function are byte arrays, and the fourth argument is an RSA key, used to decrypt some of the inputs, and the output of the function is another byte array. As another example, consider `Traverse`: according to the function documentation, it takes the root of a tree and traverses it in an order specified by the second argument. We see that reflected in the type, where the first argument has type **class**⟨Node, R6⟩ and the second argument had type **chr**[], representing the traversal order.

Function	Type Signature
`dplyr::group_indices`	⟨**class**⟨data.frame⟩, ...⟩ → **int**[]
`moments::all.cumulants`	⟨**class**⟨matrix⟩ | **dbl**[]⟩ → **class**⟨matrix⟩ | **dbl**[]
`diptest::dip`	⟨**dbl**[], **chr** | **lgl**, **lgl**, **dbl**⟩ → **class**⟨dip⟩ | **dbl**
`stabledist::cospi2`	⟨**dbl**[]⟩ → **dbl**[]
`matrixcalc::matrix.power`	⟨**class**⟨matrix⟩, **dbl**⟩ → **class**⟨matrix⟩
`data.tree::Traverse`	⟨**class**⟨Node, R6⟩, **chr**[], **any**, **any**⟩ → **list**⟨**any**⟩
`openssl::decrypt_envelope`	⟨ **raw**[], **raw**[], **raw**[], **class**⟨key, rsa⟩, **any**⟩ → **raw**[]
`dbplyr::set_win_current_group`	⟨? **chr**[]⟩ → ? **chr**[]
`openssl::sha256`	⟨ **raw**[], ? **raw**[]⟩ → **raw**[]
`forecast::initparam`	⟨?**dbl**, **any**, **any**, **any**, **chr**, **chr**, **lgl**, **dbl**[], **dbl**[], **any**⟩ → **dbl**[]

Table 2. Select Type Signatures

This section attempts to evaluate how well the proposed type system is able to describe the actual type signatures of functions. For this we focus on how often there is a single type for a particular argument; this is because union types and **any** are less accurate (and would likely require a more refined notion of subtyping or parametric polymorphism). Then, we evaluate how robust the inferred signatures are by checking that they remain valid for other inputs. Lastly, we try to see if the current proposal would be useful to programmers by allowing them to remove ad hoc checks and providing useful documentation.

6.1 Expressiveness

The first part of our evaluation attempts to shed light on how good a fit our proposed type system is with respect to common programming patterns occurring in widely used R libraries.

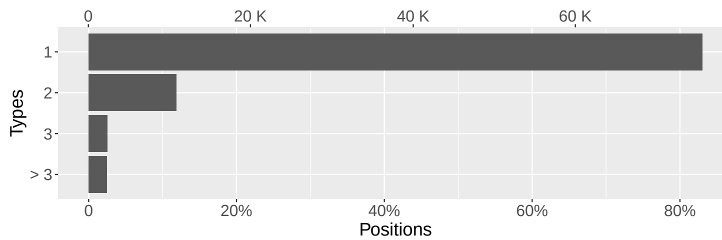

Fig. 6. Size of unions

First we look at the share of monomorphic arguments and function signatures. Monomorphic in this context means that the type is not relying on **any** or including a union. The import of monomorphism in this context is that it means our type language can accurately capture an argument's type or a function signature. We get to that number in two steps. Fig. 6 shows the number of inferred argument types and their size (in terms of members of the union). The figure shows the most functions do not require a union at all (83.1% of arguments do not have a union), and only 2.5% positions have unions with more than three members.

Types	Parameter #	%	Cumulative %
scalar	35064	33.33	33.33
class	24256	23.06	56.39
vector	13025	12.38	68.77
...	9142	8.69	77.46
null	7694	7.31	84.77
any	7614	7.24	92.01
list	3558	3.38	95.39
^vector	2923	2.78	98.17
function	1427	1.36	99.52
environment	500	0.48	100.00

Table 3. Singleton Type Categories

Table 3 provides a breakdown of types occurring in arguments without a union. Scalar, class and vector are the most common type categories. The shaded rows correspond to polymorphic types. When an argument's type is **`null`**, we say that the argument is polymorphic due to a limitation in our analysis: In R, it is common for programmers to include default values for arguments, and in many cases this value happens to be `NULL`. Our type analysis will report a `null` type for these arguments if they are *never passed a value* during testing. We interpret these instances of **`null`** as polymorphic to capture that we cannot be sure of the actual type.

Removing the aforementioned instances of polymorphism gives us 68.9 K (60.4%) monomorphic positions in a corpus of 114 K parameters. With close to 60% of monomorphic argument or return values, it is fair to say that even a simple type language provides significant benefits.

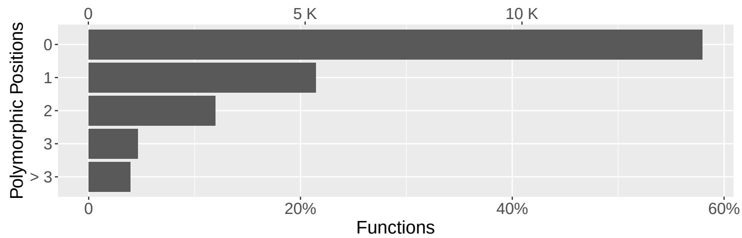

Fig. 7. Function Polymorphism

If we look at the numbers from the point of view of functions and count how many of their arguments are polymorphic, we observe that 58.0% (14.2 K) functions are monomorphic. The remaining 42.0% (10.3 K) have at least one union or polymorphic parameter or return type. Figure 7 shows the distribution of functions against the number of polymorphic arguments. Finally, we count that 38 out of the 412 packages export only monomorphic functions.

6.1.1 Discussion. A number of lessons can be drawn from the data we have gathered.

NAs. Our data supports making the presence of `NA`s explicit. Only 2923 (or 2.78%) of arguments are marked as possibly having `NA`s, thus the overwhelming majority of types appear to be `NA`-free. In practice, programmers check for them and sanitize them if they are present. Consider the `binom` package for computing confidence intervals and its `binom.profile` function. This attached code snippet highlights a data sanitization pattern: the programmer first binds the vectors into a matrix, then finds rows where both columns are not `NA`, extracts non-`NA` values and stores them into `x` and `n` respectively.

```
binom.profile <- function(x, n, conf.level=0.9, maxsteps=50, ...) {
  xn <- cbind(x = x, n = n)
  ok <- !is.na(xn[, 1]) & !is.na(xn[, 2])
  x <- xn[ok, "x"]
  n <- xn[ok, "n"]
  # ...
}
```

Scalars. The data also suggests that programmers often use scalars, and do dimensionality checks on their data. In our data 25,064 (or 33.33%) of the arguments are scalar types. While not completely surprising, this is a rather large number. Consider the `hankel.matrix` function, it takes two arguments and checks that `n` is **`int`**, that `x` is a vector, and also, indirectly, that `n` is a a scalar (this comes from the fact that it is used in the guard of a conditional which fails if `n` is not a scalar).

```
hankel.matrix <- function( n, x ) {
  ### n = a positive integer value for the order of the Hankel matrix
  ### x = an order 2 * n + 1 vector of numeric values
  if ( n != trunc( n ) ) stop( "argument n ix not an integer" )
  if ( !is.vector( x ) ) stop( "argument x is not a vector" )
  m <- length( x )
  if ( m < n ) stop( "length of argument x is less than n" )
  # ...
}
```

Nullables. The number of argument which may be `NULL` is 5057 (or 4.44%). This is a relatively small number of occurrences, but it is worth expressing the potential for the presence of `NULL` as these would likely inhibit optimizations.

Higher-Order Functions. Typetracer assigns the type **class**⟨function⟩ to function values. The number of positions that receive a function (possibly as part of a union) is 1,705, which is just 1.51% of all the positions for which we infer types. Given the small number of occurrences, it is not worth complicating the inferred types with a complete signature for these functions.

Structs. While experimenting with various design, we consider adding a struct type to capture lists with named elements that can be accessed with the `$` operators. We ended up discarding those types as they grew large and were often only representative of the example data being manipulated. Consider function `cv.model`, its argument `x` is observed to be of **class**⟨aov, lm⟩ or **class**⟨lm⟩. Internally, linear models are represented as lists with named elements. The pollution is illustrated by the lines after the function definition. They load an example data set and test the function `cv.model`. The `data(sweetpotato)` expression loads a sample data set. The fields of `sweetpotato` will be recorded when `cv.model` is called.

```
cv.model <- function(x) {
  suma2 <- sum(x$residual^2)
  gl <- x$df.residual
  promedio <- mean(x$fitted.values)
  return(sqrt(suma2/gl)*100/promedio)
}

data(sweetpotato)
model<-aov(yield~virus, data=sweetpotato)
cv.model(model)
```

Objects. While we record classes, our analysis does not deal with method dispatch. R has multiple disparate object systems called S3, S4, and R5. The `class` attribute is used by these systems to dispatch methods. S3 does single dispatch, S4 does multiple dispatch and R5 supports imperative objects. The mechanics of S4 dispatch are more complex than for S3, and users can define their own class hierarchies that we would need to incorporate in our type analysis and contract checking frameworks. We found limited use of S4 during our analysis. Coming up with a type system that accounts for all of these factors and consolidates multiple object-orientation frameworks in a single language design is an interesting problem in and of itself, one we leave for future work.

Matrices. Matrices are instances of the eponymous class, representing 10.71% of all classes occurring in types. They have a `dims` attribute indicating dimensions, and while not codified in the language semantics, many internal functions coerce vectors to matrices automatically. For example, the `rowWeightedMeans` function calculates the weighted means of rows. The programmer added a type check for `x`.

```
rowWeightedMeans <- function(x, w=NULL, rows=NULL, cols=NULL, na.rm=FALSE, ...) {
  if (!is.matrix(x)) .Defunct(msg = sprintf("'x' should be a matrix.)
  # ...
}
```

Data Frames. One of the most popular classes in R is the `data.frame` class, making up 8.15% of observed classes. Data frames and the derivative `tibble` and `data.table` types underpin much of the idiomatic usage of R. One way to deal with data frames is through the struct type, with a named field for each column of the data frame, but as mentioned previously structs introduced undue noise. Further complicating data frames is that many functions built to operate on them operate in a name-agnostic way. For instance, the `tidyverse` package ecosystem allows programmers to pass column names to functions which operate on their data frames. In base R, typical data frame use is to use string column names to select rows from the frame (unless only a single column is of interest, wherein the `$` syntax is appropriate). In sum, data frames are a popular class of R values, and have spawned many derivative data types, such as tibbles and data tables. We include a **class**⟨data.frame⟩ type to cover most use-cases, and we leave a richer type for future work on a full fledged object-oriented type system.

6.2 Robustness

We now ask *how robust are the inferred types*? To measure this, we conducted another large-scale experiment: for each package in the corpus, using the inferred type signatures as contracts we ran all of the CRAN reverse dependencies for that package. In total we ran 8,694 unique packages and recorded 98,105,161 total assertions. Overall, we found that only 1.98% of contract assertions failed. The limit on number of arguments (we record only 20) accounted for 0.07% failed assertions. We found that 97.60% of parameter types and 87.70% of function types never failed. The number of immaculate function types increases to 89.70% if we discount S3 object method dispatch. Overall, these numbers are promising, and suggest that the type signatures are indeed robust.

We break down the failed assertions by type in Table 4. Accounting for 36.36% of assertion failures are cases where a **dbl**[] is passed where a **class**⟨matrix⟩ is expected. Considering these types, we might imagine them to be compatible, as a vector is just a one-dimensional matrix. However, not allowing this coercion was a deliberate design decision, as coercion of this kind is ad hoc at best, and unfortunately not a practice codified in the language. For example, if the vector has length n, should it be a $1 \times n$ or $n \times 1$ dimensional matrix?

In a similar vein, another popular failing assertion is checking if a **dbl**[] has type **int**[], another case of commonly performed coercion. We did not include these types of coercions in our type annotation framework as programmers cannot rely on them, and it is not always the case that the coercions are safe to perform.

The second row of the table is exemplary of a pattern where vectors are passed when scalars are expected. In these cases, the functions exhibiting these assertion failures were under-tested, and can operate just as well on vectors of values. As an example, this failure occurred in functions from the `lubridate` package which provides date/time functionality. Many functions, e.g., `date_decimal` and `make_datetime` turn doubles into **class**⟨POSIXct, POSIXt⟩ (which are dates in R), and they can easily operate on vectors of doubles, producing lists of dates.

Finally, we point out that assertion failures of, e.g., **class**⟨data.frame⟩ values being passed to **class**⟨data.frame, tbl, tbl_df⟩ arguments and **class**⟨xml_node⟩ values being passed to an argument expecting other XML-like classes are related to our simplified take on class types. Our type system does not encode user-defined subtyping and coercion, which could help address these mismatches.

In addition to the number of failed contract checks, we were interested in how many functions had a parameter where a contract check failed, and overall we found this to be the case in 12.29% of functions. To subdivide this number, we discounted functions that were performing S3 dispatch, as they exhibit user-defined polymorphism which we do not handle. Removing those functions, we see that the proportion of functions with failed contract checks falls to 10.30%. These remaining functions were under-tested, as calls to these functions represent only 2.73% of recorded calls during Typetracer's run on the core corpus to infer types.

<table>
<tr><th>Passed</th><th>Arg Type</th><th>Occurrences</th><th>% Total</th><th>Cumul. %</th></tr>
<tr><td>dbl[]</td><td>class⟨matrix⟩</td><td>705,036</td><td>36.36</td><td>36.36</td></tr>
<tr><td>dbl[]</td><td>dbl</td><td>189,800</td><td>9.79</td><td>46.15</td></tr>
<tr><td>chr</td><td>class⟨bignum⟩ | raw[]</td><td>100,100</td><td>5.16</td><td>51.31</td></tr>
<tr><td>class⟨simpleError, error, condition⟩</td><td>class⟨data.frame⟩ | class⟨matrix⟩ |
class⟨randomForest⟩ | dbl[]</td><td>78,197</td><td>4.03</td><td>55.34</td></tr>
<tr><td>class⟨data.frame⟩</td><td>class⟨data.frame, tbl, tbl_df⟩</td><td>58,809</td><td>3.03</td><td>58.37</td></tr>
<tr><td>class⟨matrix⟩</td><td>class⟨timeSeries⟩</td><td>53,482</td><td>2.76</td><td>61.13</td></tr>
<tr><td>dbl[]</td><td>int[]</td><td>33,350</td><td>1.72</td><td>62.85</td></tr>
<tr><td>dbl</td><td>class⟨data.frame⟩</td><td>32261</td><td>1.66</td><td>64.52</td></tr>
<tr><td>dbl[]</td><td>class⟨data.frame⟩</td><td>31361</td><td>1.62</td><td>66.13</td></tr>
<tr><td>class⟨xml_node⟩</td><td>class⟨xml_missing⟩ | class⟨xml_nodeset⟩ |
class⟨xml_document, xml_node⟩</td><td>30,330</td><td>1.56</td><td>67.70</td></tr>
</table>

Table 4. Top contract failures

<table>
<tr><th rowspan="2">Passed</th><th rowspan="2">Arg Type</th><th colspan="2"># Args Types</th><th rowspan="2">% Failure</th></tr>
<tr><th>Failed</th><th>Total</th></tr>
<tr><td>dbl[]</td><td>class⟨matrix⟩</td><td>18</td><td>1522</td><td>1.18</td></tr>
<tr><td>dbl[]</td><td>dbl</td><td>66</td><td>5865</td><td>1.13</td></tr>
<tr><td>chr</td><td>class⟨bignum⟩ | raw[]</td><td>1</td><td>3</td><td>33.33</td></tr>
<tr><td>class⟨simpleError, error, condition⟩</td><td>class⟨data.frame⟩ | class⟨matrix⟩ |
class⟨randomForest⟩ | dbl[]</td><td>2</td><td>2</td><td>100</td></tr>
<tr><td>class⟨data.frame⟩</td><td>class⟨data.frame, tbl, tbl_df⟩</td><td>23</td><td>196</td><td>11.73</td></tr>
<tr><td>class⟨matrix⟩</td><td>class⟨timeSeries⟩</td><td>21</td><td>39</td><td>53.85</td></tr>
<tr><td>dbl[]</td><td>int[]</td><td>31</td><td>624</td><td>4.97</td></tr>
<tr><td>dbl</td><td>class⟨data.frame⟩</td><td>2</td><td>1025</td><td>0.20</td></tr>
<tr><td>dbl[]</td><td>class⟨data.frame⟩</td><td>6</td><td>1025</td><td>0.59</td></tr>
<tr><td>class⟨xml_node⟩</td><td>class⟨xml_missing⟩ | class⟨xml_nodeset⟩ |
class⟨xml_document, xml_node⟩</td><td>1</td><td>1</td><td>100</td></tr>
</table>

Table 5. Results from Table 4, broken down by occurrences of the expected type as a parameter type

Turning our attention now to arguments, we found that only 2.40% of argument types failed. Table 4 showed the runtime occurrences, but that data alone does not tell the full story, as some failures may be overrepresented if, e.g., a failing contract assertion was in a loop. We were interested in knowing for each of the most common violations in Table 4, how many different arguments had that type, and how many of those exhibited the contract failure in question. Table 5 breaks down the failed assertions by type, folding away multiple identical failed contract assertions for the same parameter position. The first row of this table reads: a value of type **dbl**[] was passed to 15 different function parameters expecting a **class**⟨matrix⟩, of which there are 1522 in total: 18 (1.18%) of these **class**⟨matrix⟩-typed parameters were passed **dbl**[] values.

We see that even though the double vector and matrix issue was wildly prevalent in the raw, dynamic contract evaluation numbers, the number of actual function argument types that were violated is very small. The story is similar with the double and integer coercion we mentioned earlier: it represents many dynamic contract failures, but very few of the **int**[]-typed arguments have their contracts violated by **dbl**[]-typed values. Row six is interesting: we see that rather often arguments expecting **class**⟨timeSeries⟩ data are passed **class**⟨matrix⟩ values. This is a quirk of the `timeSeries` package, whose functions often accept matrices and vectors, converting them to time series in an ad hoc manner. Note that the code coverage of the `timeSeries` tests, examples, and vignettes package code is only 58%, which is one possible explanation of why these contract failures are occurring: the types that Typetracer generates are only as good as the test code its run on.

Passed	Arg Type	# Args Failed	# Args with Type	% Failure
dbl[]	**int**[]	31	624	4.97
dbl	**int**	21	519	4.05
chr[]	**chr** | **null**	10	256	3.91
^**lgl**[]	**lgl**[]	8	219	3.65
^**lgl**	**lgl**[]	5	219	2.28

Table 6. Highest failure rate among popular argument types, for argument signatures whose frequency is in the 90th percentile.

Passed	Arg Type	#Args Fail	#Args w Type	% Failure
class⟨matrix⟩	**class**⟨timeSeries⟩	21	39	35.90
dbl[]	**class**⟨timeSeries⟩	21	39	35.90
class⟨data.frame⟩	**class**⟨timeSeries⟩	14	39	35.90
class⟨dtplyr_step_first, dtplyr_step⟩	**class**⟨data.frame⟩ | **class**⟨data.frame, grouped_df, tbl, tbl_df⟩ | **class**⟨data.frame, tbl, tbl_df⟩	10	45	22.22
chr	**raw**	6	31	19.35

Table 7. Highest failure rate among popular argument types, for argument signatures whose frequency is in the 80th percentile.

Table 6 presents data on the most frequently violated contracts amongst the most frequently occurring argument types. We selected argument types which were in the 90th percentile of argument type occurrences, computed the most frequent type signature violations among them, and reported the most frequently violated contracts together with the type of the value that violated that contract. The first row of the table reads: 31 function arguments with **int**[] type are passed **dbl**[] values instead, and 624 arguments have **int**[] type, representing a failure rate of 4.97%.

Had we failed to capture some key usage pattern of R with our type annotation framework, we would likely see it here, and we can see this in action if we consider Table 7, which was obtained identically to Table 6 except selecting arguments in the 80th percentile instead. The most frequent argument type violation pattern in that of **class**⟨matrix⟩, **dbl**[], and **class**⟨data.frame⟩ values passed to arguments expecting **class**⟨timeSeries⟩. This occurs in 35.90% of such arguments, and represents cases where tests did not adequately cover all valid function inputs. Separate from the issue of testing, we can capture this behaviour with user-defined subtyping or coercion, as the data types which were passed are readily convertible to **class**⟨timeSeries⟩.

In sum, we believe that this evaluation shows that the type signatures we generate from traces are quite good. Only 1.98% of contract assertions failed at runtime, representing failures in as few as 2.40% of argument types. Even though 10.30% of functions had at least one argument type involved in a failing contract check, these functions are under-tested, representing only 2.73% of calls observed while inferring types.

6.2.1 Other Observations. We drew a number of other observations from the contract assertion failure data.

First, we were curious about how many **null**-typed values flowed into non-nullable arguments, and we found that they accounted for 6.46% of contract assertion failures in 141 functions. We manually inspected some of the offending functions and observed three main patterns.

First, we found that many of these errors occurred in arguments that have a `NULL` default value. This would be the case when the programmer only tested the function by passing values to these null-by-default arguments, and clients of the package make use of the default. As an example, we observed a contract assertion failure when `inner` and `labels` were `NULL` in the following function:

```
nlme::nfGroupedData <- function (formula, data = NULL, order.groups = TRUE,
  FUN = function(x) max(x, na.rm = TRUE), outer = NULL, inner = NULL,
  labels = NULL, units = NULL) {
  # ...
}
```

The other two patterns are for arguments with no default value, where either the call results in an error (perhaps explicitly handled by the programmer), or results in valid function behaviour that was untested by the original package designer. Here, the first case can be explained by a lack of testing, and the second case is explained by programmers not fully understanding R's language semantics. For example, we observed this kind of error in the following function:

```
BBmisc::convertIntegers <- function (x) {
  if (is.integer(x))
      return(x)
  if (length(x) == 0L || (is.atomic(x) && all(is.na(x))))
      return(as.integer(x))
  # ...
}
```

Here, if `x` is `NULL`, the second branch of the conditional will be triggered (as in R, `length(NULL) == 0`), and the function will return `as.integer(NULL)`, which curiously returns `integer(0)`, a zero-length vector of integers (one might expect it to return the integer `NA` value, or error).

Next, we analyzed how often vectors were passed to arguments expecting scalars. We found that 12.73% of dynamic contract assertion failures were of this type, and these errors were present in 114 functions. Besides an outright error, this kind of contract assertion failure might indicate that a function was not well-tested, in that it was only ever tested with unit-length vectors being passed to an argument which is intended to have a vector type. Further, these errors may reveal functions that were not designed with a vector-typed argument in mind, but can in fact handle vectors of values (in R, most functions that can accept scalars can also accept vectors). As an example, consider the function `BBmisc::strrepeat`, which takes a string and repeats it a specified number of times:

```
BBmisc::strrepeat <- function (x, n, sep = "") {
  paste0(rep.int(x, n), collapse = sep)
}

BBmisc::strrepeat("a", 3) # => "aaa"
BBmisc::strrepeat(c("a", "b"), 3) # => "ababab"
```

This function was only ever tested with unit-length vectors passed to `x`, even though technically it can handle longer vectors, as per the two sample calls above. This could be attributed to poor quality testing, or misunderstanding language semantics (e.g., misunderstanding the semantics of `paste0` and `rep.int`), but we found other instances of type errors where the functions really ought to have been tested with the offending type, for instance:

```
combinat::permn <- function (x, fun = NULL, ...) {
    if (is.numeric(x) && length(x) == 1 && x > 0 && trunc(x) == x)
        x <- seq(x)
    # ...
}
```

As per the documentation, this function is intended to take a vector, and generate all permutations of elements in that vector. If given a scalar "integer" n, it will generate all permutations for the list $[1, 2, ..., n]$. Interestingly, the function was only tested by passing an integer, and not ever with a vector (even though, presumably, that is the main utility of the function).

Finally, we were curious to see what patterns of errors occurred in arguments expecting classes. Overall we found that 81.44% of assertion failures were on arguments which were expecting a class in some way (51.13% of assertion failures were on monomorphic arguments expecting a class, the remainder on polymorphic arguments with at least one option being a class).

There are two broad divisions which account for most of the class-related contract assertion failures that are not outright errors. First, we observe a class of errors related to classes being passed to arguments expecting a different, yet convertible class. For instance, we observed **class**⟨data.frame⟩ values being passed to arguments expecting tibbles or data.tables (data frames have a straightforward conversion to these classes). Second, we observe a class of errors related more to coercion between simple data types and classes. As an example, consider the aforementioned assertion failures in the `timeSeries` package, and as a further example we found many instances of **class**⟨matrix⟩, **class**⟨data.frame⟩, and vectors being passed to arguments expecting **class**⟨array⟩, a generalization of matrices.

6.2.2 Kaggle. To further validate our inferred types, we repeated the experiment discussed in this section on end-user R code found on the Kaggle competition website.

By-and-large, we saw no meaningful difference in the two data sets, with the contract assertion failure patterns being repeated from the reverse dependencies. Overall, we observed that 2.14% of all contract assertions failed while running Kaggle code. If we remove assertion failures related to our simplifying assumption that function types will not have more than 20 arguments, that number drops to a mere 0.42%. In all, 15.98% functions had at least one contract failure. There were 19,038,496 assertions in total, on 970 functions.

Passed	Arg Type	Occurrences	% Total	Cumulative %
class⟨data.table, data.frame⟩	**class**⟨matrix⟩	20002	28.15	28.15
class⟨factor⟩	^**chr**[]	18344	25.81	53.96
class⟨factor⟩	**chr**[]	7519	10.58	64.54
chr[]	**list**⟨**int**[]⟩ | ... | **list**⟨**class**⟨formula, quosure⟩⟩	5385	7.58	72.12
class⟨ixforeach, iter⟩	**class**⟨dataframeiter, iter⟩ | ... | **class**⟨iforeach, iter⟩	4139	5.82	77.94

Table 8. Top contract failures in Kaggle kernels

Passed	Arg Type	# Args Failed	# Args with Type	% Failure
chr[]	**chr**	12	304	0.04
class⟨factor⟩	**chr**[]	7	199	0.04
dbl[]	**chr**[]	6	199	0.03
^**chr**[]	**chr**[]	4	199	0.02
int	**chr**	5	304	0.02

Table 9. Highest failure rate among popular argument types in Kaggle, for argument signatures whose frequency is in the 90th percentile.

To mirror our analysis of contract assertions on the reverse dependencies of our corpus, we show in Table 8 the most frequently failing contract in Kaggle. While we don't see many overlapping entries per se, the assertions exhibit similar patterns. For instance, *data tables* (which have **class**⟨data.table, data.frame⟩) are often passed to arguments expecting a **class**⟨matrix⟩. Data tables are essentially serve the same purpose as data frames and tibbles. As it happens, data tables can be coerced to matrices if their elements are unityped, and programmers will often interchange the two, as is the case here. One function producing many of these errors is the following:

```
class::knn <- function (train, test, cl, k = 1, l = 0, prob = FALSE, use.all = TRUE) {
    train <- as.matrix(train)
    test <- as.matrix(test)
    ...
}
```

We see that `class::knn` coerces its first two arguments to matrices. On the topic of coercion, rows two and three of Table are interesting as they depict *factors* being passed to arguments expecting character vectors. **class**⟨factor⟩ typed values are known as factors in R, and they are stored as a vector of integer values corresponding to a set of character values, and their purpose is to allow for R to quickly deal with categorized data. Factors can be readily converted to characters when needed, as evidenced by these assertion failures.

Another interesting entry in Table 8 is the fifth row, where a **class**⟨ixforeach, iter⟩ is being passed to an argument expecting a long union of classes, each of the form **class**⟨X, iter⟩. This is likely an instance where the user defined their own class, `ixforeach`, and wanted to use the `iterators` package (the user called `iterator::iter` with a **class**⟨ixforeach⟩, and it gained the class `iter` on return). As we mentioned, accounting for object-orientation in the type system is beyond the scope of this work, and such an inclusion would allow us to better type situations like this.

Table 9 mirrors Table 6 in showing contract violations on the most frequently occurring argument types. Here, our manual analysis has revealed similar failure patterns. In the case of vectors being passed to scalars, we find functions which can take vectors but were only tested with scalars (e.g., `stringr::str_to_upper` which converts a vector of characters to upper case, and `dplyr::anti_join`, which can join by a vector of column names but was only ever tested with a scalar). We also see possibly-NA character vectors being passed to NA-free character vectors. These assertion failures arise from a lack of testing: the offending functions are `str_trim`, `str_sub`, and `str_replace_all` from the `stringr` package. These functions are actually wrappers for other functions which have the correct argument type (`^`**chr**`[]`).

6.2.3 Discussion. Overall, the analysis discussed in this section has revealed two broad categories of contract assertion failures: those related to coercion, and those related to a class hierarchy. Our type system does not account for coercion as coercion in R is ad hoc at best, and it is implemented on a function-by-function basis, even in the core R packages. As for errors related to a class hierarchy, we aim to tackle this in future work, as designing a full fledged object-oriented type system for a language like R is outside of the scope of this work.

6.3 Usefulness of the Type Checking Framework

There are a number of ways to check the types of function parameters in R. The default and most common way[9] is to use the `stopifnot` function from the R base package. It takes a number of R expressions which all should evaluate to true otherwise a runtime exception is thrown with

[9] 87.6% of all runtime checks in the whole of CRAN

a message quoting the corresponding failed expression. For example, the following code checks whether a given parameter x is a scalar string:

```
stopifnot(is.character(x), length(x) == 1L, !is.na(x))
```

Besides stopifnot, there are 4 packages in CRAN[10] that focus on runtime assertions: *assertive*, *ensurer*, *assertr* and *assertthat*. *assertive* and *ensurer* have not been updated since 2016 and 2015 respectively, and *assertr* is used by only 2 other packages and currently focuses on checking properties of data frames. Only the *assertthat* package is maintained and used (with 211 reverse dependencies). The advantage of *assertthat* over the R's default is that it provides much better error messages.

One way to asses the usefulness of our type checking system is to find out how many of the existing type checking constrains could be replaced by ContractR. To measure this, we have extracted all calls to stopifnot and assertthat assertions and checked which among them could be either completely replaced by ContractR or at least partially simplified by removing a portion of an assertion expression. This is useful, because a common pattern is that the first part of the parameter assertion checks its type while the rest checks its value. In the example above, the whole expression could be replaced by a **chr** type check.

Out of the 412 packages, 153 use runtime assertions. Together there are 1,995 asserts in 1,264 functions. Among these, ContractR can replace 1,005 (50.4%) assertion calls across 114 packages and 688 functions. Furthermore, additional 1,223 (61.3%) asserts in packages 125 packages and 859 functions could have been simplified.

Checking the type of function parameters is not something that is seen often in the R code. In the whole of CRAN, there are only 32.3K asserts in 15.9K functions define in 2.4K packages. One may speculate that this is the case due to the verbosity and inconvenience of the existing assertion tools. Our system can infer type annotations for existing functions automatically. This can remove or partially remove over 61.3% of existing assertions.

7 CONCLUSION

Retrofitting a type system for the interactive and exploratory programming style of R is hard: The language is poorly specified and builds upon an eclectic mix of features such as laziness, reflection, dynamic evaluation and ad-hoc object systems. Our intent is to eventually propose a type system for inclusion in the language, but we are aware that for any changes to be accepted by the community, they must have clear benefits without endangering backwards compatibility. As a step towards this, we focus on a simpler problem: instead of an entire type system, we limited the scope of our investigation to ascribing types to function signatures. To this end, we designed a simple type language which found a compromise between simplicity and usefulness by focusing on the most widely used features of R. We presented Typetracer, a tool for inferring types for function signatures from runnable code, and ContractR, an easy-to-use package for R which allows users to specify function type signatures and have function arguments checked for compliance at runtime.

We evaluated our design by running Typetracer on a corpus of 412 of the most widely used R packages on CRAN, inferring signatures for exported functions, and testing those inferred signatures on the 8,694 reverse dependencies of the corpus. Overall, we found that our simple design fits quite well with the existing language: Nearly 80% of functions are either monomorphic or have only one single polymorphic argument. When we tested the types inferred by Typetracer during our evaluation, we found that only 1.98% of contract assertions failed. Furthermore, we found that our type language and contract checking framework would be useful to programmers,

[10]Packages are available on the CRAN website: https://cran.r-project.org/web/packages/

eliminate or otherwise simplify 61.3% of existing type checks and assertions in user code. In sum, we believe that our simple type language design is a solid foundation for the eventual type system for R.

ACKNOWLEDGEMENTS

This work was partially funded by NSERC.

REFERENCES

Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. arXiv:cs.PL/2004.10657

Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic Inference of Static Types for Ruby. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* https://doi.org/10.1145/1926385.1926437

Esben Andreasen, Colin S. Gordon, Satish Chandra, Manu Sridharan, Frank Tip, and Koushik Sen. 2016. Trace Typing: An Approach for Evaluating Retrofitted Type Systems. In *European Conference on Object-Oriented Programming (ECOOP).* https://doi.org/10.4230/LIPIcs.ECOOP.2016.1

Richard A. Becker, John M. Chambers, and Allan R. Wilks. 1988. *The New S Language.* Chapman & Hall, London.

Gavin Bierman, Martin Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP).*

Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. 2009. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*. https://doi.org/10.1145/1640089.1640110

Aviral Goel and Jan Vitek. 2019. On the Design, Implementation, and Use of Laziness in R. *Proc. ACM Program. Lang.* 3, OOPSLA (2019). https://doi.org/10.1145/3360579

Ross Ihaka and Robert Gentleman. 1996. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics* 5, 3 (1996), 299–314. http://www.amstat.org/publications/jcgs/

Uwe Ligges. [n. d.]. 20 Years of CRAN (Video on Channel9. In *Keynote at UseR!*

André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2014. Typed Lua: An Optional Type System for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications (Dyla'14).* Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2617548.2617553

Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *European Conference on Object-Oriented Programming (ECOOP).* https://doi.org/10.1007/978-3-642-31057-7_6

Python Team. 2020. Type Hints for Python. https://docs.python.org/3/library/typing.html.

Ole Tange et al. 2011. Gnu parallel-the command-line power tool. *The USENIX Magazine* 36, 1 (2011).

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed Scheme. In *Symposium on Principles of Programming Languages (POPL).*

Alexi Turcotte and Jan Vitek. 2019. Towards a Type System for R. In *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '19).* Association for Computing Machinery, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.1145/3340670.3342426

Julien Verlaguet. 2013. Hack for HipHop. CUFP, 2013, http://tinyurl.com/lk8fy9q.